Прерывайте вежливо
Грубая сила это не выход

http://www.ddj.com, 9.04.2008, Герб Саттер. Перевод — Пеленицын А. М.

Остановка потоков или задач, которые больше не нужны, важна для эффективности. Но как вы её осуществляете?

Мы хотим иметь возможность остановить запущенный поток или задачу, когда обнаружили, что он больше не нужен или мы хотим завершить его. Как мы видели в двух предыдущих статьях, при простом параллельном поиске мы можем остановить один поток, когда другой нашёл нужное, и когда одновременно запущены два альтернативных алгоритма для вычисления одного результата, мы можем остановить более медленный, когда второй нашел ответ. [1,2] Останов потоков или задач позволяет нам вернуть обратно их ресурсы, включая «локи», и задействовать их в другой работе.

Но как именно остановить поток или задачу, которые больше не нужны или не желательны? Таблица 1 резюмирует четыре основные пути и то, как они поддерживаются на самых распространённых платформах. Давайте изучим их по очереди.

	1. Kill	2. Tell, don't take no for an answer	3. Ask politely, and accept rejection	4. Set flag politely, let it poll if it wants
Tagline	Shoot first, check invariants later	Fire him, but let him clean out his desk	Tap him on the shoulder	Send him an email
Summary	A time-honored way to randomly corrupt your state and achieve undefined behavior	Interrupt at well-defined points and allow a handler chain (but target can't refuse or stop)	Interrupt at well-defined points and allow a handler chain, but request can be ignored	Target actively checks a flag – can be manual, or provided as part of #2 or #3
Pthreads	pthread_kill pthread_cancel (async)	pthread_cancel (deferred mode)	*n/a*	Manual
Java	Thread.destroy Thread.stop	*n/a*	Thread.interrupt	Manual, or Thread.interrupted
.NET	Thread.Abort	*n/a*	Thread.Interrupt	Manual, or Sleep(0)
C++0x	*n/a*	*n/a*	*n/a*	Manual
Guidance	**Avoid,** almost certain to corrupt transaction(s)	OK for languages without exceptions and unwinding...	Good, conveniently automated	Good, but requires more cooperative effort (can be a plus!)

Таблица 1: Основные возможности по завершению/прерыванию потоков.

Вариант 1: (ты не должен) «убивать»

Первый вариант, который почти всегда ошибочен, это «убить» нужный поток или задачу немедленно посреди любого совершаемого им действия. Эта форма безрассудного убийства доступна в API большинства платформ и фреймвёрков, включая достопочтенный `kill -9` в UNIX: функция `pthread_kill` (или `pthread_cancel` в асинхронном режиме) библиотеки Pthreads, `Thread.destroy` или `Thread.stop` в Java и `Thread.Abort` в .NET.

Каждая из ведущих платформ «переизобрела» эту ловушку, потому как это выглядит довольно простой идеей на первый взгляд, до тех пор пока ты не поймешь, что практически невозможно написать корректный код, исполнение которого может быть внезапно прервано в случайной, непредсказуемой точке.

Основная проблема с Вариантом 1 это то, что он является чрезвычайной мерой с чрезвычайными последствиями: редко встречается ситуация «убийства» только одного потока или задачи. Это действие чревато не только остановкой указанной работы, но также повреждением текущего процесса или других процессов. Есть шансы, что поток будет на пол пути к завершению операции перевода объекта или данных из одного допустимого состояния в другое. Например, данные могут быть частично записаны в буфер; или задача перевода денег может снять деньги с одного счёта, но ещё не положить их на заданный счёт. Теперь прибавьте сюда оптимизаторы в компиляторах, процессоры, подсистемы кэша, которые, как обычно, преобразуют ваш код и упорядоченно исполняют его, и вы, как правило, не имеете представления, просто прочитав исходный код, какие значения в памяти могут быть прочитаны или записаны, и в каком порядке, и, таким образом, вы не можете предсказать последствия прерывания этих действий в случайной точке.

«Убийство» потока или задачи посреди какой-либо работы обычно означает, что мы оставим за собой состояние, которое было повреждено обычно довольно случайным и непредсказуемым образом; и/или мы упустим ресурсы, принадлежавшие потоку или задаче, такие как все «локи», которыми они владели.

