

Operating Systems

Threads Scheduling

Me

February 25, 2016

- 1 Понятия потока и контекста потока. Переключение контекста.
- 2 Простые алгоритмы планирования (SJF, RR).
- 3 Приоритетное планирование (STCF, MLFQ).
- 4 Честное планирование (Linux CFS, Lottery Scheduling)

Поток исполнения и контекст потока

Поток (thread, поток исполнения) - набор исполняемых инструкций процессора

- указатель команд (instruction pointer, IP, RIP в x86-64) - указывает на текущую инструкцию;
- регистр флагов (flag register, state register, RFLAGS в x86-64) - определяет поведение команд;
- указатель стека (stack pointer, RSP в x86-64) - указывает на вершину стека (указатель стека часто относят к регистрам общего назначения);
- прочие регистры общего назначения (и не только);
- процесс - поток связан с процессом (работает в рамках какого-то процесса).

Переключение контекста

- сохраняем контекст одного потока в известное место (например, на стек этого потока);
- восстанавливаем контекст другого потока из известного места;
- если нужно переключаем процессы (например, меняем таблицу страниц);
- x86-32 поддерживает аппаратное переключение контекста

Переключение контекста

- сохраняем контекст одного потока в известное место (например, на стек этого потока);
- восстанавливаем контекст другого потока из известного места;
- если нужно переключаем процессы (например, меняем таблицу страниц);
- x86-32 поддерживает аппаратное переключение контекста, но никто им не пользовался, потому что в x86-64 такой возможности больше нет.

Пример переключения контекста для x86-64

```
1 ; void switch_threads(void **old_sp, void *new_sp); // C signature
2
3 switch_threads:
4     pushq %rbp          ; save volatile registers
5     pushq %rbx
6     pushq %r12
7     pushq %r13
8     pushq %r14
9     pushq %r15
10
11     movq %rsp, (%rdi) ; save SP
12     movq %rsi, %rsp   ; restore SP
13
14     popq %r15          ; restore volatile register
15     popq %r14
16     popq %r13
17     popq %r12
18     popq %rbx
19     popq %rbp
20
21     ret
```

Когда переключаться между потоками?

- когда поток сам добровольно отдаст управление (невывесняющая многозадачность)
 - добровольно вызовет перепланирование;
 - совершит блокирующую операцию (mutex lock, condtion wait);
- в обработчике прерывания (вытесняющая многозадачность)
 - например, по прерыванию таймера;

Примитивная задача планирования

Список задач (даже пока не потоков) дан заранее:

Примитивная задача планирования

Список задач (даже пока не потоков) дан заранее:

- про каждую задачу известна ее длительность;

Примитивная задача планирования

Список задач (даже пока не потоков) дан заранее:

- про каждую задачу известна ее длительность;
- задача работает от начала и до конца (не прерываясь, не вытесняясь);

Примитивная задача планирования

Список задач (даже пока не потоков) дан заранее:

- про каждую задачу известна ее длительность;
- задача работает от начала и до конца (не прерываясь, не вытесняясь);
- какой смысл вообще что-то придумывать в такой ситуации?

Пример

A 6

B 4

C 3

A	B	C
---	---	---

Figure : Example Schedule

Пример

A 6

B 4

C 3

A B C

- $TOTAL = 6 + 4 + 3 = 13$

Figure : Example Schedule

Пример

A 6

B 4

C 3



- $TOTAL = 6 + 4 + 3 = 13$
- $WAIT_A = 6$
- $WAIT_B = 6 + 4 = 10$
- $WAIT_C = 6 + 4 + 3 = 13$

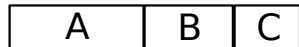
Figure : Example Schedule

Пример

A 6

B 4

C 3



- $TOTAL = 6 + 4 + 3 = 13$
- $WAIT_A = 6$
- $WAIT_B = 6 + 4 = 10$
- $WAIT_C = 6 + 4 + 3 = 13$
- $WAIT_{AVG} = \frac{(6+10+13)}{3} = 9. (6)$

