گزارش آزمایشگاه مهندسی نرم‌افزار

پاییز ۱۴۰۱

دانشکده‌ی مهندسی کامپیوتر

گزارش ۱: آشنایی با نحوه پروفایل برنامه (Profiling)

مهرانه نجفی(۹۷۱۰۴۷۰۷) رستا روغنی(۹۷۱۰۵۹۶۳)

۱

۱.۱ بررسی برنامه با YourKit

در کلاس JavaCup سه تابع main ، eval و temp داریم اما هدف ما مطالعه‌ی مستقیم آن‌ها برای پیدا کردن مشکل نیست. پس ابتدا بدون بررسی کد، برنامه را با YourKit ران می‌کنیم و اعداد ۱ و ۲ و ۳ را به ترتیب به آن ورودی می‌دهیم. پس از مدتی می‌بینیم که حافظه‌ی heap سرازیر می‌شود و به ارور می‌خوریم:

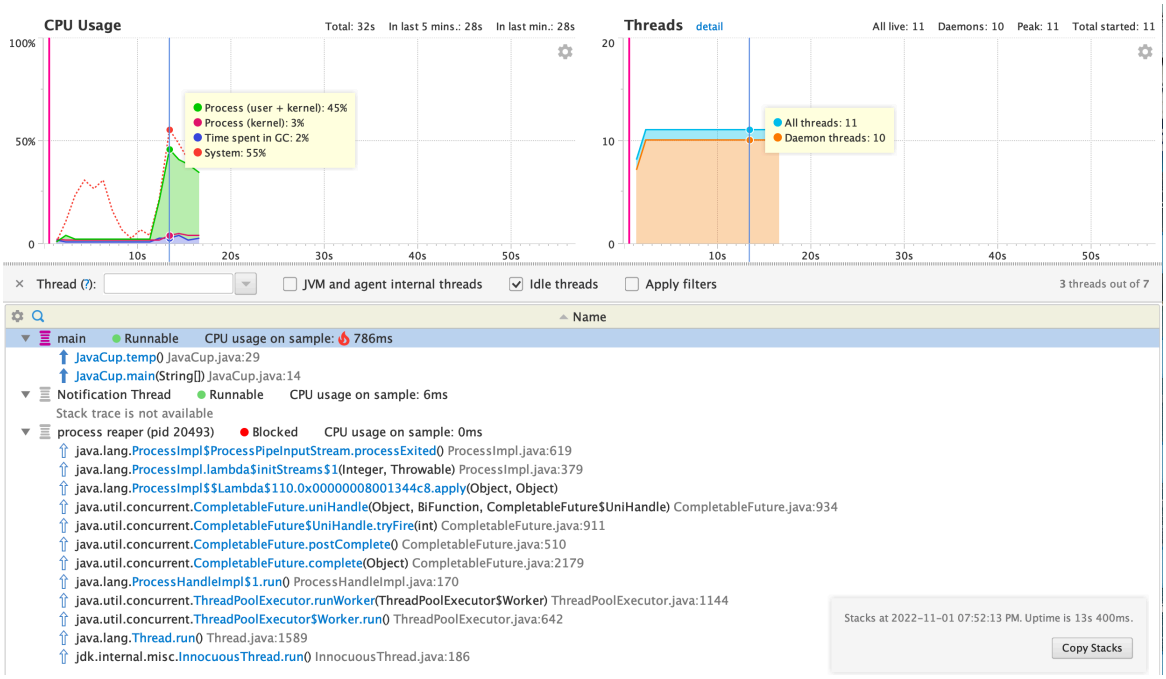
```
/Users/rostaroghani/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java -agentpath:/Applications/YourKit-Java-Profiler-2022.9.app/Conter
[YourKit Java Profiler 2022.9-b170] Log file: /Users/rostaroghani/.yjp/log/JavaCup-20492.log
Press number1:
1
Press number2:
2
Press number3:
3
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid20492.hprof ...
Heap dump file created [5842552909 bytes in 35.051 secs]
Exception in thread "main" java.lang.OutOfMemoryError Create breakpoint : Java heap space
    at java.base/java.util.Arrays.copyOf(Arrays.java:3481)
    at java.base/java.util.ArrayList.grow(ArrayList.java:237)
    at java.base/java.util.ArrayList.grow(ArrayList.java:244)
    at java.base/java.util.ArrayList.add(ArrayList.java:454)
    at java.base/java.util.ArrayList.add(ArrayList.java:467)
    at JavaCup.temp(JavaCup.java:30)
    at JavaCup.main(JavaCup.java:14)

Process finished with exit code 1
```

شکل ۱: OutOfMemoryError

از برنامه Snapshot می‌گیریم تا بررسی کنیم که مشکل از کجای برنامه‌است.

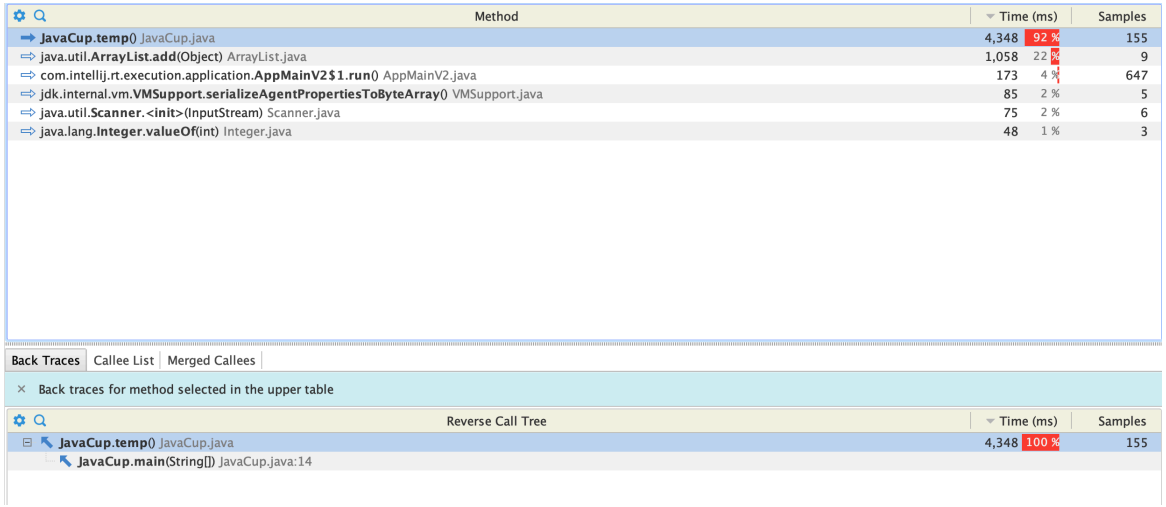
- بخش CPU

در زمانی که استفاده از CPU به بیشترین مقدار خود (بیشتر از ٪۵۰) رسیده‌است، ریسه‌ی main بیشترین استفاده را داشته‌است که با backtrace به تابع temp می‌رسیم (که در داخل تابع main صدا زده‌شده‌است).



