Operating Systems

Threads, Processes and Scheduling

Me

October 6, 2016

Поток исполнения

- Поток исполнения (aka поток) - память содержащая команды и некоторый контекст, определяющий состояние потока исполнения
  - набор команд и то что нужно для их исполнения.
- Контекст потока - окружение в котором исполняются команды и состояние CPU:
  - доступная память:
    - таблица страниц определяет всю доступную память (если вы ее используете);
    - память с командами и данными;
    - стек;
  - регистры процессора.

Память потока

- Различные потоки исполнения *могут* "разделять" общую память:
  - использовать один и тот же набор команд;
  - работать с одними и теми же данными;
  - однако потоки могут быть и изолированы друг от друга, например, иметь свою таблицу страниц.
- Стек у каждого потока свой:
  - стек важная область памяти для исполнения команд потока;
  - стек хранит локальные переменные функций;
  - на стек, *обычно*, сохраняются адреса возврата при вызове функций.

Состояние CPU

- Состояние CPU включает:
  - регистры общего назначения:
    - например, для x86-64 это регистры RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R9-R15;
  - флаговый регистр (или регистры):
    - регистры флагов используются для организации условных переходов или условного исполнения;
    - например, для x86-64 это регистр RFLAGS;
  - прочие регистры, участвующие в вычислениях:
    - например, xmm/avx/прочие регистры x86 используемые для floating-point арифметики и SIMD инструкций, которые не будут интересовать нас в домашних заданиях.

Многопоточность

- Одновременно могут исполняться сразу несколько потоков:
  - если на одном кристалле находится сразу несколько полноценных или не очень (aka Hyper Threading) вычислительных ядер;
  - если у вас просто несколько CPU в системе.
- Потоки могут исполняться *почти* одновременно:
  - одновременность в контексте многопточности - плохое слово (вообще понятие времени очень странное в данном контексте);
  - в каждый момент времени на конкретном CPU исполняется только один поток, но между потоками можно быстро переключаться создавая иллюзию одновременной работы.

Кооперативная и вытесняющая многопточность

- Кооперативная многопоточность
  - поток работает на CPU до тех пор, пока он сам не решить "отдать" CPU кому-нибудь другому;
  - часто в некоторых контекстах такие потоки называют корутинами;
  - считается, что программировать используя кооперативную многопточность проще (впрочем, это очень-очень-очень спорное утверждение).
- Вытесняющая многопоточность
  - поток может быть смещен (вытеснен) с CPU в любой момент без предупреждения;
  - например, ОС может смещать поток с CPU, если он отработал достаточно долго, ну или если он сам решил "отдать" CPU.

Переключенние потоков

- Для переключения потоков, очевидно, нужно подменить контекст одного потока, контекстом другого потока и передать управление коду другого потока
  - чтобы уметь переключиться назад на старый поток необходимо сохранить его контекст, чтобы его можно было восстановить;
  - переключением с одного потока на другой поток занимается какой-то код, этот код тоже использует регистры и какую-то память;
  - соответственно, это код не должен в процессе сохранения состояния испортить само состояние;
  - соответственно, код переключающий потоки должен быть доступен в каждом потоке - все потоки должны иметь какую-то общую память.

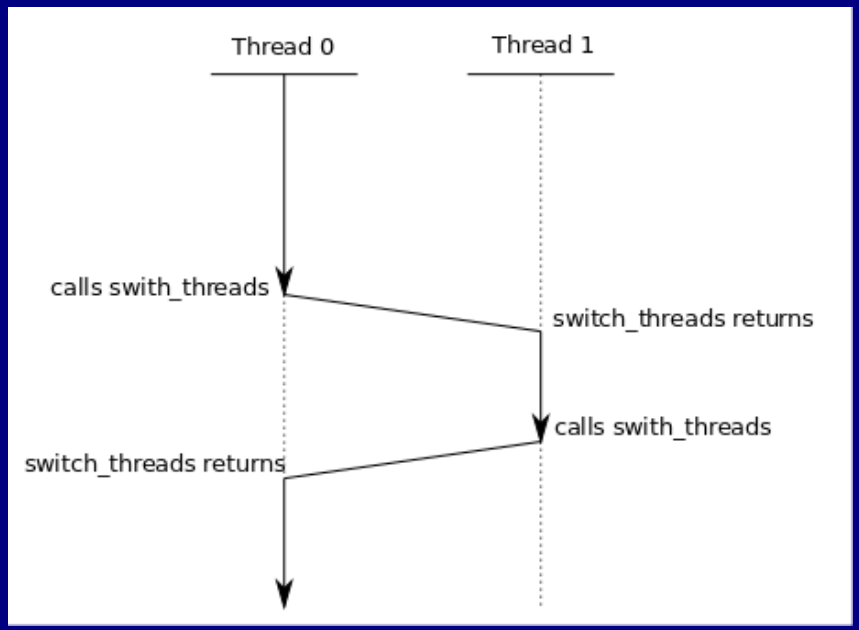
Пример переключения для x86

```
1   .text
2 switch_threads:
3   pushq %rbx
4   pushq %rbp
5   pushq %r12
6   pushq %r13
7   pushq %r14
8   pushq %r15
9   pushfq
10
11  movq %rsp, (%rdi)
12  movq %rsi, %rsp
13
14  popfq
15  popq %r15
16  popq %r14
17  popq %r13
18  popq %r12
19  popq %rbp
20  popq %rbx
21
22  ret
```

```
1 void switch_thread(void **prev, void *next);
```

