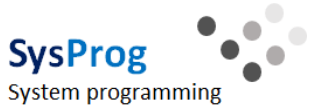
Московский Государственный Технический Университет им. Н.Э. Баумана

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

к.т.н.
Папулин Сергей Юрьевич
papulin_bmstu@mail.ru

Москва, 2021

Планировщик задач в Linux



Основные темы

- Создание процесса и потока
- Планировщик выполнения задач
- Completely Fair Scheduler

Запуск процесса

Создание процесса

- Создание задачи происходит посредством вызова **fork**() и **exec**() (одной из семейства)
- **fork**() клонирует текущую задачу, обновляет PID и некоторые другие значения
- **exec**() загружает исполняемый код в адресное пространство и начинает выполнят его
- **fork**() реализован с использованием системного вызова **clone**(). **clone**() принимает флаги, которые указывают на то, какие ресурсы должны быть общими между родительским и дочерним процессами.
- **clone**() вызывает **do_fork**()
- **do_fork**() вызывает **copy_process**(), который выполняет большинство работы

Создание процесса

copy_process():

1. Вызывает **dup_task_struct**() для создания нового стека, **thread_info** структуры и **task_struct** для нового процесса
2. Гарантирует, что количество дочерних процессов не превышено для текущего пользователя
3. Очищает и сбрасывает поля **task_struct** уникальные для создаваемого процесса (большинство полей остается без изменений)
4. Устанавливает поле **state** в **task_struct** в **TASK_UNINTERRUPTABLE**
5. Вызывает **copy_flags**() для обновления **flags** в **task_struct**
6. Назначает новый **PID** посредством **alloc_pid**()
7. В зависимости от установленных флагов в **clone**(), **copy_process**() дублирует или делает общими ресурсы, такие как дескрипторы файлов
8. Возвращает указатель на новую **task_struct**

Создание потока

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

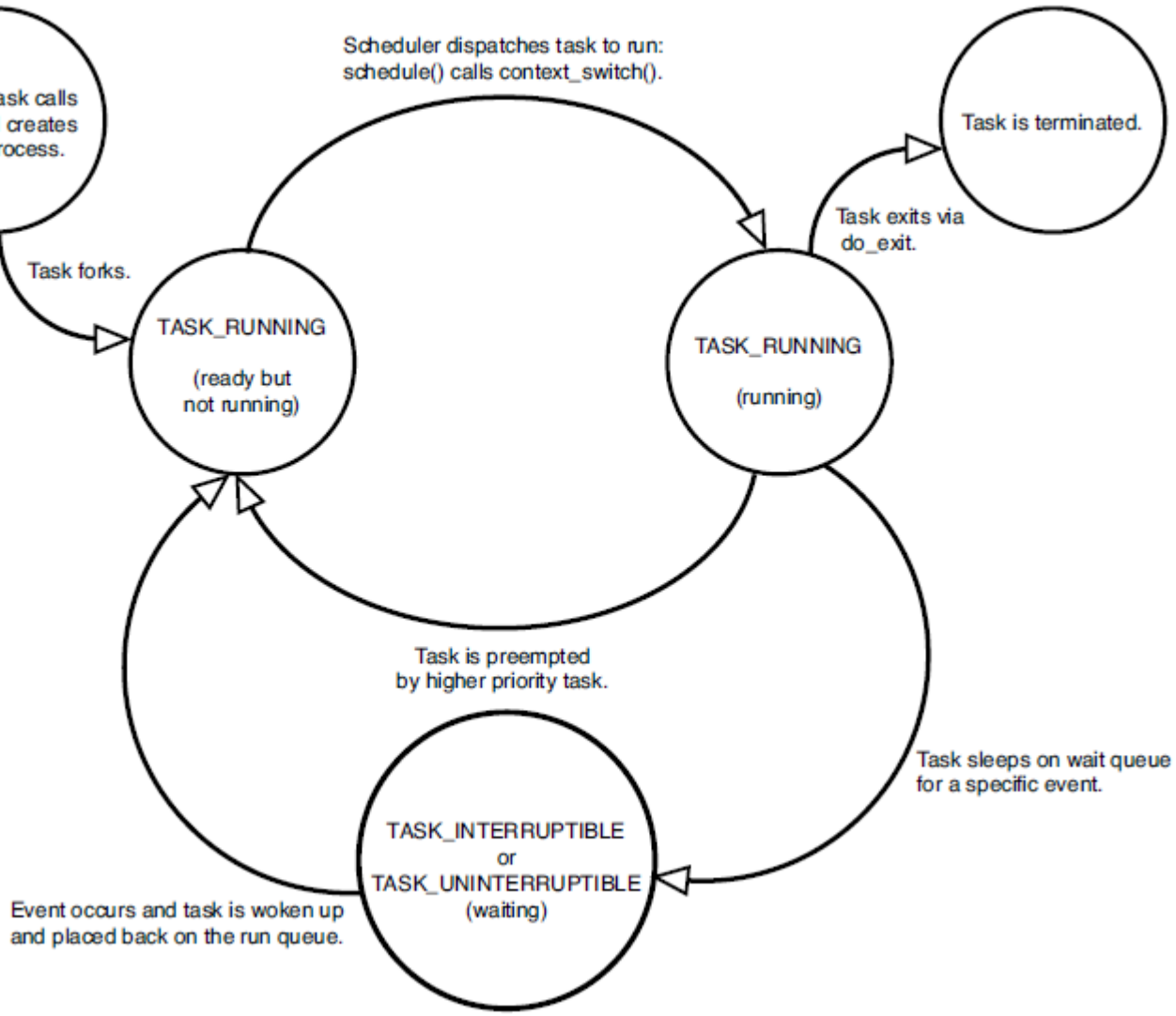
`CLONE_VM` – родительский и дочерний процессы имеют общее адресное пространство