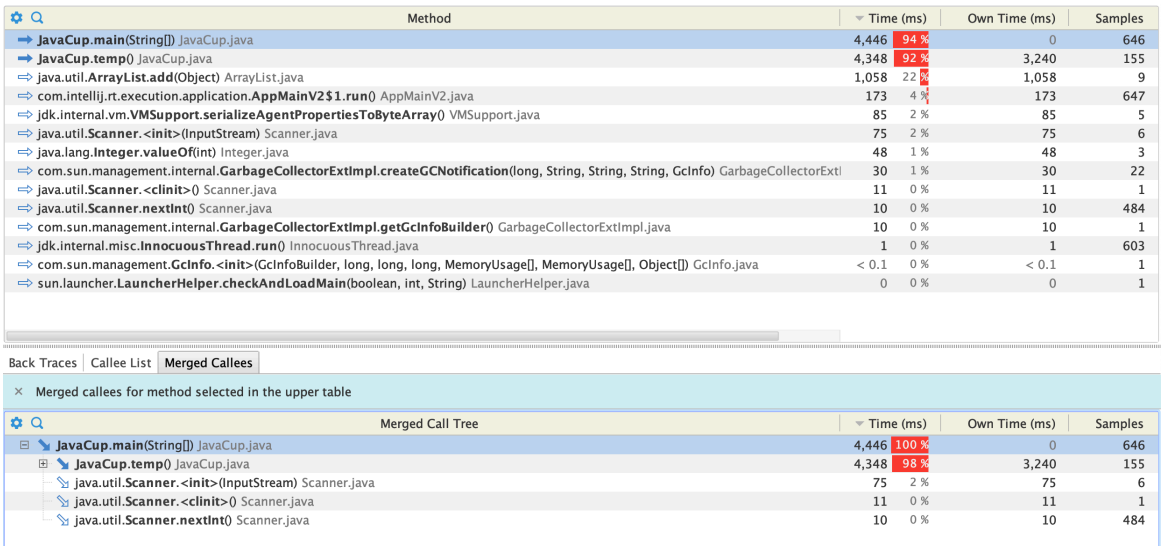
شکل ۲: CPU Usage

همچنین در Hot spots مشخص شده‌است که بیشترین زمان اجرا (۹۲٪) مربوط به JavaCup.temp است:



شکل ۳: Hot spots

استفاده از CPU را می‌توان در بخش Method lists هم بررسی کرد که در اینجا می‌بینیم تابع main بیشترین زمان اجرا (۹۴٪) را داشته است که البته با توجه به Merged callees مشخص می‌شود که ۹۸ درصد از آن زمان تابع temp در حال اجرا بوده‌است.



Method	Time (ms)	Own Time (ms)	Samples
JavaCup.main(String[]) JavaCup.java	4,446 94 %	0	646
JavaCup.temp() JavaCup.java	4,348 92 %	3,240	155
java.util.**ArrayList.add**(Object) ArrayList.java	1,058 22 %	1,058	9
com.intellij.rt.execution.application.**AppMainV2$1.run**() AppMainV2.java	173 4 %	173	647
jdk.internal.vm.**VMSupport.serializeAgentPropertiesToByteArray**() VMSupport.java	85 2 %	85	5
java.util.**Scanner.<init>**(InputStream) Scanner.java	75 2 %	75	6
java.lang.**Integer.valueOf**(int) Integer.java	48 1 %	48	3
com.sun.management.internal.**GarbageCollectorExtImpl.createGCNotification**(long, String, String, String, GcInfo) GarbageCollectorExtI	30 1 %	30	22
java.util.**Scanner.<clinit>**() Scanner.java	11 0 %	11	1
java.util.**Scanner.nextInt**() Scanner.java	10 0 %	10	484
com.sun.management.internal.**GarbageCollectorExtImpl.getGcInfoBuilder**() GarbageCollectorExtImpl.java	10 0 %	10	1
jdk.internal.misc.**InnocuousThread.run**() InnocuousThread.java	1 0 %	1	603
com.sun.management.**GcInfo.<init>**(GcInfoBuilder, long, long, long, MemoryUsage[], MemoryUsage[], Object[]) GcInfo.java	< 0.1 0 %	< 0.1	1
sun.launcher.**LauncherHelper.checkAndLoadMain**(boolean, int, String) LauncherHelper.java	0 0 %	0	1

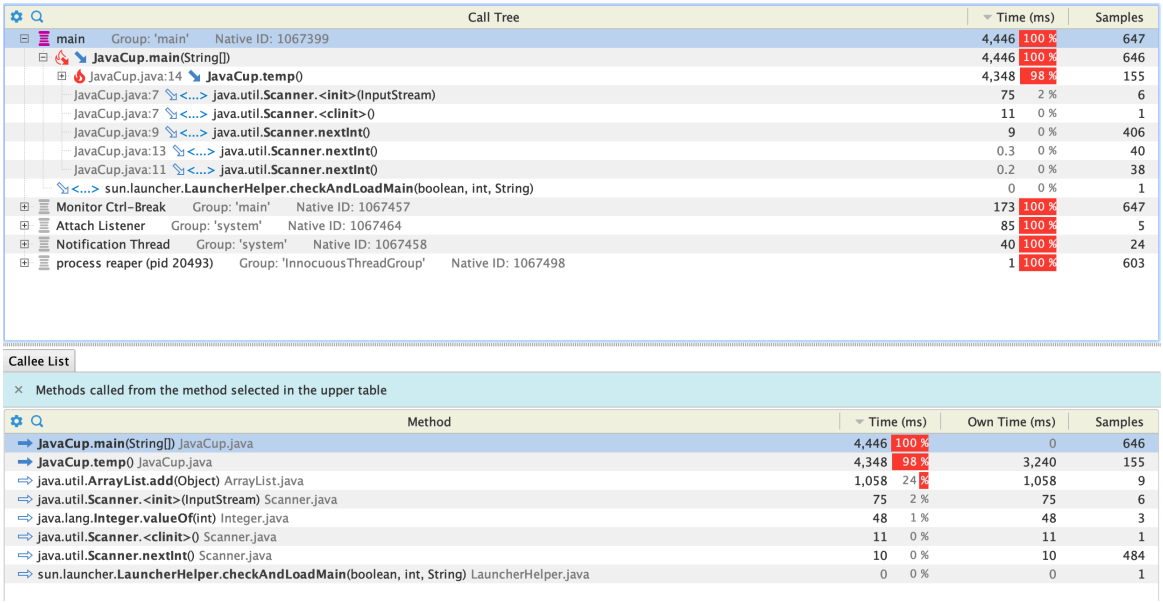
Back Traces | Callee List | Merged Callees

Merged callees for method selected in the upper table

Merged Call Tree	Time (ms)	Own Time (ms)	Samples
JavaCup.main(String[]) JavaCup.java	4,446 100 %	0	646
JavaCup.temp() JavaCup.java	4,348 98 %	3,240	155
java.util.**Scanner.<init>**(InputStream) Scanner.java	75 2 %	75	6
java.util.**Scanner.<clinit>**() Scanner.java	11 0 %	11	1
java.util.**Scanner.nextInt**() Scanner.java	10 0 %	10	484

شکل ۴: Method lists

اطلاعات مشابهی را در Call tree هم می‌توان دید:



Call Tree	Time (ms)	Samples
main Group: 'main' Native ID: 1067399	4,446 100 %	647
JavaCup.main(String[])	4,446 100 %	646
JavaCup.java:14 **JavaCup.temp**()	4,348 98 %	155
JavaCup.java:7 <...> java.util.**Scanner.<init>**(InputStream)	75 2 %	6
JavaCup.java:7 <...> java.util.**Scanner.<clinit>**()	11 0 %	1
JavaCup.java:9 <...> java.util.**Scanner.nextInt**()	9 0 %	406
JavaCup.java:13 <...> java.util.**Scanner.nextInt**()	0.3 0 %	40
JavaCup.java:11 <...> java.util.**Scanner.nextInt**()	0.2 0 %	38
<...> sun.launcher.**LauncherHelper.checkAndLoadMain**(boolean, int, String)	0 0 %	1
Monitor Ctrl-Break Group: 'main' Native ID: 1067457	173 100 %	647
Attach Listener Group: 'system' Native ID: 1067464	85 100 %	5
Notification Thread Group: 'system' Native ID: 1067458	40 100 %	24
process reaper (pid 20493) Group: 'InnocuousThreadGroup' Native ID: 1067498	1 100 %	603