Figure : Example Schedule

Пример

A 6

B 4

C 3

C B A

Figure : Example Schedule

Пример

A 6

B 4

C 3

C B A

- $TOTAL = 6 + 4 + 3 = 13$

Figure : Example Schedule

Пример

A 6

B 4

C 3

C B A

- $TOTAL = 6 + 4 + 3 = 13$
- $WAIT_C = 3$
- $WAIT_B = 3 + 4 = 7$
- $WAIT_A = 6 + 4 + 3 = 13$

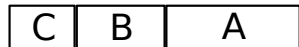
Figure : Example Schedule

Пример

A 6

B 4

C 3



- $TOTAL = 6 + 4 + 3 = 13$
- $WAIT_C = 3$
- $WAIT_B = 3 + 4 = 7$
- $WAIT_A = 6 + 4 + 3 = 13$
- $WAIT_{AVG} = \frac{(3+7+13)}{3} = 7. (6)$

Figure : Example Schedule

Shortes Job First

Shortest Job First (SJF) - гарантирует минимальное *среднее* время ожидания завершения задачи.

Shortes Job First

Shortest Job First (SJF) - гарантирует минимальное *среднее* время ожидания завершения задачи.

Доказательство: ну это же очевидно!

Shortest Job First

Доказательство:

- планируем N задач, с временами работы T_0, T_1, \dots, T_{N-1} ;
- допустим, что в расписании с наименьшим $WAIT_{AVG}$ есть две задачи s и t , такие что $T_s > T_t$, но s идет раньше чем t ;
- поменяем местами s и t в расписании;
- все задачи до t и начиная с s имеют тоже самое время ожидания;
- все задачи начиная с t и до s имеют меньшее время ожидания;
- получили расписание с меньшим $WAIT_{AVG}$ - противоречие.

- задачи могут обращаться к медленным внешним устройствам или ждать внешних событий:
 - задача, которая только ждет - бесполезно занимает процессор (естественная точка перепланирования);

- задачи могут обращаться к медленным внешним устройствам или ждать внешних событий:
 - задача, которая только ждет - бесполезно занимает процессор (естественная точка перепланирования);
- задачу стоит снять с процессора:
 - если задача не критична к времени отклика, то стоит использовать прерывания;
 - если внешнее устройство обрабатывает запросы долго, то стоит использовать прерывания;
 - если время ожидания не предсказуемо (получение пакета по сети или ожидание нажатия клавиши), то стоит использовать прерывания;

Пример

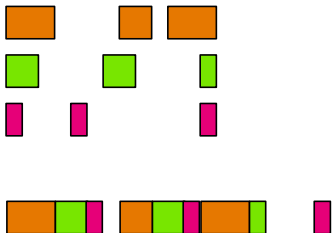


Figure : Schdeule with IO

- в расписании могут появляться "дыры" (все ждут завершения IO)
- интуиция - хотим, максимально параллельный IO:
 - реалистично для сети или клавиатуры;
 - не реалистично, например, для диска - общий ресурс.

Пример

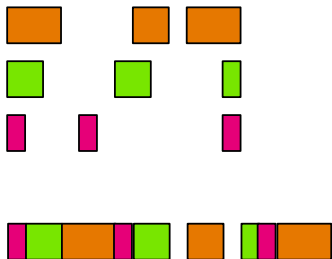


Figure : Schdeule with IO

- задачу с ближайшим IO пропускаем вперед:
 - чем раньше задача начнет IO, тем лучше;
 - чем больше задач будет выполнять IO параллельно, тем лучше;
 - при прочих равных, задачу с большим IO стоит пустить раньше;
 - получаем SJF с учетом IO;

Ограничения

- мы не знаем заранее время до следующего IO (или завершения)
 - мы можем его примерно оценить основываясь на "истории" работы задачи;

