Непрерывное профилирование в облаке с помощью eBPF

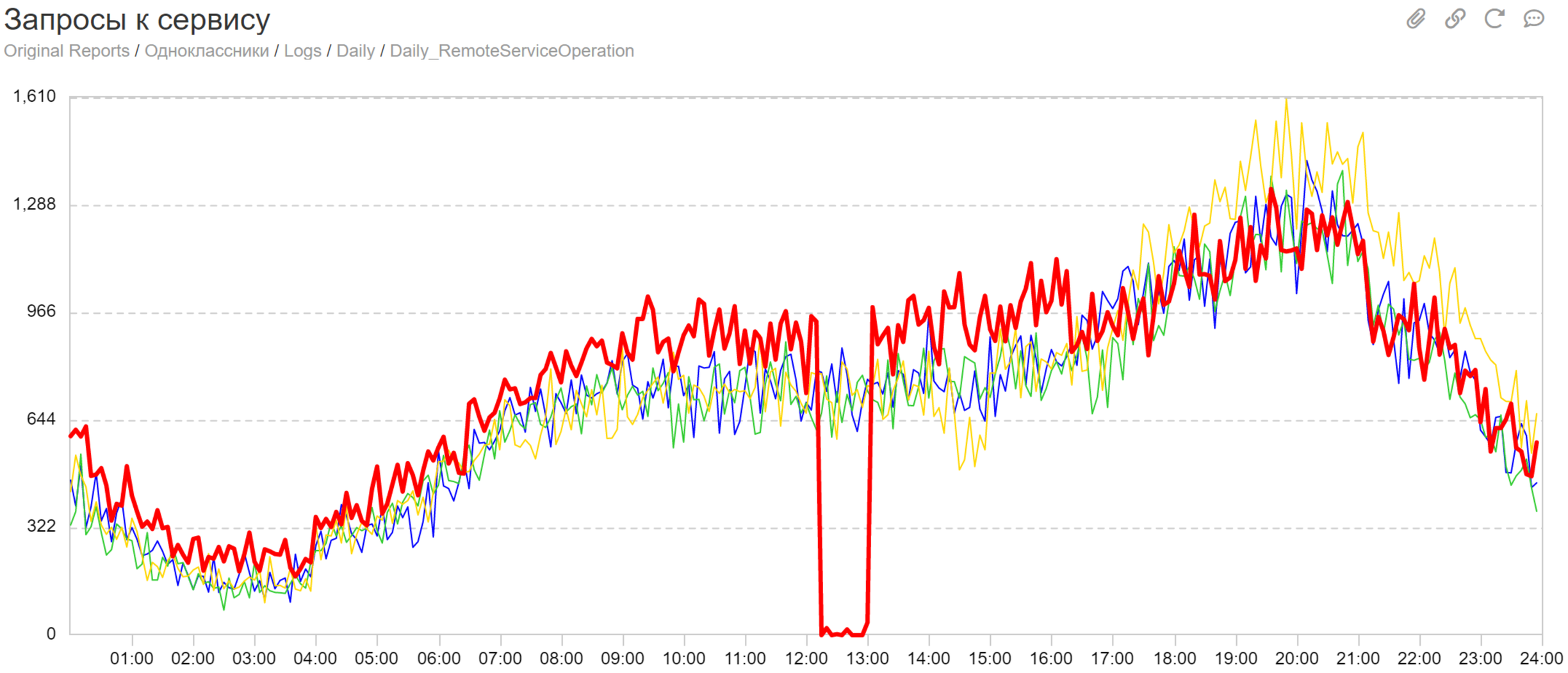
В Одноклассниках

(включая VK Видео, VK Звонки, RuStore)

- 10K серверов в 7 ЦОД
- 50K контейнеров в one-cloud*
- 1000 PB данных
- 4 Tbps
- до 100K rps на контейнер

* https://www.youtube.com/watch?v=eJShMFzDV6g

Мониторинг


Запросы к сервису
Original Reports / Одноклассники / Logs / Daily / Daily_RemoteServiceOperation
1,610
1,288
966
644
322
0
01:00
02:00
03:00
04:00
05:00
06:00
07:00
08:00
09:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
21:00
22:00
23:00
24:00

DC

Search

in DC

Event	Output	Duration	Interval			Skip	Min%			
cpu	flamegraph	30		☐ Split	☐ Reverse	0	0	Extra start options	Extra stop options	PROFILE

DC

Search in DC

Event: cpu | Output: flamegraph | Duration: 30 | Interval | Split | Reverse | Skip: 0 | Min%: 0 | Extra start options | Extra stop options | PROFILE

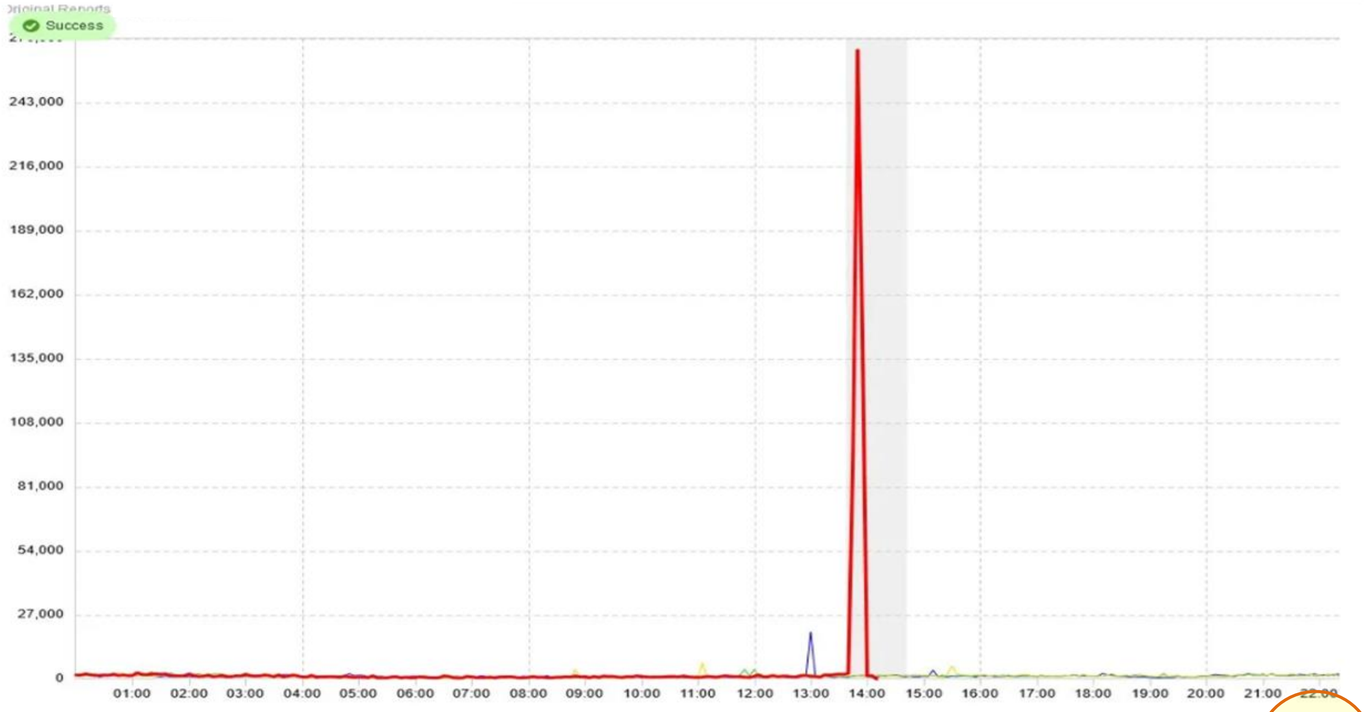
org/apache/cassandra/concurrent/ThreadWithPriority.run:84
java/lang/Thread.run:748
java/util/concurrent/ThreadPoolExecutor$Worker.run:624
java/util/.. java/util/concurrent/ThreadPoolExecutor.runWorker:1149
java/util/.. org/apache/cassandra/net/MessageDeliveryTask.run:60
java/ut.. org/apache/cassandra/service/RangeSliceVerbHandler.doVerb:51
java/u.. org/apache/cassandra/db/RangeSliceCommand.executeLocally:146
java/u.. org/apache/cassandra/db/ColumnFamilyStore.getRangeSlice:1969
sun/mi.. org/ap.. o.. org/apache/cassandra/db/ColumnFamilyStore.getSequentialIterator:1844
Unsafe.. org/ap.. o.. org/apache/cassandra/db/RowIteratorFactory.getIterator:62 org/apache/cassandra/db/RowI..
__pth.. org/ap.. o.. org/apache/cassandra/io/sstable/SSTableReader.getScanner:1276 org/apache/cassandra/db/RowI..
entry.. org/ap.. o.. org/apache/cassandra/io/sstable/SSTableReader.getScanner:1290 org/apache/cassandra/db/Row..
do_sy.. org/ap.. j.. org/apache/cassandra/io/sstable/SSTableScanner.<init>:70 org/apache/cassandra/io/sstable/SSTabl.. org/apache/cassandra/utils/..
__x6.. org/ap.. j.. org/apache/cassandra/io/sstable/SSTableReader.openDataReader:1538 org/apache/cassandra/io/sstable/SSTabl.. org/apache/cassandra/utils..
do_f.. org/a.. j.. org/apache/cassandra/io/compress/CompressedRandomAccessReader.op.. org/apache/cassandra/io/util/RandomA.. org/apache/cassandra/utils..
fute.. or.. o.. org/apache/cassandra/io/compress/Compress.. org/apache/cassandra.. org/apache/cassandra/io/util/RandomA.. org/apache/cassandra/io/ss..
fut.. or.. j.. org/apache/ca.. or.. org/apache/cassan.. o.. org/apach.. o.. org/apache/cassan.. o.. com/google/common/collect/..
sch.. ja.. j.. java/io/Rando.. ja.. java/io/R.. j.. com/google/common/collect/..
__s.. j.. j.. java/io/Rand.. su.. java/io/R.. s.. org/apache/cassandra/io/ss..
j.. j.. java/io/Rand.. Op.. java/io/R.. O.. org/apache/cassandra/io/s..
j.. j.. fileOpen fileOpen org/apache/cassandra/io/s..
j.. __open __open org/apache.. org/apache/..
_.. entry_S.. entry.. org/apache.. org/apache..
e.. do_sysc.. do_sy.. org/a.. org/apache..
d.. __x64.. __x6.. java/.. o.. or..
do_sy.. do_s.. java/.. s..
do_sy.. do_s.. rea..
do_f.. do.. read
pat.. pa.. en..
do..

чем процесс занят сейчас

-=РАЗБОР ИНЦИДЕНТОВ=-
24 из 40 в сети

Мониторщик

Чёт ошибки скакнули: https://charts.****.io/reports?link=123456
Глянь пжлст



14:15

Не понятно... пока расследовал, само прошло 16:46

Напишите сообщение...

Проблема

- Чем процесс был занят во время инцидента?

Решение

- Непрерывное профилирование
- Всегда и везде
- Исторические данные

async-profiler

https://github.com/jvm-profiling-tools/async-profiler/

Releases / v2.6

Continuous profiling

apangin released this 10 Jan 2022 · 69 commits to master since this release v2.6 ce91abe

async-profiler

https://github.com/jvm-profiling-tools/async-profiler/

Releases / v2.6

Continuous profiling

apangin released this 10 Jan 2022 · 69 commits to master since this release v2.6 ce91abe

```
$ profiler.sh --loop 1m -f /var/log/profile-%t.html <pid>
```

```
Jun 10  14:18  profile-20220610-141800.html
Jun 10  14:19  profile-20220610-141900.html
Jun 10  14:20  profile-20220610-142000.html
Jun 10  14:21  profile-20220610-142100.html
...
```

Pyroscope Docs Blog

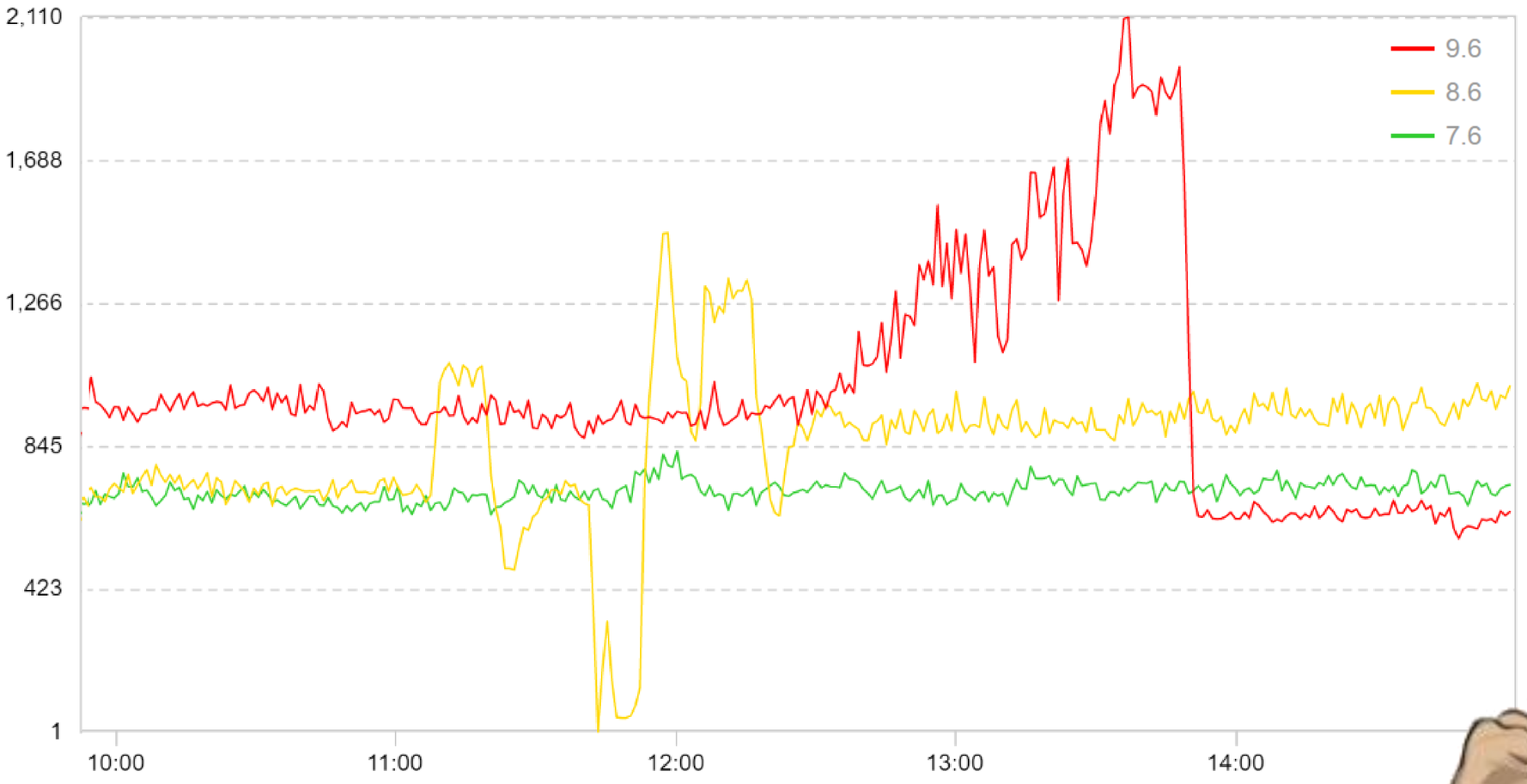
Open Source Continuous Profiling Platform

Java

Pyroscope uses async-profiler to collect profiling data from Java applications.