Callee List

Methods called from the method selected in the upper table

Method	Time (ms)	Own Time (ms)	Samples
JavaCup.main(String[]) JavaCup.java	4,446 100 %	0	646
JavaCup.temp() JavaCup.java	4,348 98 %	3,240	155
java.util.**ArrayList.add**(Object) ArrayList.java	1,058 24 %	1,058	9
java.util.**Scanner.<init>**(InputStream) Scanner.java	75 2 %	75	6
java.lang.**Integer.valueOf**(int) Integer.java	48 1 %	48	3
java.util.**Scanner.<clinit>**() Scanner.java	11 0 %	11	1
java.util.**Scanner.nextInt**() Scanner.java	10 0 %	10	484
sun.launcher.**LauncherHelper.checkAndLoadMain**(boolean, int, String) LauncherHelper.java	0 0 %	0	1

شکل ۵: Call tree

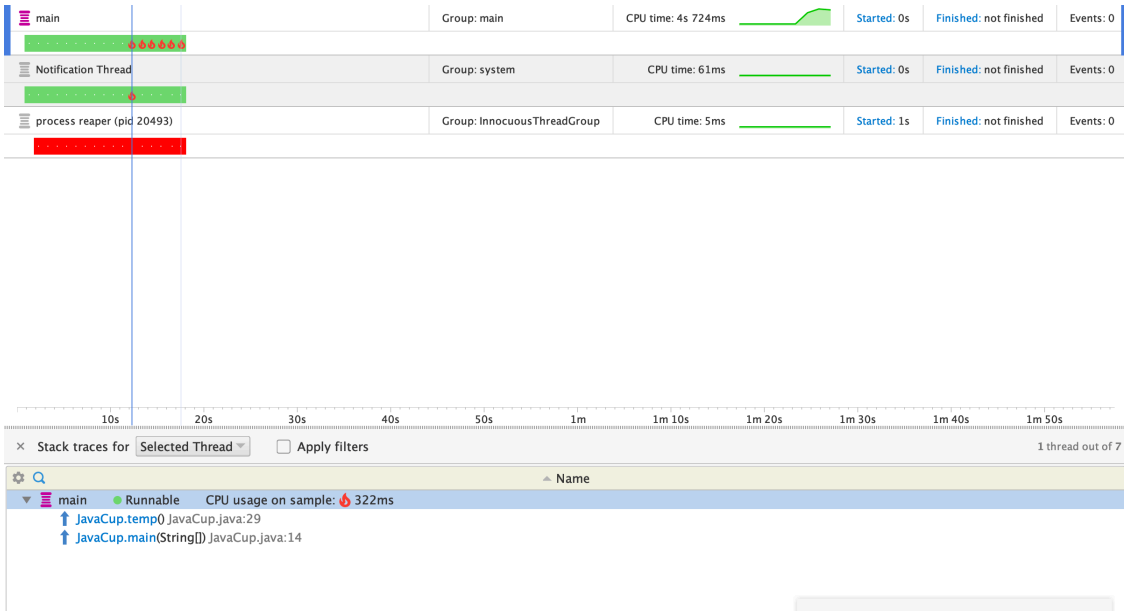
- بخش Memory

در آخرین لحظه‌ی اجرای برنامه و قبل از کرش، استفاده از Memory Heap به حداکثر خود رسیده بود که اینجا هم به شکل بازگشتی به temp می‌رسیم.

شکل ۶: Heap Memory

- بخش Threads

اگر اولین زمانی که استفاده از CPU توسط ریسه‌ی main زیاد شده است را بررسی کنیم (و یا زمان‌های بعد از آن که با آتش نشان داده‌شده‌اند)، اسم تابع temp آورده‌شده‌است.



شکل ۷: Threads

بنابراین با بررسی نمودارهای بخش‌های مختلف، دیدیم که تابع temp بیشترین مصرف منابع را دارد.

۲.۱ درست کردن تابع temp

ابتدا تابع را بررسی می‌کنیم:
این تابع یک ArrayList به نام a درست می‌کند و با استفاده از دو حلقه for تودرتو، اعداد زیر را به آرایه اضافه می‌کند:

$$
\begin{gathered}
0, 1, 2, \ldots, 9999 \\
1, 2, 3, \ldots, 10000 \\
2, 3, 4, \ldots, 10001 \\
\vdots \\
9999, 10000, 10001, \ldots, 29998
\end{gathered}
$$

```java
public static void temp() {
    ArrayList a = new ArrayList();
    for (int i = 0; i < 10000; i++)
    {
        for (int j = 0; j < 20000; j++) {
            a.add(i + j);
        }
    }
}
```

شکل ۸: temp

برای بهتر کردن زمان اجرا، به‌جای ArrayList از Array استفاده می‌کنیم چون اندازه Array یک مقدار مشخص و static است (برخلاف ArrayList که می‌شود اندازه را تغییر داد) بنابراین زمان اجرای آن کمتر است. سایز آرایه هم که برابر است با $20000 * 10000 = 200000000$.
همچنین چون در Array تابع add نداریم، یک متغیر index تعریف می‌کنیم که ابتدا برابر با صفر است و هربار که یک عضو به تابع اضافه می‌شود، به مقدار index هم یک عدد اضافه می‌شود و مشخص می‌کند که عضو بعدی در چه خانه‌ای از آرایه باید قرار بگیرد.

```java
public static void temp() {
    int[] a = new int[200000000];
    int index = 0;
    for (int i = 0; i < 10000; i++)
    {
        for (int j = 0; j < 20000; j++) {
            a[index] = i + j;
            index = index + 1;
        }
    }
}
```

شکل ۹: new temp

این‌بار با اجرای برنامه و دادن ورودی‌های ۱ و ۲ و ۳، به هیچ مشکلی نمی‌خوریم و برنامه خروجی تابع eval را چاپ کرده و به پایان می‌رسد.

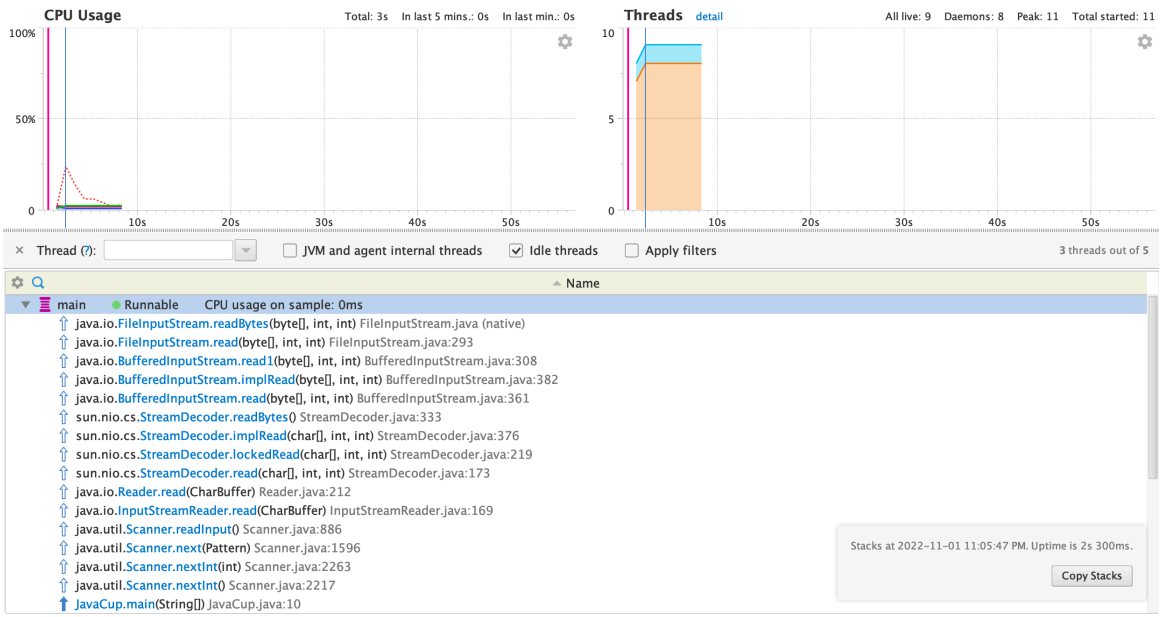
```
/Users/rostaroghani/Library/Java/JavaVirtualMachines/openjdk-19.0.1/Contents/Home/bin/java -agentpath:/Applications/YourKit-Java-Profiler-2022.9.app/Conten
[YourKit Java Profiler 2022.9-b170] Log file: /Users/rostaroghani/.yjp/log/JavaCup-22335.log
Press number1:
1
Press number2:
2
Press number3:
3
NO

Process finished with exit code 0
```