- сохраняет состояние старого потока на стек:
  - через *prev* (регистр rdi) возвращается указатель на сохраненное состояние;
  - *next* (регистр rsi) - указатель на сохраненное состояние нового потока;

Пример переключения для x86



- Функция *switch_threads* изменяет внутри себя регистр *rsp*
  - т. е. входим мы в функию пользуясь стеком *Thread 0*, а выходим уже используя стек *Thread 1*;
  - следовательно *ret* берет адрес возврата со стека *Thread 1*, т. е. *ret* передает управление коду *Thread 1*.

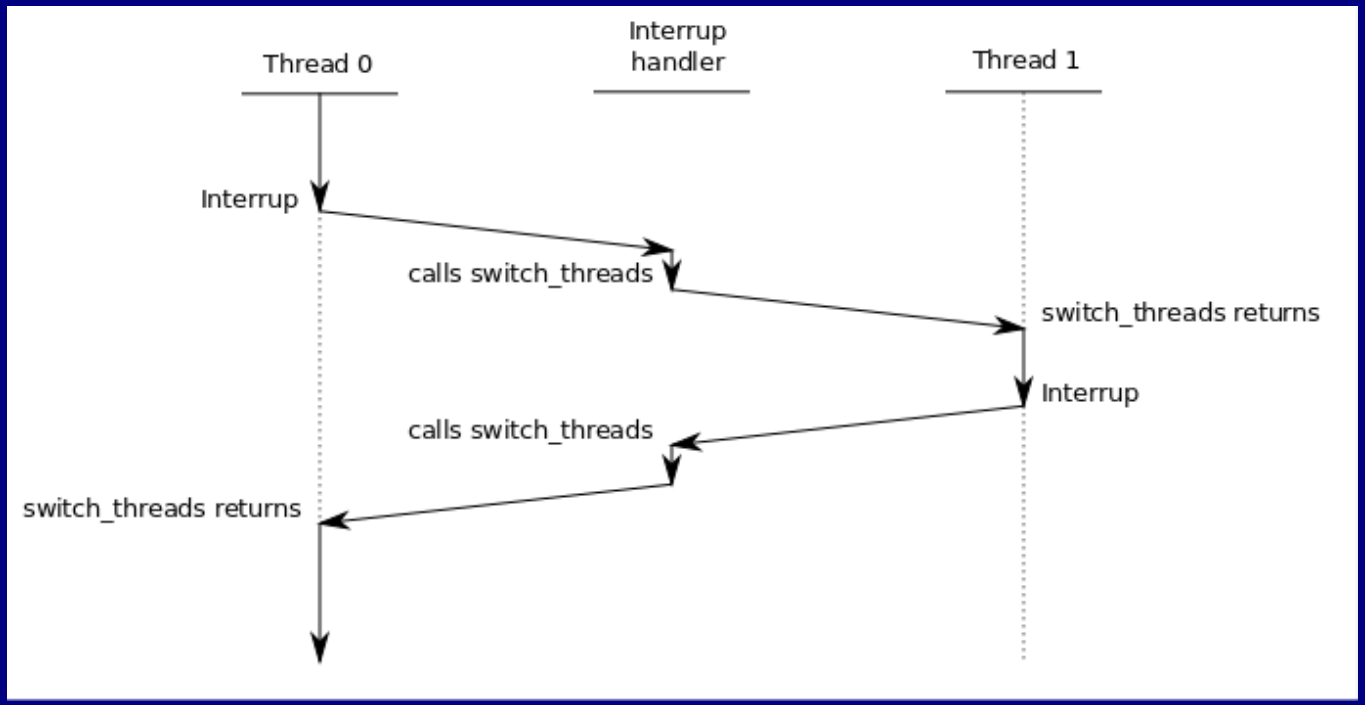
Пример переключения для x86

- *switch_threads* восстанавливает состояние сохраненное другим вызовом *switch_threads*
  - как переключиться на поток в первый раз?
  - при создании нового потока необходимо аллоцировать память под стек;
  - мы можем искуственно "положить" на стек потока нужное "состояние";
- под состоянием понимаются не только регистры сохраняемые командами *pushq* и *pushfq*, но и адрес возврата используемый инструкцией *ret*
  - по завершению *switch_threads* управление передается инструкцией *ret* по адресу возврата сохраненному на стеке;
  - в качестве адреса разумно использовать адрес основной функции потока (main для потока).

