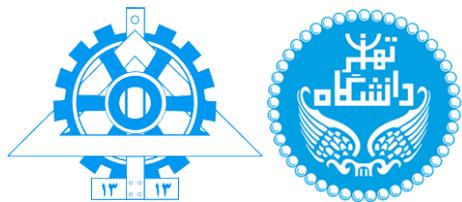


دانشگاه تهران
دانشکده مهندسی برق و کامپیوتر



سیستم های نهفته‌ی بی‌درنگ
تمرین کامپیوتری دوم: استفاده از سنسورهای تلفن همراه

هاتف رضایی 810100149، آرین فیروزی 810100196، شهزاد ممیز 810100272، محمد رضا محمد هاشمی 810100206

استاد: دکتر مدرسی، دکتر کارگهی

بهار 1404

سوالات مربوط به Perfetto

سوال 1: از وقتی که درخواست خواندن داده به یک سنسور داده شده تا گرفتن داده چه اتفاقاتی در سطح سیستم‌عامل افتاده است؟
توضیح خود را با خروجی Perfetto توضیح داده و توجیه کنید.

پاسخ:

در سطح سیستم‌عامل از لحظه فرآخوانی خواندن داده از یک سنسور تا تحویل نهایی آن به اپلیکیشن، مراحل زیر رخ می‌دهد:

مرحله 1. فرآخوانی در فضای کاربر (User Space)

1. متدهای بالا (مثل `SensorManager.readSensor()`) در لایه فریمورک اجرا می‌شود.

2. این متدهای طریق `Binder transaction` درخواست را به `SensorService` می‌فرستند.
در این لحظه، یک `thread` از `context switch` به `thread` بایندر رخ می‌دهد.

مرحله 2. گذر به فضای هسته (Kernel Space)

3. در لایه `SensorService` سنسور فرآخوانی `(read())` را روی فایل دستگاه (مثل `dev/sensorX`) انجام می‌دهد.
4. این باعث فرآخوانی سیسکال `sys_read` می‌شود.

مرحله 3. مسیر درون کرنل (Kernel)

5. درایور مربوطه وصل می‌کند.

6. تابع `driver->read()` در درایور سنسور اجرا می‌شود:

اغلب با فرآخوانی متدی `I2C` یا `SPI` برای درخواست انتقال داده.

چون سخت‌افزار ممکن است هنوز داده آماده نکرده باشد، درایور `thread` جاری را به حالت خواب (`sleep`) می‌فرستد و تابع `sched_switch` ظاهر می‌شود.

مرحله 4. آماده‌سازی داده در سخت‌افزار و IRQ

7. سنسور سخت‌افزاری پس از آماده‌سازی داده، یک `IRQ - Interrupt Request` به کنترلر می‌فرستد.

8. کنترلر با اینترپریت کرنل را مطلع می‌کند:

در UI Perfetto می‌بینیم که ایونت irq_handler روی یک هسته CPU اجرا شده.

این تابع اولیه پردازش را انجام داده و سپس با wake_up()، sensor thread متعلق را بیدار می‌کند.

مرحله ۵. ادامه خواندن و بازگشت

۹. پس از بیداری، sensor thread دوباره توسط شدولر (scheduler) روی یک هسته (مثلًا CPU3) اجرا می‌شود.

۱۰. در این مرحله متد sensor_read_channel() را فراخوانی می‌کند تا داده آمده را با استفاده از I²C/SPI به buffer کرنل منتقل کند.

۱۱. سپس user داده را به buffer فضای کاربر بر می‌گرداند و sys_exit_read copy_to_user اجرا می‌شود.

مرحله ۶. انتقال مجدد via Binder به اپلیکیشن

۱۲. SensorService با خواندن نتیجه sys_read، داده را در قالب یک Binder transaction دیگر به اپلیکیشن ارسال می‌کند.

۱۳. اپلیکیشن روی thread اصلی اجرا می‌شود تا داده سنسور را دریافت و پردازش کند.