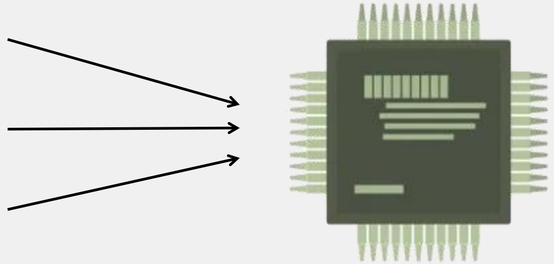
Включим везде async-profiler

System CPU utilization, millicores



Как работает async-profiler*

- perf счётчик на каждый поток
  - x сотни потоков
  - x до 50 контейнеров на сервер

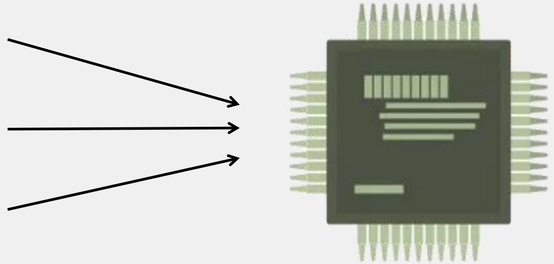


high system %cpu

* https://www.youtube.com/watch?v=QiGrTvsCZmA

Как работает async-profiler*

- perf счётчик на каждый поток
  - x сотни потоков
  - x до 50 контейнеров на сервер



high system %cpu

- файловый дескриптор на счётчик
- память для kernel стеков

  `kernel.perf_event_mlock_kb = 516 (64 потока)`

* https://www.youtube.com/watch?v=QiGrTvsCZmA

Per-thread счётчик

✖ Плохо масштабируется

Решение: per-CPU счётчики

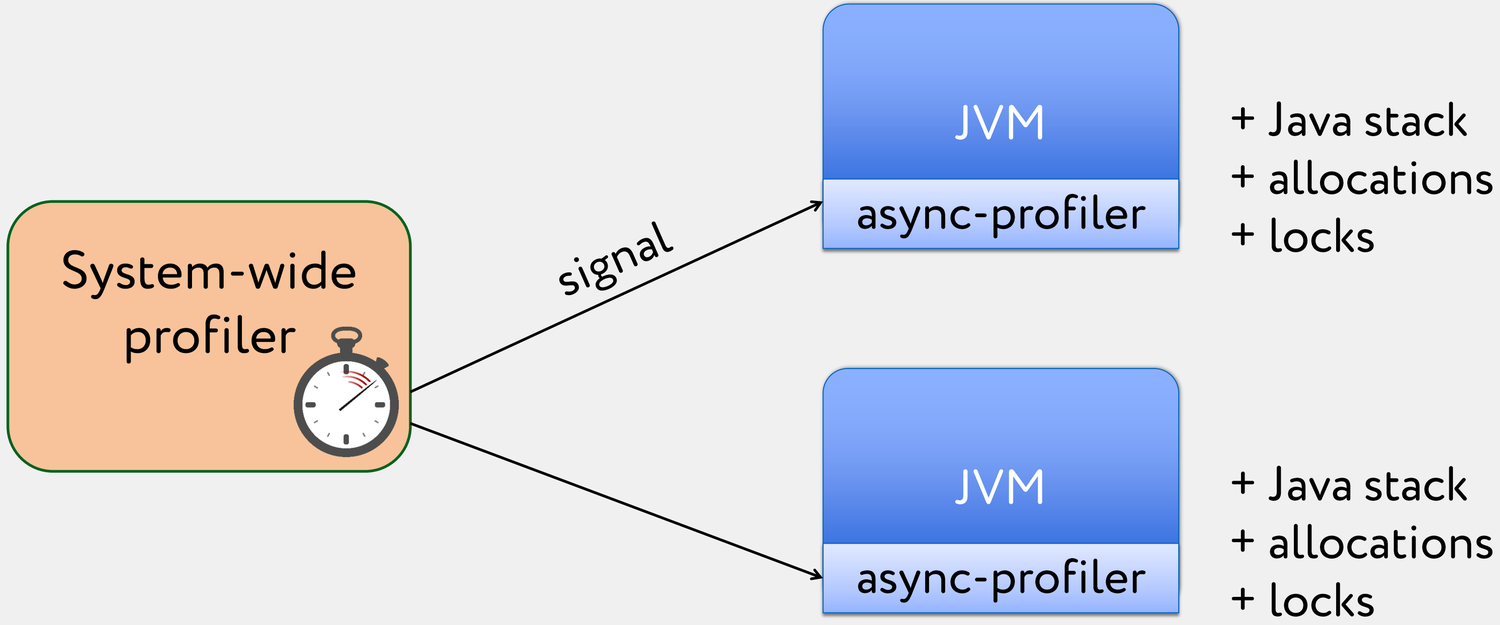
- No contention
- System wide

perf

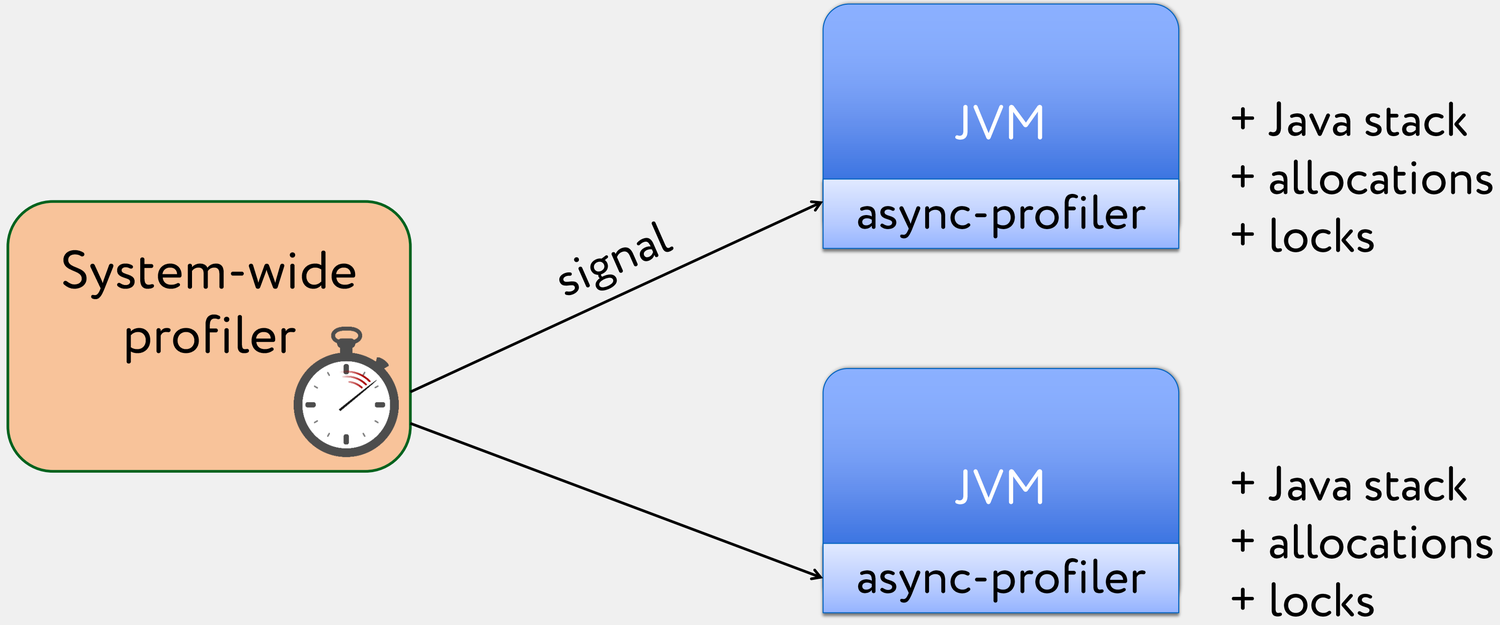
- ✚ System wide
- ✚ Low overhead

- ✖ Не Java-friendly
- ✖ -XX:+PreserveFramePointers (оверхед до 5%)
- ✖ Не показывает interpreted методы
- ✖ Случайные JVM краши (JDK-8173361)
- ✖ Нет профилирования аллокаций и локов

Интеграция System-wide profiler с Java



Интеграция System-wide profiler с Java

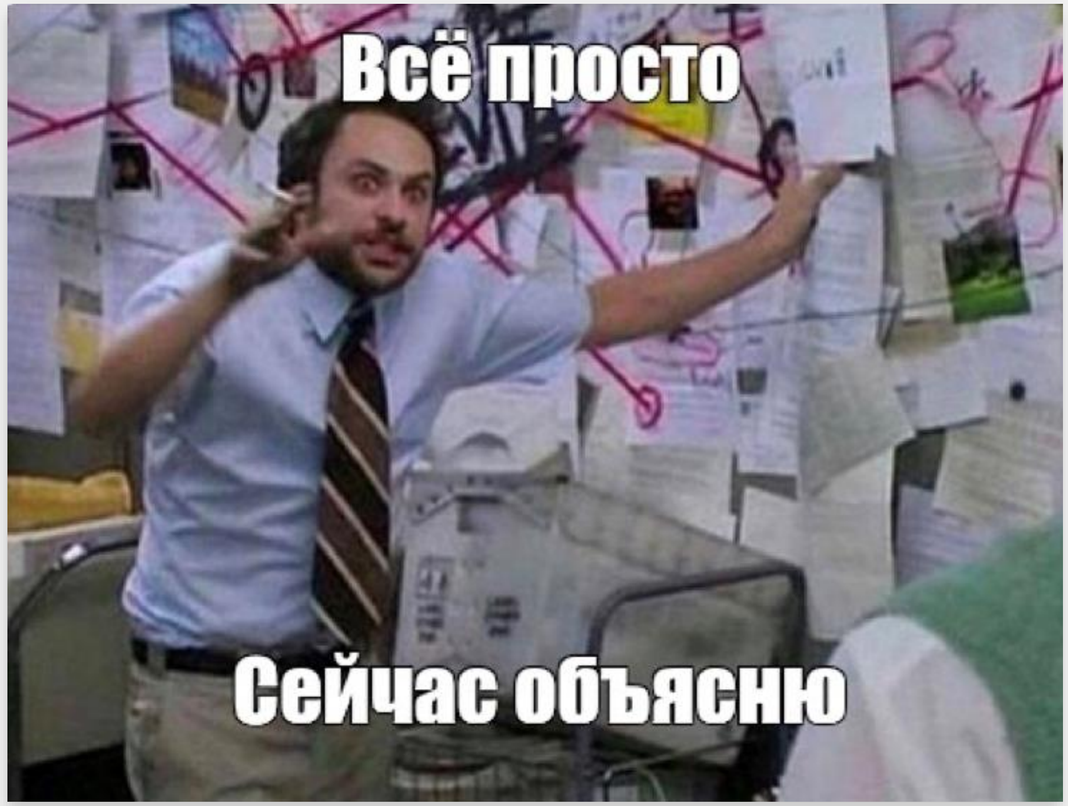


- Как послать сигнал от perf счётчика нужному процессу/потоку?

eBPF

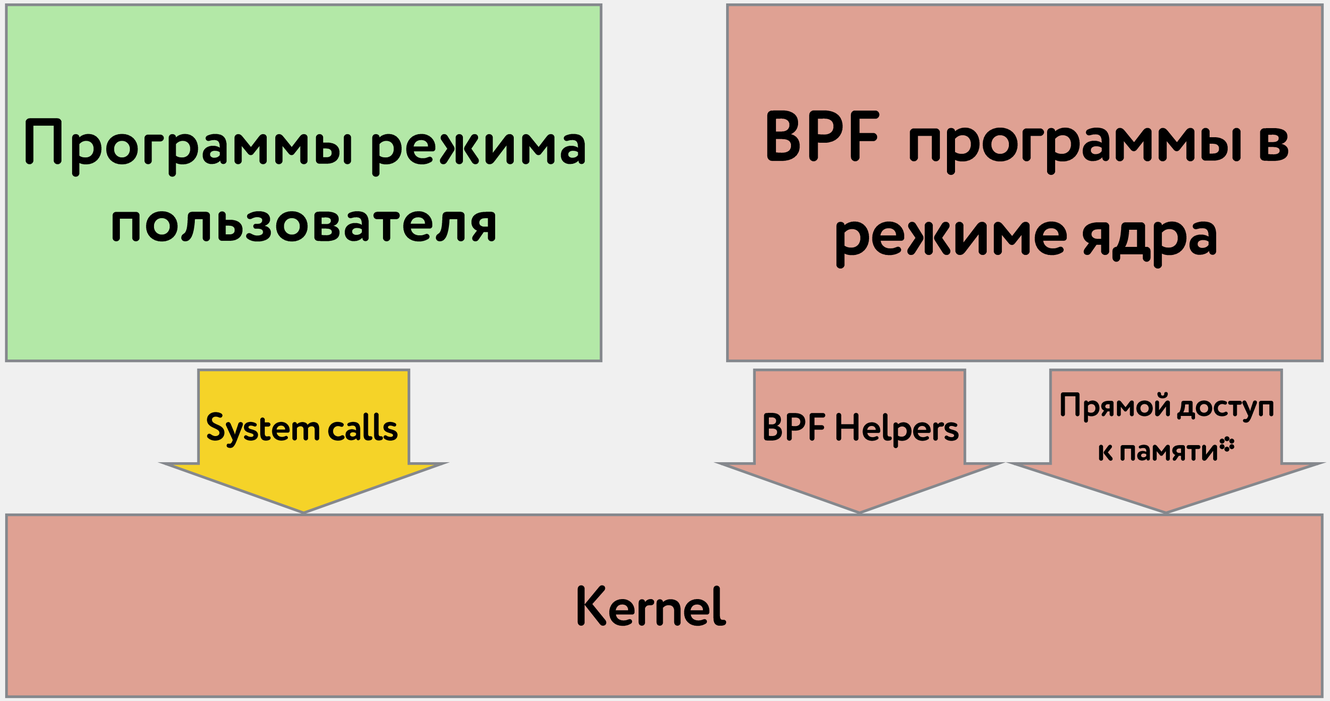
Краткое введение в eBPF (или просто BPF)

- BPF в 1992г – “BPF classic”
  - Berkeley Packet Filter
- С 2013г eBPF – ”Extended BPF”



eBPF https://ebpf.io/

BPF– микропрограммы в ядре Linux



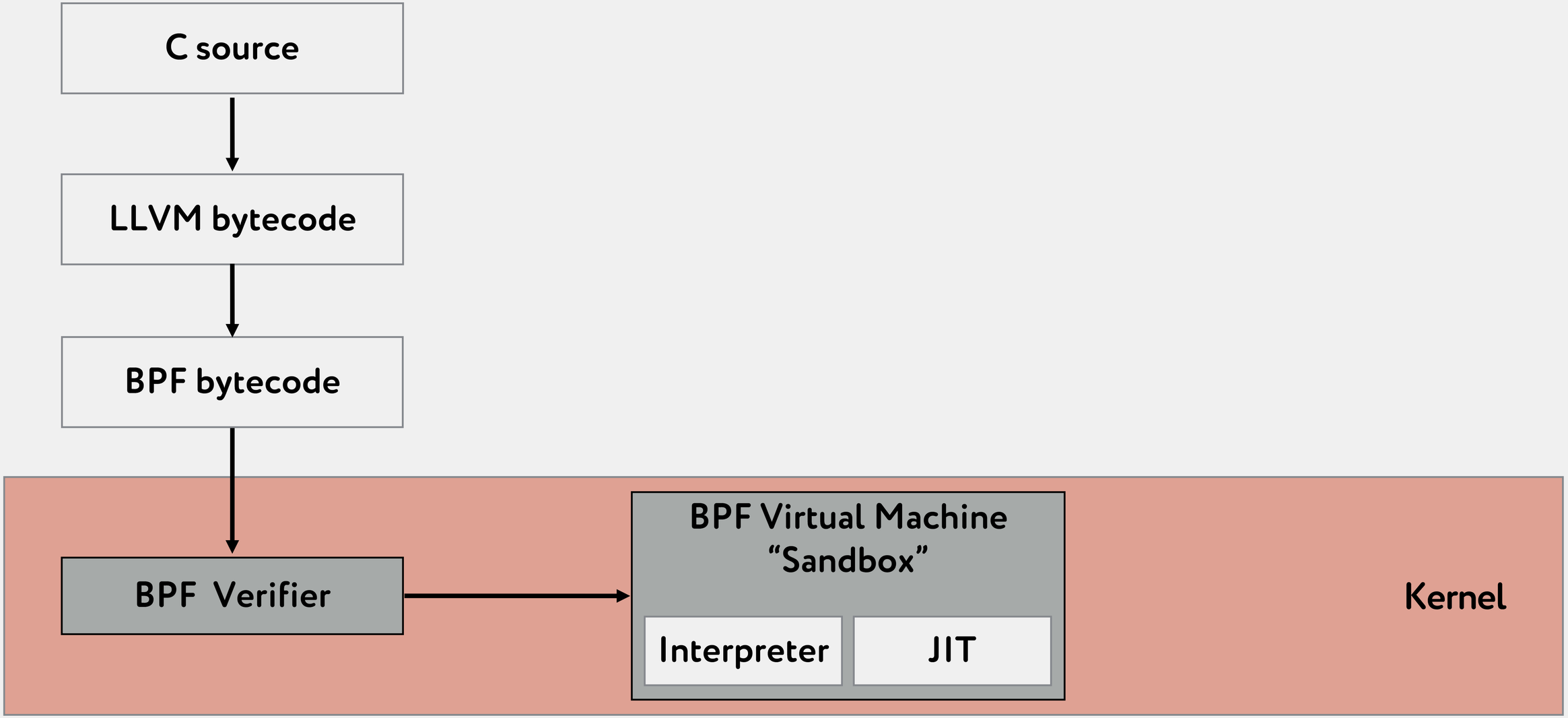
- **Быстродействие**
  - Syscalls: 100-200 наносекунд
  - BPF Helpers: прямой вызов
  - Прямой доступ к **некоторым** структурам ядра
- **Безопасность и удобство режима пользователя**
  - Верификация корректности
  - Динамическая загрузка

