# **Задание 1**

Многозадачность в современных системах реализована через переключение между процессами с помощью планировщика. Какие операции при этом замедляют работу компьютера?

*Приведите ответ в свободной форме со своим комментарием.*

# **Решение 1**

Операции сохранения и восстановления (переключения) контекста процессов из оперативной памяти (или свопа, в худшем случае).

При использовании планировщика CFS процессам выделяется равное количество времени, например на 10 процессов 50 мс, то есть процессу отводится 5мс ресурса процессора. Это, конечно, сферический конь в вакууме, поскольку процессору перед началом обработки процесса необходимо получить информацию о том с чем он, собственно, будет работать. Предположим что это 0.5мс. Следовательно полезных действий процессор будет выполнять 4.5мс, остальное время (с точки зрения пользователя) тратится бесполезно.

# **Задание 2**

В каких случаях используется планировщик SCHED\_DEADLINE

*Приведите ответ в свободной форме*

# **Решение 2**

Планировщик deadline лучше всего подходит для систем реального времени, в которых более важны операции чтения, чем записи (например web-сервер, база данных). Максимальное время выполнения операции чтения в таком планировщике меньше, чем максимальное время выполнения операции записи. Основная его обязанность - выполнить задачу за фиксированное реальное время.

# **Задание 3**

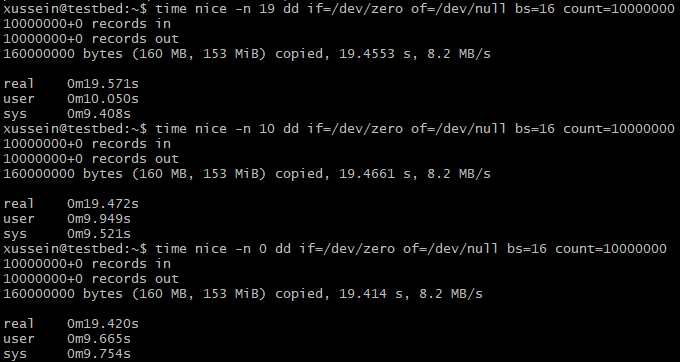
Запустите следующий код, имитирующий нагрузку типа ввод / вывод, с значениями nice 19, 10, 0 и измерьте время исполнения с помощью утилиты time. Объясните получившееся различие во времени исполнения для разных запусков.

dd if=/dev/zero of=/dev/null bs=16 count=10000000

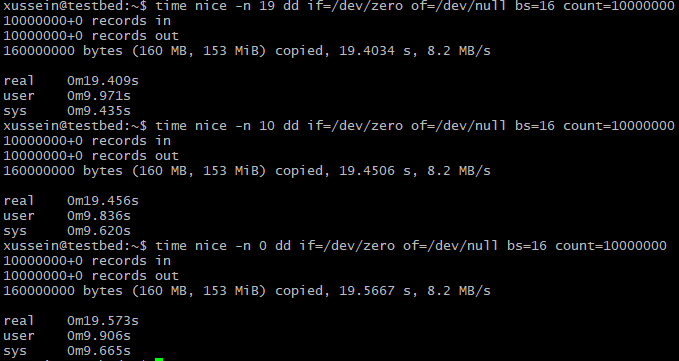
*Ответ приведите в виде снимка экрана с комментариями в свободной форме*

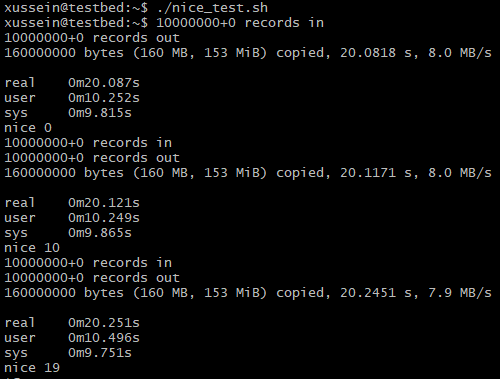
# **Решение 3**

Параметр nice определяет уступчивость процесса, то есть насколько процесс позволяет другим процессам идти впереди себя. Чем больше значение (диапазон от -20 до 19), тем более процесс уступчивый и тем медленнее выполняется.

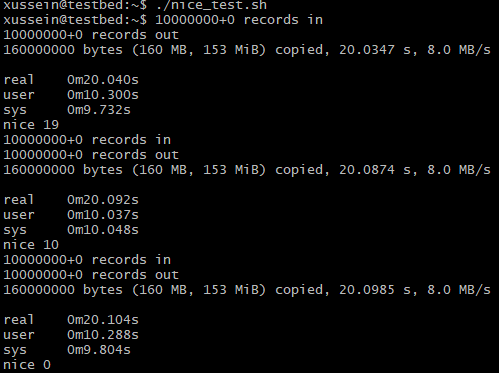


Наличие более низкого nice не является гарантией более быстрого выполнения процесса, как видно на скрине ниже. Почему? Видимо погода на Марсе неподходящая.



