САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Шаго Павел Евгеньевич

22.Б07-ПУ, 01.03.02 Прикладная математика и информатика

Планировщики

Научный руководитель

к.ф.-м.н., доцент Корхов В. В.

Санкт-Петербург

25 декабря 2024 г.

Содержание

Введение

С усложнением систем построенных человеком, появилась необходимость этими системами управлять. Значительные запросы на построение таких управляющих механизмов приходят из автоматизации и компьютеризации. Так, почти любое современное производство немыслимо без компьютера или микроконтроллера, который, например, управляет станком.

Крайне распространено Unix-подобное ядро Linux для операционных систем (ОС). Этот компонент содержит огромное количество планировщиков, среди которых 3 различных реализации планировщика потоков (процессов) ОС. Однако кроме них существуют также и специализированные решения, например, планировщики сетевого трафика или ввода-вывода, всего в ядре Linux содержится более 20 различных планировщиков и более 6 из них работают постоянно практически на всех системах.

Улучшения планировщиков имеют особую важность, поскольку они сразу же затрагивают большое число систем которые их используют. Такие улучшения несомненно приводят к значительному повышению общей производительности всех систем.

К планировщикам необходимо предъявлять различные требования в зависимости от задач, которые они выполняют. Например, при работе с потоками важно учитывать ограничения необходимые для реализации напрямую взаимодействующих с планировщиком клиентов. Так, для реализации сильных семафоров необходимо накладывать на планировщик дополнительное условие (ограничение) описывающее его работу. В системах реального времени планировщики должны гарантировать выполнение задач в строго определенные временные интервалы, что требует высокой предсказуемости и низкой задержки при переключении контекста. Кроме того, в целом мы стремимся к тому, чтобы планировщики обладали минимальными затратами на выполнение (англ. runtime cost) и высокой эффективностью, что позволяет оптимально использовать доступные ресурсы системы и обеспечивать высокую производительность при минимальных затратах.

Настоящая работа посвящена изучению современных подходов к разработке и оптимизации планировщиков, включая анализ их преимуществ, недостатков и возможных сфер применения. Также внимание уделено вопросам моделирования задач планирования, выбора алгоритмов и оценки их производительности.

1 Планировщики ОС

1.1 Переключение контекста и роль планировщика

Переключение контекста (англ. context switch) — это процесс сохранения состояния текущего выполняющегося процесса и восстановления состояния следующего процесса, который будет запущен на выполнение.

Это фундаментальная операция в многозадачных системах, позволяющая эффективно использовать процессорное время. Однако ключевой вопрос заключается в том, какой процесс выбрать следующим для выполнения. Ответ на этот вопрос дает алгоритм планирования, реализуемый в компоненте ядра операционной системы, известном как планировщик процессов.

Планировщик процессов играет центральную роль в обеспечении стабильной и эффективной работы операционной системы, особенно в условиях множества одновременно выполняющихся задач. Без эффективного планировщика система могла бы столкнуться с ситуациями, когда одни процессы потребляют непропорционально много ресурсов, в то время как другие остаются без необходимой мощности для выполнения своих задач.

В современных операционных системах планировщики постоянно эволюционируют, интегрируя новые технологии и методы для улучшения производительности и надежности. Например, адаптивные алгоритмы планирования могут динамически изменять приоритеты процессов в зависимости от текущих условий и требований, обеспечивая более гибкое и эффективное использование ресурсов.

Таким образом, планировщики являются неотъемлемым компонентом операционных систем, обеспечивая их стабильную работу, высокую производительность и способность адаптироваться к постоянно меняющимся условиям эксплуатации.

2 Алгоритмы планирования

2.1 Алгоритм первый пришел первый вышел (FIFO)

FIFO (First in first out) — классический метод планирования, следует принципу «первым пришел — первым обслужен». Процессы обрабатываются в порядке их поступления. Этот алгоритм отвечает за непосредственное распределение процессорного времени между потоками. Планировщик определяет, какому потоку и на какой период времени будет выделен процессор. Этот уровень также включает координацию выполнения потоков на многоядерных или многопроцессорных системах, обеспечивая эффективное использование аппаратных ресурсов.итм прост в реализации, но редко применяется в чистом виде, поскольку не учитывает особенности задач и может приводить к неэффективному использованию ресурсов. Он лучше всего подходит для длительных, требовательных к вычислительным ресурсам процессов, которые требуют операций ввода-вывода.

2.2 Алгоритм кратчайшей работы следующей (SJN)

Алгоритм кратчайшей работы следующей (англ. shortest job next) выбирает для выполнения тот процесс, который требует наименьшего процессорного времени (англ. burst time). Одной из основных трудностей является оценка времени выполнения процесса, которая не всегда может быть точной. Вытесняющая модификация этого алгоритма сортирует процессы по оставшемуся времени выполнения, позволяя быстро завершать короткие процессы и эффективно снижать общее время оборота.