BPF в облаке Одноклассников one-cloud

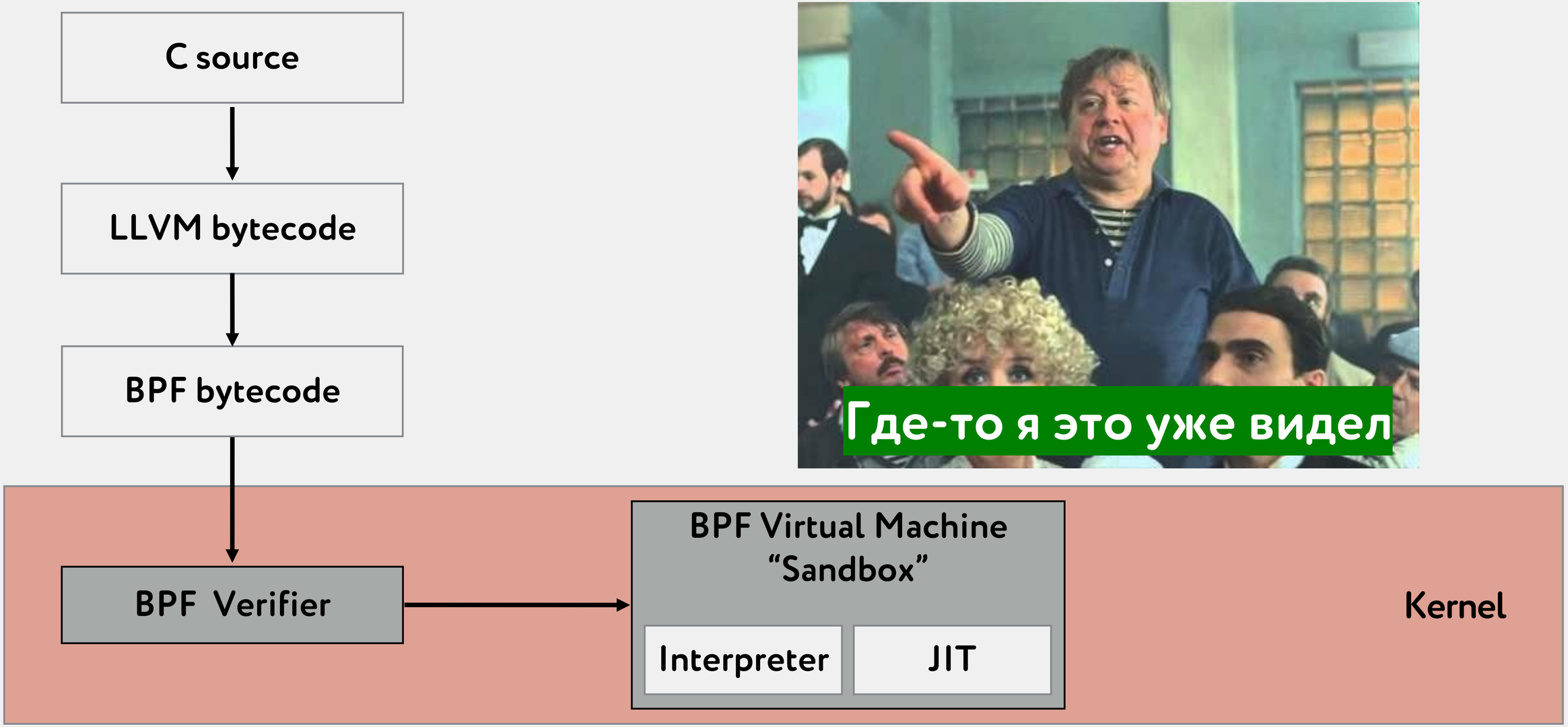
- **Шейпер сетевого трафика**
  https://habr.com/ru/company/odnoklassniki/blog/572206/

- **Непрерывный сбор perf метрик по контейнерам**
  instructions, cycles, cache-misses, cpu-migrations etc.

Жизненный цикл BPF программы



Жизненный цикл BPF программы



Возможности BPF

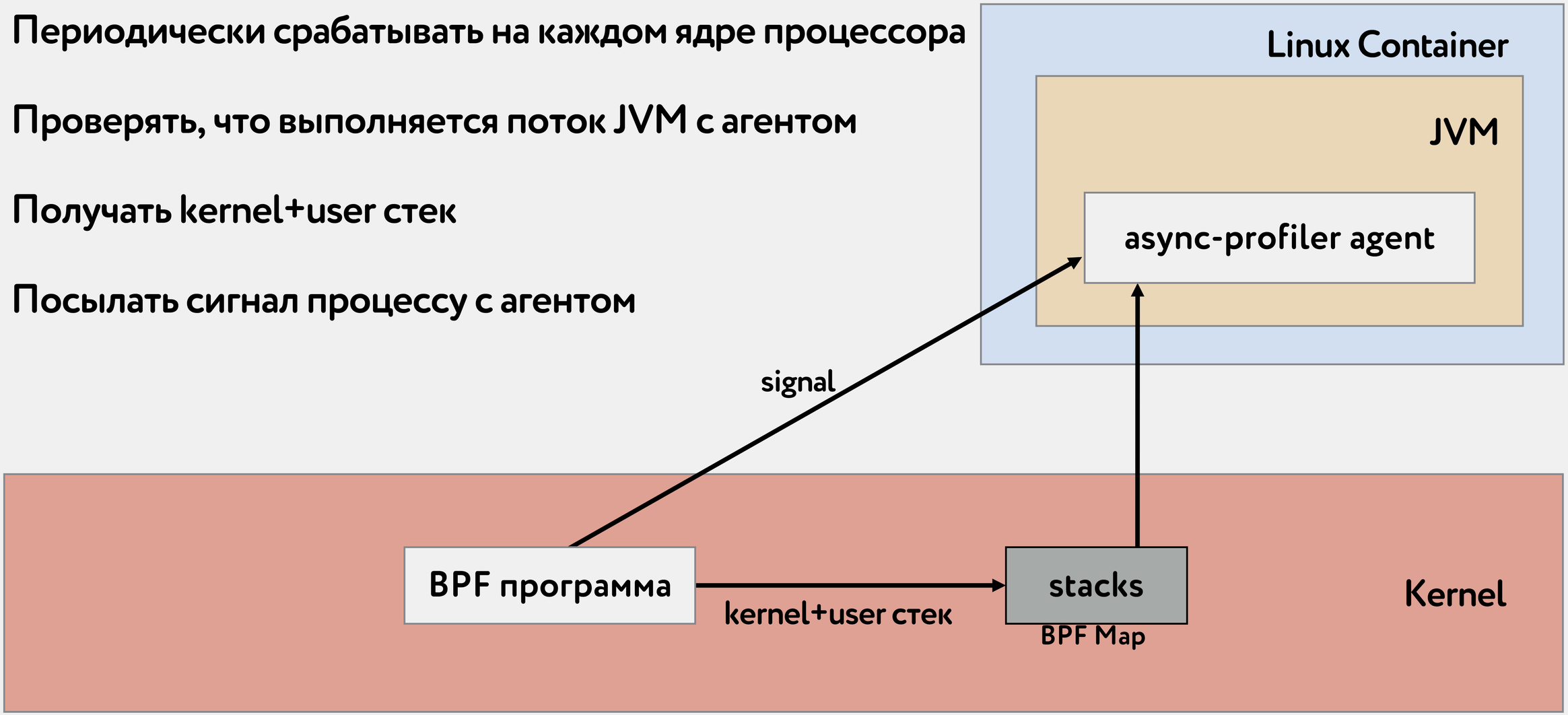
- **Привязка к событиям в Kernel (типы программ):**
  сеть, perf events, tracepoints, kprobes, uprobes …
- **Вспомогательные методы:** BPF Helpers
- **Хранение состояния:** BPF Maps
  - Хэш-таблицы, массивы, кольцевые буферы, …
  - Доступ из BPF и user-space, создание только из user-space
- **Возможности зависят от версии ядра Linux**
  Нам нужно 5.7 или выше
  https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

Ограничения BPF

- **Только «простые» микропрограммы**
  - Нет неограниченных циклов, динамической аллокации, потоков, ввода-вывода и т.п.
- **Ограничения на размер:** стека, числа инструкций, ветвлений
- **Статически доказуемая корректность**
  Верификатор может отвергать корректный код

Требования к BPF программе для профилирования

- Периодически срабатывать на каждом ядре процессора
- Проверять, что выполняется поток JVM с агентом
- Получать kernel+user стек
- Посылать сигнал процессу с агентом



BPF программа типа perf_event

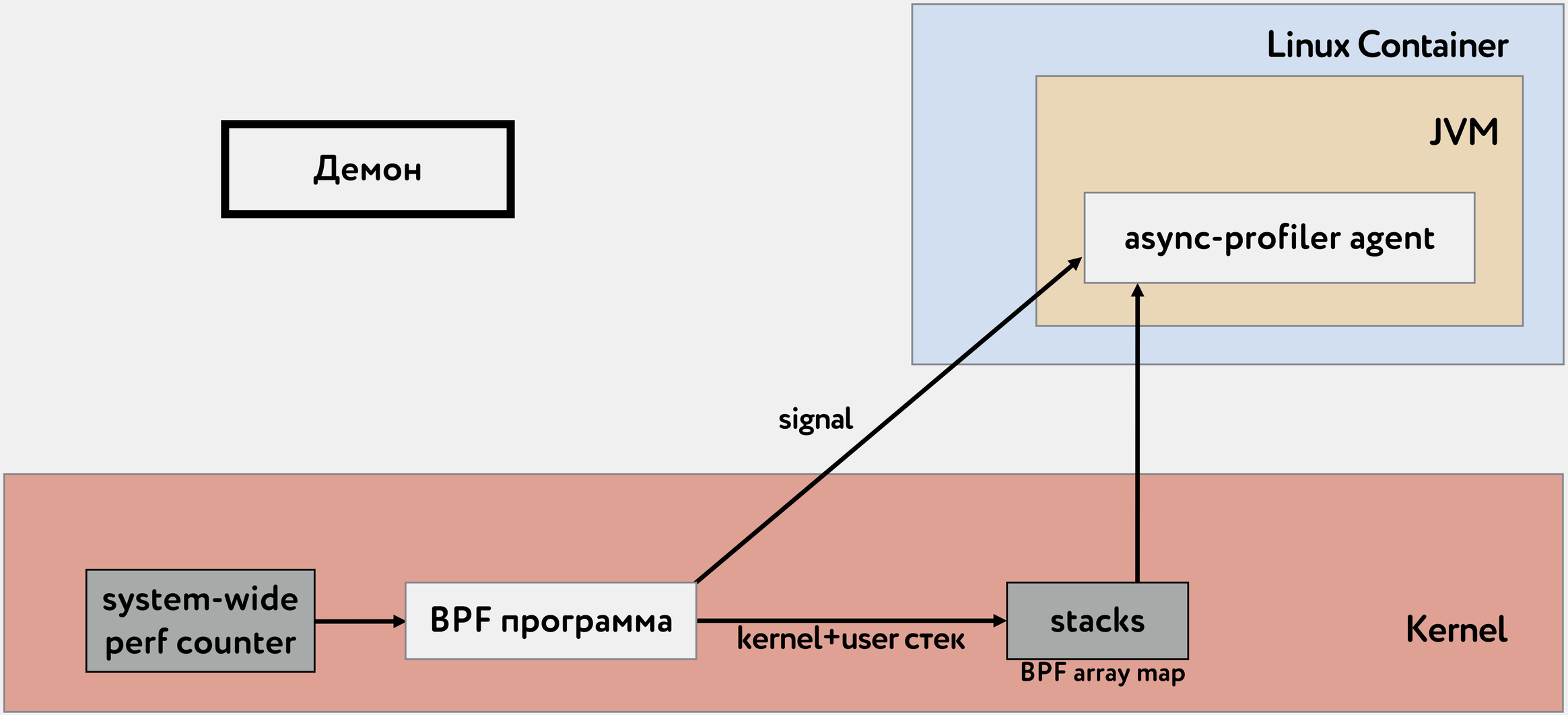
- Можно привязывать к perf счетчикам

```
// Точка входа
int do_perf_event(struct bpf_perf_event_data *ctx) {
…
}
```

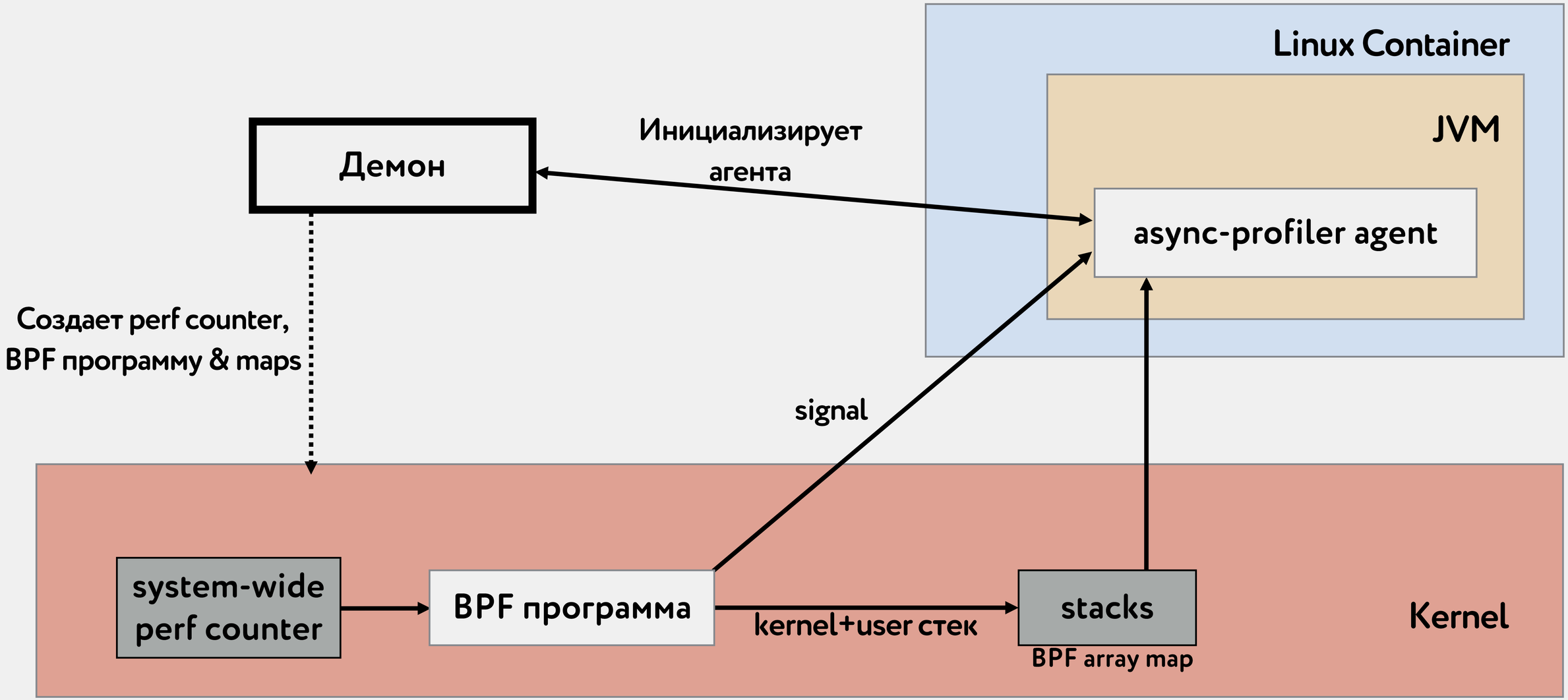
Нужные нам BPF Helpers уже есть

- **Проверить что выполняется поток процесса JVM с агентом**
  - **bpf_get_current_pid_tgid**: получить thread id + process id
  - **bpf_get_current_task**: получить struct task_info (tid, pid, sched_policy, …)
- **Получить user+kernel стек**
  - **bpf_get_stack**
- **Посылает сигнал процессу с агентом**
  - **bpf_send_signal_thread**

