Про планировщики и их виды

Задача планирования возникает из желания реализации многопоточности на одном процессоре (ну или на нескольких, но всё же каждый процессор должен обслуживать сразу много нитей, иначе интерактивная работа на машине невозможна).

Самый простой способ реализации — так называемая **кооперативная многозадачность.**
По большому счету нужна структура нити и три операции:

1. Создать нить

```
Thread* CreateThread(void* threadRoutine);
```

2. Переключить нить

```
void SwitchThread();
```

3. Завершить нить

```
void FinishThread();
```

По смыслу, функция SwitchThread и называется **планировщиком**, он занимается тем, что переключает нить с текущей активной на следующую активную. Текущая нить останавливает исполнение и ждет в очереди активных нитей, через некоторое время получая управление обратно, при этом нить не должна заметить переключения (необходимо сохранить контекст нити — все регистры общего назначения, указатель стека, счетчик команд, слово состояния), и продолжить исполняться как если бы SwitchThread была бы простой сишной функцией которая вернула управление в точку из которой её позвали.

SwitchThread должна быть реализована на низком уровне, потому что требуется изменять стек и запись активации, чтобы передать управление в другую нить. SwitchThread должна уметь менять stack pointer и сохранять предыдущий SP в дескрипторе отработавшей нити.

При описанной системе планирования, каждая программа должна делать какую то полезную работу, затем вызывать ThreadSwitch(); Легко понять что это доставляет немало проблем — багованная или злая нить может на всегда захватить управление и остановить всю работу. К тому же, если мы пишем какую то вычислительную программу, использующую циклы с большим числом итераций, нам нужно будет звать ThreadSwitch на каждой итерации, или каждые N итераций. Это безусловно скажется на производительности потому что компилятору будет труднее такое оптимизировать, а они как известно очень умные и часто занимаются конвейеризацией вычислений, что требует перестановки команд.

Хорошо, на пользователя полагаться не можем, значит потоки должна переключать система. Это приводит к идее вытесняющей многозадачности.

Идея в использовании прерываний и реакции на эти прерывания переключением потока.

Например, механизм TS - time sharing:

- Используется сигнал системного таймера, прерывает планировщик каждый квант времени.
- Каждая нить получает квант времени
- Если нить не закончила вычисления в течении своего кванта её приостанавливают и ставят в конец очереди.

Со статическим квантом есть проблемы - слишком короткий заставит процессор переключать нити большую часть времени, при слишком большом кванте приведет к слишком большому времени реакции. Поэтому в системах реального времени есть реальный и разделенный классы планирования. Нити реального времени прерываются по таймеру и их могут вытеснить только более приоритетные нити реального времени. Разделенные же вытесняются нитями разделенными и реального времени.

Расширяя эту идею приходим к планировщикам с приоритетами. (я бы тут могла объяснить на примерах почему нам некоторые нити могут требовать более быстрой обработки нежели другие но мне очень лень и это вполне очевидно)

Есть разные реализации таких планировщиков но самым распространенным способом деления нитей на приоритеты является составление нескольких очередей (по очереди для каждого уровня приоритета).

Это такая очередь очередей — самая приоритетная очередь будет исполняться пока станет пустой, и тд. То есть, самые неприоритетные нити исполнятся только когда все очереди с более высоким приоритетом будут пусты.

Приоритет вычисляется динамически. Если система хочет повысить среднее время реакции на действия пользователя, то системе нравятся нити которые что-то быстренько посчитали и ушли ждать какие то данные, а нити, которые долго и настырно что-то считают и их приходится прерывать на “пол пути” сохраняя все что она там насчитала, система не любит. Поэтому, приоритеты так и выставляются:

- нитям, освободившим процессор в результате запроса на ввод/вывод приоритет повышается
- нитям которые были сняты по таймеру приоритет понижается

Понятно, что бесконечно повышать приоритет нельзя — иначе некоторые нити вовсе никогда не выполнятся. В UNIX системах приоритет задачи не может повысится сверх определенного базового уровня. Там приоритет процесса складывается из статического приоритета (nice
level) и штрафа за использование процессора (CPU penalty). Оба эти значения
представляют собой положительные целые числа; чем больше их сумма, тем
ниже приоритет задачи. Nice level наследуется от родительского процесса. Штраф начисляется, когда система отнимает у задачи управление по исчерпанию кванта времени, и сбрасывается, когда задача сама уступает процессор. Nice level можно поменять системным вызовом, но обычный пользователь может его лишь повысить.