Организация вытесняющей многопоточности

- Имея *switch_threads* легко организовать кооперативную многопоточность
  - достаточно узнать, где хранится состояние следующего потока.
- Как организовать "насильное" вытеснение потока?
  - необходимо "прерывать" исполняющийся на CPU код, и вызвать код, который выполнит переключение;
  - для этого можно использовать прерывания (например, прерывание от таймера):
    - обработчик прерывания прерывает исполняемый код, но в остальном выполняется в контексте прерваного кода;
    - мы можем просто вызвать *switch_threads* из обработчика прерывания;
    - не забудьте про EOI перед переключением.

Организация вытесняющей многопоточности

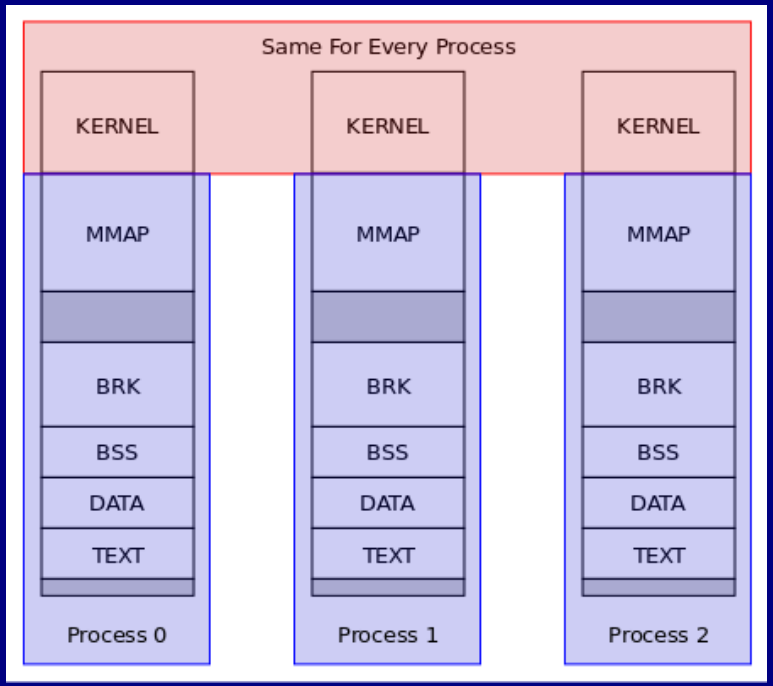


- CPU при вызове обработчика прерываний вытесняет исполняемый код с CPU;
- обработчик прерывания вызывает *switch_threads*.

Изоляция и группировка ресурсов

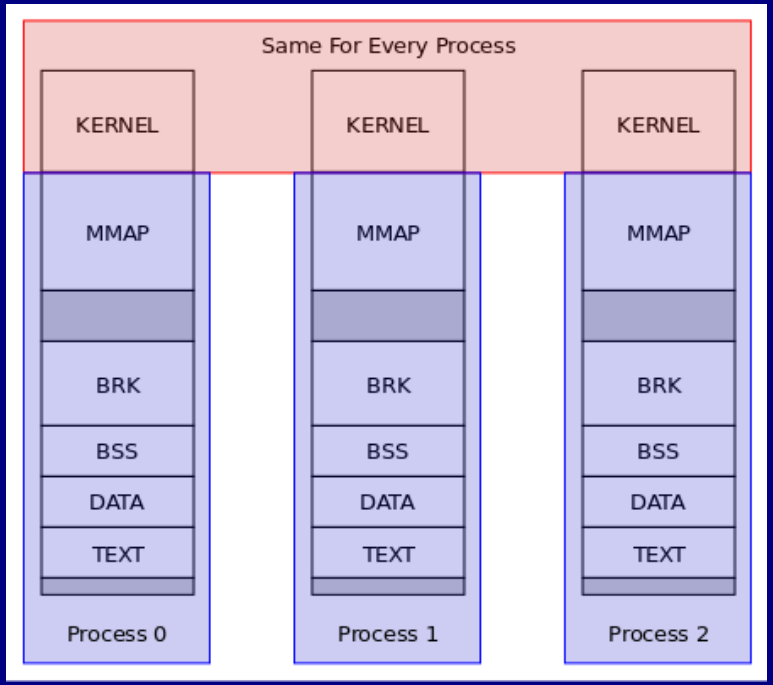
- Иногда хочется изолировать одни потоки от других:
  - чтобы ошибки в одном потоке не могли повлиять на другие потоки;
  - наиболее распространненый случай - раздельная память для потоков.
- Процесс - контейнер ресурсов ОС:
  - процесс группирует вместе ресурсы и потоки, которые работают с этими ресурсами (потоков может быть много, но не меньше одного);
  - потоки внутри одного процесса используют одну и ту же память (таблицу страниц);
  - в идеале процессы независимы друг от друга и не знают друг о друге, но по желанию они могут использовать общие участки памяти;
  - в зрелых ОС существуют другие ресурсы кроме памяти (например, файловые дескрипторы).

Организация памяти процесса



- Ядро ОС отображается в адресное пространство каждого процесса:
  - в таблице страниц каждого процесса есть записи для памяти ядра;
  - эти записи должны запрещать непривилигерованный доступ к этой памяти.