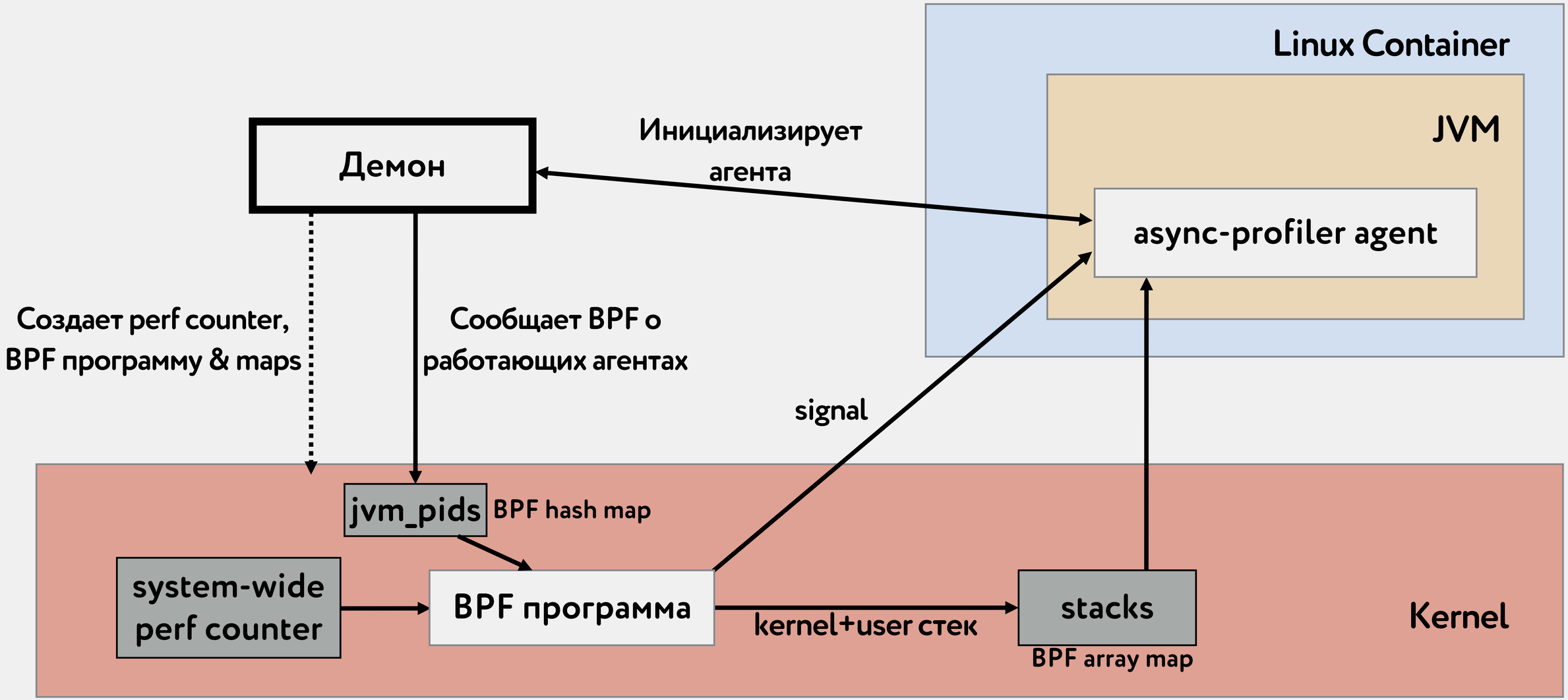
User-space демон: связующее звено между агентом и BPF



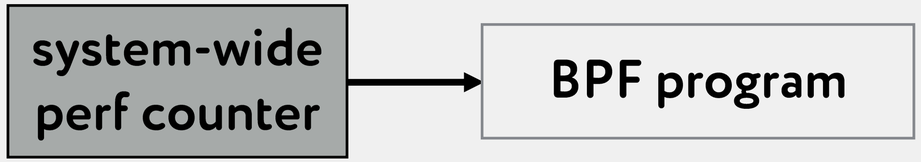
User-space демон: связующее звено между агентом и BPF



User-space демон: связующее звено между агентом и BPF



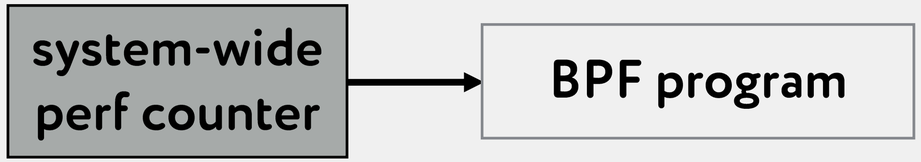
Демон: инициализация perf counter и BPF программы



Демон: инициализация perf counter и BPF программы

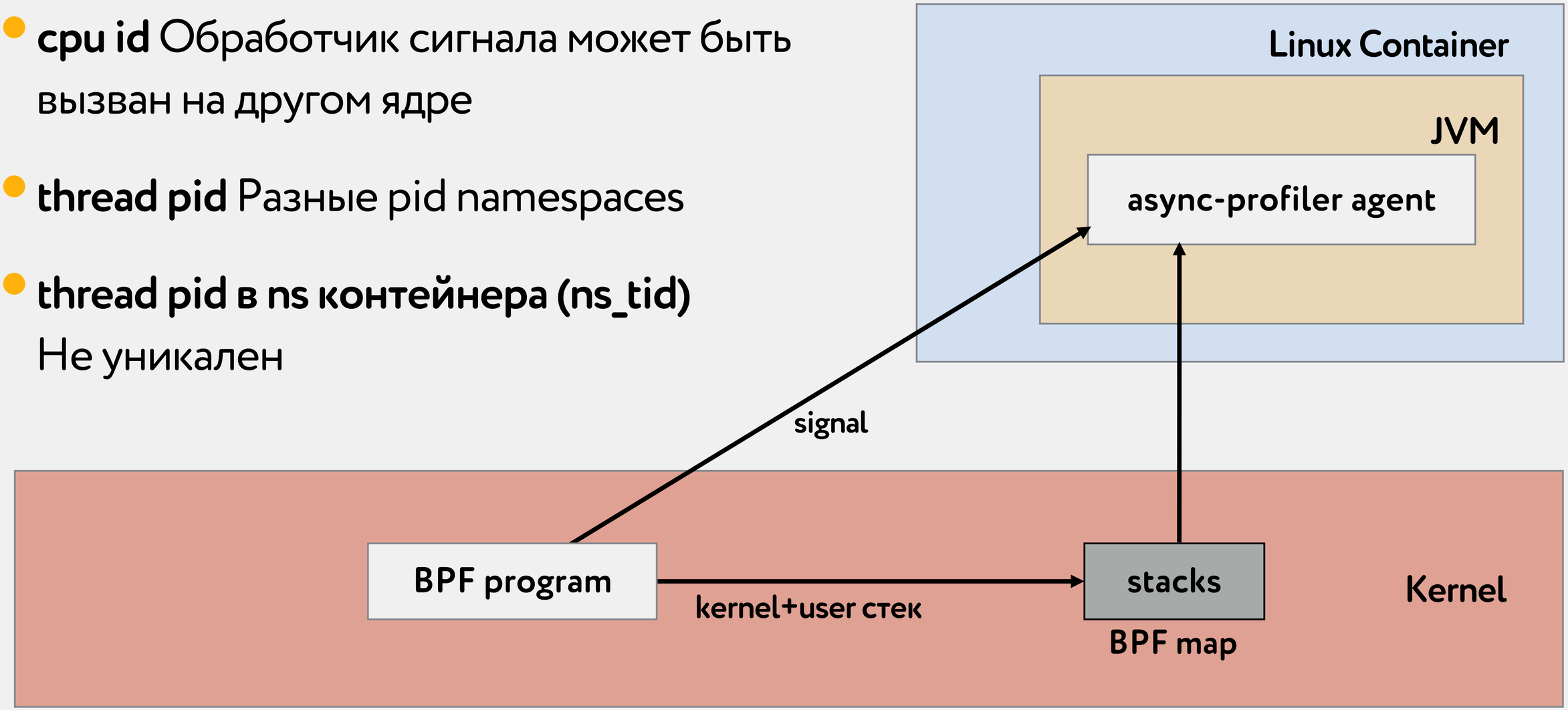
Прямо из Java! Используем one-nio https://github.com/odnoklassniki/one-nio

```
// Загружаем BPF программу и создаем BPF maps (требует libbpf)
var prog = BpfProg.load("profile.bpf.o", ProgType.PERF_EVENT);

//Создаем счетчик
var perfCounter = Perf.openGlobal(   // per-cpu
    PerfEvent.HW_REF_CPU_CYCLES,     // счетчик тактов процессора
    Perf.ANY_PID,                    // system-wide
    PerfOption.period(10_000_000L)); // событие раз в 10 млн тактов

// Привязываем BPF программу к событию perf счетчика
prog.attach(perfCounter);

// Запускаем
perfCounter.enable();
```



Какой ключ использовать в BPF Map stacks?

- **cpu id** Обработчик сигнала может быть вызван на другом ядре
- **thread pid** Разные pid namespaces
- **thread pid в ns контейнера (ns_tid)** Не уникален

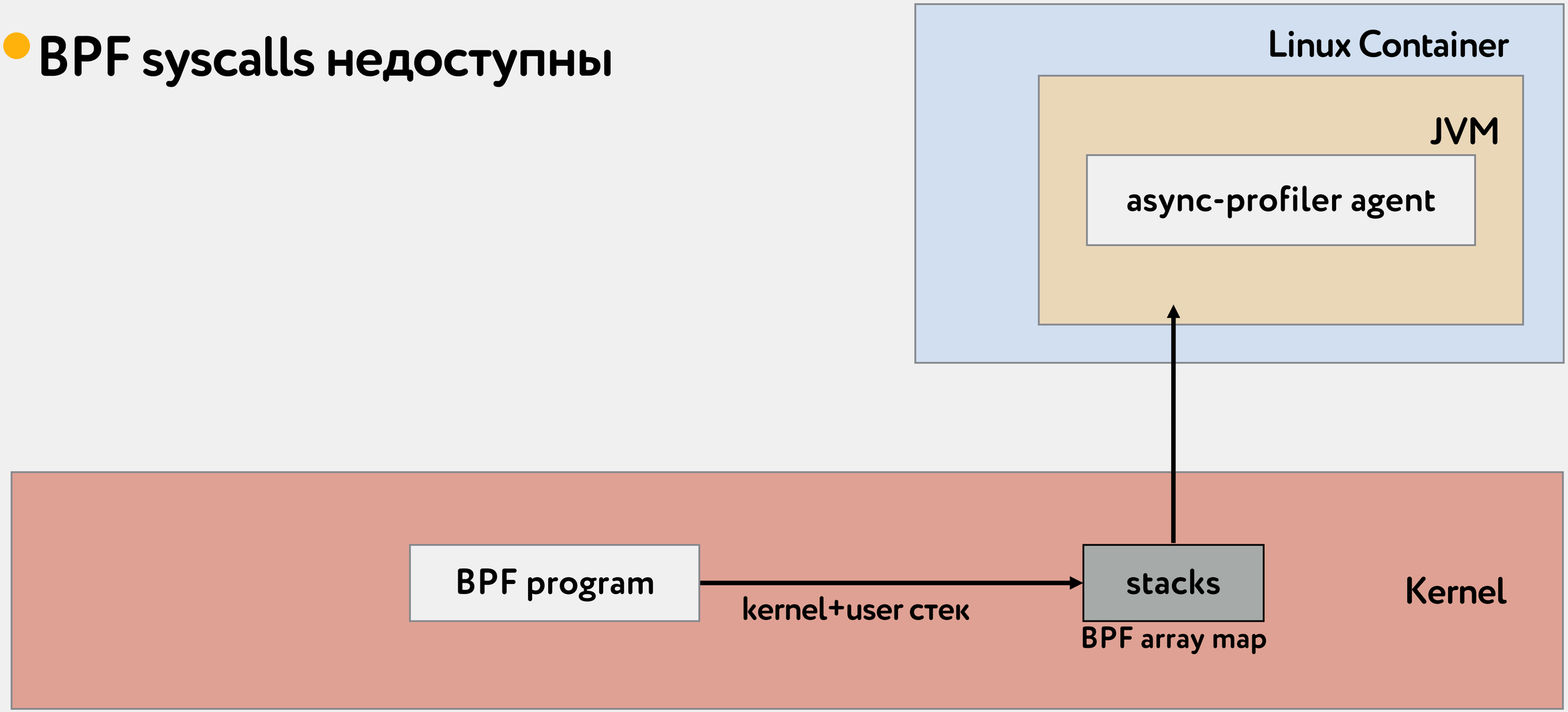


Хэш-таблица для передачи стеков

- Каждый элемент – буфер 1024 байт
- Индекс элемента вычисляется и в агенте, и в BPF:
  `index = (ns_tid + salt) % length`
- Свой `salt` для каждого агента
- Получаем ns_tid :
  `bpf_get_ns_current_pid_tgid` (kernel **5.7** и выше)
- Возможны коллизии
  Размер: 256K элементов * 1024 байт = 256Мб