`CLONE_FS` – родительский и дочерний процессы имеют общие данные по файловой системе (корневой каталог, рабочую директорию, права доступа (**unmask**))

`CLONE_FILES` – родительский и дочерний процессы имеют общие открытые файлы

`CLONE_SIGHAND` – родительский и дочерний процессы имеют общую таблицы обработки сигналов

Состояния процессов в Linux



Состояния процесса:

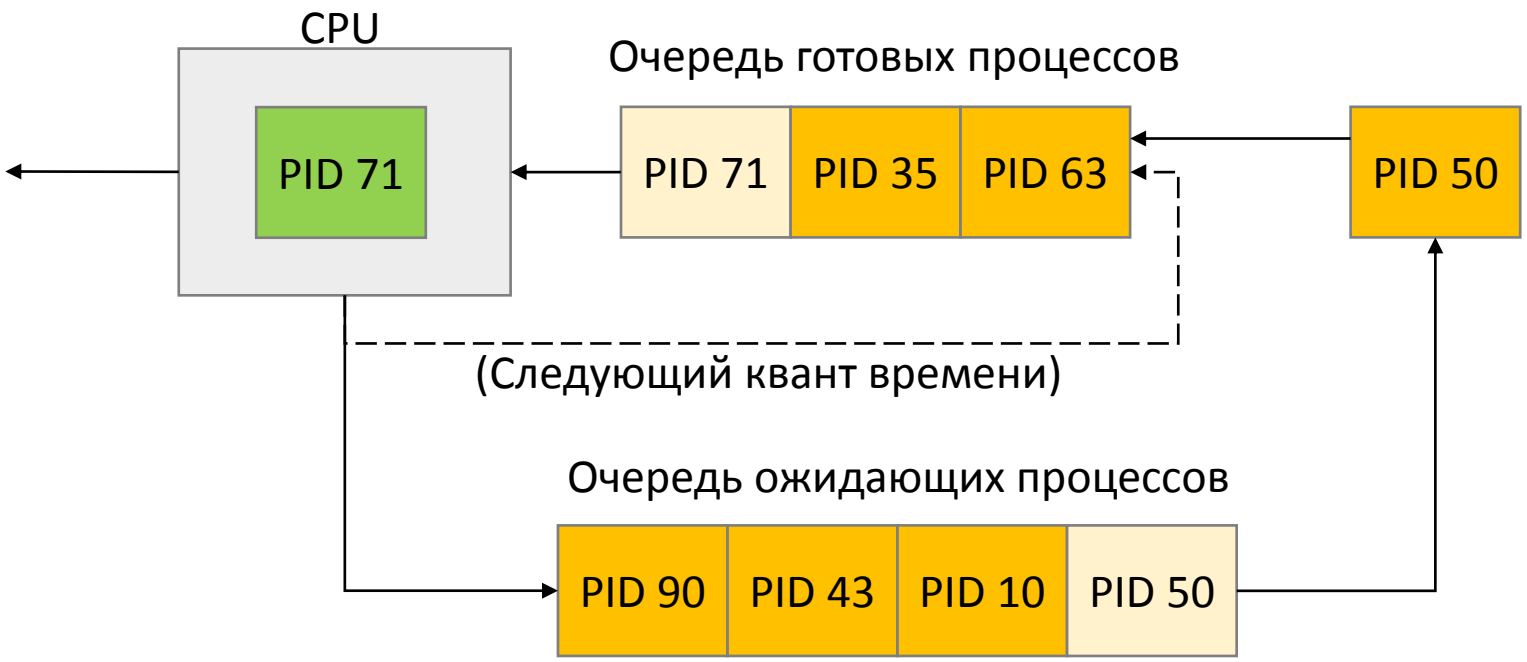
- `TASK_RUNNING`
- `TASK_INTERRUPTIBLE`
- `TASK_UNINTERRUPTIBLE`
- `__TASK_TRACED`
- `__TASK_STOPPED`