Организация памяти процесса



- Остальная память у каждого процесса своя, но есть но:
  - по обоюдному согласию процессы могут иметь общие участки памяти;
  - память не обязательно физически разделена (хей-хей, page fault).

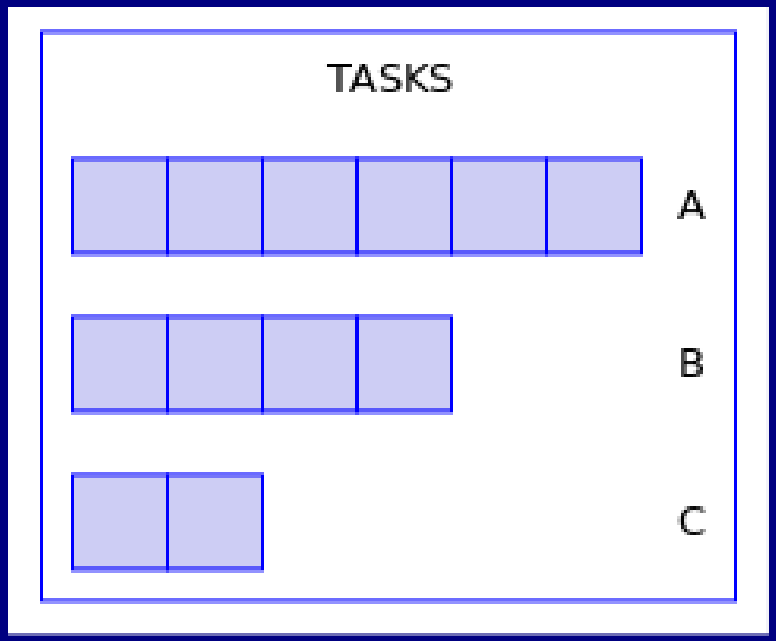
Финальные замечания про процессы

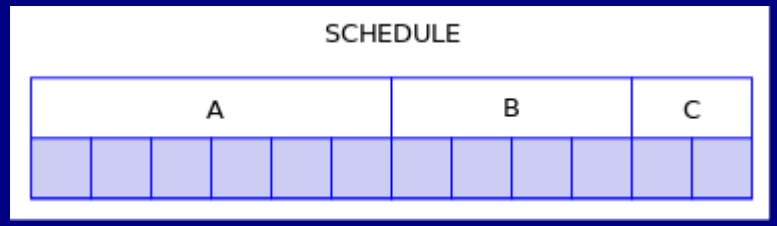
- Процесс - абстракция ОС созданная с использованием аппаратной поддержки:
  - в объяснении выше использовалась таблица страниц для организации памяти;
  - мы полагались на прерывания, для переключения потоков внутри процесса и между процессами.
- Не трудно создать ОС, в которой такой абстракции не будет:
  - например, если аппаратной поддержки нет или вам просто не нужна такая абстракция;
  - примеры существуют: MS-DOS;
  - абстракции не даются бесплатно - вы платите за это производительностью.

Планирование потоков

- Мы знаем что-такое потоки и как между ними переключаться
  - осталось разобраться с мелочами - когда и на какой конкретно поток переключаться.
- Начнем с простой и нереалистичной задачи:
  - мы заранее знаем все задачи, которые нужно выполнить;
  - и про каждую задачу знаем сколько времени потребуется на выполнение;
  - более того мы выполняем каждую задачу от начала до конца без переключений, т. е. нужно только определить порядок выполнения задач.