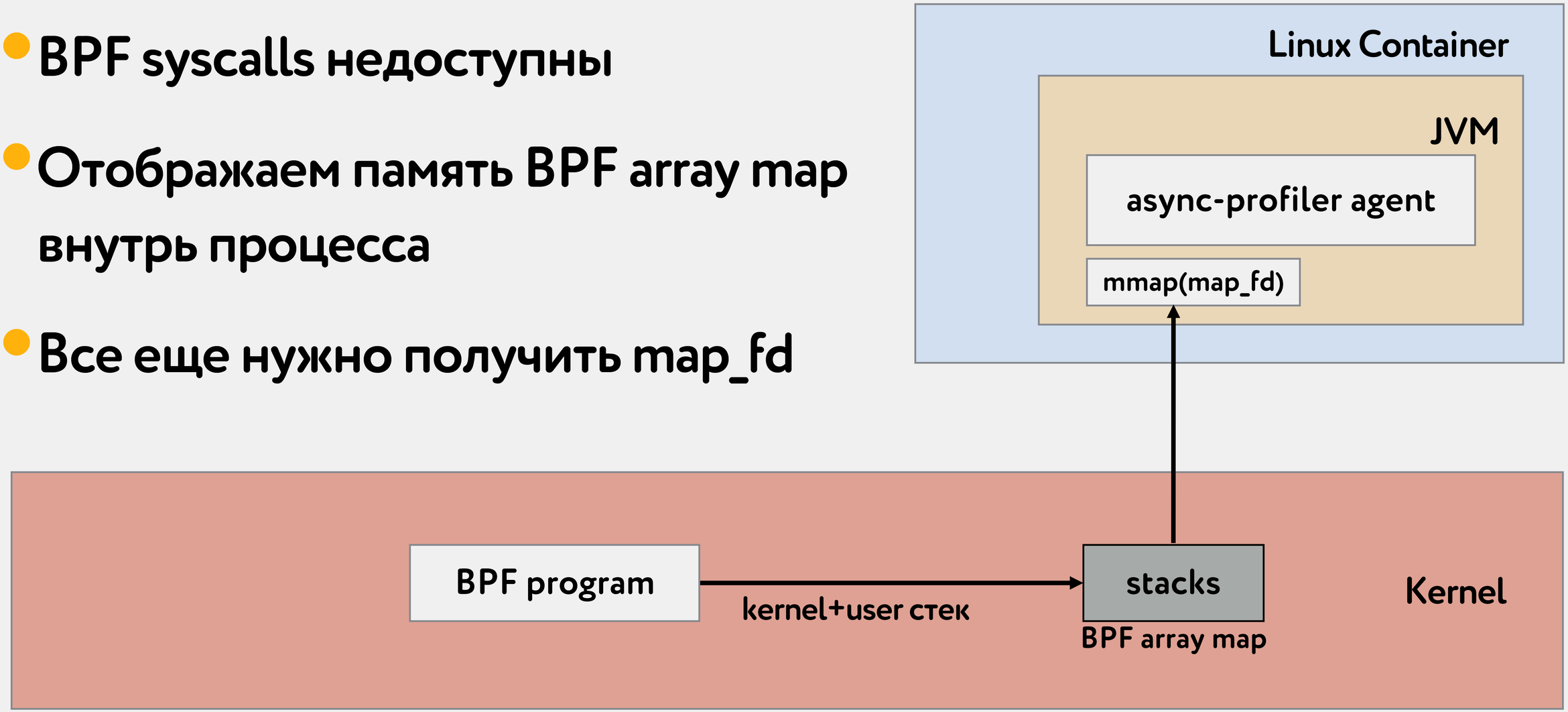
Доступ к BPF maps из контейнера

- BPF syscalls недоступны



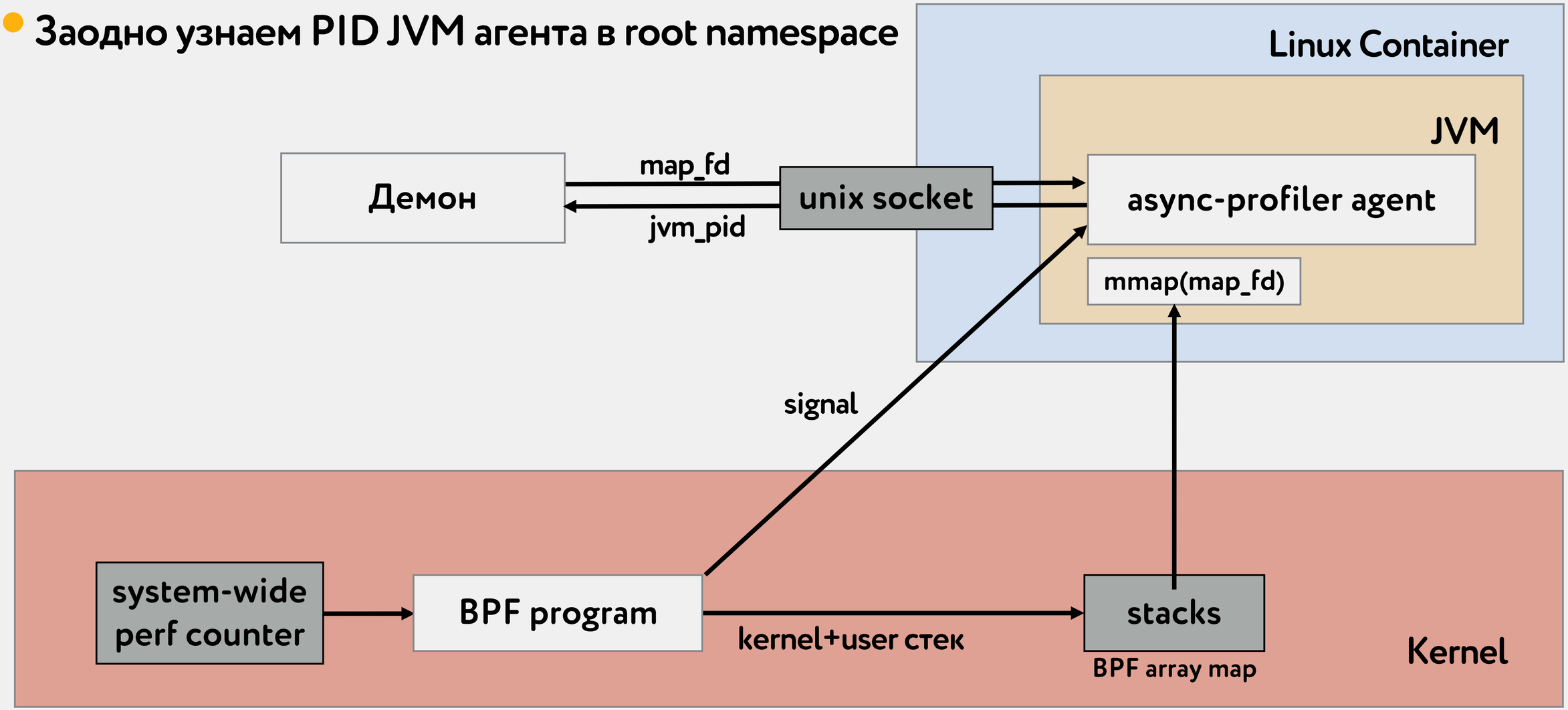
Доступ к BPF maps из контейнера

- BPF syscalls недоступны
- Отображаем память BPF array map внутрь процесса
- Все еще нужно получить map_fd



Передаем map_fd через unix socket

- Заодно узнаем PID JVM агента в root namespace



Демон: протокол инициализации агента через unix socket

И снова используем one-nio!

```
// Инициализируем сокет
var sock = Socket.createUnixSocket(SOCK_SEQPACKET);
sock.bind("/path/to/socket", NO_PORT, 0);
sock.listen(256);

// Ждем подключения async-profiler
Socket s = sock.accept();

// получаем PID JVM с агентом async-profiler
int jvm_pid = bytesToInt(s.getOption(Socket.SOL_SOCKET, Socket.SO_PEERCRED));

// передаем дескриптор BPF map handle и salt
int salt = hash(jvm_pid);
s.sendMsg(new Msg(intToBytes(salt)).withFd(bpfMapStacks.fd()), 0);
```

BPF: читаем pid, tid, policy из task_struct

```
// task_struct объявлен в kernel headers
#include <linux/sched.h>
…
// Точка входа
int do_perf_event(struct bpf_perf_event_data *ctx) {
  int tid, pid, policy;

  // Получаем указатель на структуру в Kernel
  struct task_struct* task = (struct task_struct*) bpf_get_current_task();
  tid = task->pid;
  pid = task->tgid;
  policy = task->policy;
…
}
```

BPF: читаем pid, tid, policy из task_struct

```
// task_struct объявлен в kernel headers
#include <linux/sched.h>
…
// Точка входа
int do_perf_event(struct bpf_perf_event_data *ctx) {
  int tid, pid, policy;

  // Получаем указатель на структуру в Kernel
  struct task_struct* task = (struct task_struct*) b
  tid = task->pid;
  pid = task->tgid;
  policy = task->policy;
…
}
```

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        unsigned int                    __state;

#ifdef CONFIG_PREEMPT_RT
        /* saved state for "spinlock sleepers" */
        unsigned int                    saved_state;
#endif

        /*
         * This begins the randomizable portion of task_struct. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

        void                            *stack;
        refcount_t                      usage;
        /* Per task flags (PF_*), defined further below: */
        unsigned int                    flags;
        unsigned int                    ptrace;
```

800 строк кода

- Зависимость от kernel headers: собирать отдельно для каждой сборки Kernel

Переносимый BPF: Compile-Once Run-Everywhere (CO-RE)

- Связывание по именам полей в момент загрузки

```
// Объявляем структуру прямо в коде, добавляя магические флажки
struct task_struct {
    int pid, tgid, policy;
} __attribute__((preserve_access_index));

int do_perf_event(struct bpf_perf_event_data *ctx) {
  int tid, pid, policy;

  struct task_struct* task = (struct task_struct*) bpf_get_current_task();
  // CO-RE чтения: смещения полей вычисляются "автомагически" в libbpf
  bpf_probe_read(&tgid, sizeof(tgid), __builtin_preserve_access_index(&task->tgid));
  bpf_probe_read(&pid, sizeof(pid), __builtin_preserve_access_index(&task->pid));
  bpf_probe_read(&policy, sizeof(policy), __builtin_preserve_access_index(&task->policy));
…
}
```

- Работает на любом Kernel

BPF: получение user+kernel стека

```
void* p_stack = XXX; // область памяти длиной 1024 байт
int len = bpf_get_stack(ctx, p_stack, 1024, 0 /*kernel stack*/);
if (len >= 0 && len < 1024) {
    len = bpf_get_stack(ctx, p_stack + len, 1024 - len, BPF_F_USER_STACK);
}
```

BPF Helper:

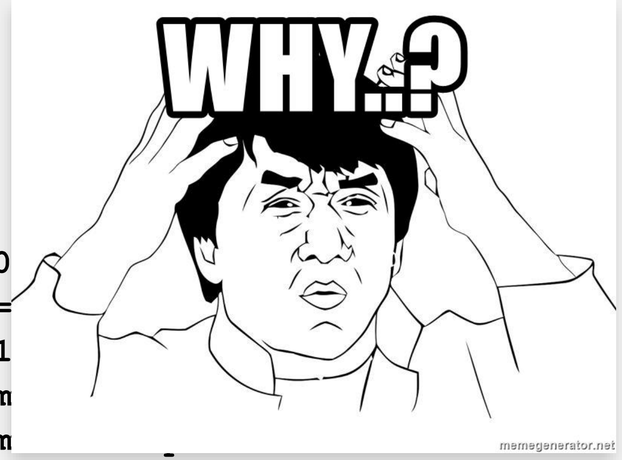
```
bpf_get_stack(ctx, buf, size, flags)
```

BPF: ошибка верификатора при загрузке программы

```
void* p_stack = XXX; // область памяти длиной 1024 байт
int len = bpf_get_stack(ctx, p_stack, 1024, 0 /*kernel stack*/);
if (len >= 0 && len < 1024) {
    len = bpf_get_stack(ctx, p_stack + len, 1024 - len, BPF_F_USER_STACK);
}
```

... (куча строк дампа пропущена)

```
; len = bpf_get_stack(ctx, p_stack + len, 1024 - len, BPF_F_USER_STACK);
280: (bf) r1 = r6
281: (b7) r4 = 256
282: (85) call bpf_get_stack#67
R0=inv(id=0) R1_w=ctx(id=0,off=0,imm=0) R2_w=map_value(id=0,off=24,ks=4,vs=1040,umax_value=2040
0x7f8)) R3_w=inv(id=0,umin_value=8,umax_value=1016,var_off=(0x0; 0x3f8)) R4_w=inv256 R6=ctx(id=
R7=map_value(id=0,off=0,ks=4,vs=1040,imm=0) R8=inv(id=14,umax_value=536870911,var_off=(0x0; 0x1
R9_w=inv(id=15,umax_value=126,var_off=(0x0; 0x7f)) R10=fp0 fp-8=mmmmmmmm fp-16=mmmm???? fp-24=m
32=mmmm???? fp-40=mmmmmmmm fp-48=mmmmmmmm fp-56=mmmmmmmm fp-64=mmmmmmmm fp-72=mmmmmmmm fp-80=mm
88=mmmmmmmm fp-96=mmmmmmmm fp-104=mmmmmmmm fp-112=mmmmmmmm fp-120=mmmmmmmm fp-128=map_value
invalid access to map value, value_size=1040 off=2064 size=1024
```



R2 max value is outside of the allowed memory range

BPF: модель ограничений верификатора

```
void* p_stack = XXX; // область памяти длиной 1024 байт
int len = bpf_get_stack(ctx, p_stack, 1024, 0 /*kernel stack*/);
if (len >= 0 && len < 1024) {
    len = bpf_get_stack(ctx, p_stack + len, 1024 - len, BPF_F_USER_STACK);
}
```

```
bpf_get_stack(ctx, buf, size, flags)

buf ∈ [p_stack, p_stack + 1024)

size ∈ (0, 1024]

buf + size ∈ (p_stack, p_stack + 2048)
```

BPF: обходим ограничения верификатора