Планировщик выполнения задач

Планировщики выполнения

- FIFO
- Round Robin
- O(n)
- O(1)
- Completely Fair Scheduler

RR планирование выполнения



Назначение планировщика

- Планировщик задач – важная часть ОС и определяет, какую задачу выполнять и как долго
- Позволяет выполнять попеременно несколько процессов на одном CPU, в результате чего складывается впечатление, что они работают одновременно. В реальности количество одновременно выполняемых процессов ограничено количеством CPU
- Является ключевым компонентом многозадачных систем
- ОС с вытеснением определяют, когда завершить выполнение задачи и какая задача далее должна быть назначена CPU
- Промежуток времени в течение которого должна выполняться задача называется timeslice
- Процесс замены задачи новой называется переключение контекста (context switch)

Назначение планировщика

- Значение timeslice определяет как долго процесс может выполняться до того, как будет вытеснен.
- Политика планировщика должна определять, какое будет значение timeslice по умолчанию
  - Если слишком большое значение, то система будет плохо реагировать на внешние команды
  - Если очень маленькое, система будет неэффективной, так как процессор будет тратить больше времени на переключение между задачами

Типы задач

Различные задачи могут иметь разные требования

- **Задачи, ориентированные на ввод-вывод (I/O-bound)**

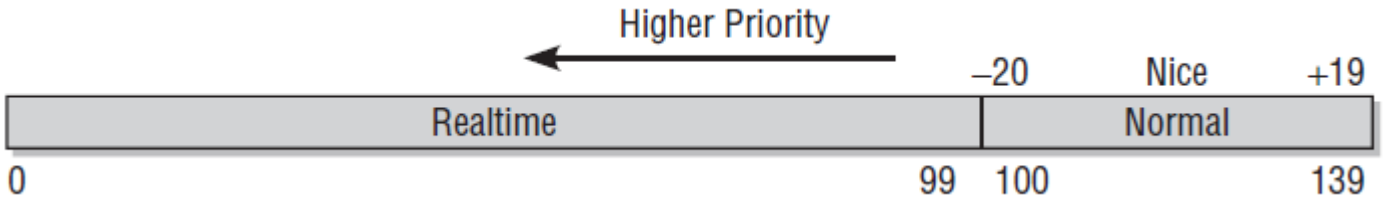
  Большую часть времени тратят на ожидание операций ввода-вывода (доступ к устройствам хранения, сетевым устройства, клавиатуре и пр.)