- мы не знаем заранее время до следующего IO (или завершения)
 - мы можем его примерно оценить основываясь на "истории" работы задачи;
- список задач не дан заранее
 - SJF теряет свою оптимальность в этом случае (трудно планировать без полной информации и без возможности изменить решение);
 - для оптимального планирования нам нужно вытеснять задачи в произвольные моменты времени (вытесняющая многозадачность);

- мы не знаем заранее время до следующего IO (или завершения)
 - мы можем его примерно оценить основываясь на "истории" работы задачи;
- список задач не дан заранее
 - SJF теряет свою оптимальность в этом случае (трудно планировать без полной информации и без возможности изменить решение);
 - для оптимального планирования нам нужно вытеснять задачи в произвольные моменты времени (вытесняющая многозадачность);
- IO зависит от внешних условий (пользователь, нагрузка, другие задачи);

Round Robin

Сделаем ограничения более реалистичными (переходим от задач к потокам):

- мы можем снимать поток с процессора в любое время (не спрашивая разрешения у потока):
 - когда задача запускает операцию ввода/вывода;
 - по сигналу от таймера;

Сделаем ограничения более реалистичными (переходим от задач к потокам):

- мы можем снимать поток с процессора в любое время (не спрашивая разрешения у потока):
 - когда задача запускает операцию ввода/вывода;
 - по сигналу от таймера;
- больше мы ничего не знаем о потоке.

Round Robin

Если мы ничего не знаем о потоках - дадим каждому по чуть-чуть по очереди:

Round Robin

Если мы ничего не знаем о потоках - дадим каждому по чуть-чуть по очереди:

- нужно ограничить время работы на процессоре (квант времени)
 - поток может "заблокироваться" раньше, чем истечет квант времени;

Round Robin

Если мы ничего не знаем о потоках - дадим каждому по чуть-чуть по очереди:

- нужно ограничить время работы на процессоре (квант времени)
 - поток может "заблокироваться" раньше, чем истечет квант времени;
- все активные потоки выполняются по очереди
 - если поток отработал квант, ставим его в конец очереди;
 - если поток "разблокировался", ставим его в конец очереди;

Round Robin

Если мы ничего не знаем о потоках - дадим каждому по чуть-чуть по очереди:

- нужно ограничить время работы на процессоре (квант времени)
 - поток может "заблокироваться" раньше, чем истечет квант времени;
- все активные потоки выполняются по очереди
 - если поток отработал квант, ставим его в конец очереди;
 - если поток "разблокировался", ставим его в конец очереди;
- Round Robin дает гарантии реального времени:
 - зная, сколько всего потоков мы можем посчитать максимальное время ожидания;

Что не так с Round Robin?

Рассмотрим планирование нескольких задач из двух классов:

Что не так с Round Robin?

Рассмотрим планирование нескольких задач из двух классов:

- 1 CPU bounded - не блокируются, постоянно вырабатывают свой квант:
 - например, какие-то численные расчеты попадают в этот класс;

Что не так с Round Robin?

Рассмотрим планирование нескольких задач из двух классов:

- ❶ CPU bounded - не блокируются, постоянно вырабатывают свой квант:
 - например, какие-то численные расчеты попадают в этот класс;
- ❷ IO bounded - работают мало, ждут долго - не вырабатывают свой квант:
 - например, текстовый редактор большую часть времени ждет ввода;

Что не так с Round Robin?



CPU bounded



IO bounded

Active: 

Blocked:


Schedule: 

Figure : Round Robin Example

Что не так с Round Robin?



CPU bounded



IO bounded

Active:

Blocked:

Schedule:

Figure : Round Robin Example

Что не так с Round Robin?

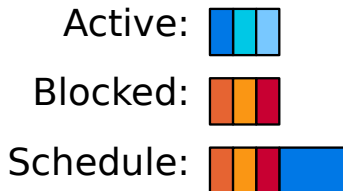
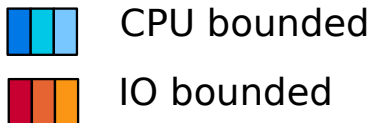


Figure : Round Robin Example

Что не так с Round Robin?