Ещё немного про **честное планирование**:

- Используется в компьютерах коллективного пользования: разделяемый хостинг, облачные инфраструктуры.
- Пользователи платят за процессорное
  время, которое нельзя выдать одним куском: на примере хостинга,
  процессорное время складывается из всех нажатий пользователей.
- Вводится понятие периода справедливости / кванта справедливости ,
  например, 1 секунда.
- Процессы делятся на группы (в Linux - cgroup).
- Каждой группе выделяем долю времени в каждом периоде справедливости.
- Если пользователь заплатил за 10% процессора, то ему каждую секунду будут выделять 100 мс.
- Когда интервал справедливости начался, мы планируем процессы, которые еще не съели свою долю, если они готовы к исполнению
- В отличие от классического планировщика разделенного времени, где решение о планировании можно принять за константное время (поменять приоритет у текущего процесса + засунуть его в какую-нибудь очередь), честный
  планировщик требует поиска по группам. Получаем, что время планирования
  зависит от количества групп и процессов соответственно.

Ещё есть такой прикол как **инверсия приоритетов**. Это когда высокоприоритетные процессы ждут низкоприоритетные (например, из-за несогласованности данных)

- При расчете времени реакции на событие разработчик системы реального времени должен принимать во внимание не только время исполнения кода, непосредственно обрабатывающего это событие, но и времена работы всех критических секций во всех нитях, которые могут удерживать мутексы, необходимые обработчику , ведь обработчик не сможет продолжить исполнение, пока не захватит эти мутексы, а произвольно снимать их нельзя, потому что они сигнализируют, что защищаемый ими разделяемый ресурс находится в несогласованном состоянии.

- Если высокоприоритетная нить пытается захватить мутекс, занятый низкоприоритетной нитью, то в определенном смысле получится, что эта нить будет работать со скоростью низкоприоритетного процесса .
- Низкоприоритетные нити могут не получать управление в течение длительного времени. Соответственно, на то же время будет заблокирована высокоприоритетная нить, которая пытается захватить мутекс, захваченный низкоприоритетной.
- Аналогичная проблема может возникать также при работе с синхронными примитивами гармонического взаимодействия — линками транспьютера, рандеву языка Ада и т. д.

Что делать?

Наследование приоритета:

Обычно наследование контролируется флагом в параметрах мутекса или в параметрах системного вызова, захватывающего мутекс. Если высокоприоритетная нить пытается захватить мутекс с таким флагом, то приоритет нити, удерживающей этот мутекс, приравнивается приоритету нашей нити. Таким образом, в каждый момент времени реальный приоритет нити, удерживающей мутекс, равен наивысшему из приоритетов нитей, ожидающих этого мутекса.

Потолок приоритета:

Приоритет нити, удерживающей мутекс, приравнивается наивысшему из приоритетов нитей, которые могут захватить этот мутекс. Разумеется, во время исполнения определить потолок приоритета невозможно, он должен устанавливаться программистом как параметр мутекса.

И **наследование**, и **потолок** приоритета применимы только к мутексам и другим примитивам синхронизации, для которых можно указать, какая именно нить их удерживает. Точнее говоря, для борьбы с инверсией приоритета необходимо, чтобы приоритет повышался для нити, которая будет освобождать семафор. Для мутексов это всегда та же нить, которая его захватывала, но для семафоров-счетчиков это неверно. Нам всегда надо знать, какая нить сидит в мутексе, потому что мы повышаем ей приоритет с приходом высокоприоритетной. По выходу из мутекуса надо вернуть его вниз