2.3 Алгоритм очереди с приоритетами

Приоритетное планирование назначает каждому процессу числовой приоритет, определяющий его порядок выполнения. Высокий приоритет (низкое числовое значение) означает более раннее выполнение. Однако этот метод сталкивается с проблемой старвации (голодания), когда процессы с низким приоритетом не получают процессорного времени из-за непрерывного выполнения процессов с высоким приоритетом. Для решения этой проблемы часто используется механизм старения, который постепенно повышает приоритет процессов, долго находящихся в очереди на исполнение.

2.4 Алгоритм Раунд-Робин (RR)

Алгоритм Раунд-Робин (RR) основан на вытесняющей многозадачности и предоставляет каждому процессу фиксированный квант времени (q) для выполнения. После истечения этого времени процесс вытесняется и помещается в конец очереди. Этот метод обеспечивает равномерное распределение процессорного времени и высокую отзывчивость системы.

Однако выбор размера кванта времени критичен: слишком большой квант приводит к поведению, схожему с FIFO, а слишком маленький увеличивает накладные расходы на переключение контекста. Также Раунд-Робин может быть неэффективен для процессов с интенсивным вводом-выводом.

2.5 Алгоритм многоуровневых очередей

Метод многоуровневых очередей классифицирует процессы по различным критериям и распределяет их по нескольким очередям с разными уровнями приоритетов. Внутри каждой очереди может применяться свой алгоритм планирования, например, Раунд-Робин или FIFO. Этот подход обеспечивает гибкость и позволяет оптимизировать время отклика для критически важных процессов, одновременно эффективно используя ресурсы для фоновых задач.

2.6 Модификация алгоритма 2.5

Модификация этого метода с обратной связью позволяет динамически изменять приоритеты процессов на основе их поведения, предотвращая старвацию и улучшая общую производительность системы. Например, процессы, долго ожидающие выполнения, могут получать повышение приоритета, а активно работающие процессы — понижаться.

3 Классификация планировщиков

3.1 Пакетные планировщики

Пакетные планировщики предназначены для выполнения длительных задач, требующих значительных вычислительных ресурсов. Частые прерывания процессов в таких планировщиках не предусмотрены, что позволяет эффективно использовать процессорное время для интенсивных вычислений.

3.2 Интерактивные планировщики

Интерактивные планировщики стремятся минимизировать время отклика, обеспечивая отзывчивость пользовательского интерфейса на действия пользователя, такие как перемещение курсора мыши или нажатие клавиш. Это особенно важно для персональных компьютеров, где скорость реакции системы напрямую влияет на удобство и эффективность работы.

3.3 Планировщики реального времени

Планировщики реального времени разработаны для систем, требующих соблюдения временных рамок выполнения задач (дедлайнов). Они часто используются в критически важных приложениях, где нарушение временных ограничений может привести к серьезным последствиям. В таких системах первостепенным является соблюдение дедлайнов, а другие показатели, такие как время отклика, могут быть менее значимыми.

4 Уровни планирования

4.1 Долгосрочное планирование

На долгосрочном уровне принимается решение о том, какие новые задачи будут добавлены в систему для исполнения. Это включает управление загрузкой задач в систему и контроль за количеством активных процессов.

4.2 Среднесрочное планирование

Среднесрочное планирование занимается управлением памятью, определяя, какие программы временно выгрузить во вторичную память для оптимизации доступного пространства и повышения эффективности работы системы.

4.3 Краткосрочное планирование

Краткосрочное планирование отвечает за непосредственное распределение процессорного времени между потоками. Планировщик определяет, какому потоку и на какой период времени будет выделен процессор. Этот уровень также включает координацию выполнения потоков на многоядерных или многопроцессорных системах, обеспечивая эффективное использование аппаратных ресурсов.

5 Метрики планирования

5.1 Время поступления T_A

Время поступления процесса — момент, когда процесс становится готовым к выполнению.

5.2 Время ожидания T_W

Время ожидания представляет собой период, в течение которого процесс ожидает возможности быть обработанным центральным процессором.

5.3 Время выполнения T_S

Время выполнения — период, в течение которого процесс активно исполняется на процессоре.

5.4 Время оборота T_R

Время оборота включает в себя общее время, затраченное процессом в системе, включая периоды ожидания и активного выполнения.

5.5 Пример оценки метрик

Рассмотрим пример для иллюстрации важности выбора порядка выполнения процессов. Допустим, одновременно поступают три процесса с различным временем выполнения.