- **Задачи, ориентированные на использование CPU**

  Большую часть времени тратят на выполнения кода (аналитические пакеты, обработка видео)

Планировщик должен учитывать различия, чтобы обеспечить своевременный отклик системы на команды пользователя и устройства ввода-вывода и при этом выполнять долгие задачи, требующие больших вычислительных ресурсов.

Приоритет задач



Задачи реального времени имеют статический приоритет. Это гарантирует, что процесс с некоторым приоритетом всегда может быть вытеснен процессом с более низким приоритетом

Приоритет реального времени от 0 до MAX_RT_PRIO (100) - 1

Архитектура планировщика

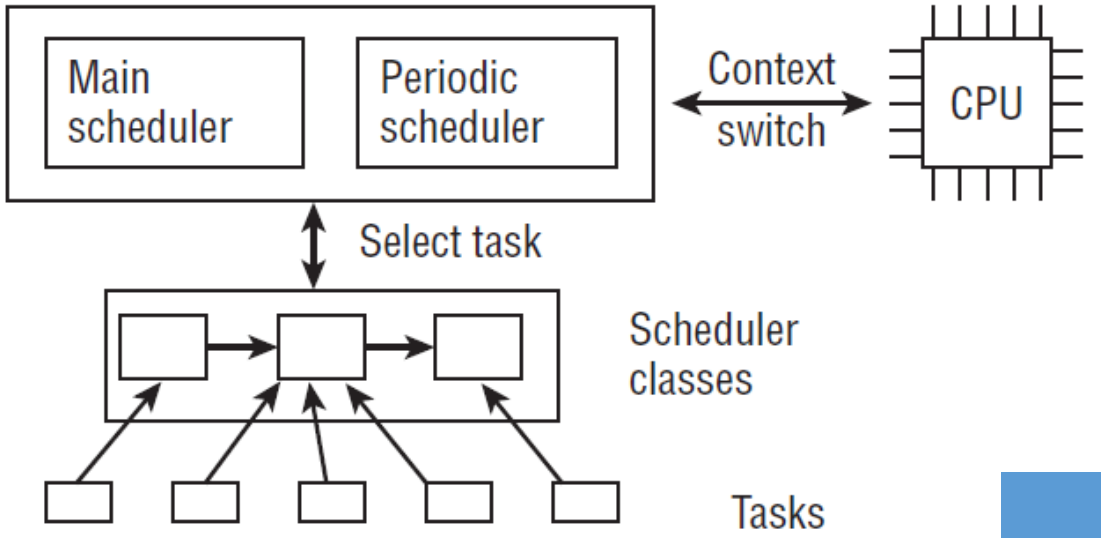
- Задачи готовые к выполнению находятся в очереди runqueue

- Классы планирования используются для принятия решения о том, какую далее запускать задачу

Ядро поддерживает несколько видов планирования (policies):

- Абсолютно честное планирование (Completely Fair Scheduling – CFS)
- Планирование задач реального времени (Real-Time Scheduling)
- Планирование холостой задачи (idle task)

- После того как выбрана задача, выполняется переключение контекста



Архитектура планировщика

- Каждый класс планирования имеет разный приоритет
- Планировщик перебирает классы в порядке приоритета
- Класс с наивысшим приоритетом имеющий готовый к выполнению процесс побеждает и определяет, какой будет запущен следующий процесс
- Каждая задача принадлежит только одному из классов
- Каждый класс отвечает за управление его задачами

CFS – класс для нормальных процессов (SCHED_NORMAL)
RTS – класс для процессов реального времени (SCHED_RR, SCHED_FIFO)

Вызов планировщика

- Планировщик вызывается, когда
  - срабатывает таймер с определенным периодом (periodic scheduling)
  - задача переходит в спящий режим или покидает CPU по другой причине (main scheduling)
- При вызове планировщика проверяется необходимость переключения задачи
- Если необходимо, то устанавливается специальный флаг (need_resched)

Период выполнения задачи

- CFS вместо фиксированного времени определяет пропорцию процессорного времени так, чтобы оно зависело от текущей нагрузки. Вычисляемая пропорция зависит от nice-значения и играет роль веса. Процесс с низким nice-значением получит больший вес, с высоким – низкий вес.
- Когда процесс готов к выполнению, на основе того, как много он уже потребил процессорного времен, принимается решение запускать его или нет.
- Если он отработал меньшую порцию, чем текущий выполняемый процесс, то новый процесс будет запущен, если нет, то он будет запланирован на запуск позднее.