: Context Switch

user thread ↔ kernel thread

بین IRQ handler و

: Scheduler

تصمیم می‌گیرد کدام thread و در چه هسته‌ای اجرا شود.

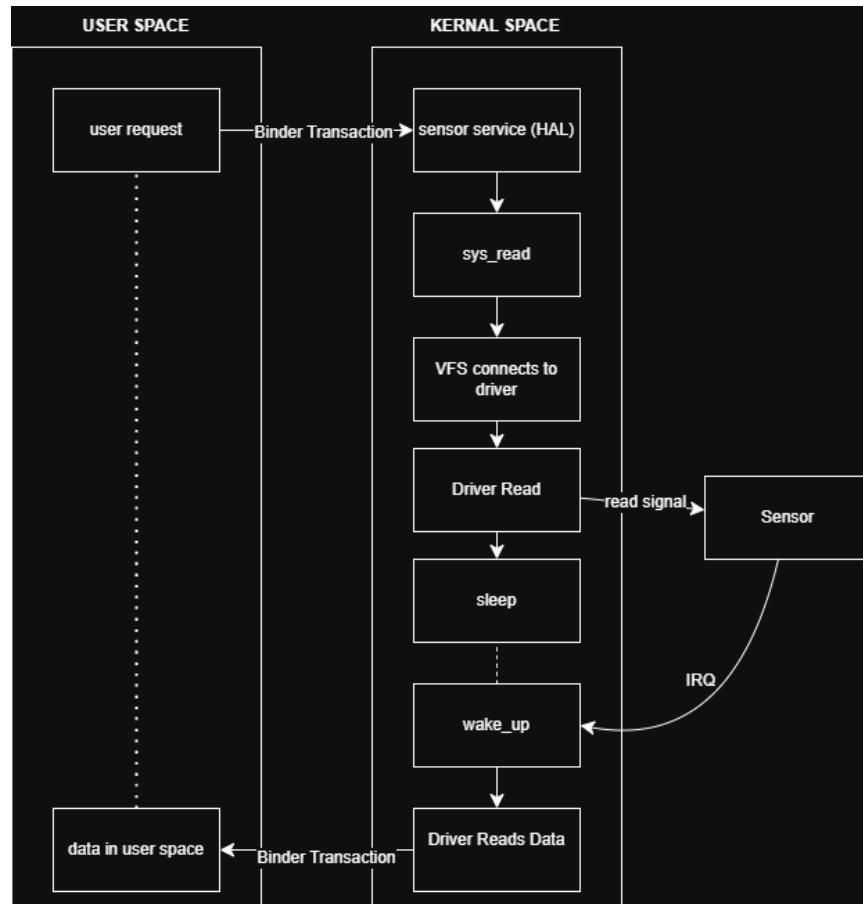
: CPU Frequency Scaling

ممکن است فرکانس CPU در طول اجرای IRQ و پردازش در این تغییر کند.

: Binder Overhead

دو بار ترانسیکشن (به و از SensorService) که latency افزوده ایجاد می‌کند.