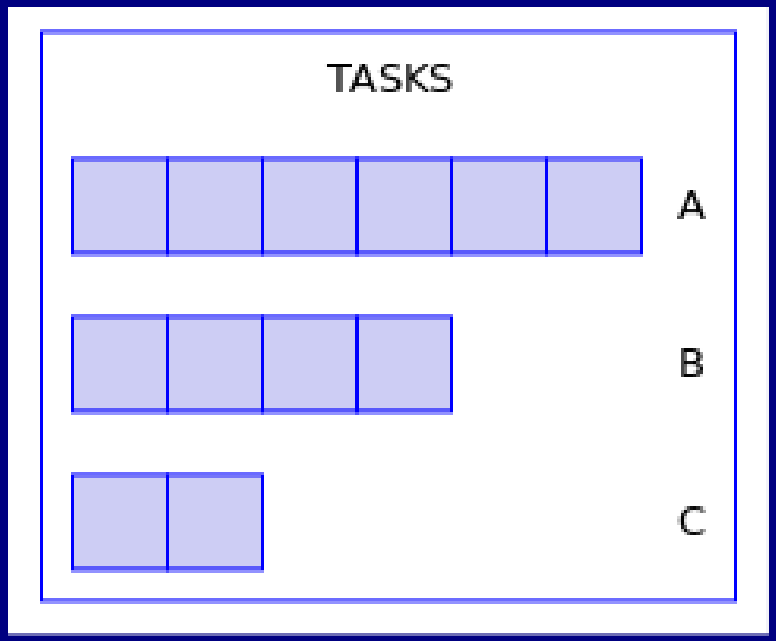
SJF 1/2

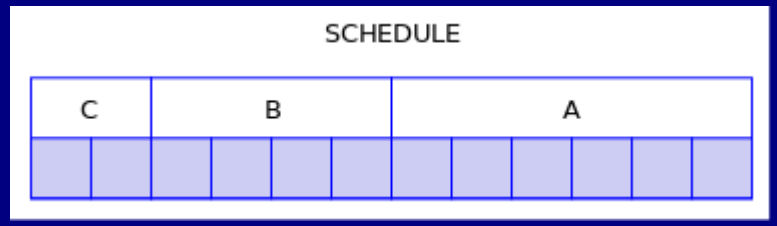




- 3 задачи, по 6, 4 и 2 единиц;
- суммарно 12 единиц времени;
- среднее время ожидания заверешения: $\frac{6+10+12}{3} = 9.(3)$

SJF 2/2





- теже 3 задачи, по 6, 4 и 2 единиц;
- суммарно 12 единиц времени (о чудо!);
- среднее время ожидания заверешения: $\frac{2+6+12}{3} = 6.(6)$

Shortest Job First

- Упорядочив задачи по времени исполнения от меньшей к большей получим оптимальное по среднему времени ожидания расписание
  - отсюда и Shortest Job First (SJF).

Блокировка потока

- Потоки могут быть заблокированы:
  - поток может ожидать ввода от пользователя - даже самый быстрый пользователь очень медленный по сравнению с CPU;
  - поток может ожидать получения данных по сети;
  - поток может запросить доступ к медленному устройству и ждать прерывания от него (например, диск);
  - другими словами поток может быть заблокирован в ожидании завершения операции ввода/вывода.
- Заблокированным потокам нет смысла отдавать CPU:
  - все что они могут делать, так это ждать завершения IO.

Динамическое создание и завершение

- Потоки создаются и завершаются динамически:
  - новый поток может быть создан в любой момент;
  - т. е. все потоки заранее не известны;
  - существующий поток может завершиться;
  - т. е. мы не знаем время необходимое потоку для завершения.

Round Robin

- Самый простой вариант планирования в реалистичных условиях - отдавать процессор активным потокам по очереди:
  - все задачи ожидающие CPU организованы в очередь;
  - каждой задаче выделяется квант времени;
  - задача снимается с CPU по истечении кванта;
  - задача может отдать CPU самостоятельно перед истечением кванта;
  - незавершившаяся задача снятая с CPU возвращается в конец очереди.