$\beth$ есть 3 процесса и их burst (время исполнения одного процесса)

$T_{BURST_1} = 30,\ T_{BURST_2} = 5,\ T_{BURST_3} = 5$

Если мы будем выпонять сначала 1 потом 2 потом 3, то

$T_{W_1} = 0,\ T_{W_2} = 30,\ T_{W_3} = 35 \implies T_{W_{AVG}} = 21\frac{2}{3}$

А выполнять по порядку 3, 2, 1, то

$T_{W_3} = 0,\ T_{W_2} = 5,\ T_{W_1} = 10 \implies T_{W_{AVG}} = 5$

Таким образом видим, что изменение порядка выполнения может значительно снизить среднее время ожидания.

6 Реализации планировщиков потоков в Linux

6.1 Brain Fuck Scheduler (BFS)

BFS был разработан Кон Коливасом в 2009 году как альтернативный планировщик для ядра Linux, его задача предоставить планировщик с более простым алгоритмом, который не требует настройки эвристики или параметров настройки (англ. tuning parameters) для адаптации производительности к определенному типу вычислительной нагрузки. Коливас утверждал, что эти настраиваемые параметры сложны для понимания обычным пользователем, особенно в плане взаимодействия нескольких параметров друг с другом, и заявлял, что использование таких параметров настройки часто может привести к улучшению производительности в конкретном целевом типе вычислений ценой ухудшения производительности в общем случае.

BFS улучшает отзывчивость на настольных компьютерах Linux с менее чем 16 ядрами. В отличие от других планировщиков, BFS использует единую глобальную очередь для всех процессов, что упрощает управление и устраняет необходимость в сложных механизмах балансировки нагрузки между процессорами (ядрами).

Каждому процессу присваивается виртуальный дедлайн, рассчитываемый на основе текущего времени, приоритета процесса и фиксированного интервала времени. Это позволяет планировщику определять порядок выполнения задач, отдавая предпочтение процессам с более ранними дедлайнами.

6.2 Completely Fair Scheduler (CFS)

CFS был разработан венгерским программистом из компании Red Hat Инго Молнаром и был включен в ядро Linux начиная с версии 2.6.23. Этот планировщик ориентирован на задачи класса SHED_NORMAL (т.е. на задачи, не имеющие ограничений на выполнение в реальном времени). CFS старается максимизировать две характеристики: использование процессора (ядер процессора) и интерактивную производительность.

В отличие от предыдущего планировщика O(1), использовавшегося в версиях ядра Linux 2.6, который поддерживал и переключал очереди выполнения активных и истекших задач, реализация планировщика CFS основана на очередях выполнения на каждом процессоре (ядре), узлами которых являются

упорядоченные по времени планируемые объекты, которые хранятся отсортированными в виде красно-черных деревьев. CFS отказывается от старого понятия фиксированных временных срезов с определенными приоритетами и вместо этого стремится предоставить справедливую долю процессорного времени планируемым сущностям.

Планируемая сущность (англ. task) — минимальный объект который может спланировать ядро Linux. Однако ядро также может управлять группами потоков, целыми многопоточными процессами и даже всеми процессами определенного пользователя. Такая конструкция приводит к концепции планируемых объектов, где задачи группируются и управляются планировщиком как единое целое. Для того чтобы эта конструкция работала, каждый дескриптор задачи task_struct содержит поле типа sched_entity, которое представляет группу планирования к которой принадлежит эта задача.

Каждая очередь типа cfs_rq на каждом процессоре (ядре) сортирует структуры sched_entity по времени в красно-черное дерево (или «rbtree» на языке Linux), где самый левый узел занимает группа, получившая наименьший кусок времени выполнения (который сохраняется в поле vruntime сущности). Узлы индексируются по времени выполнения процессором в наносекундах. Для каждого процесса также рассчитывается максимальное время выполнения, представляющее собой время, в течение которого процесс должен был бы выполняться на идеальном процессоре. Это время, в течение которого процесс ожидал запуска, деленное на общее количество процессов.

Когда планировщик вызывается для запуска нового процесса происходит следующая последовательность действий:

1. Выбирается самый левый узел дерева планирования (так как он будет иметь наименьшее время выполнения) и отправляется на выполнение.

2. Если процесс просто завершает выполнение, он удаляется из системы и дерева планирования.

3. Если процесс достигает своего максимального времени выполнения или иным образом останавливается (добровольно или через прерывание), он снова вставляется в дерево планирования на основе его нового затраченного времени выполнения.

4. Из дерева выбирается новый крайний левый узел, и итерация повторяется.

Если процесс проводит большую часть времени в состоянии сна, то значение затраченного времени будет низким, и он автоматически получит повышение приоритета, когда ему это понадобится. Следовательно, такие задачи не получают меньше процессорного времени, чем постоянно выполняющиеся.