۸. خلاصه کامل مسیر

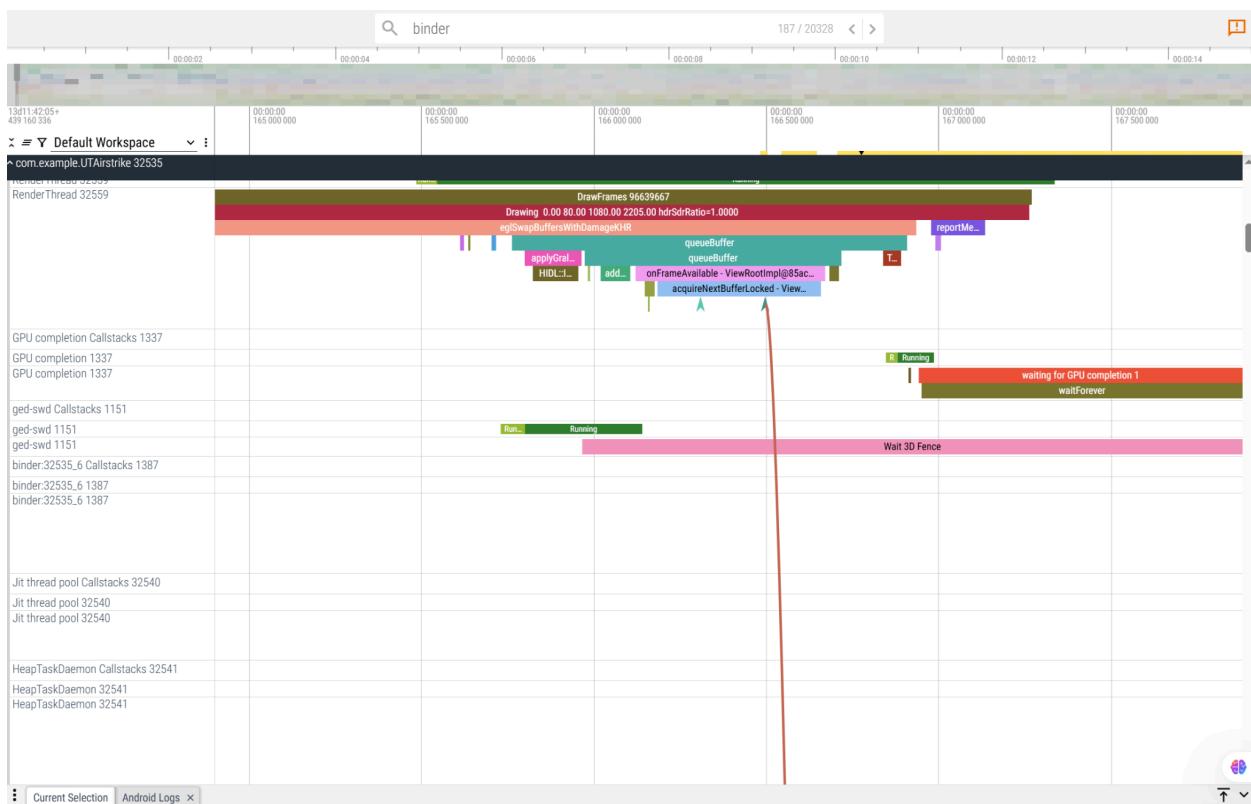
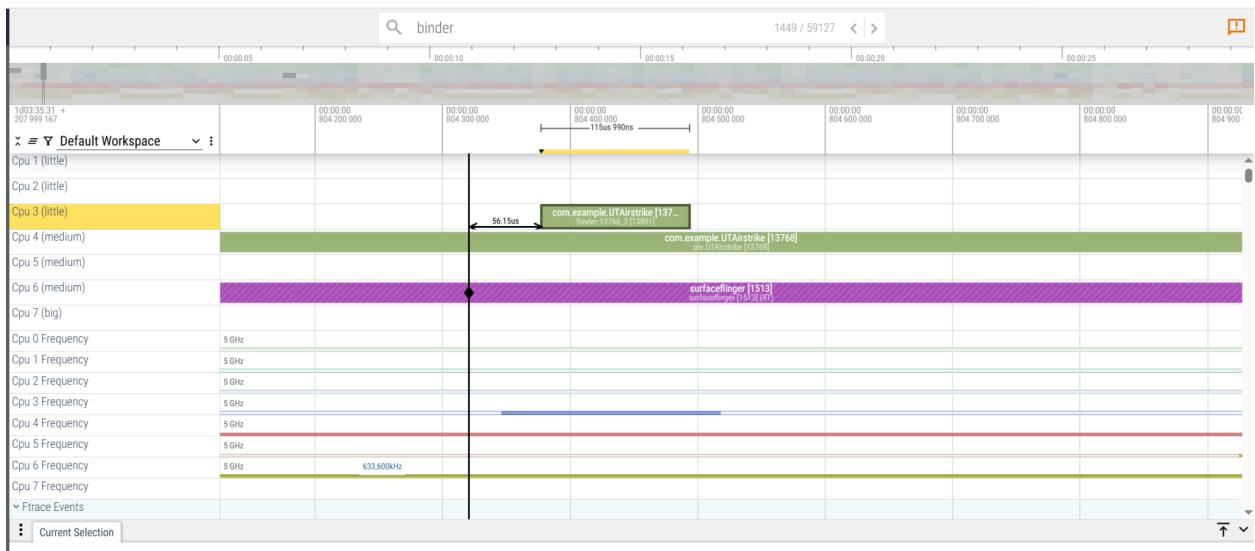