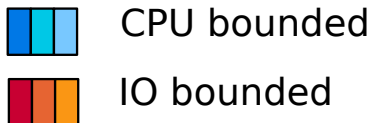


Figure : Round Robin Example

Что не так с Round Robin?



CPU bounded



I/O bounded

Active: 


Blocked: 

Schedule: 

Figure : Round Robin Example

Что не так с Round Robin?

 CPU bounded

 IO bounded

Active: 

Blocked: 

Schedule: 

Figure : Round Robin Example


Что не так с Round Robin?



CPU bounded



IO bounded

Active: 

Blocked: 

Schedule: 

Figure : Round Robin Example

Что не так с Round Robin?

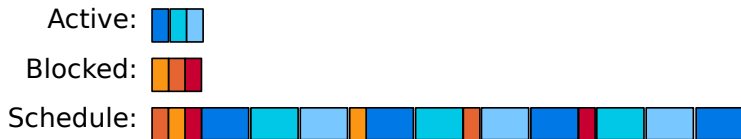
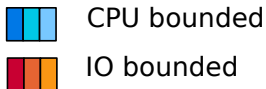


Figure : Round Robin Example

Что не так с Round Robin?

- IO bounded потоки получают много меньше CPU, а ждут столько же:
 - это кажется не честным;
 - IO bounded потоки - это, часто, интерактивные потоки (т. е. пользователь не доволен);

Что не так с Round Robin?

- IO bounded потоки получают много меньше CPU, а ждут столько же:
 - это кажется не честным;
 - IO bounded потоки - это, часто, интерактивные потоки (т. е. пользователь не доволен);
- нужна возможность учитывать разные классы задач:
 - в примере мы могли ставить IO bounded задачи в начало (осторожно, может приводить к старвации!);
 - приоритетизация задач:
 - IO bounded vs CPU bounded;
 - внутри класса можно варьировать квант времени;

Приоритетное планирование

Назначим каждому потоку приоритет. Приоритет может определять:

- порядок - более приоритетные идут вперед;
- размер кванта - более приоритетные работают дольше;

Приоритетное планирование

Назначим каждому потоку приоритет. Приоритет может определять:

- порядок - более приоритетные идут вперед;
- размер кванта - более приоритетные работают дольше;

Приоритет может назначаться

- статически (например, SJF - приоритет определяется длиной задачи);
- динамически - приоритет меняется в процессе работы (например, SCTF);

Shortest Completion Time First

То же самое, что и SJF, но теперь мы можем снять задачу с процессора:

- мы можем захотеть перепланировать, если появилась новая задача;
- приоритет - оставшееся время выполнения задачи (динамический - изменяется, когда задача выполняется);

Старвация в SCTF

Priority Queue:    

Schedule:

Figure : SCTF Example

Старвация в SCTF

Priority Queue:



Schedule:



Figure : SCTF Example

Старвация в SCTF

Priority Queue:



Schedule:



Figure : SCTF Example

Старвация в SCTF

Priority Queue:



Schedule:



Figure : SCTF Example

Старвация в SCTF

Priority Queue:



Schedule:



Figure : SCTF Example

Старвация в SCTF

Priority Queue:



Schedule:



Figure : SCTF Example

Старвация в SCTF

Priority Queue:



Schedule:



Figure : SCTF Example

Старвация в SCTF

Priority Queue:



Schedule:



Figure : SCTF Example

Multi Level Feedback Queue

Заведем несколько очередей (по очереди на приоритет):

- исполняем потоки из самой приоритетной очереди;

Multi Level Feedback Queue

Заведем несколько очередей (по очереди на приоритет):

- исполняем потоки из самой приоритетной очереди;
- очередь потока меняется в зависимости от "истории":
 - выработал свой квант - понижаем приоритет и увеличиваем квант;
 - не выработал свой квант - повышаем приоритет и уменьшаем квант;