Пример

- Представим, что запущены два процесса: видеокодек (CPU) и текстовый редактор (I/O)
- Если оба процесса имеют одинаковое nice-значение, то оба будет назначены процессору по 50% мощности.
- Текстовый редактор не будет использовать большую часть выделенного процессорного времени, так как он в основном будет в заблокированном состоянии, ожидая ввода пользователя
- Видеокодек наоборот будет использовать больше чем 50% процессорного времени
- Однако когда текстовый редактор проснется в ответ на ввод пользователя, планировщик CFS определит, что он использовал меньше, чем назначенные ему 50% и поэтому отработал меньше чем видеокодек.
- Планировщик вытеснит видеокодек и запустит текстовый редактор

Fair Scheduling

- CFS основан на простой идее: моделирование планирования задач таким образом, как если бы система имела идеальный, совершено многозадачный процессор. В такой системе каждая задача получала бы 1/n мощности процессора, где n – количество готовых к выполнению процессов
- Или это можно представить, как назначение каждой задаче бесконечно малый период выполнения. А за некоторый измеряемый период складывалось бы впечатление, что все процессы выполняются одновременно.
- CFS запускает каждую задачу на некоторое время, затем следующую задачу, которая выполнялась меньше всего. Планировщик вычисляет как долго задача должна выполнятся как функция от общего количества задача готовых к выполнению
- Nice-значение используется для определения доли процессорного времени, которое получит задача
- Каждая задача запускается на период времени (timeslice) пропорциональный её весу деленному на общий вес всех задач

Fair Scheduling

- CFS устанавливает целевую задержку (latency), которая определяется как суммарное время, в течение которого все процессы должны быть запущены
- Данный период контролируется параметром **sysctl_sched_latency**
- Например, если период задержки 10ms, то 2 одинаково взвешенных процесса будут работать в течение 5ms. Если 5 задача, то по 2ms каждая
- Существует минимальный период выполнения задачи, который позволяет избежать затрат на переключение контекста
- Контролируется параметром **sysctl_sched_min_granularity**

