Кейс №3

🔑 Объяснение

Основной поток `main`, с которого начинается исполнение метода `CareTipsProvider#composeCareTips`, помещает в пул `careThreadPool` столько задач `OwnerCareTask`, сколько владельцев питомцев заведено в базе данных (из коробки 10, см. таблицу `owners`). Поскольку пул создаётся с числом потоков, равным числу доступных ядер процессора, может так статься, что число задач будет больше, чем пул может взять в работу одновременно. В целом, это абсолютно нормальная ситуация, и на этот случай в пуле `Executors.newFixedThreadPool` предусмотрена очередь, в которую попадают все задачи, не успевшие сразу встать на исполнение. Но есть одно “но”.

Исполняемые задачи `OwnerCareTask`, в свою очередь, тоже помещают в пул свои дочерние задачи `PetCareTask` в количестве, равном числу питомцев у каждого владельца (как правило, 1, реже 2).

Это приводит к тому, что вскоре после запуска пул оказывается заполнен задачами `OwnerCareTask`, каждая из которых ждёт возможности поместить в пул свою одну или две задачи `PetCareTask`, но не может этого сделать из-за того, что пул переполнен.

Таким образом, пул не может освободиться, потому что задачи в нём не завершены, а задачи не завершаются, потому что им нужно место в пуле.

Инструменты анализа не обнаруживают этот, казалось бы, простой deadlock, потому что участвующие в нём потоки находятся в состоянии `WAITING`, вызванном методом `Unsafe.park` при обращении к `Future.get`, который в свою очередь вызывается из `ExecutorService.invokeAll`:

```
"pool-2-thread-3" #197 [228694] prio=5 os_prio=0 cpu=0.77ms elapsed=13.17s
tid=0x00007aee5a5770c0 nid=228694 waiting on condition  [0x00007aee2f1fe000]
   java.lang.Thread.State: WAITING (parking)
  at jdk.internal.misc.Unsafe.park(java.base@21.0.3/Native Method)
  - parking to wait for  <0x0000000619166668> (a
java.util.concurrent.FutureTask)
  at
java.util.concurrent.locks.LockSupport.park(java.base@21.0.3/LockSupport.java:2
21)
  at
java.util.concurrent.FutureTask.awaitDone(java.base@21.0.3/FutureTask.java:500)
  at java.util.concurrent.FutureTask.get(java.base@21.0.3/FutureTask.java:190)
  at
java.util.concurrent.AbstractExecutorService.invokeAll(java.base@21.0.3/Abstrac
tExecutorService.java:252)
  at
org.springframework.samples.petclinic.service.care.OwnerCareTask.call(OwnerCare
Task.java:39)
...
```

Это пример потока из пула. Главный же поток `main` имеет стек со схожей верхушкой, потому что тоже зовёт `invokeAll`:

```
"main" #1 [228456] prio=5 os_prio=0 cpu=4416.50ms elapsed=17.80s
tid=0x00007aee5802d060 nid=228456 waiting on condition  [0x00007aee5ed18000]
   java.lang.Thread.State: WAITING (parking)
  at jdk.internal.misc.Unsafe.park(java.base@21.0.3/Native Method)
  - parking to wait for  <0x00000006190bccc0> (a
java.util.concurrent.FutureTask)
  at
java.util.concurrent.locks.LockSupport.park(java.base@21.0.3/LockSupport.java:2
21)
  at
java.util.concurrent.FutureTask.awaitDone(java.base@21.0.3/FutureTask.java:500)
  at java.util.concurrent.FutureTask.get(java.base@21.0.3/FutureTask.java:190)
  at
java.util.concurrent.AbstractExecutorService.invokeAll(java.base@21.0.3/Abstrac
tExecutorService.java:252)
  at
org.springframework.samples.petclinic.service.care.CareTipsProvider.composeCare
Tips(CareTipsProvider.java:46)
...
```

Точно также потоки выглядят, когда просто стоят в пуле без дела, в ожидании, пока их кто-нибудь задействует.

Проблема воспроизводится до тех пор, пока число задач меньше или равно числу ядер процессора. Как только ядер становится хотя бы на одно больше, клубок разматывается. Правда, чем меньше разница между ядрами и родительскими задачами, тем медленнее выполняются дочерние задачи. В пределе, когда разница составляет 1 (задач `OwnerCareTask` 10, а ядер 11), все задачи `PetCareTask` по началу вынуждены исполняться всего одним ядром, т.е. по сути не распараллеливаться.

✅ Варианты решения

- Использовать два разных пула потоков для `OwnerCareTask` и `PetCareTask`.
- ~~Создавать `ThreadPoolExecutor` вручную, чтобы задать переменное число потоков параметрами `core` & `max`:~~
- Отправлять задачи на исполнение не все разом методом `invokeAll`, а по одной, проверяя занятость пула. Не вместившиеся задачи, например, исполнять текущим потоком.
- Сравнивать в начале число задач и число доступных потоков в пуле; при превалировании числа задач отправлять их на исполнение пачками, а не все сразу.
- *(ваш вариант)*