```
void* p_stack = XXX; // область памяти длиной 1024
int len = bpf_get_stack(ctx, p_stack, 1024, 0 /*kernel stack*/);
switch (len) {
  case 0:
    len = bpf_get_stack(ctx, p_stack + 0, 1024, BPF_F_USER_STACK);
    break;
  case 8:
    len = bpf_get_stack(ctx, p_stack + 8, 1016, BPF_F_USER_STACK);
    break;
  case 16:
  ...
}
```

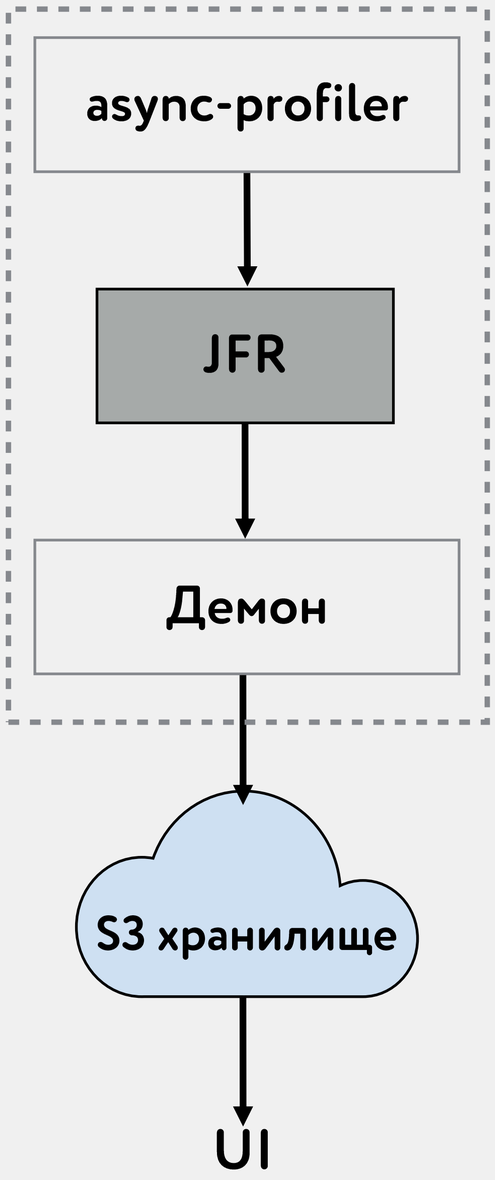
Хорошо что в C есть #define ;)

Развертывание демона профилирования в облаке

- Запуск демона средствами one-cloud
  - в привилегированном контейнере
- GraalVM native image
  - 58Mb, включая bpf, libonenio.so, libasyncProfiler.so
- Обновление BPF программы на лету
  - с сохранением BPF Maps
- Изменение параметров JVM контейнеров

Центральное хранилище профилей

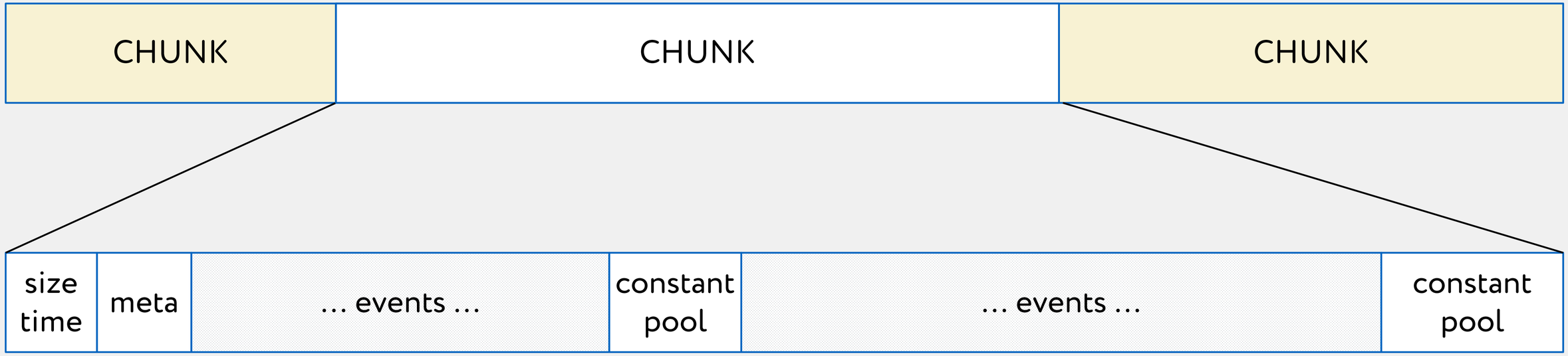
- **Агент пишет в формате JFR (Java Flight Recorder)**
- **Демон загружает в S3-совместимое хранилище**
  26 июня: OK S3 at Hydra Conference (Вадим Цесько)
  https://hydraconf.com/talks/93d5676508ff44b5a780f1e61ae00756/
- **UI читает из хранилища**



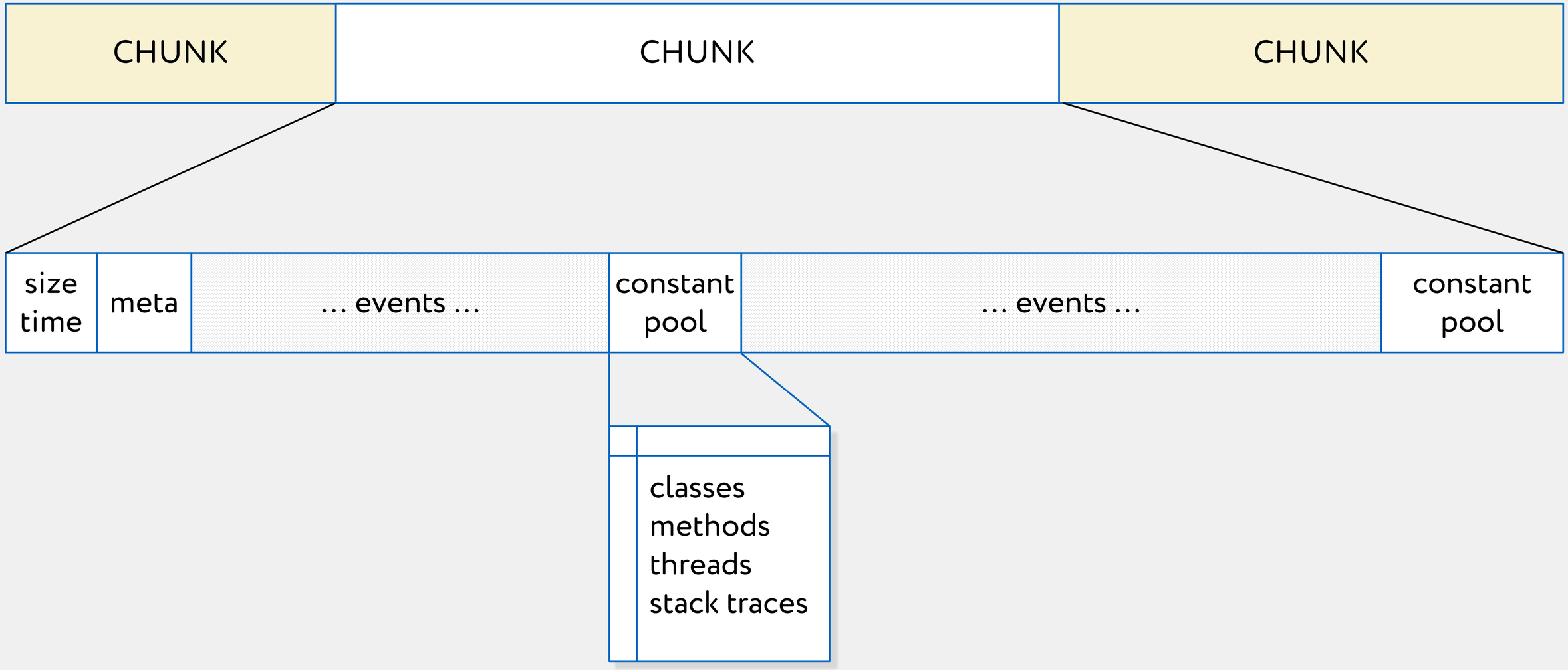
Формат хранения JFR

- Совместим с JDK Mission Control, есть API
- Хранит каждое событие
- Расширяемый
- Компактный: varint, словари

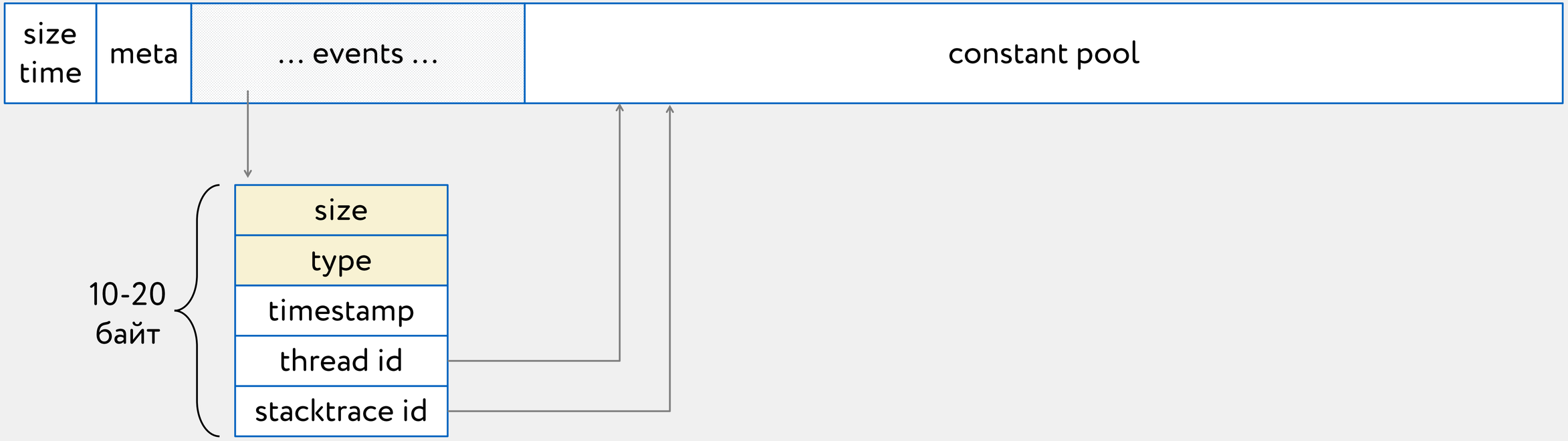
Структура JFR



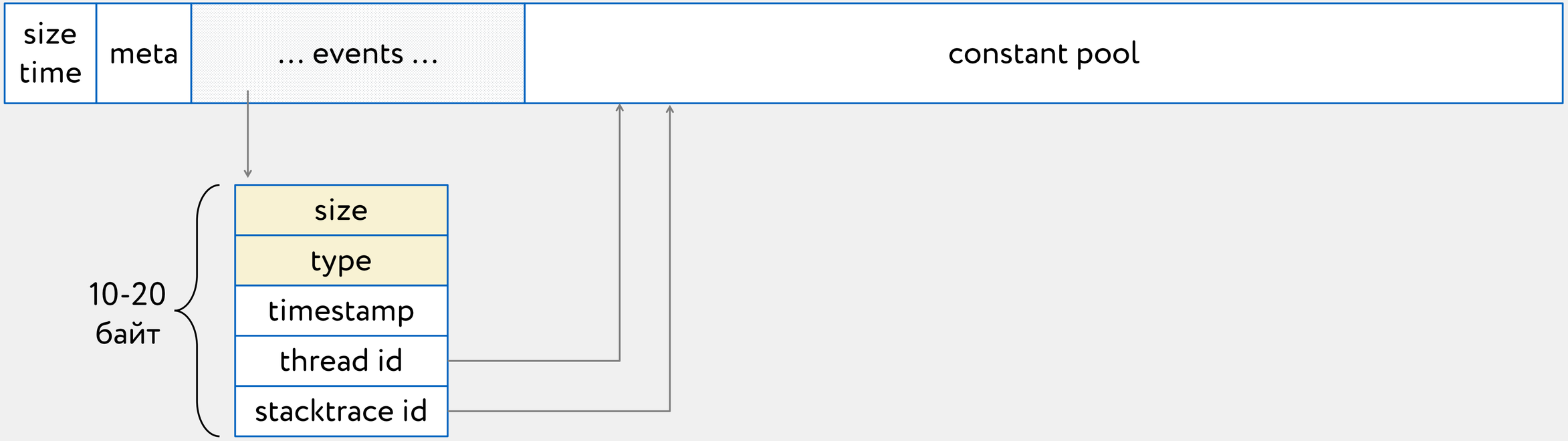
Структура JFR



Структура JFR



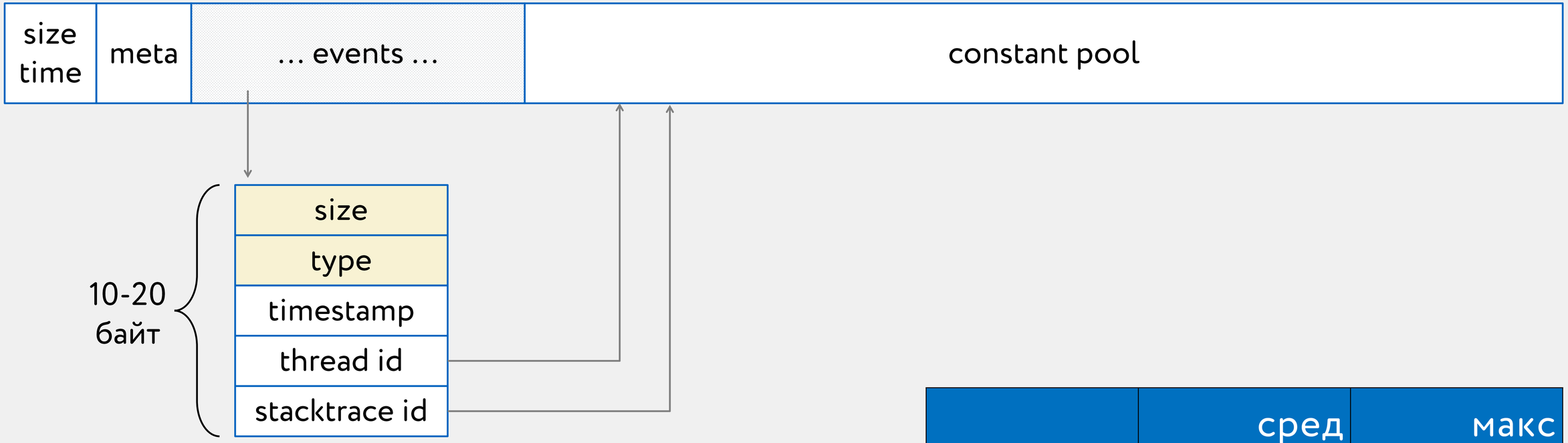
Структура JFR



~10 млн событий в сутки

(при 100% загрузке одного ядра)

Структура JFR



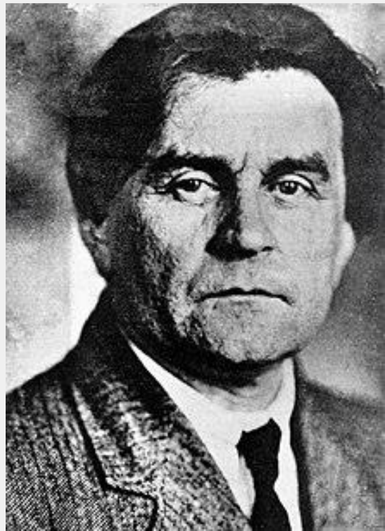
~10 млн событий в сутки

(при 100% загрузке одного ядра)

	сред	макс
исходный	400 MB	4 GB
сжатый	100 MB	600 MB

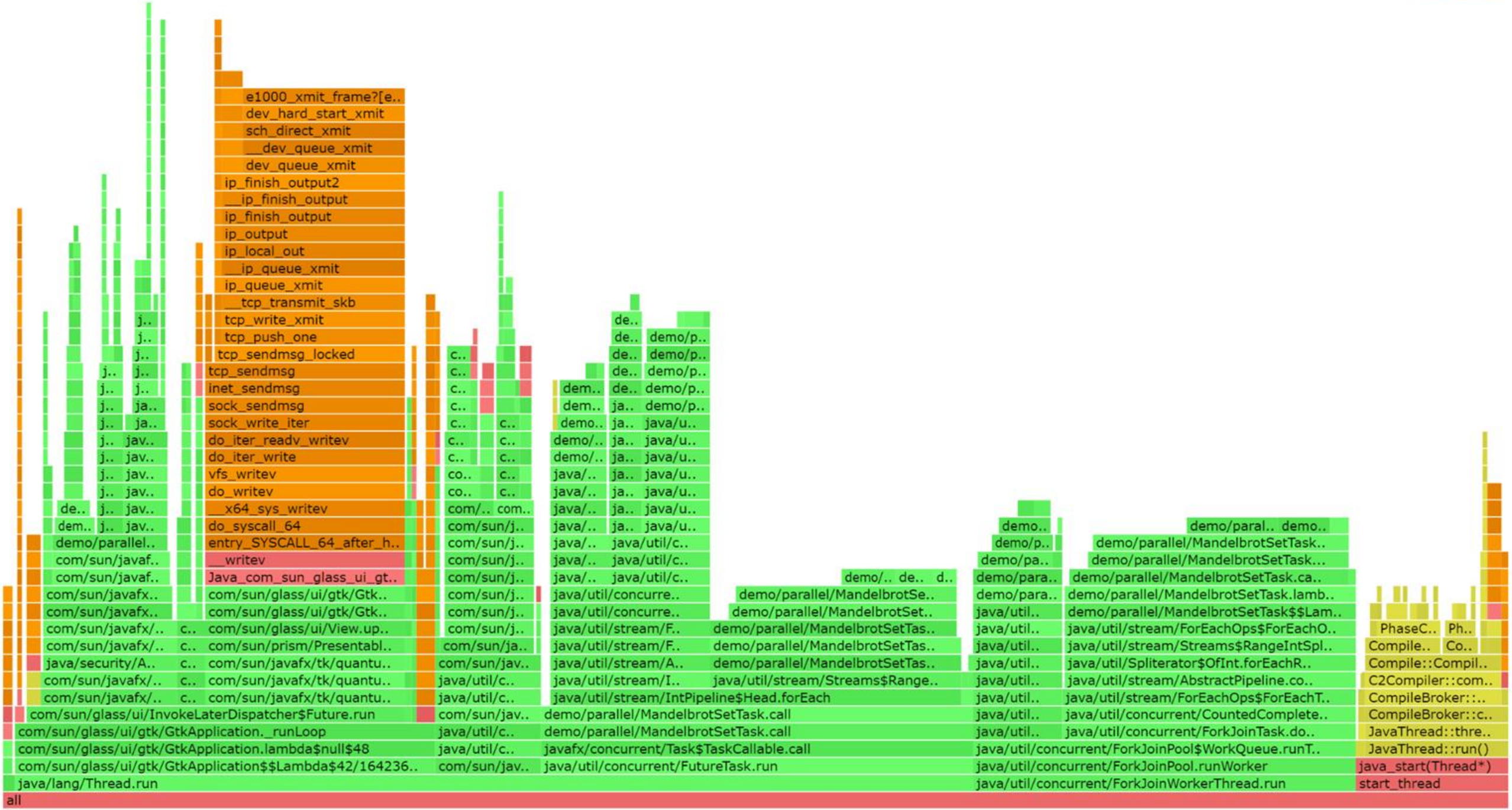
Как показать 4 GB
на одном экране?

Рисуем квадратики



Flame Graph

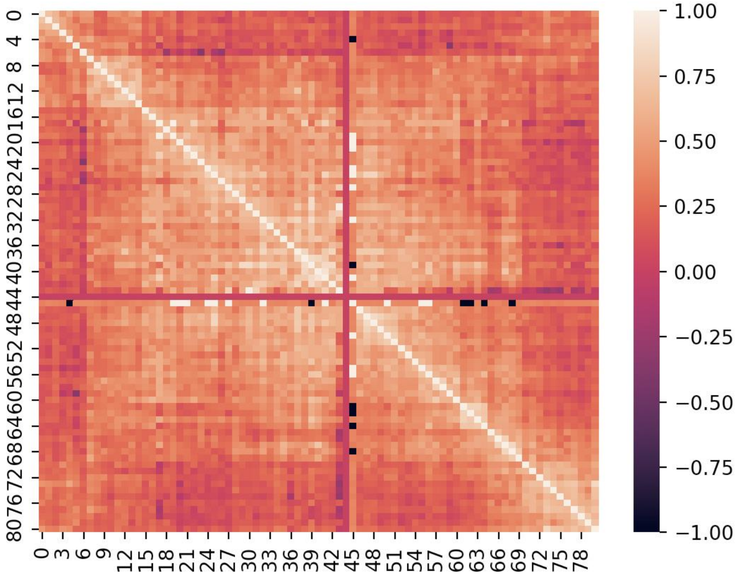
Produced by async-profiler

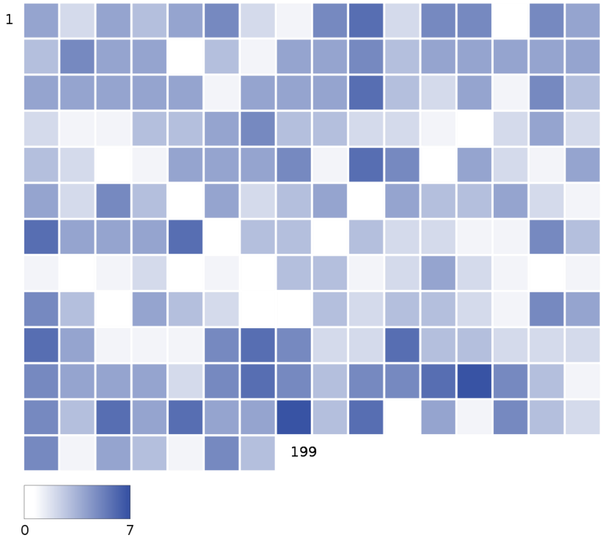


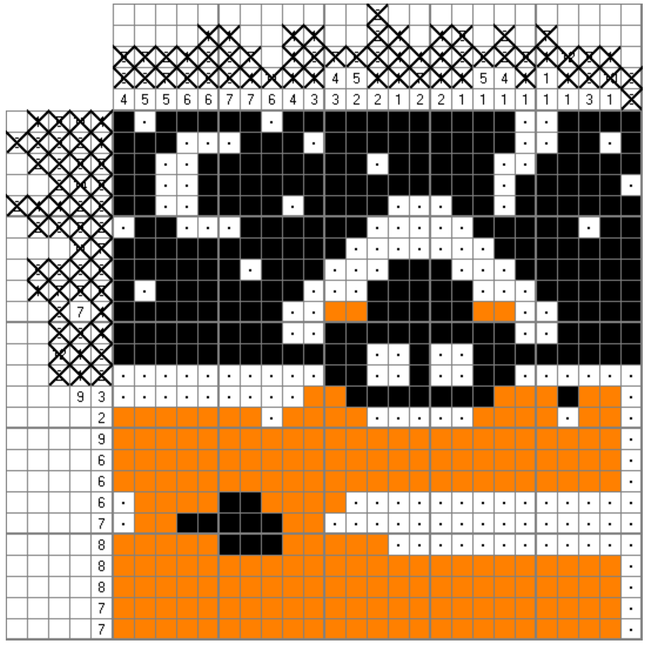
Что хотим?

- Визуально определять проблемы в профиле за сутки
- Смотреть на профиль в разном масштабе