CFS. Время выполнения задачи

- Реальное время выполнения

$$curr\_time = now$$

$$curr \rightarrow sum\_exec\_runtime \ += curr\_time - exec\_start$$

$$exec\_start = curr\_time$$

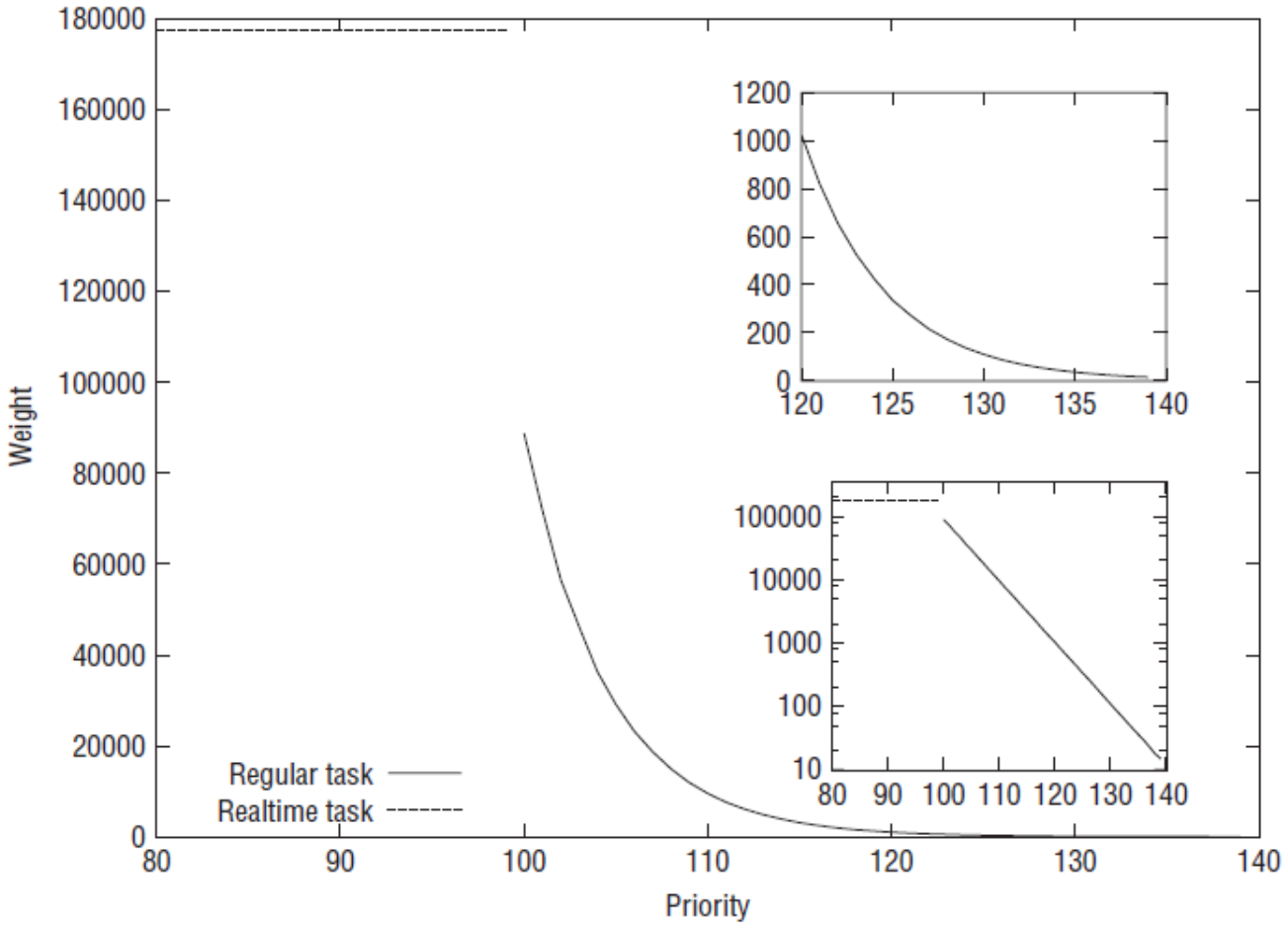
- Виртуальное время выполнения

$$delta\_exec\_weighted \ = \ delta\_exec \times \frac{NICE\_0\_LOAD}{curr \rightarrow load.weight}$$

$$curr \rightarrow vruntime \ += delta\_exec\_weighted$$

CFS. Приоритет

```
static const int prio_to_weight[40] = {
 /* -20 */     88761, 71755, 56483, 46273, 36291,
 /* -15 */     29154, 23254, 18705, 14949, 11916,
 /* -10 */     9548, 7620, 6100, 4904, 3906,
 /* -5 */      3121, 2501, 1991, 1586, 1277,
 /* 0 */       1024, 820, 655, 526, 423,
 /* 5 */       335, 272, 215, 172, 137,
 /* 10 */      110, 87, 70, 56, 45,
 /* 15 */      36, 29, 23, 18, 15, };
```



CFS. Назначаемый период выполнения задачи

- Реальный timeslice

$$\text{sysctl\_sched\_latency} = \text{sysctl\_sched\_latency} \times \frac{\text{nr\_running}}{\text{sched\_nr\_latency}}$$

$$\text{timeslice} = \frac{\text{sysctl\_sched\_latency} \times \text{se} \rightarrow \text{load.weight}}{\text{cfs\_rq} \rightarrow \text{load.weight}}$$

- Виртуальный timeslice

$$\text{vtimeslice} = \text{timeslice} \times \frac{\text{NICE\_0\_LOAD}}{\text{se} \rightarrow \text{load.weight}}$$