شکل ۱۰: program's output

دوباره Snapshot گرفته و بخش‌هایی را که قبلا بررسی کرده بودیم را بررسی می‌کنیم:

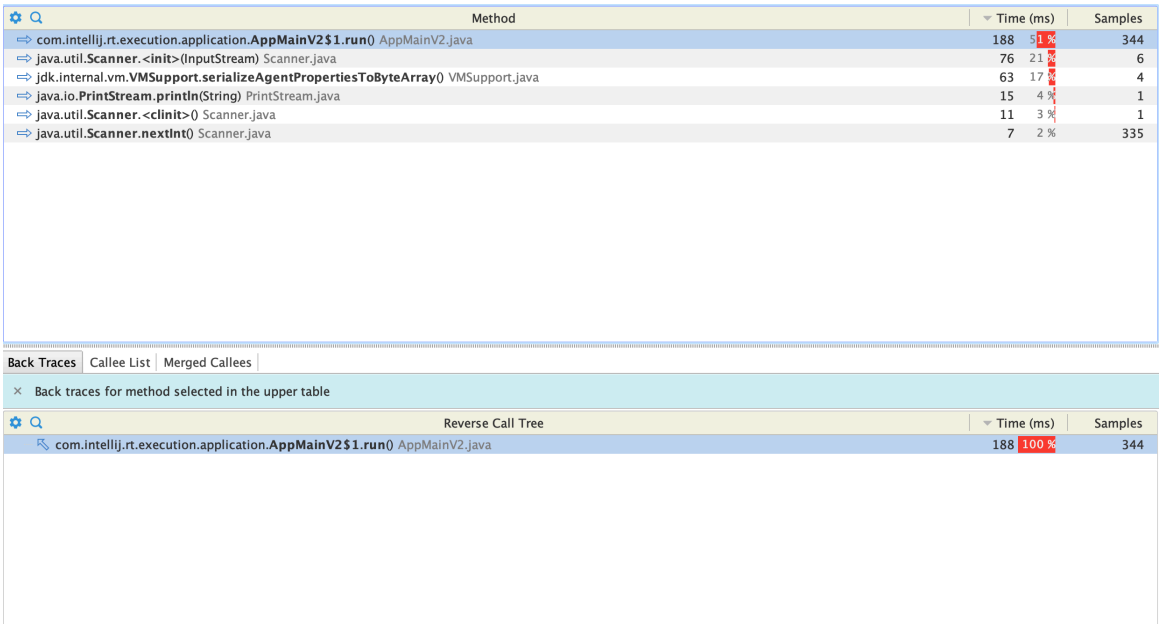
- بخش CPU

استفاده از CPU در هیچ‌زمانی به بالای ٪۵۰ نرسیده است و اگر اوج را بررسی کنیم هم اشاره‌ای به تابع temp نشده‌است و مربوط به تابع readBytes است:



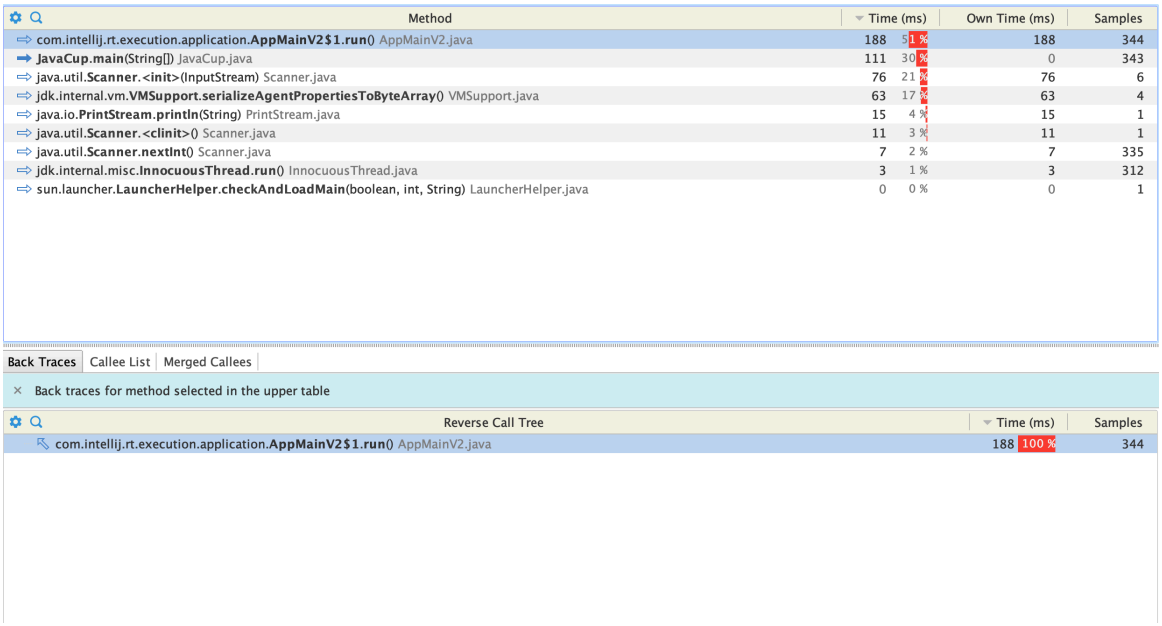
شکل ۱۱: CPU Usage

در بخش Hot spots هم نامی از تابع temp برده‌نشده‌است و java.util.Scanner استفاده‌ی بیشتری داشته‌است:



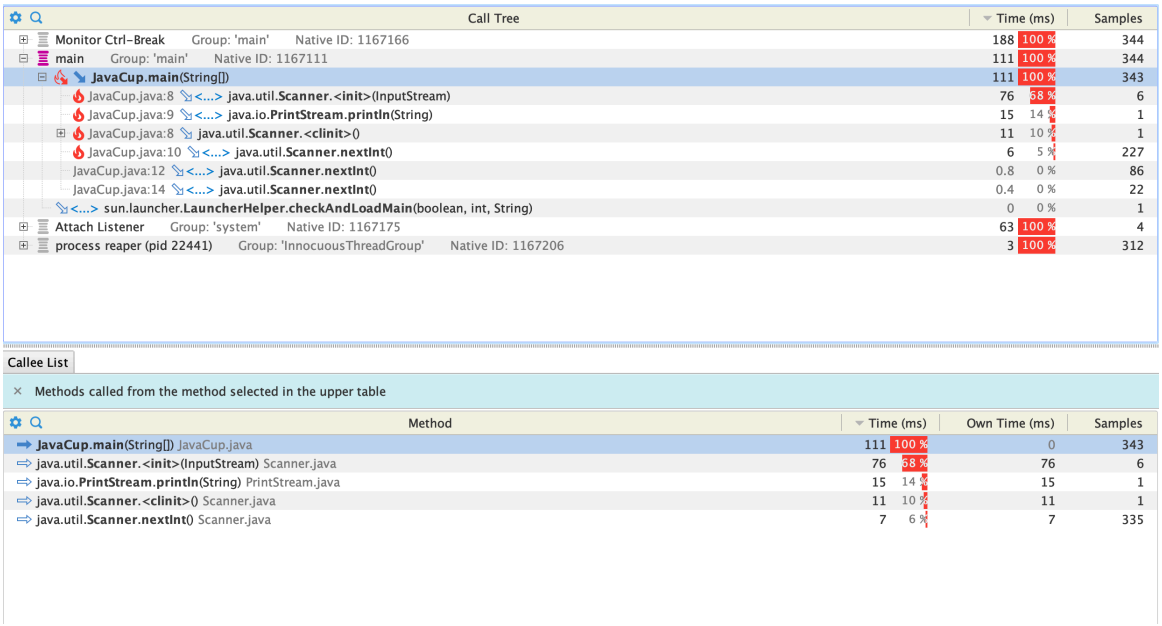
شکل ۱۲: Hot spots

نهایت استفاده‌ی تابع main ، ٪۳۰ است درحالیکه قبلا ٪۹۴ بود.



شکل ۱۳: Method lists

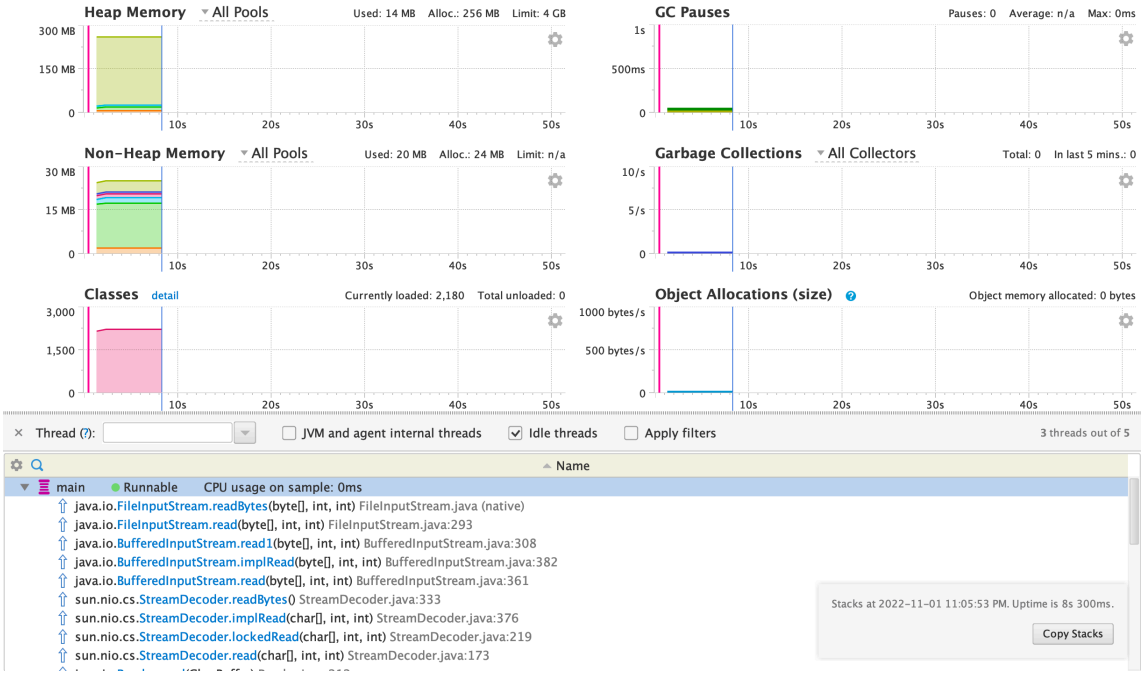
این بار در Call tree می‌بینیم که در تابع main بیشترین استفاده مربوط به java.util.Scanner بوده است (با ٪۶۸).



شکل ۱۴: Call tree

- بخش Memory

نمودار حافظه سیر صعودی نداشته‌است و بیشترین بخش آن مربوط می‌شود به تابع .readBytes

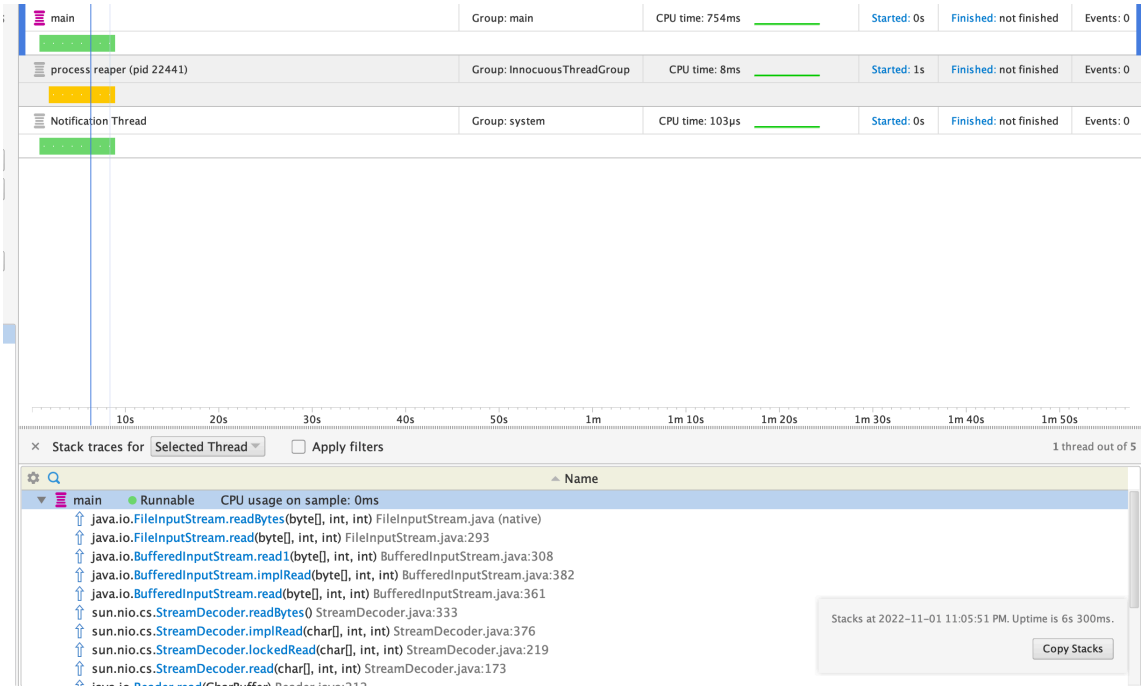


شکل ۱۵: Heap Memory

- بخش Threads

در نهایت در بخش ریسه‌ها هم می‌بینیم که برخلاف دفعه‌ی پیش، جایی نبوده‌است که مصرف CPU بطور قابل

توجه بالا بوده باشد و در زمان‌های مختلف تابع $java.io.FileInputStream.readBytes(byte[], int, int)$ بیشترین مصرف را داشته‌است.



شکل ۱۶: Threads

بنابراین تغییری که دادیم موفقیت‌آمیز بود. فایل‌های Snapshot به ترتیب با نام‌های FirstRun و SecondRun همراه با کد جدید Profiling test پیوست شده‌اند.