- Получать привычный FlameGraph за любой диапазон
- Сравнивать диапазоны
- Искать проблемные фреймы RegEx’ом

HeatMap - много данных на одной картинке

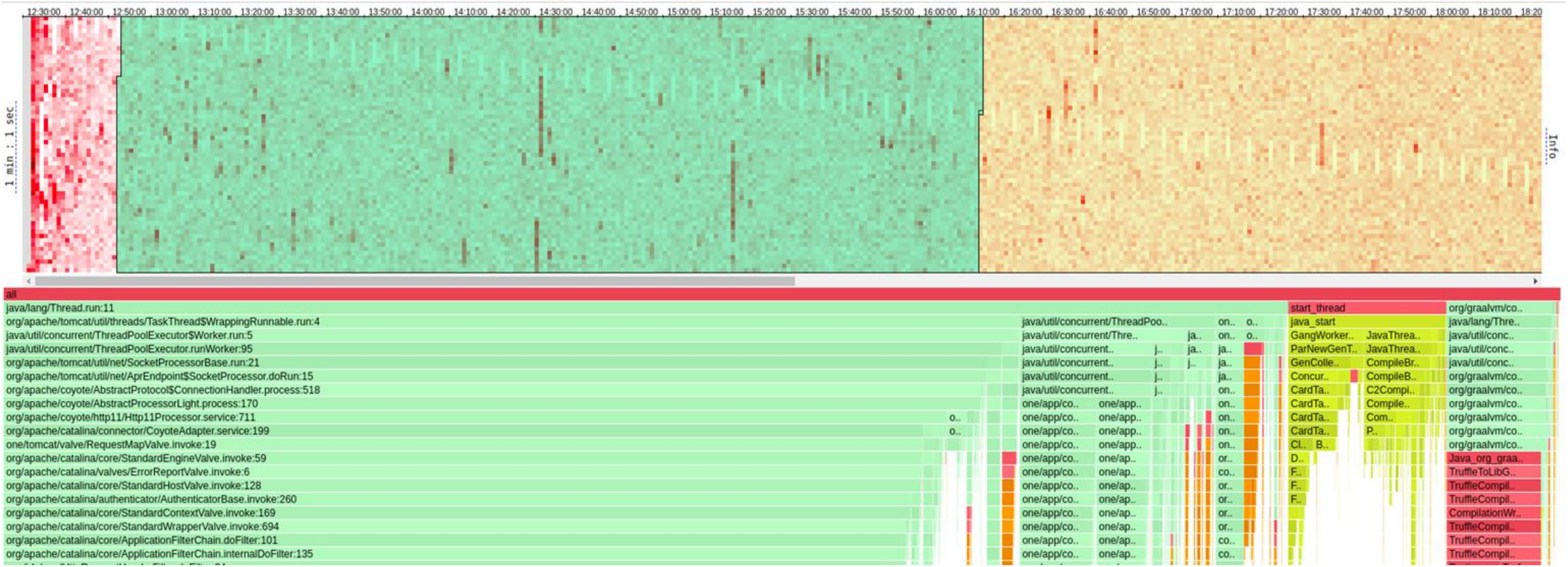






https://www.brendangregg.com/HeatMaps/utilization.html

HeatMap + FlameGraph



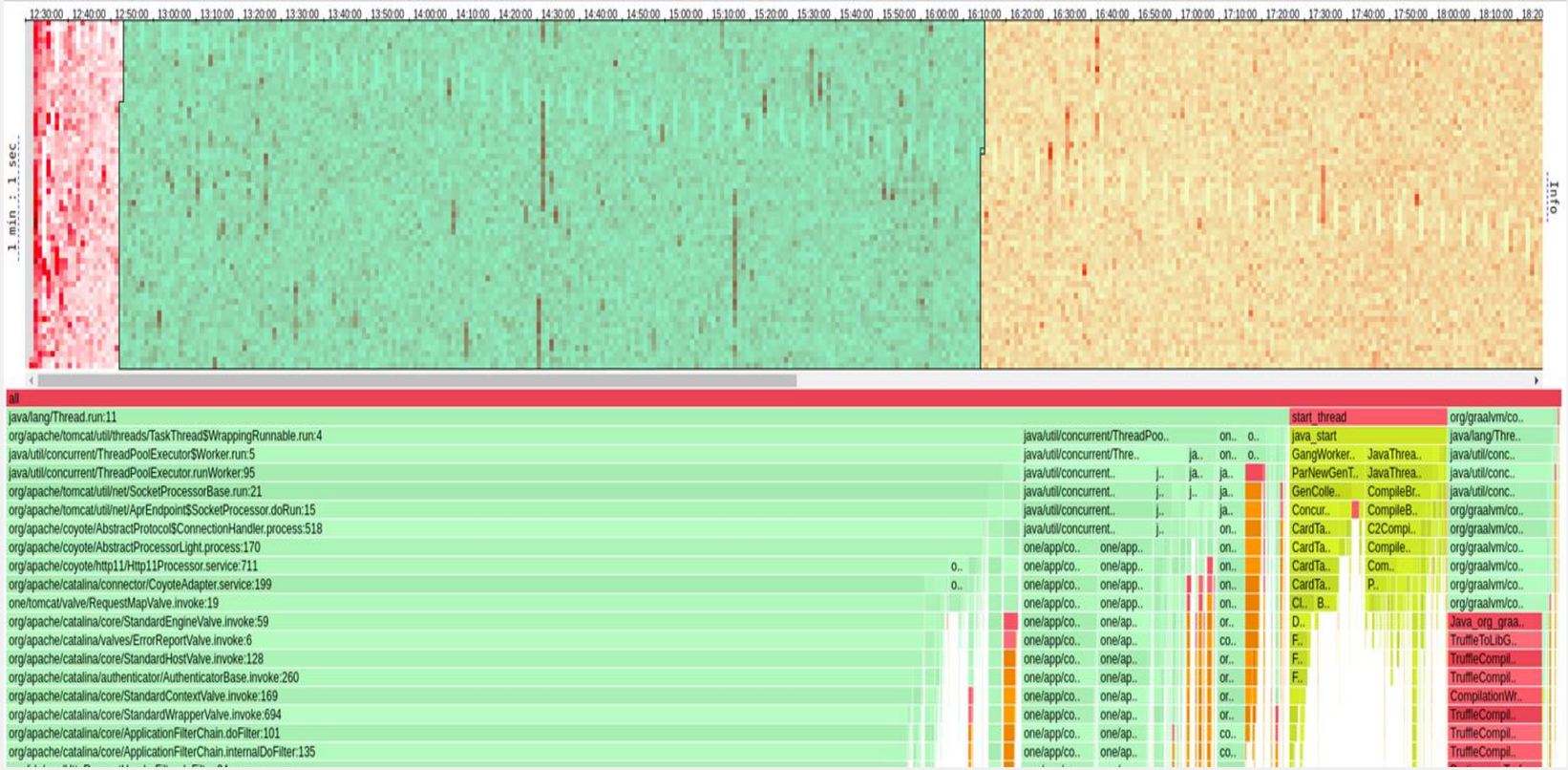
Demo

Вместо тысячи слов!

Что нужно для отрисовки HeatMap?

На JS

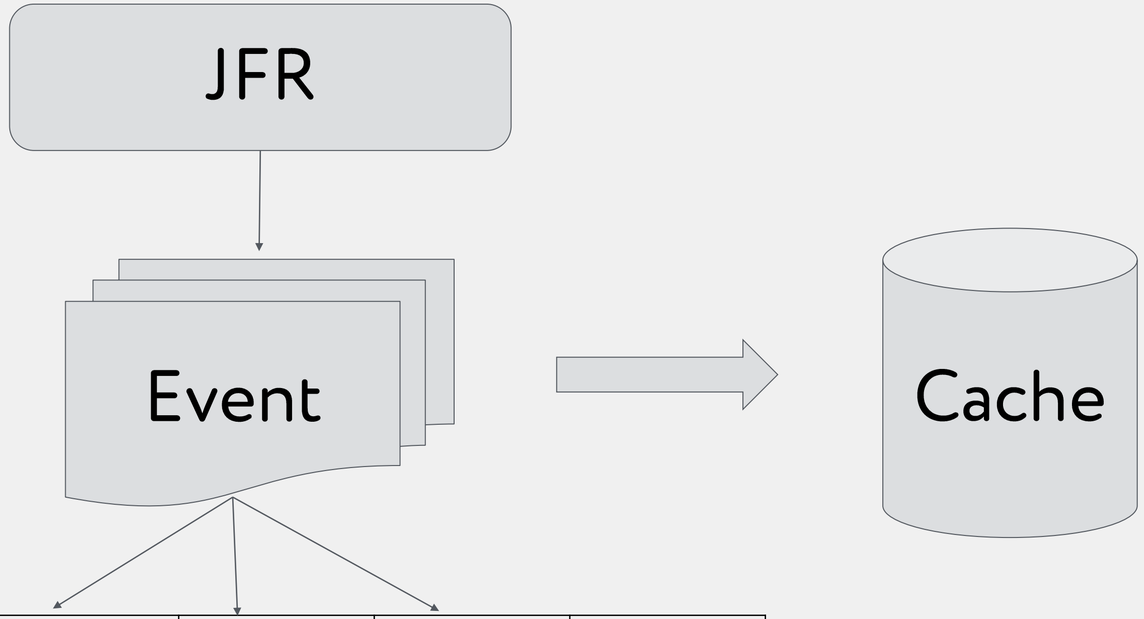
- <iframe>
- <iframe>



Что нужно для отрисовки HeatMap?

На Java

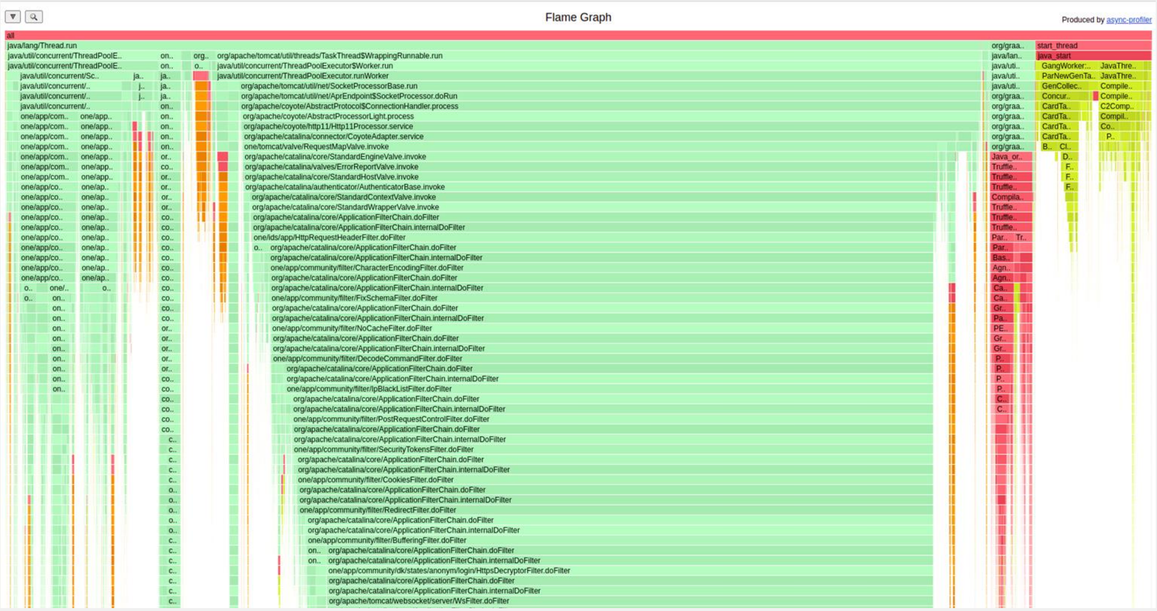
- Прочитать профиль
- Заполнить массив
- Положить в кэш



0 мс	20 мс	40 мс	60 мс	80 мс	100 мс
0	9	7	3	1	10

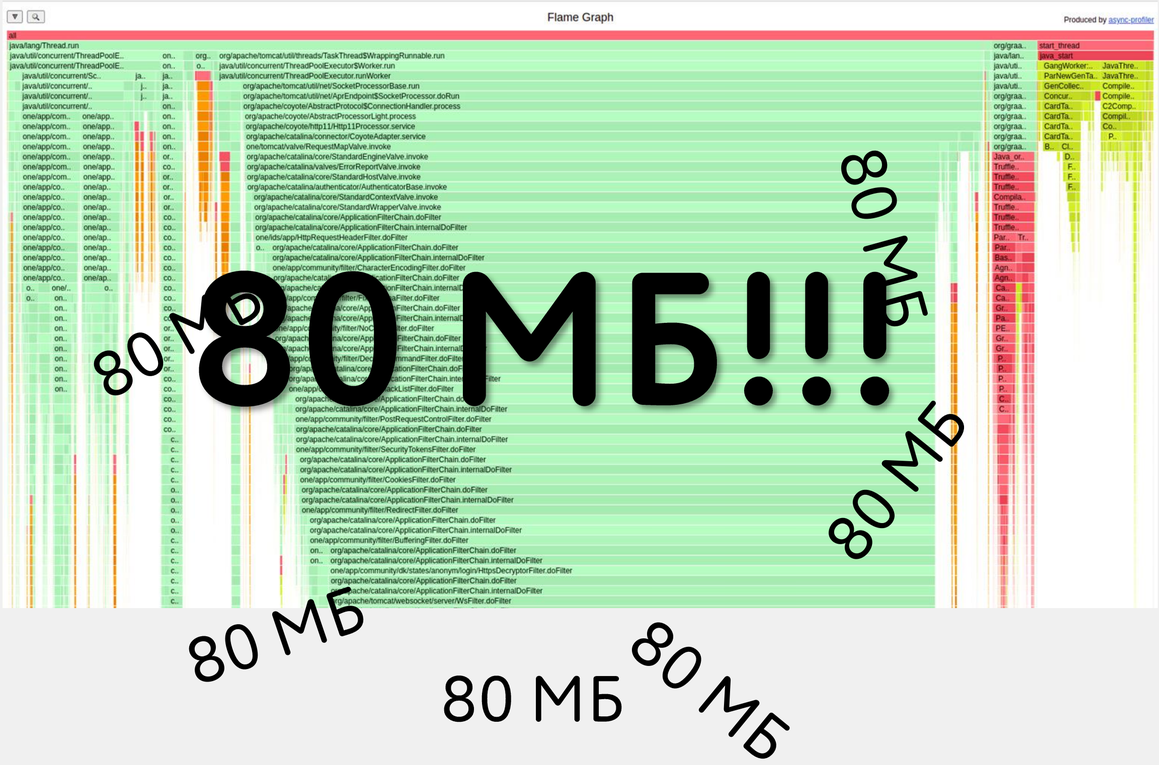
Что нужно для отрисовки HeatMap?

Flame Graph



Что нужно для отрисовки HeatMap?

Flame Graph



Можно лучше!

Single-page HTML

- Профилировать локально
- Хранить бесконечно
- Прикладывать к таскам
- Пересылать
- Хвастаться

Single-Page HTML?

- FlameGraph за любой период
- Гранулярность: 20 мс
- 1-4 Гб данных (!!!)
- 6 млн сэмплов (!!!)

Тут мог бы быть слайд с моими страданиями

(но коллеги сказали, что все и без меня умеют страдать)

Байтики!

	JFR (компактный!)	HTML (не компактный, Base64)
Без сжатия		
GZIP		

Байтики!

	JFR (компактный!)	HTML (не компактный, Base64)
Без сжатия	1.3 ГБ	
GZIP		

Байтики!

	JFR (компактный!)	HTML (не компактный, Base64)
Без сжатия	1.3 ГБ	
GZIP	133 МБ	

Байтики!

	JFR (компактный!)	HTML (не компактный, Base64)
Без сжатия	1.3 ГБ	8 МБ
GZIP	133 МБ	

Байтики!

	JFR (компактный!)	HTML (не компактный, Base64)
Без сжатия	1.3 ГБ	8 МБ CPU + 18 МБ Alloc
GZIP	133 МБ	

Байтики!

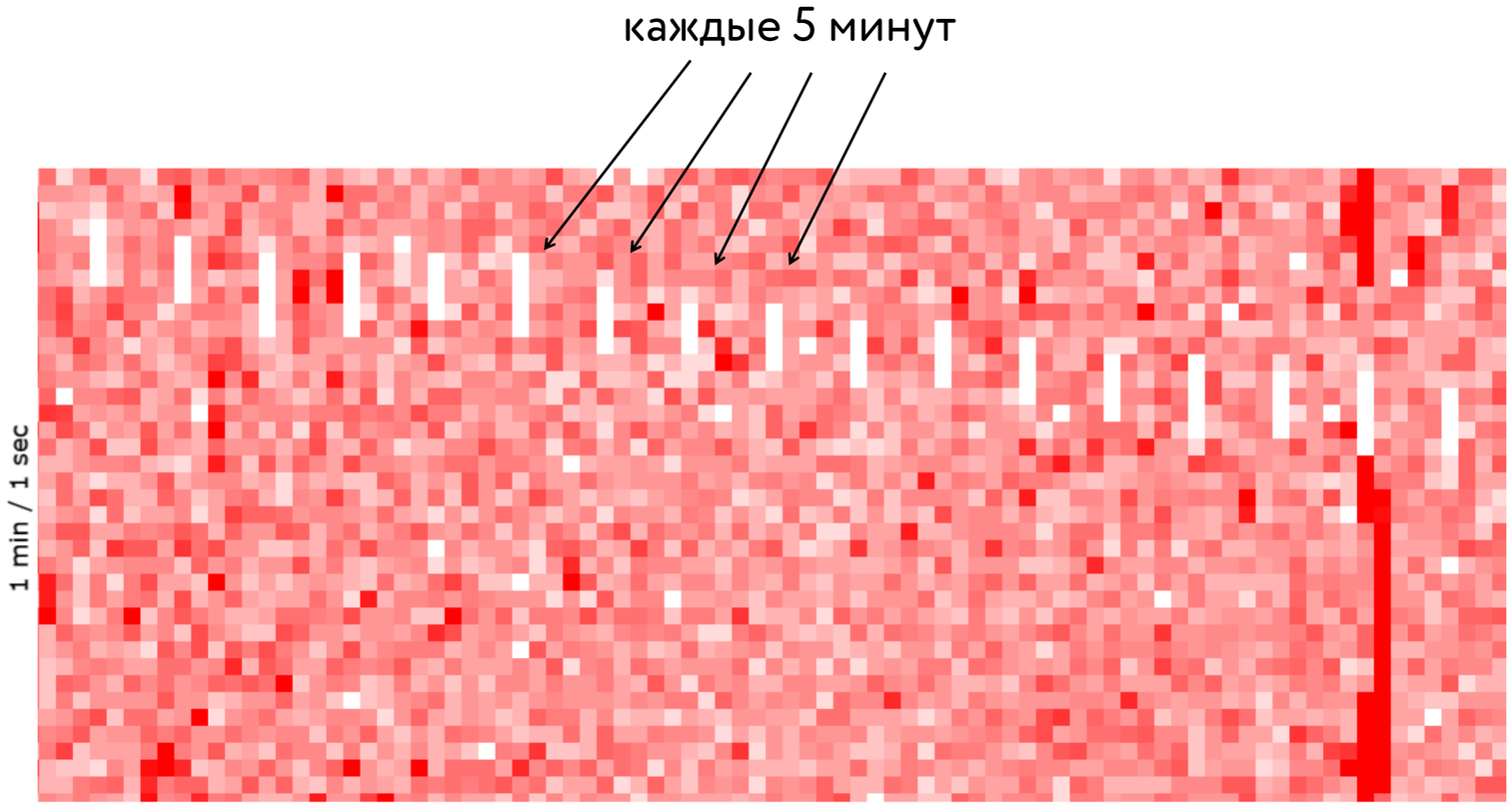
	JFR (компактный!)	HTML (не компактный, Base64)
Без сжатия	1.3 ГБ	26 МБ
GZIP	133 МБ	

Байтики!

	JFR (компактный!)	HTML (не компактный, Base64)
Без сжатия	1.3 ГБ	26 МБ
GZIP	133 МБ	15 МБ

Запуск в эксплуатацию

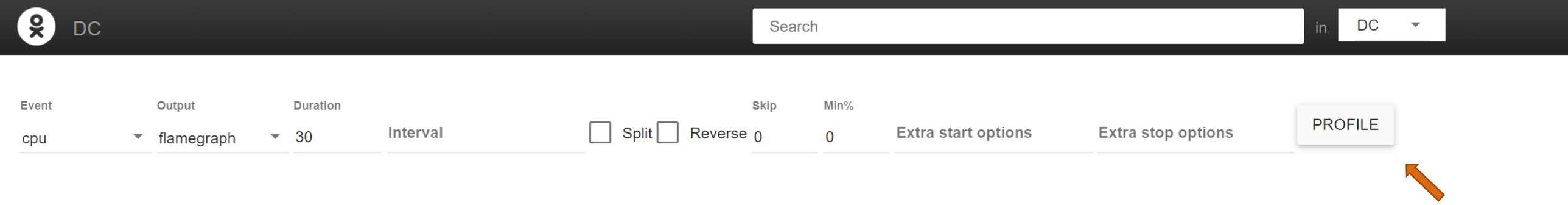
Белые пятна



JDK-8185348: Major performance regression in JVMTI Method functions

Патчим JVM прямо в памяти!