CFS. Единица планирования

Единица планирования

```
struct sched_entity {
        struct load_weight      load;
        struct rb_node          run_node;
        struct list_head        group_node;
        unsigned int            on_rq;
        u64                     exec_start;
        u64                     sum_exec_runtime;
        u64                     vruntime;
        u64                     prev_sum_exec_runtime;
        u64                     last_wakeup;
        u64                     avg_overlap;
        u64                     nr_migrations;
        u64                     start_runtime;
        u64                     avg_wakeup;
```

vruntime – виртуальное время выполнения процесса, которое есть реальное время взвешенное на количество готовых к выполнению процессов. Измеряется в наносекундах

CFS. Обновление времени выполнения

- **update_curr**() обеспечивает обновление **vruntime** .
- **update_curr**() запускается системным таймеров или когда процесс запускается или блокируется
- После вычисления времени выполнения текущего процесса, он передает значение **__update_curr**()
- **__update_curr**() взвешивает время с учетом количества готовых к выполнению процессов и обновляет **vruntime** текущего процесса

CFS. Выбор задачи

- CFS решает какую задачу запустить следующей по наименьшему значению **vruntime**
- CFS использует красно-черное дерево для управления списком готовых к выполнению задач. Значения узлов дерева соответствует значению **vruntime**
- Красно-черное дерево сбалансированно и узел с наименьшим значением всегда крайний левый
- Если нет готовых к выполнению процессов, то назначается холостая задача (idle task)
- Задача добавляет в дерево посредством **enqueue_entity**()
- Задача удаляется из дерева, когда задача заблокирована, остановлена или выполняется на CPU (?). Используется функция **dequeue_entity**()

CFS. Выбор задачи

- Основная функция планирования – **schedule**()
- Используется ядром для вызова планировщика задач
- **schedule**() вызывает функцию **pick_next_task**() для выбора следующей задачи

CFS. Засыпание и пробуждение

- Спящие задачи не могут быть назначены на выполнение. Задачи переходят в спящий режим, например, в при ожидании завершения I/O операции. Спящая задача всегда ожидает события, которые пробудит её и переведет в состояние готовности к выполнению (runnable)

- Когда задача переходит в спящий режим, она помечается как спящая, помещается в очередь ожидания, удаляется из дерева процессов на запуск и вызывается функция schedule() для выбора новой задачи на выполнение

- Существует два состояния спящей задачи:

  - TASK_INTERRUPTIBLE
  - TASK_UNINTERRUPTABLE (игнорирует сигналы)

CFS. Очередь ожидания

- Очередь ожидания – связанный список задач, ожидающий некоторого события. Когда событие происходит, задачи в очереди пробуждаются
- Пробуждение выполняется функцией **wake_up**()
- Функция вызывает **try_to_wake_up**() для каждой задачи, которая
  - устанавливает задачи в состояние TASK_RUNNING,
  - добавляет задачу в красно-черное дерево посредством **enqueue_task**() и
  - устанавливает флаг **need_resched**, если приоритет пробудившейся задачи выше чем текущей

CFS. Смена контекста

- Переключение контекста – замена одной выполняемой задачи другой
- Выполняется посредством функции **context_switch**(), которая вызывается функцией **schedule**(), когда задача выбрана на выполнение
- Переключение контекста включает:
  - Замену виртуальной памяти задачи посредством **switch_mm**()
  - Смену состояние процессора (сохранение и восстановление стека и значений регистров) посредством **switch_to**()

CSF. Вытеснение

- Ядро использует флаг **need_resched** для обозначения о необходимости перепланирования
- Флаг устанавливается:
  - функцией **scheduler_tick**(), когда процесс должен быть вытеснен
  - функцией **try_to_wake_up**(), когда пробуждается процесс с более высоким приоритетом, чем выполняемый
- Каждый раз, когда ядро возвращается в пространство пользователя из системного вызова или из обработка прерываний, проверяется флаг **need_resched**
- Если флаг установлен, то ядро запускает **schedule**(), чтобы запустить планировщик
- С Linux 2.6 код ядра может быть вытеснен. Любая задача может быть перепланирована, если ядро находится в безопасном для перепланирования состоянии

Источники

Linux kernel development / Robert Love. — 3rd ed.

Professional Linux kernel architecture / Wolfgang Mauerer.

Understanding the Linux / Daniel P. Bovet and Marco Cesati. — 3rd ed.