Представьте на минуту специальную ситуацию относительно «локов»: если «убитый» поток владел «локом», то это потому что он использовал (и, возможно, изменял) какие-то данные защищённые этим «локом». Убийство его в таком состоянии имеет два возможным последствия. Первое, на некоторых платформах, «лок» освободится, что сделает повреждённое состояние видимым для других частей программы. Второе, на других платформах, «лок» не освободится, что заблокирует другие части программы, которые уже ждали или в последствии попытаются подождать этого же самого «лока». Возможно это прозвучит неожиданно, но второе следствие, как правило, лучше, потому что, как минимум, оно не позволяет системе увидеть данные, оставленные в повреждённом состоянии. Конечно, лучше даже не взводить курок и не повреждать данные с самого начала.

Коротко: давайте остановим убийства. Вариант 1 почти всегда ошибочен, потому что он, вероятно, приведёт к повреждению, по меньшей мере, всего процесса, и может также повредить другие процессы — включая даже процессы на других машинах, если «убитый» поток был посреди выполнения некой важной операции ввода-вывода. В большинстве случаев, когда кто-то пытается использовать `pthread_kill`, `Thread.stop` или одну из их разновидностей, программист не подозревает о дополнительных затратах на которые он в действительности подписывается. Будьте осторожны и не используйте эту возможность, если только Вы действительно не хотите «положить» процесс или машину без всяких попыток аккуратной зачистки.

Есть два случая, когда Вариант 1 может быть приемлем, один редкий, другой очень редкий:

- Если Вы можете доказать, что данный поток занят исключительно чтением памяти и что он не владеет ресурсами, его «убийство» может быть безопасным.
- Если вы сознательно желаете завершить и перезапустить данный процесс (не только поток) и, возможно, даже целую текущую машину, не попытавшись зачистить поврежденное состояние, тогда «убийство» может быть допустимо. Например, в системе, которая использует три избыточных и независимых компьютера или процессора, которые не разделяют данных, когда один из них ведет себя неадекватно, может быть приемлемо «убить» и перезапустить его изолировано.

Лирическое отступление: точки завершения/прерывания

В отличие от Варианта 1, все три оставшиеся альтернативы разделяют один существенный общий пункт: заданный поток или задача может быть остановлен только во вполне

определённых точках исполнения, называемых «точками завершения» или «точками прерывания», которые чаще всего встречаются, когда поток блокирован при выполнении одного из следующих действий:

- Ожидает захвата мьютекса, получения семафора или другой синхронизации.
- Присоединяется (join) к другому потоку или задаче.
- «Спит» (sleep).

В Вариантах 2, 3 и 4 это и есть точки, в которых поток или задача могут быть прерваны. Это по-прежнему возлагает тяжёлую ношу на плечи автора кода потока или задачи: код должен быть готов к прерыванию в таких точках и, в особенности, вам необходимо либо восстановить инварианты перед выполнением одного из таких вызовов, когда вы можете быть прерваны, либо подготовить восстановление инвариантов когда вы уже прерваны. Давайте посмотрим, на что же это похоже, рассмотрев оставшиеся три Варианта.

Вариант 2: безапелляционный, нет права на отказ.

Вариант 2 состоит в том, чтобы следовать модели POSIX threads (Pthreads) отложенного завершения, осуществляемой `pthread_cancel`: ждать пока заданный поток достигнет следующей вполне определённой точки завершения, остановить его и запустить цепочку обработчиков завершения (если таковые есть), которые программа установила и которые служат схожей цели, что и деструкторы/функции разрушения (destructor/dispose functions) в современных языках. Это намного лучше, чем Вариант 1.

Основной недостаток Варианта 2 в том, что запросы на завершение не могут быть проигнорированы или отловлены; у целевого потока нет выбора кроме как быть остановленным в следующей точке завершения, и если процедура завершения была запущена, то остановить ее нельзя. Это разумный дизайн для языка, в котором нет исключений или объектов с деструкторами/функциями разрушения (обработчики завершения имитируют последние), но это по большей части не подходит для современных языков, которые обладают исключениями и знают, как отлавливать и восстанавливаться после ошибок и продолжать корректное исполнение. Таким образом, Вариант 2 подходит для таких языков как C и Fortran, если счесть приемлемой «насильную» остановку потока, но не вполне удовлетворителен при использовании современных языков, которые снабжены более изощрёнными механизмами восстановления после ошибок, или в случаях, когда ваши потоки или задачи могут обоснованно хотеть обработать запросы на завершение и продолжить работу или вообще игнорировать их — ни то, ни другое не допустимо в рамках Варианта 2.