۲

Brute Force ۱.۲

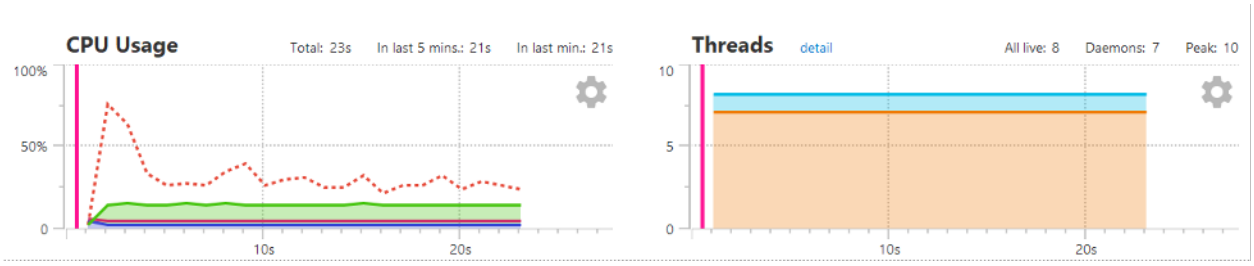
اول از الگوریتم brute force با اردر زمانی n^2 استفاده کردیم. همچنین اعداد آرایه رندوم انتخاب می‌شوند (به علت تعداد بالایشان) که کران بالایشان ۱۰۰۰۰۰۰ است و تعداد اعداد آرایه را هم ۱۰۰۰۰۰ درنظر گرفتیم (چون می‌خواستیم که تغییرات واضح‌تر دیده‌شوند که در تعداد اعداد پایین این اختلاف آنقدر مشخص نبود). پس از اجرای profiling در yourkit ، ترمینال به شکل:

```
[YourKit Java Profiler 2022.9-b171] Log file: C:\Users\najaf\.yjp\log\Sorting-21528.log
10 14 38 54 57 59 75 76 80 104 141 147 151 158 186 189 213 219 232 266 276 292 298 299 305 344 368 387 417 419 421 428 449 451 451 452 464 467
 488 514 531 533 533 544 550 556 563 564 566 583 583 588 600 601 611 611 621 623 637 645 665 668 679 685 716 720 722 723 727 736 738 747 768 774
 776 808 826 837 841 843 861 868 869 870 875 885 908 940 953 955 955 961 988 989 995 996 1001 1021 1022 1030 1036 1047 1047 1051 1068 1081 1092
 1105 1106 1137 1149 1173 1182 1187 1188 1193 1221 1236 1241 1242 1246 1256 1262 1303 1306 1306 1309 1311 1319 1334 1341 1343 1350 1355 1371 1376
  1393 1408 1418 1420 1423 1447 1448 1454 1462 1465 1471 1474 1474 1475 1495 1496 1510 1542 1564 1569 1573 1577 1592 1596 1604 1617 1619 1622
 1628 1641 1644 1646 1657 1694 1710 1711 1723 1735 1747 1763 1785 1803 1806 1809 1817 1821 1828 1845 1849 1859 1881 1881 1884 1888 1913 1916 1930
  1944 1961 2001 2006 2024 2038 2045 2072 2093 2101 2124 2171 2178 2178 2187 2194 2217 2219 2221 2241 2259 2259 2278 2282 2307 2307 2309 2316
 2316 2342 2346 2350 2358 2369 2374 2376 2400 2425 2437 2442 2443 2445 2451 2459 2468 2478 2491 2494 2494 2499 2510 2519 2555 2567 2579 2579 2586
  2598 2600 2604 2610 2617 2623 2628 2629 2629 2630 2652 2654 2657 2661 2682 2692 2694 2696 2711 2720 2730 2731 2734 2741 2752 2762 2783 2785
 2787 2791 2793 2799 2809 2811 2827 2837 2839 2844 2849 2856 2858 2872 2890 2899 2913 2916 2921 2968 2988 2993 2996 2999 3004 3042 3062 3066 3069
  3112 3121 3156 3157 3176 3179 3181 3184 3184 3208 3221 3221 3248 3254 3255 3257 3259 3293 3298 3308 3314 3318 3332 3340 3344 3356 3361 3372
 3391 3406 3431 3434 3438 3440 3517 3521 3537 3560 3566 3570 3604 3605 3609 3612 3627 3636 3641 3642 3667 3671 3672 3674 3686 3691 3693 3702 3714
```

شکل ۱۷: Output

و مصرف منابع به شکل زیر است:

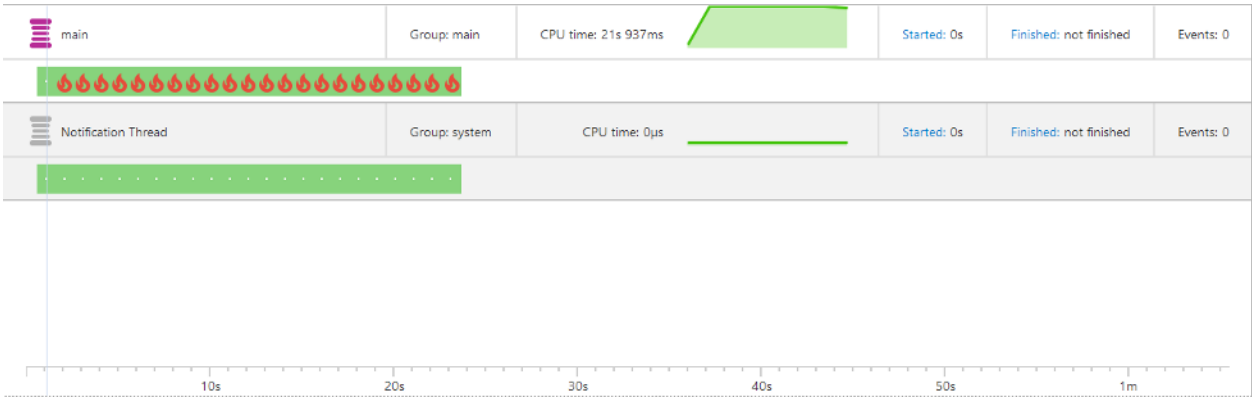
- بخش CPU



شکل ۱۸: CPU Usage

می‌بینیم که در طول اجرا سیستم پیک درگیری ۷۵ درصدی داشته است. تعداد ۸ ترد هم در حال اجرا بودند.

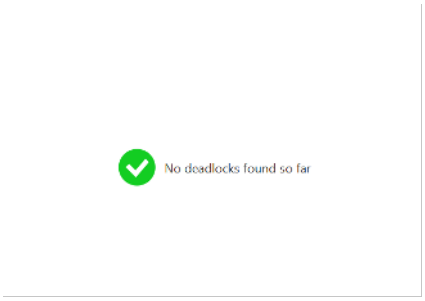
- بخش Threads



شکل ۱۹: Threads

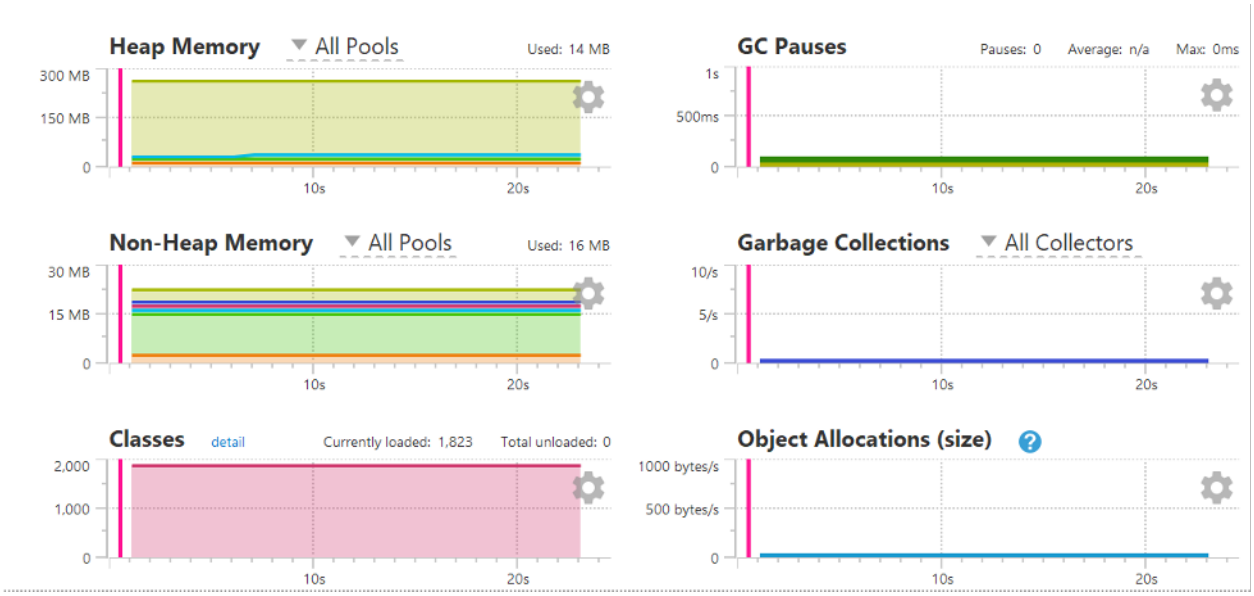
حدودا اجرای کد ۲۲ ثانیه طول کشیده که برای عملیات سورت کردن بسیار بالاست.