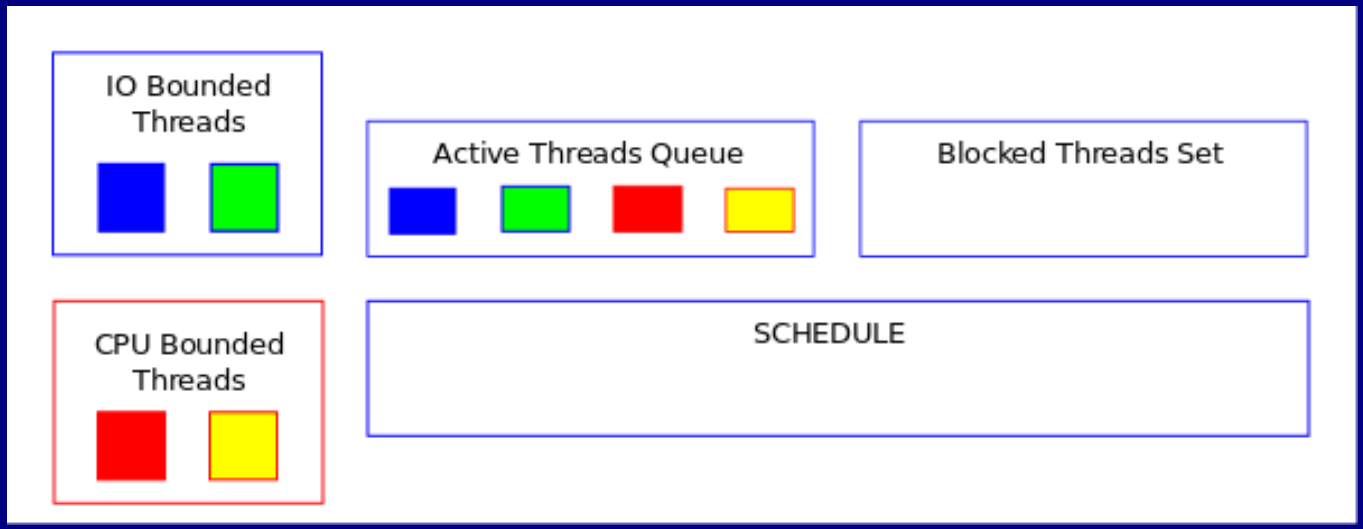
Round Robin, pros

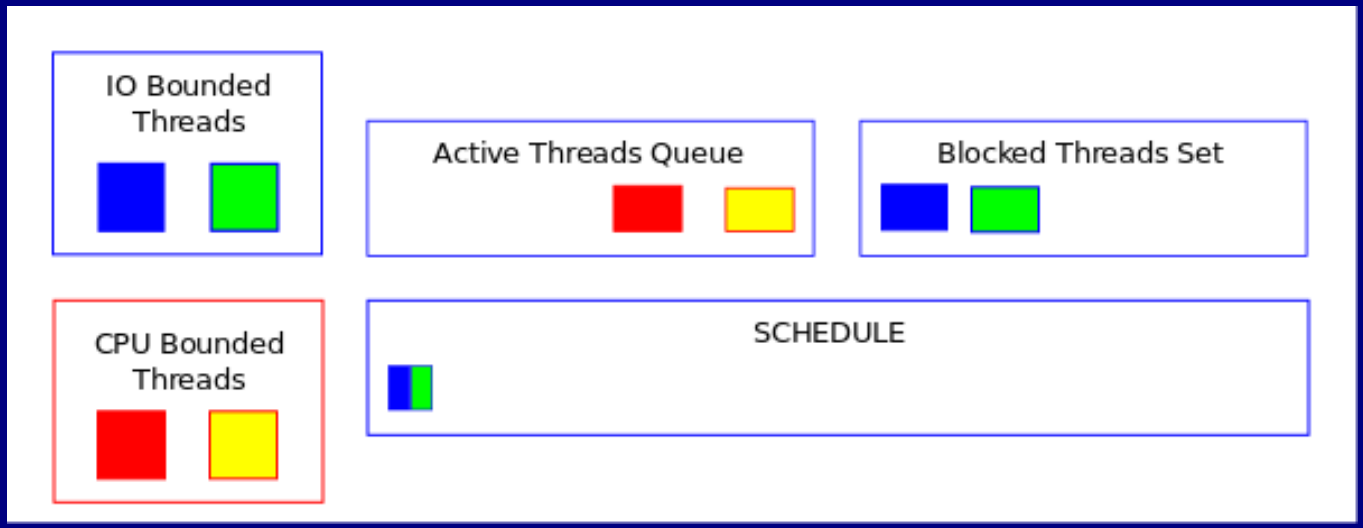
- Round Robin - является простым реалистичным алгоритмом планирования:
  - выбор следующего потока требует $O(1)$;
  - зная ограничение на количество потоков, мы можем ограничить максимальное время ожидания.
- Гарантия, когда время реакции системы на событие (прерывание) жестко ограничено сверху - гарантия жесткого реального времени:
  - т. е. Real Time на самом деле - это не про скорость;
  - Round Robin сам по себе не нарушает гарантию реального времени.

Round Robin, cons 1/3

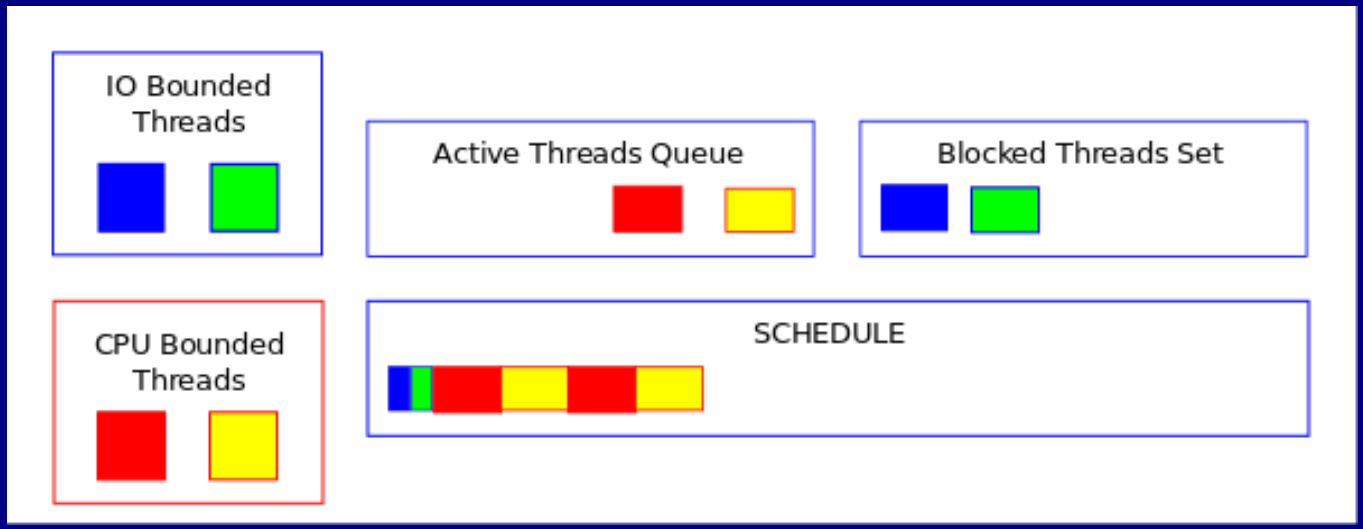
- IO Bounded поток - поток, работа которого ограничена скоростью IO:
  - такие потоки часто не вырабатывают свой квант полностью, а отдают CPU раньше заблокировавшись на IO;
  - типичный пример: текстовый редактор.
- CPU Bounded поток - работа потока ограничивается скоростью CPU:
  - такие потоки редко не вырабатывают свой квант полностью, они всегда хотят получить CPU в свое пользование;
  - типичные примеры: численные расчеты, задачи рендеринга.