Вариант 3: попросить вежливо

Вариант 3 — следовать модели прерывания, общей для современных языков и фреймвёрков, включающих Java и .NET, которые обозначены как `Thread.interrupt` и `Thread.Interrupt`, соответственно. Как и в Варианте 2, заданный поток продолжает исполнение до тех пор, пока он не достигнет следующей точки прерывания, в которой, как это происходит в большинстве систем, реализующих Вариант 3, прерывание заявляет о себе при помощи выброса исключения из вызова wait/join/sleep. То есть, в отличие от Варианта 2, указанный поток может отловить и обработать такое исключение также как и любое другое исключение, включая некоторые дополнительные возможности.

Также как и в Pthreads, поток может просто вначале позволить деструкторам/разрушителям (disposers) и предложениям finally полностью раскрутить стек, а потом завершиться. В отличие от Pthreads, он может решить раскрутить стек частично, до тех пор, пока не будет найден обработчик, который поймает и обработает исключение, а затем продолжит нормальное выполнение. Также в отличие от Pthreads, заданный поток может немедленно поймать и полностью проигнорировать исключение.

Это вежливое прерывание, оно общепринято в современных автоматизированных средствах работы с прерываниями.

Вариант 4: сотрудничать

Наконец, Вариант 4, который вы можете и должны использовать вместе с Вариантом 3, это модель, полностью основанная на сотрудничестве, когда заданный поток может проверить, не просил ли его кто-нибудь прервать работу. Эта проверка может выполняться между точками завершения (если вы хотите использовать Варианты 3 и 4 вместе) или вместо точек завершения (если вы хотите использовать только Вариант 4). Мы видели Вариант 4 в действии в двух предыдущих статьях [1,2]: при простом параллельном поиске, когда один «рабочий» нашёл ответ и записал его в разделяемую область [памяти], другие «рабочие» могут время от времени проверять эту область и останавливать свою работу, увидев что кто-то другой нашёл ответ.

Как насчёт библиотечных/системных вызовов?

Что делать с библиотечными вызовами, которые не рассчитаны на то, чтобы быть прерванными. Если с вами не сотрудничают, то так тому и быть. Только на стреляйте! Грубая сила это не выход.

Что делать, если вам нужно вызвать функцию операционной системы (возможно, в режиме ядра), которая не может быть прервана. Ответ всё тот же: если с вами не сотрудничают, так тому и быть. Не стреляйте! Кстати, вы могли заметить тенденцию последнего времени: современные ОС — на пути к тому, чтобы сделать все вызовы прерываемыми. Например, в Windows Vista почти все API файловой системы и системы ввода-вывода поддерживают прерывания, таким образом вы можете остановить их, не дожидаясь завершения их работы. Это не должно удивлять, после того как мы стали учитывать важность прерывания параллельного кода.

Итог

Прерывайте вежливо. Всегда используйте Варианты 3 и 4, которые позволяют потоку или задаче принять участие в принятии решения о том, должен ли он и как зачистить свою работу и/или продолжить её. Оповещайте поток о запросах на прерывание только в строго определённых предсказуемых wait/join/sleep-точках и убедитесь в том, что пишите код который будет безопасным, если прерывание произойдёт в одной из этих точек. Заметьте, что Варианты 3 и 4 предоставляют строгое надмножество возможностей, предоставляемых Вариантом 2: все, что вы можете запрограммировать выбрав Вариант 2, вы также можете получить, используя Варианты 3 или 4. Избегайте безоговорочного Варианта 2 не давать потоку возможности принять участие в принятии решения. Даже если вы используете Pthreads, которая не поддерживает Вариант 3, у вас есть вариант написать Вариант 4 самостоятельно.

Наконец, никогда не «убивайте» поток или задачу, как в Варианте 1, если только вы не можете доказать, что вы действительно попали в один из редких случаев, когда эта сомнительная практика безопасна и «положить» целый процесс (или больше) без мало-мальски аккуратной зачистки не страшно. В реальном мире большая часть попыток «убить» процесс в случайном месте непростительны; каждая из основных поточных платформ или окружений начинали с этого, но теперь мы знаем наверняка — грубая сила это не выход.

Библиография

[1] H. Sutter. "Going Superlinear" (Dr. Dobb's Journal, март 2008).

[2] H. Sutter. "Super Linearity and the Bigger Machine" (Dr. Dobb's Journal, март 2008).