Также, если запустить три процесса одновременно:

Видно что процесс с nice=19 выполнялся больше процессорного времени. Хотя на системные вызовы было затрачено меньше времени. При следующем запуске:



nice снова никаких гарантий не дает. Хотя в моем конкретном случае такое поведение nice связано с малой нагрузкой системы. **В целом, более низкий nice может только пообещать, что процесс, скорее всего, будет выполнен быстрее, чем процесс с высоким nice.**

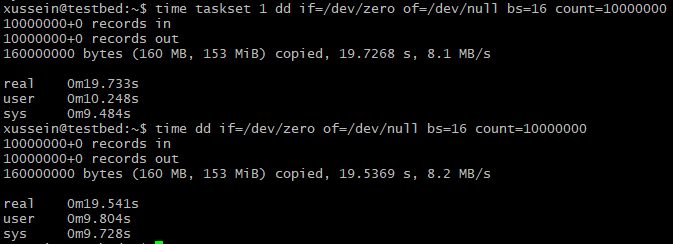
# **Задание 4**

Повлияет ли на реальное время исполнения запуск кода из Задания 3 на одном ядре вместо нескольких? Напишите почему да или почему нет. Проверьте своё предположение с помощью утилиты taskset

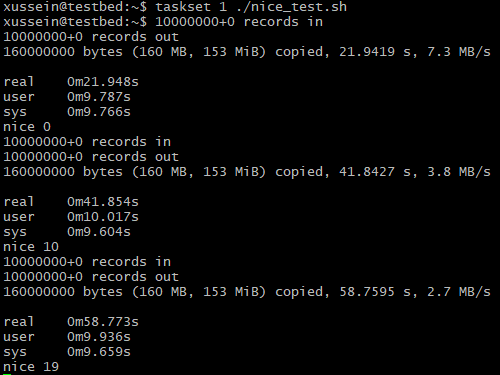
*Ответ приведите в виде снимка экрана с комментариями в свободной форме*

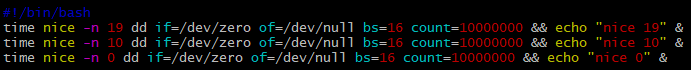
# **Решение 4**

В таком формате разницы никакой нет, dd и так работает только с одним процессором:

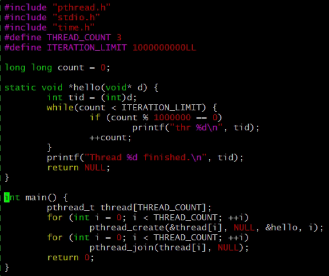


Если запускать три процесса одновременно:

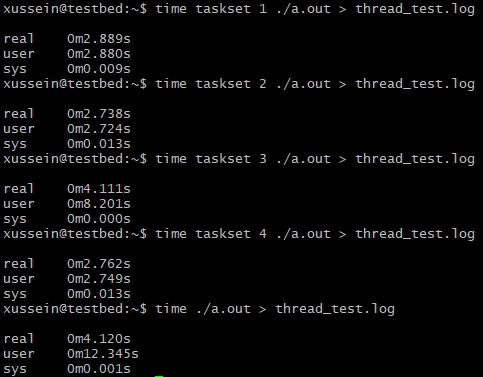




то, конечно, и время выполнения будет больше - выполняются три одинаковые задачи на одном процессоре. Ну хоть nice работает как положено. Для максимальной наглядности возьмем следующий код (лектор приводил в пример):



Здесь мы запускаем 3 треда и перебрасываем между ними переменную count, они ее доводят с 0 до 1 миллиарда добавлением единицы. В итоге получаем следующую картину:



В системе 4 ядра. Ситуация с одним и двумя ядрами происходит из-за того, что у ядер (которые сообщаются попарно, как правило) есть свой (малый) кэш, в котором хранится “жонглируемая” переменная во время смены контекста. Если выполнять программу на 3х (через taskset) и просто выполнить программу получается что user time меньше… Рискну предположить, что это связано с явным указанием 3 процессоров на выполнение для 4х процессорной системы. Мол так программа не лезет с вопросами к 4му ядру… На 4х ядрах скорость выполнения +- равна скороста на 2х ядрах. Ну и больше всего времени уходит на выполнение задачи без taskset. В этом случае переменная передается из одной пары ядер в память, потом в третий процессор через RAM. Скорость выполнения падает из-за обращения к RAM.

# **Задание 5**

При каких условиях лучше увеличить time slice планирования, а когда лучше уменьшить?

*Приведите ответ в свободной форме со своим комментарием.*

# **Решение 5**

Если отзывчивость системы не особо нужна можно увеличить размер гранулы. Например, если речь идет о каких-нибудь математических вычислениях, В таком случае смен контекста станет меньше, длительность выполнения процесса увеличивается. Процессор меньше занимается утилитарной деятельностью, больше пользовательской. Если нужна большая отзывчивость, для геймеров например, то гранулу можно уменьшить. Прерываний будет больше, но задачи переключаются быстрее.

# **Задание 6**

Запустите код из Задания 3 с приоритетом по умолчанию. Одновременно запустите веб-браузер с разными приоритетами. Создайте условия, чтобы получить различную отзывчивость браузера при использовании разных приоритетов. Запишите условия и опишите результаты.

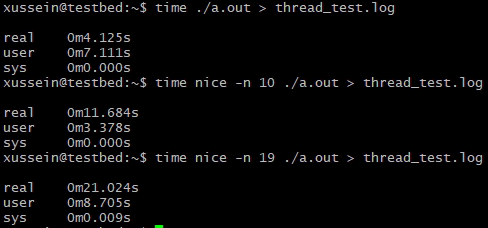
*Приведите ответ в свободной форме со своим комментарием.*

# **Решение 6**

# Запустим на фоне следующий скрипт:

# 

Во время выполнения скрипта запускаем тестовую программку из предыдущего примера:



При прочих равных, ниже приоритет - больше времени выполнения.

# **Задание 7**

В каких ситуациях планировщик ввода / вывода noop может быть производительней cfq?

*Приведите ответ в свободной форме со своим комментарием.*

# **Решение 7**

NOOP - самый примитивный алгоритм и минимально взаимодействует с процессором, не расходуя его ресурс. В случае если NOOP взаимодействует с устройствами, у которых seek time минимален (то есть Flash, SSD etc) и эти устройства сами знают как оптимизировать IO (RAID, SAN) то он оказывается потенциально более эффективен чем CFQ.