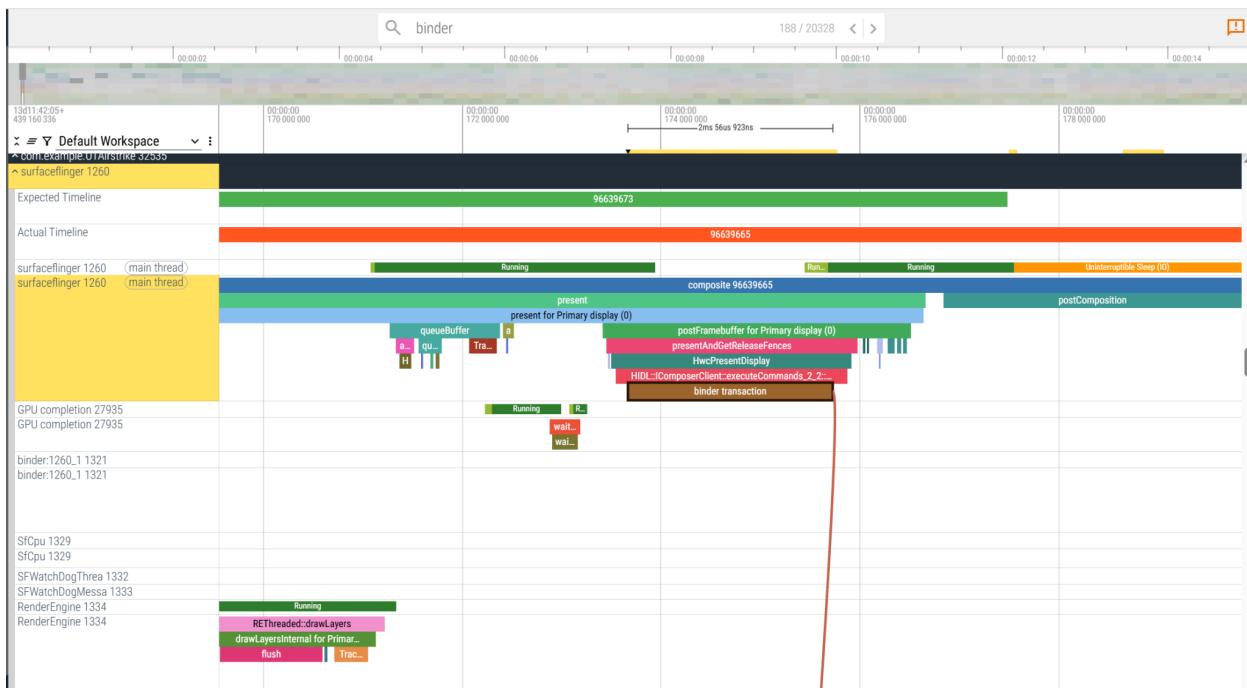
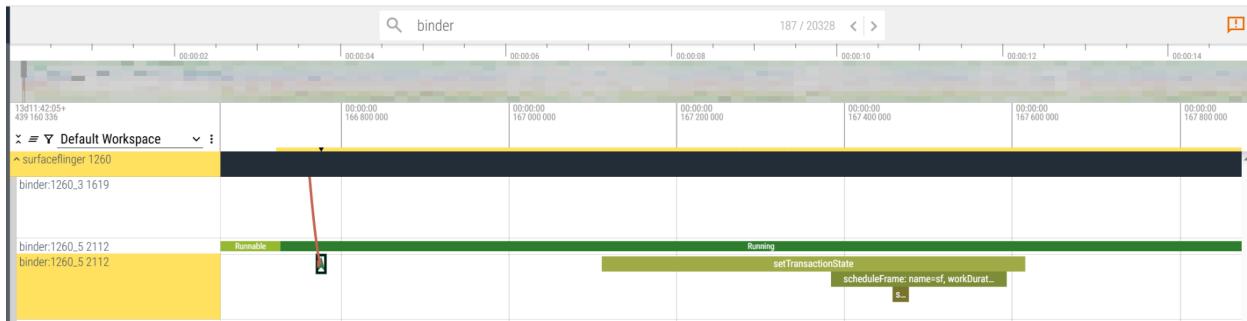
با این زنجیره اجرایی در سطح سیستم عامل، دقیقاً می‌توان دید که هر بخش از سخت‌افزار و نرم‌افزار چه نقشی دارد، در چه زمانی context switch رخ می‌دهد، چیپست چگونه IRQ را مدیریت می‌کند و داده سنسور چگونه به فضای کاربر بازمی‌گردد.