Multi Level Feedback Queue

Заведем несколько очередей (по очереди на приоритет):

- исполняем потоки из самой приоритетной очереди;
- очередь потока меняется в зависимости от "истории":
 - выработал свой квант - понижаем приоритет и увеличиваем квант;
 - не выработал свой квант - повышаем приоритет и уменьшаем квант;
- все потоки начинают с наибольшим приоритетом и наименьшим квантом;
 - в оригинальной версии очередь определялась размером программы;
 - Fernando J. Corbató получил премию Тьюринга (не только за это);

Предыдущие подходы к планированию пытались:

- оптимизировать время ожидания или уменьшить простаивание (SJF);
- предоставить гарантии на время отклика (RR);
- подобрать оптимальный "приоритет" для задачи (MLFQ);

Честное планирование

Предыдущие подходы к планированию пытались:

- оптимизировать время ожидания или уменьшить простаивание (SJF);
- предоставить гарантии на время отклика (RR);
- подобрать оптимальный "приоритет" для задачи (MLFQ);

Теперь посмотрим на проблему с другой стороны:

- давайте честно делить ресурсы CPU между потоками (процессами, пользователями);

Честное планирование

Честное планирование не совсем очевидная вещь:

- потоки могут делать IO в произвольные моменты времени
 - нельзя сказать потоку работать какое-то конкретное время;
 - время работы можно ограничить только сверху;
- могут появляться новые потоки
 - не очевидно как должна работать "честность" при изменениях условий;

Честное планирование

Честное планирование не совсем очевидная вещь:

- потоки могут делать IO в произвольные моменты времени
 - нельзя сказать потоку работать какое-то конкретное время;
 - время работы можно ограничить только сверху;
- могут появляться новые потоки
 - не очевидно как должна работать "честность" при изменениях условий;
- если вы не знаете что делать - используйте рандом;

Лотерейное планирование

У каждого потока (процеса или пользователя) есть набор билетов

- при распределении ресурсов выбирается случайный номер билета
 - ресурс получает владелец выигравшего билета;
 - вероятность выигрыша определяется количеством билетов;
- свои билеты можно передавать/разделять:
 - процесс может разделить свои билеты между потоками;
 - пользователь может делить свои билеты между процессами;
 - если поток ждет другой поток, то он может передать ему свои билеты;
- можно создавать новые билеты или уничтожать старые.

Completely Fair Scheduler

История планировщиков Linux:

- 1 Linux 1.2 - Round Robin;
- 2 Linux 2.2 - поддержка SMP (многопроцессорное планирование);
- 3 Linux 2.4 - добавили планировщик со сложностью $O(N)$ (он обходил все потоки в системе);
- 4 Linux 2.6 - $O(1)$ scheduler (это название), вариация на тему MLFQ;
- 5 Linux 2.6 - Completely Fair Scheduler, "честное" разделение CPU на основе динамических приоритетов без рандомизации;

Completely Fair Scheduler

- у каждого потока есть virtual runtime, чем меньше virtual runtime тем выше приоритет:
 - *активные* потоки хранятся в красно-черном дереве упорядоченными по virtual runtime;
 - virtual runtime изменяется, когда поток работает (т. е., упрощенно, изменение $\text{virtual runtime} = k * \text{CPU time}$);

Completely Fair Scheduler

- у каждого потока есть virtual runtime, чем меньше virtual runtime тем выше приоритет:
 - *активные* потоки хранятся в красно-черном дереве упорядоченными по virtual runtime;
 - virtual runtime изменяется, когда поток работает (т. е., упрощенно, изменение $\text{virtual runtime} = k * \text{CPU time}$);
- какой virtual runtime взять для нового (или разблокированного) потока? Берем наибольшее значение из:
 - минимальный virtual runtime в дереве;
 - virtual runtime потока (если это не новый поток);