- بخش Deadlock

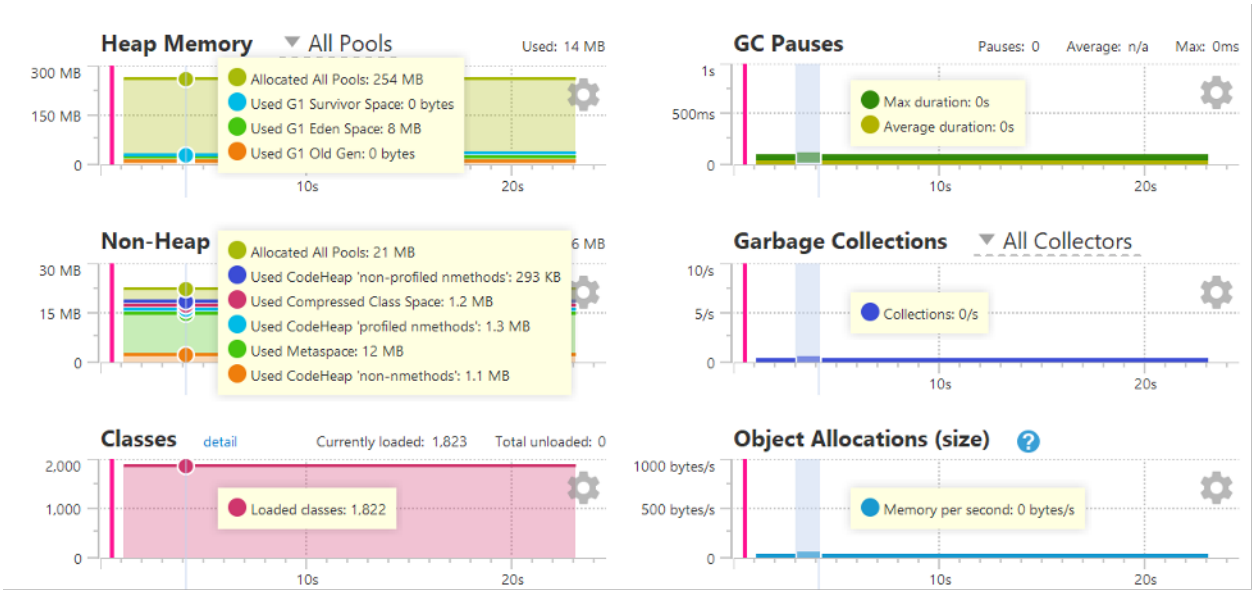


شکل ۲۰: Deadlock

- بخش Memory



شکل ۲۱: Memory



شکل ۲۲: Memory

به نسبت منابع زیادی برای یک عملیات ساده مانند سورتینگ استفاده شده است.

Java's Sort ۲.۲

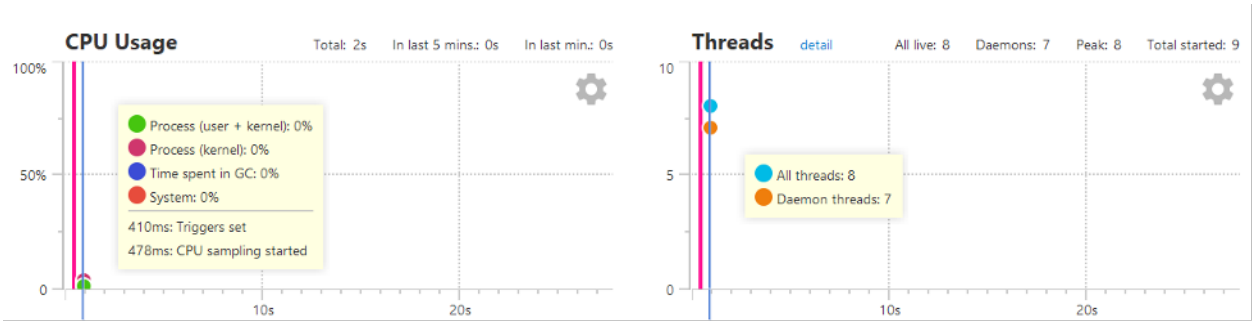
در مرحله بعد از الگوریتم سورت خود جاوا که از اردر زمانی $nlog(n)$ است استفاده می‌کنیم. این اردر زمانی برای سورت کردن بهترین اردر زمانی ممکن است. پس از اجرای profiling با Yourkit ترمینال به شکل:

```
[YourKit Java Profiler 2022.9-b171] Log file: C:\Users\najaf\.yjp\log\SortingFaster-12028.log
Modified arr[] : [999993, 999973, 999971, 999963, 999951, 999946, 999946, 999942, 999940, 999937, 999935, 999931, 999931, 999929, 999919, 999913,
  999910, 999909, 999884, 999865, 999852, 999831, 999817, 999810, 999808, 999804, 999801, 999787, 999786, 999780, 999768, 999733, 999729, 999700,
  999697, 999695, 999692, 999688, 999666, 999659, 999653, 999644, 999639, 999618, 999600, 999594, 999591, 999585, 999578, 999527, 999519, 999518,
  999514, 999513, 999508, 999508, 999506, 999484, 999472, 999464, 999462, 999458, 999450, 999427, 999422, 999416, 999388, 999374, 999362, 999354,
  999348, 999341, 999330, 999327, 999327, 999304, 999281, 999266, 999253, 999245, 999224, 999216, 999205, 999196, 999189, 999186, 999138, 999103,
  999088, 999088, 999085, 999080, 999059, 999039, 999021, 999005, 998971, 998964, 998939, 998930, 998929, 998927, 998913, 998889, 998880, 998832,
  998818, 998757, 998752, 998735, 998726, 998725, 998722, 998709, 998700, 998696, 998679, 998673, 998648, 998641, 998641, 998634, 998628, 998604,
  998596, 998593, 998593, 998586, 998583, 998580, 998568, 998547, 998545, 998544, 998528, 998517, 998517, 998512, 998486, 998465, 998464, 998431,
  998429, 998389, 998387, 998366, 998360, 998288, 998282, 998275, 998258, 998251, 998229, 998227, 998223, 998193, 998176, 998162, 998146, 998122,
  998118, 998109, 998105, 998092, 998078, 998074, 998072, 998067, 998001, 997991, 997984, 997980, 997966, 997954, 997943, 997933, 997904, 997903,
  997901, 997901, 997887, 997885, 997867, 997828, 997809, 997793, 997785, 997771, 997763, 997760, 997748, 997745, 997744, 997732, 997718, 997708,
  997705, 997702, 997698, 997687, 997664, 997658, 997653, 997653, 997648, 997646, 997637, 997626, 997592, 997590, 997582, 997578, 997577, 997560,
  997557, 997542, 997541, 997530, 997501, 997498, 997494, 997463, 997436, 997404, 997392, 997390, 997375, 997372, 997370, 997368, 997367, 997358,
  997333, 997326, 997324, 997317, 997307, 997273, 997258, 997248, 997229, 997227, 997215, 997183, 997163, 997128, 997124, 997124, 997116, 997113,
```