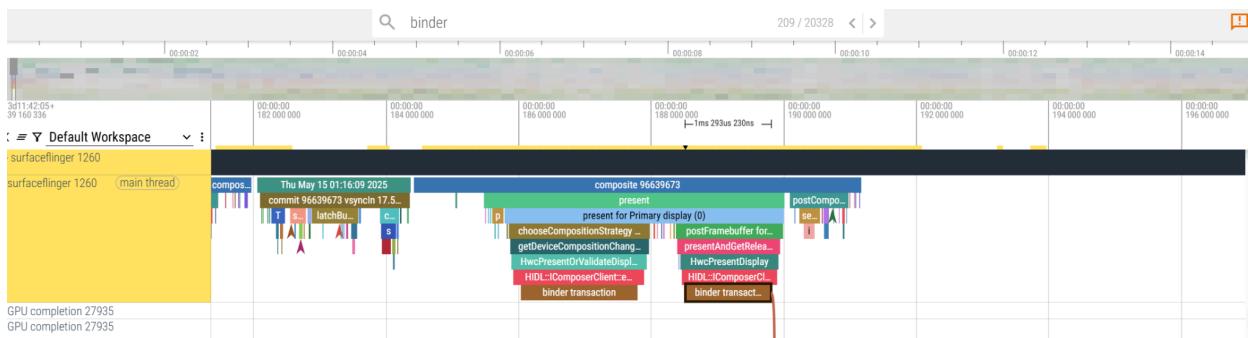
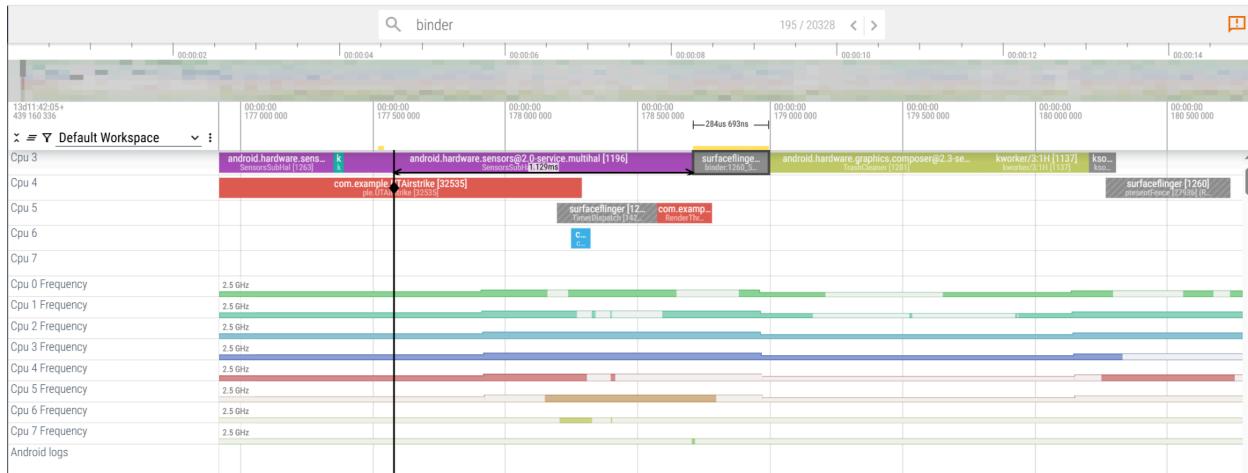
بعنوان مثال در پرفتو، SensorDataProcessing ، به اندازه i duration = 16.83 us بوده است. که پراسس را از زمانی که ریکویست داده می‌شود تا زمانی که دریافت شده است را اندازه گیری می‌کند.