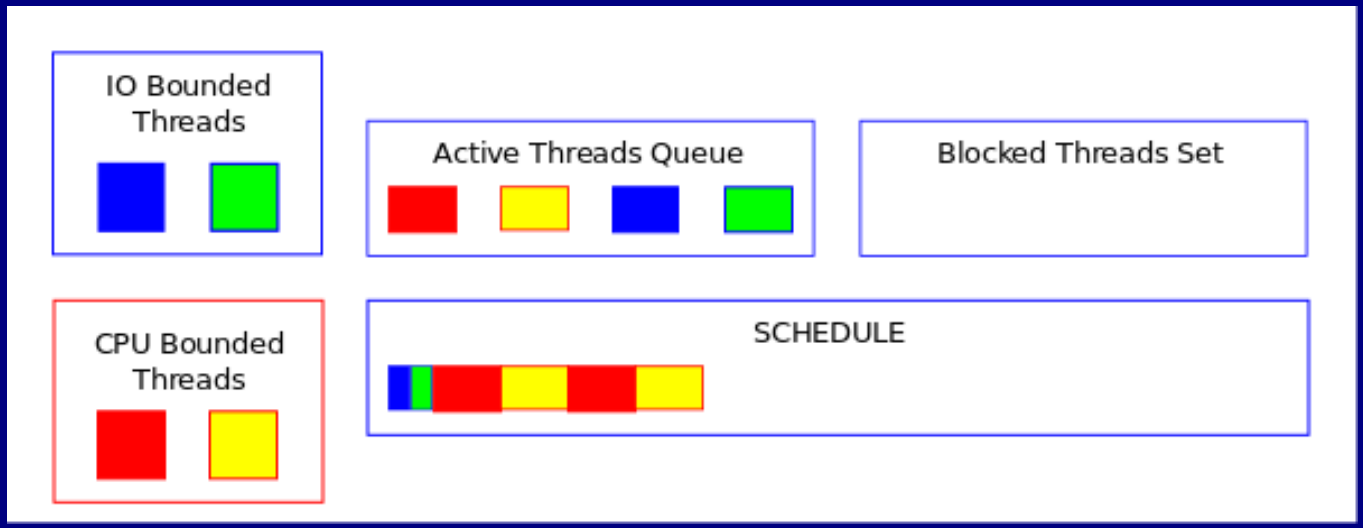
Round Robin, cons 2/3



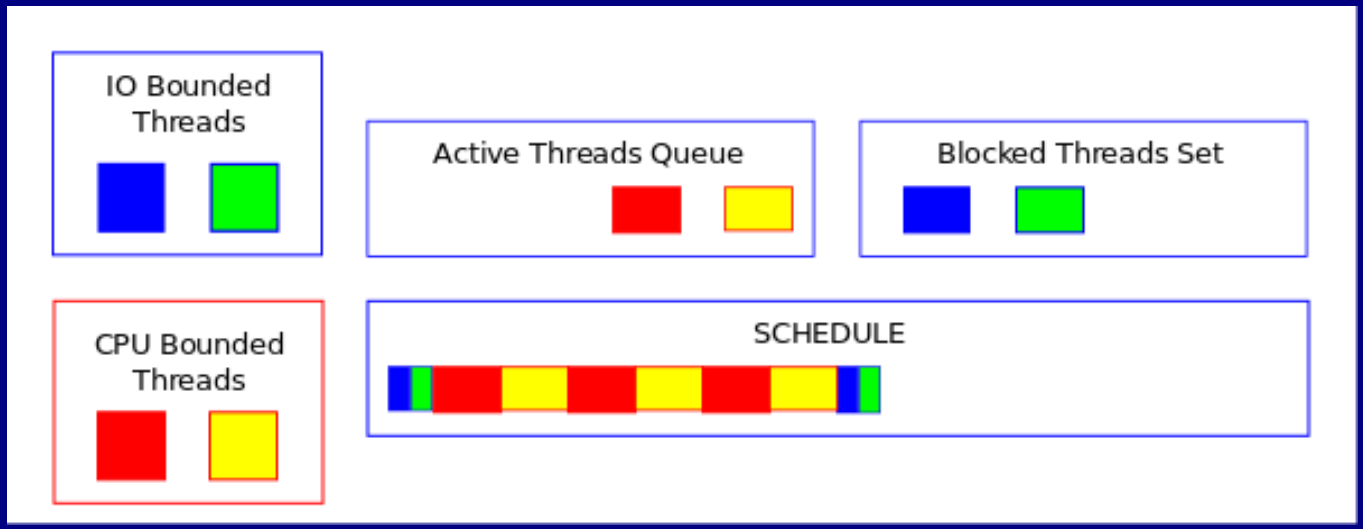


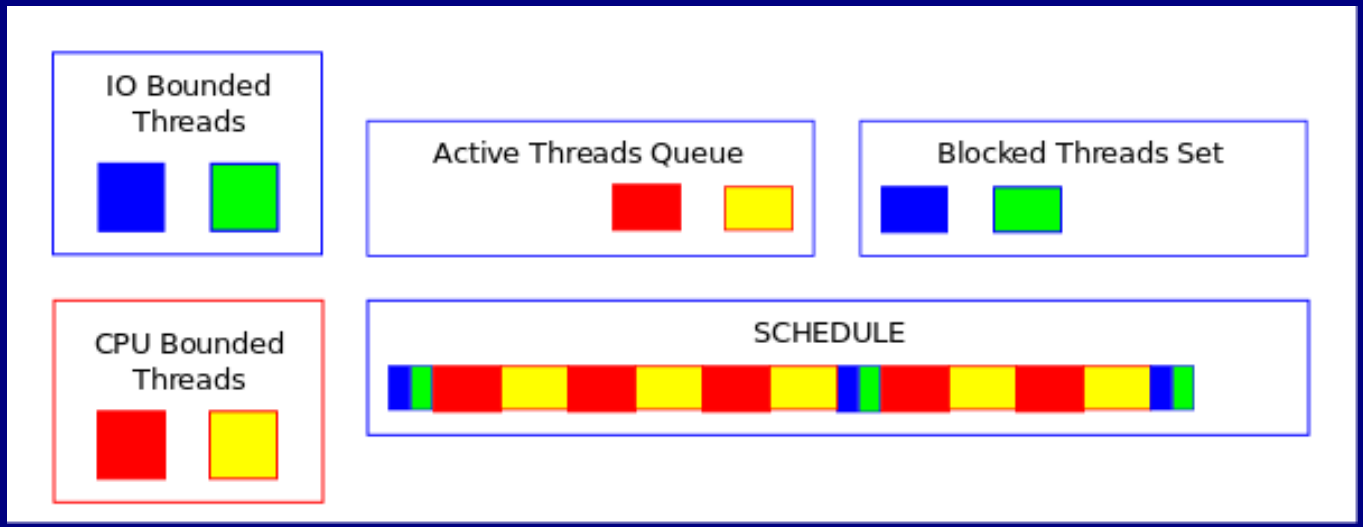
Round Robin, cons 2/3





Round Robin, cons 3/3





Честность Round Robin

- IO Bounded задачи при Round Robin получают меньше CPU:
  - они не вырабатывают свой квант и блокируются - не выработанное время никак не компенсируется.
- IO Bounded задачи часто являются интерактивными, т. е. работают с пользователем, а пользователь не любит ждать
  - однако когда поток разблокируется он встает в конец очереди и ждет пока вся очередь отработает.
- Итого IO Bounded задачи получают меньше CPU, а задержка по доступу к CPU у них такая же как и у всех
  - не все задачи одинаковые, и может потребоваться разделение задач на классы и приоритизация.

Completely Fair Scheduler 1/2

- Планировщик по-умолчанию в Linux Kernel
  - пытается гарантировать идеальную честность отслеживая "виртуальное" отработанное время каждого потока;
  - поток с нименьшим "виртуальным" временем получает CPU;
  - выбор следующего потока требует $O(logn)$.

Completely Fair Scheduler 2/2

- Как обрабатывать разблокированные потоки вновь созданные потоки?
  - нельзя просто верунть разблокированные задачи в очередь, т. к. их "виртуальное" время может сильно отстать от других и они станут "слишком приоритетными";
  - для очереди потоков поддерживается "минимальное виртуальное" время, для новых и разблокированных задач проверяется, что их "виртуальное" время не сильно отстает от "минимального виртуального" времени.

Лотерейное планирование 1/2

- Самый "простой" способ обеспечить честность - рандомизация:
  - выдадим каждому потоку набор лотерейных билетов;
  - вероятность получения CPU в каждом "розыгыше" определяется количеством билетов у потока.
- Статические и динамические приоритеты:
  - потокам можно выдавать разное количество билетов согласно приоритету;
  - кроме того, потоки могут временно передавать билеты друг другу и изменять свои приоритеты;
  - например, поток A пытается получить доступ к ресурсу X, но ресурс X уже занят потоком B - пусть A передаст свои билеты B, чтобы он раньше закончил работу с X.

Лотерейное планирование 2/2

- При честной рандомизации возможны выбросы
  - рандомизация гарантирует математическое ожидание времени CPU, но возможны отклонения от математического ожидания;
  - чтобы ограничить отклонения, мы можем "удалять" выигравший билет;
  - когда все билеты были "удалены", мы раздаем билеты заново;
  - выбросы все еще возможны, но они ограничены (почти).

Q&A