شکل ۲۳: Output

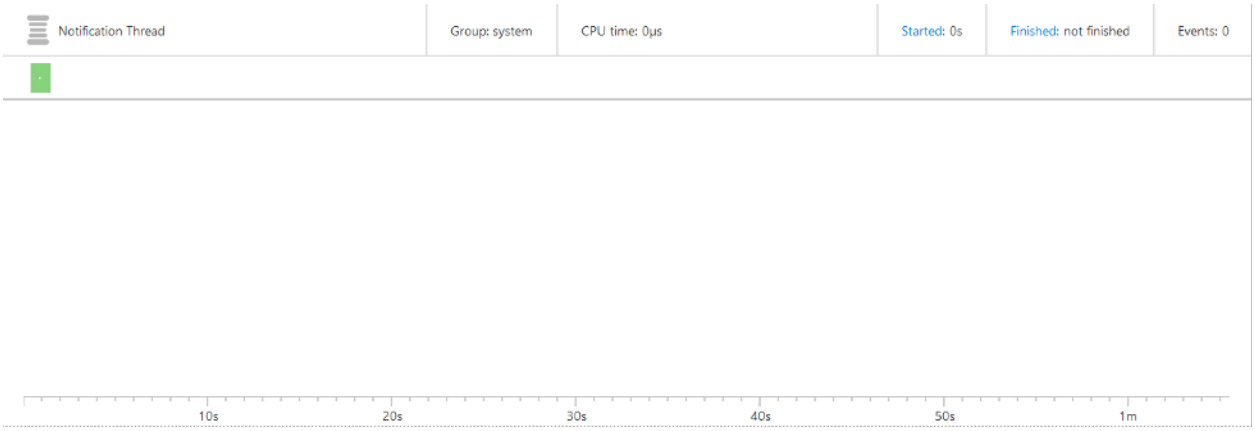
و مصرف منابع به صورت زیر است:

- بخش CPU



شکل ۲۴: CPU Usage

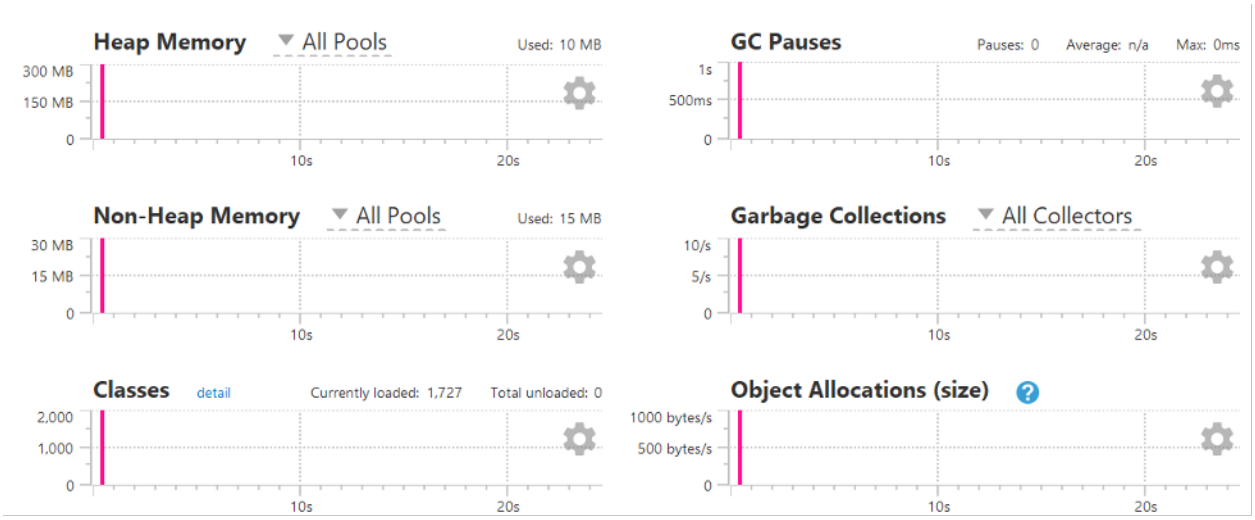
- بخش Threads



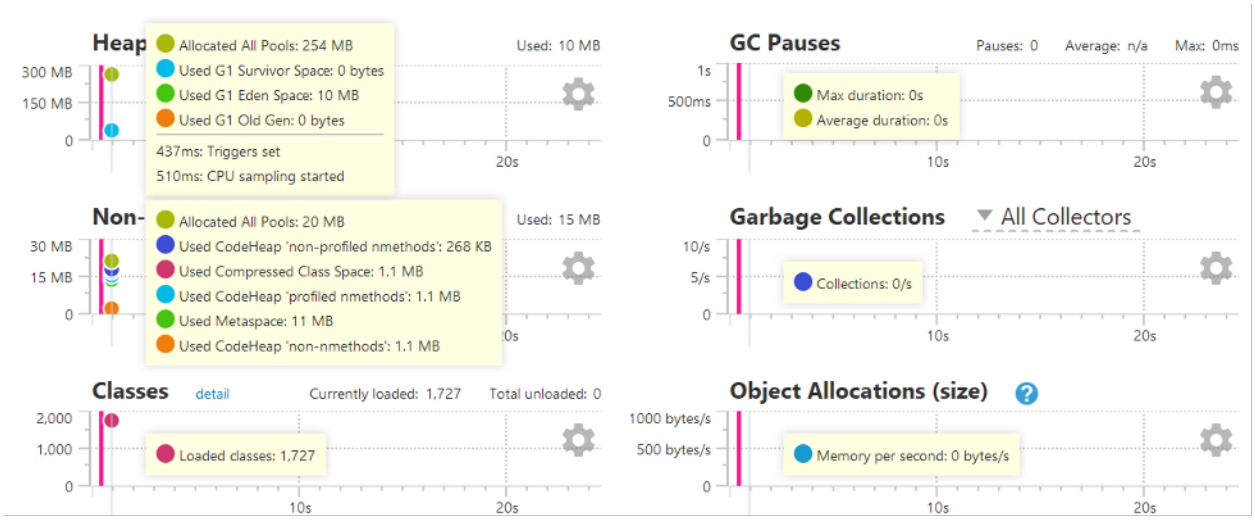
شکل ۲۵: Threads

زمانی که برای اجرای این کد صرف شده کمتر از ۱ میکروثانیه بوده و عملکرد عالی‌ای داشته‌است.

- بخش Memory



شکل ۲۶: Memory



شکل ۲۷: Memory

منابع مصرفی بسیار ناچیز بودند. این الگوریتم در مقایسه با الگوریتم پیشین عالی عمل و منابع ناچیزی مصرف کرده است. در الگوریتم اول هرچه تعداد اعداد آرایه بیشتر شود منابع مصرفی هم به شدت افزایش پیدا می‌کنند. بنابراین برای مدیریت منابع بسیار مهم است که از الگوریتم‌های مناسب استفاده کنیم.
کدهای این بخش در فولدر با نام $hw1\_2$ قرار داده‌شده‌اند. همچنین اسنپ‌شات‌های برنامه با نام‌های Sorting و SortingFaster ضمیمه‌شده‌اند.