```
// Workaround for JDK-8185348
char* func = (char*)libjvm->findSymbol("_ZN6Method26checked_resolve_jmethod_idEP10_jmethodID");
if (func != NULL) {
    mprotect(align(func), align(func_size), PROT_READ | PROT_WRITE | PROT_EXEC);
    memcpy(func, patch, size);
    mprotect(align(func), align(func_size), PROT_READ | PROT_EXEC);
}
```



On-demand vs. continuous profiler

- Две копии libasyncProfiler.so
- Разные сигналы
- Перехват JVM функций по цепочке

DC

Search in DC

Event: cpu | Output: flamegraph | Duration: 30 | Interval | Split | Reverse | Skip: 0 | Min%: 0 | Extra start options | Extra stop options | PROFILE

CPU overhead

Enter regexp to search:

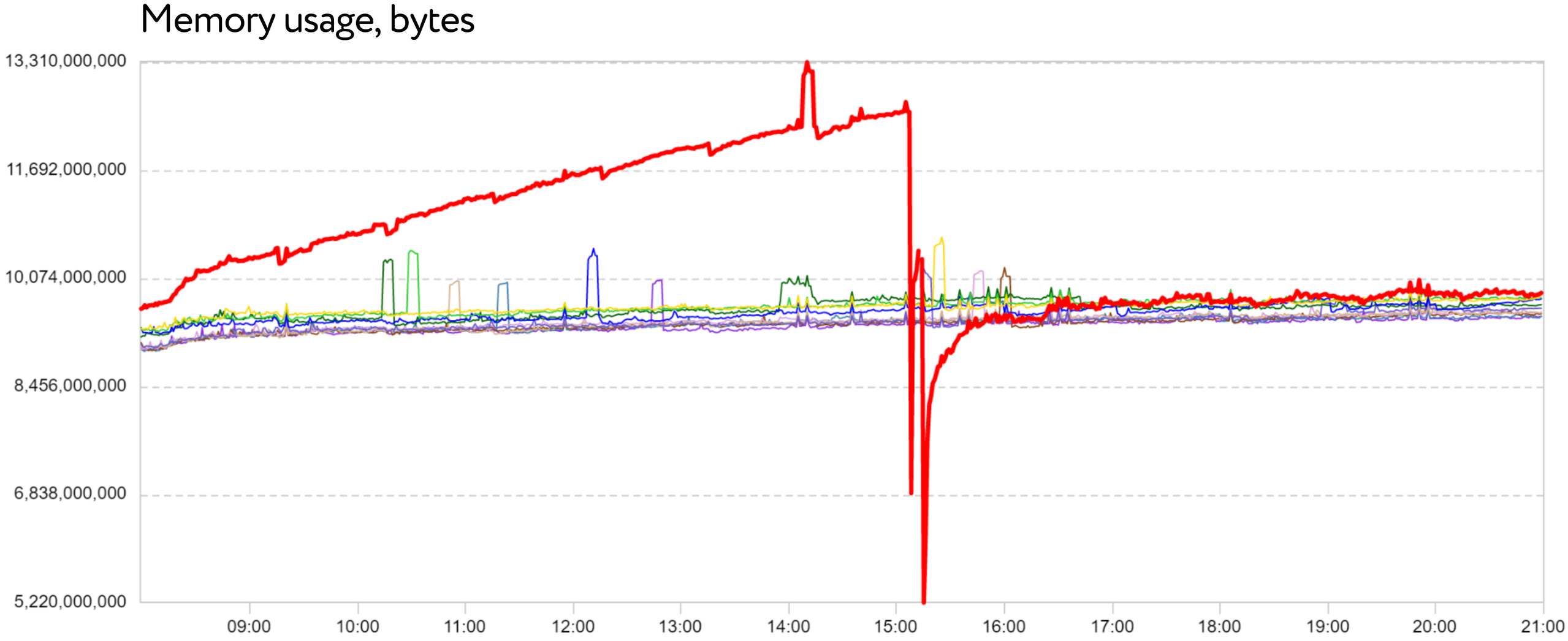
BpfClient

OK Отмена

__GI___pthread_getspecific
Profiler::recordSample(void*, unsigned long long, int, Event*) [clone .constprop.767]
BpfClient::signalHandler(int, siginfo_t*, void*)
__restore_rt
Unsafe_Park
sun/misc/Unsafe.park
java/util/concurrent/locks/LockSupport.park
java/util/concurrent/locks/AbstractQueuedSynchronizer$ConditionObject.await
java/util/concurrent/LinkedBlockingQueue.take
java/util/concurrent/ThreadPoolExecutor.getTask
java/util/concurrent/ThreadPoolExecutor.runWorker
java/util/concurrent/ThreadPoolExecutor$Worker.run
java/lang/Thread.run
all

Matched: 0.05% ✗

RAM overhead



В цифрах

- Интервал сэмплирования: 10M тактов (~200 Hz)
- CPU overhead < 0.1%
- RAM overhead < 200 MB
- 8.5 TB / месяц профилей
- 2 месяца успешной работы в одном ЦОД

Что получили

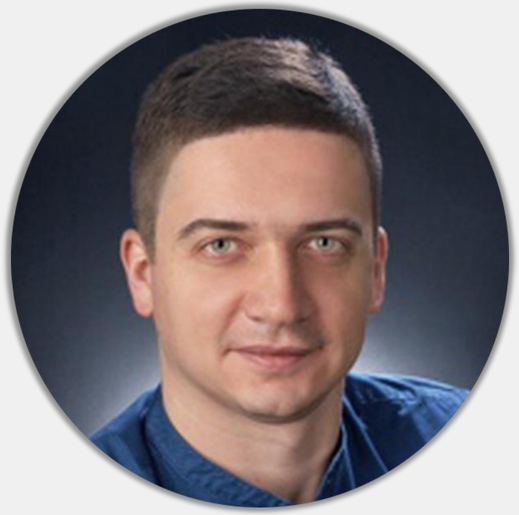
1. Профилирование CPU + Памяти
   - для всех Java приложений
   - всегда включено
2. Сравнение графиков нагрузки (diff)
3. Визуализация кода с точностью до 20 мс

Где потрогать

- async-profiler agent
  https://github.com/jvm-profiling-tools/async-profiler/tree/bpfclient
- BPF profiling program
  https://github.com/odnoklassniki/async-profiler-bpf
- BPF & Unix Sockets in Java
  https://github.com/odnoklassniki/one-nio
- Single-page heatmap
  https://github.com/Artyomcool/async-profiler/tree/single-page-heatmap

experimental

Наша команда



leonid.talalaev@corp.mail.ru

artem.drozdov@corp.mail.ru

andrey.pangin@corp.mail.ru

https://oktech.ru