Сложность алгоритма, который вставляет узлы в очередь выполнения cfs_rq

планировщика CFS, составляет $O(logN)$, где N – общее количество групп. Выбор следующего объекта для запуска осуществляется за постоянное время, поскольку самый левый узел всегда находится в кэше.

6.3 Earliest eligible virtual deadline first (EEVDF)

Начиная с версии ядра Linux 6.6 (29 октября 2023 года) CFS заменен (в качестве стандартного планировщика) на планировщик EEVDF.

Впервые был описан в статье 1995 года «Earliest Eligible Virtual Deadline First», написанной Ионом Стоикой и Хусейном Абдель-Вахабом. EEVDF использует понятия виртуального времени, подходящего времени, виртуальных запросов и виртуальных сроков для определения приоритета планирования. Также обладает свойством, что когда планируемая сущность продолжает запрашивать обслуживание, объем полученного обслуживания всегда находится в пределах максимального квантового размера, на который сущность имеет право.

Подробно EEVDF в данной работе не разбирается, в целом этот планировщик достаточно похож на BFS.

7 Планирование в пользовательском пространстве

7.1 Введение

Несмотря на все усилия разработчиков ОС иногда бывает полезно построить свой планировщик который будет работать в пользовательском пространстве и использовать собственные модели, например, для более эффективного переключения между потоками в сетевых задачах.

7.2 Планировщик в Go

Горутина (англ. goroutine) — легковесный (по сравнению с планируемой сущностью) поток, управляемый планировщиком рантайма языка Go. Планировщик Go основан на модели M:N (M горутин на N потоках ОС) и использует следующие ключевые компоненты:

1. G (Goroutine): Представляет собой структуру, описывающую горутину, включая её стек, состояние и другие метаданные.

2. M (Machine): Соответствует потоку ОС. M выполняет горутины.

3. P (Processor): Представляет собой контекст выполнения, необходимый для выполнения горутин, включая локальные очереди горутин и управление ресурсами.

При запуске программы Go (рантайм языка) инициализирует пул P контекстов, количество которых определяется переменной GOMAXPROCS. По умолчанию GOMAXPROCS устанавливается равным количеству доступных ядер процессора, но это значение может быть изменено пользователем.

Далее работа планировщика описывается следующей последовательностью:
1. Создание горутин: При создании новой горутины она помещается в очередь готовых задач (англ. run queue).

2. Выбор P: Планировщик выбирает P для выполнения горутины. Обычно используется локальная очередь P для минимизации синхронизации.

3. Назначение M: Каждому P назначается M. Если все M заняты, новые M создаются по мере необходимости до достижения GOMAXPROCS.

4. Выполнение: M выполняет горутину, извлекая её из очереди P. При завершении горутина освобождается, и M может перейти к следующей задаче.

Заключение

В данной работе были рассмотрены основные подходы и алгоритмы планирования в операционных системах и пользовательском пространстве, их реализации в различных контекстах. Были проанализировали ключевые аспекты планировщиков, такие как роль переключения контекста, особенности алгоритмов планирования. Особое внимание было уделено планировщикам, реализованным в ядре Linux.

В контексте развития вычислительных систем и повышения требований к их производительности и отзывчивости, результаты данной работы формируют важный теоретический и прикладной фундамент для дальнейших исследований. Они подчеркивают ключевые аспекты, такие как минимизация задержек, равномерное распределение ресурсов и снижение сложности настройки планировщиков, что является важным для построения высокоэффективных и адаптивных систем.

Таким образом, проведённое исследование не только предоставляет всесторонний анализ существующих решений, но и намечает направления для совершенствования планировщиков. В будущем планируется уделить внимание дальнейшей интеграции планировщиков с адаптивными алгоритмами что обеспечит ещё более высокую производительность и стабильность систем. Это позволит решать актуальные задачи, стоящие перед современными вычислительными средами, и повысить эффективность их работы в различных сферах.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Joseph Y-T. Leung Handbook of Scheduling: Algorithms, Models, and Performance Analysis CRC Press 2004

[2] Andrew S. Tanenbaum, Herbert Bos MODERN OPERATING SYSTEMS Pearson Education 2023

[3] Dror G. Feitelson, Larry Rudolph Job Scheduling Strategies for Parallel Processing Springer 2001

[4] William Stallings Operating Systems: Internals and Design Principles Pearson Education 2020

[5] Michael J. Morrison Resource Management and Scheduling in Multitasking Operating Systems CRC Press 2017

[6] Arden Durell Go Scheduler Internals 2018

[7] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, Henry M. Levy Scheduler activations: effective kernel support for the user-level management of parallelism ACM 1992

[8] Xiaomin Zhu, Yabing Zha, Ling Liu, Peng JiaoGeneral Framework for Task Scheduling and Resource Provisioning in Cloud Computing Systems IEEE 2016