ts=56.15 us

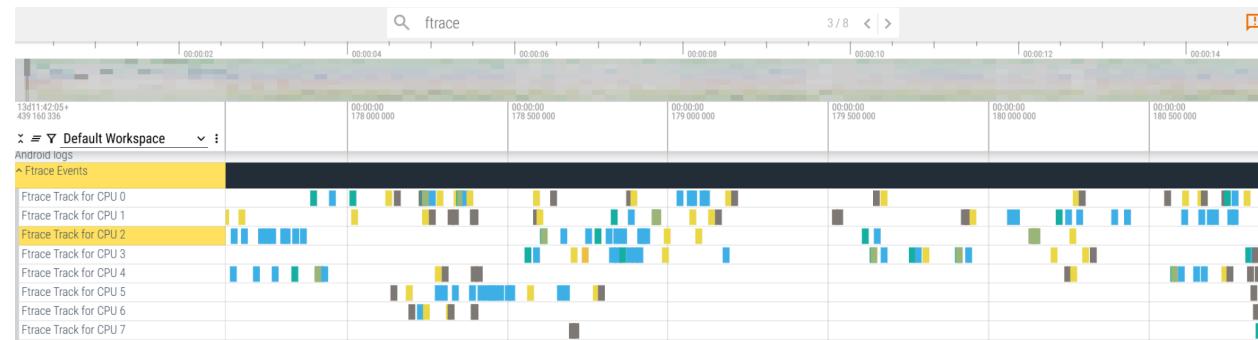
Binder timeline:



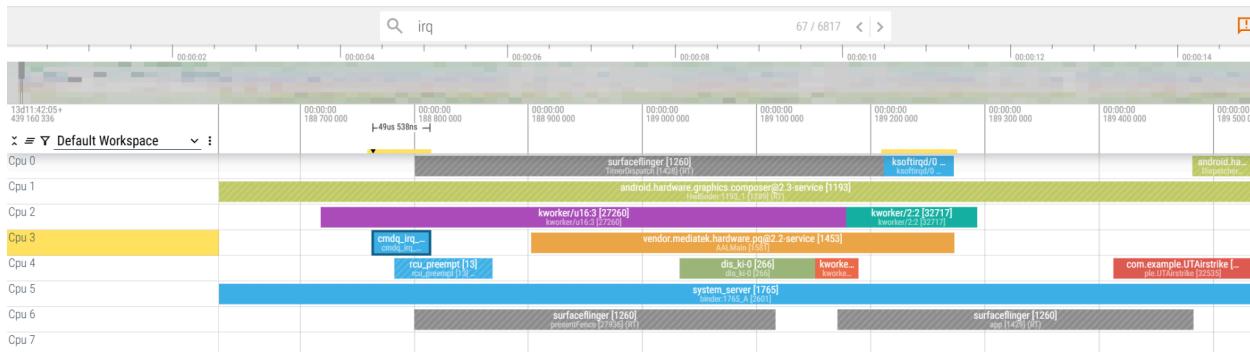




Ftrace timeline:

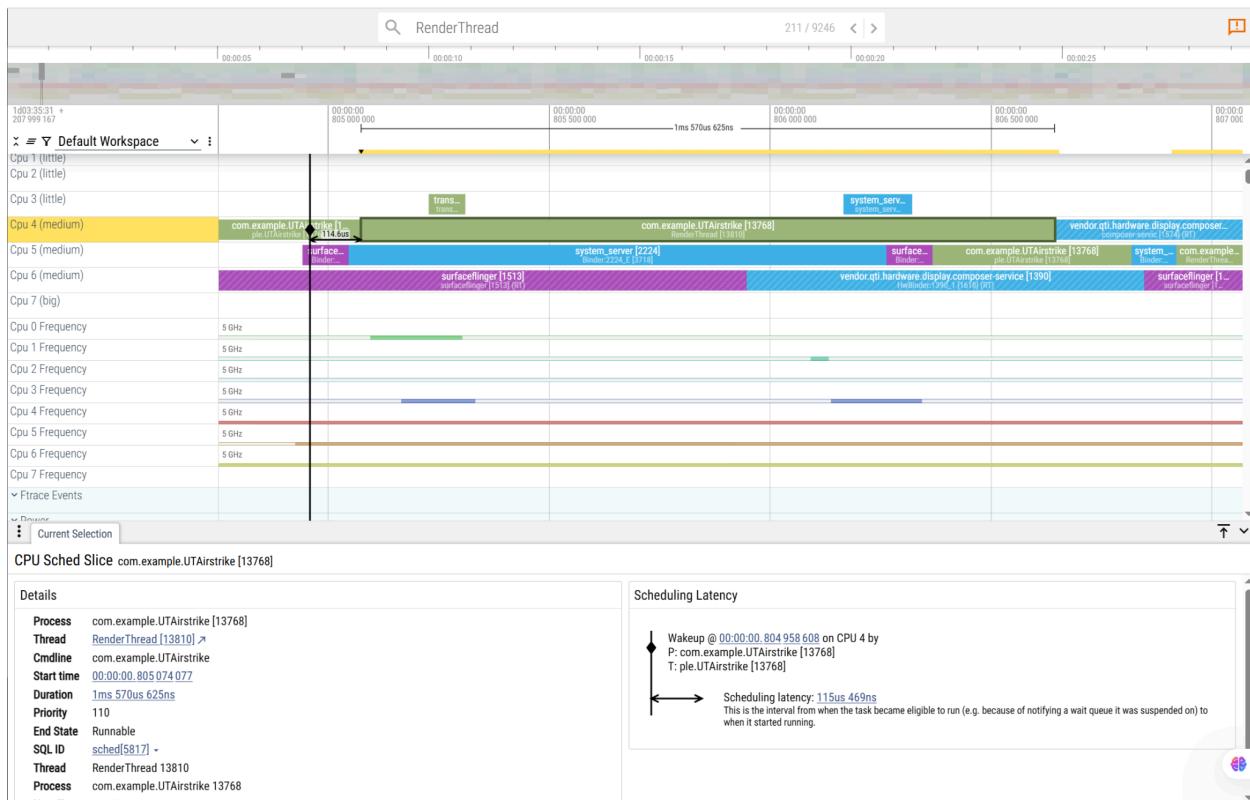


Irq:



Render thread:

114.6 us



سوال 2. آیا در فرآخونی های سیستمی، تعارض انتظار مشغول یک Thread استفاده از کتابخانه مربوط به گرافیک و بروزرسانی سنسور ها وجود دارد؟ پاسخ خود را چگونه توجیه می کنید؟

تعارض در یک سیستم عامل زمانی رخ میدهد که دو ترد بخواهند همزمان از یک ریسурс مشترک استفاده کنند و یک قفل را کنند. در این مورد میبینیم که چنین اتفاقی نمی افتد. در سطح سیستم عامل، فرآخونی های مرتبط با کتابخانه گرافیک و بروزرسانی

سنسورها روی مسیرهای مستقل (هر کدام با `driver lock` خاص خود) انجام می‌شوند و بنابراین هیچ‌گونه قفل مشترک یا تعامل مستقیمی بین آن‌ها وجود ندارد. تنها نقطه مشترک ممکن، اسکرولر است که تردها را برای بهره‌برداری از CPU زمان‌بندی می‌کند.

دلیل:

1. ساختار مجازی درایورها

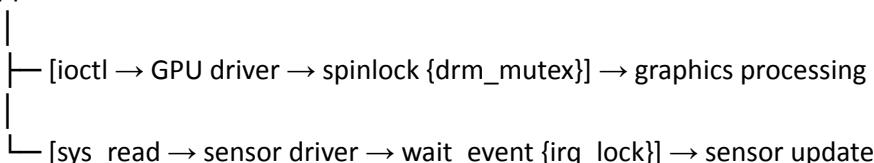
درایور GPU از طریق `ioctl` و DRM با هسته گرافیکی در تعامل است.

درایور سنسور با `sys_read` و `irq` کار می‌کند.

هر درایور از کلاس قفل مجازی استفاده می‌کند؛ بنابراین هیچ (contention) روی قفل واحد رخ نمی‌دهد.

مسیر اجرای مستقل در کرنل

Application



مشاهده می‌شود که `irq_lock` و `drm_mutex` دو قفل کاملاً جدا هستند.

تنها محل بالقوه تداخل زمان‌بندی CPU:

چون هر دو ترد گرافیک و ترد سنسور برای اجرا روی هسته‌های محدود CPU رقابت می‌کنند، ممکن است یک ترد تازمان خالی شدن CPU توسط ترد دیگر منتظر بماند؛ این همان کانتکست سویچ است، نه بلاک شدن روی یک قفل مشترک.

معیارهایی مثل `latency` و `runqueue length` نشان می‌دهند که این تداخل صرفاً در سطح scheduling است.

از نگاه `syscalls`، هیچ فرآخوانی‌ای وجود ندارد که عمدتاً «یک ترد گرافیکی» را تا پایان کار «یک ترد سنسور» بلاک کند.

همه تداخل‌های مشاهده شده (اگر باشد) ناشی از اولویت‌ها و زمان‌بندی CPU هستند، نه قفل یا resource contention داخلی کرنل.

با این استدلال مبتنی بر طراحی modular درایورها و مکانیزم scheduling هسته لینوکس، می‌توانیم مطمئن باشیم که در سطح هیچ تعارض قفل‌محور بین گرافیک و سنسورها وجود ندارد.

- بیشترین بار پردازش اشغال شده برای CPU مربوط به سنسورها است یا پردازش‌های گرافیکی؟

همانطور که در تصویر نیز دیده میشود، $cpu_sensor = 192.898172$ میلی ثانیه و $gpu_sensor = 4236.768251$ است و بیشترین بار پردازش برای پردازش گرافیکی است.

```

Enter query and press Cmd/Ctrl + Enter
1 WITH usage AS (
2     SELECT
3         SUM(CASE
4             WHEN LOWER(proc.name) LIKE '%sensor%'
5             THEN s.dur ELSE 0 END) AS sensor_cpu_ns,
6
7         SUM(CASE
8             WHEN LOWER(proc.name) LIKE '%gpu%'
9             OR LOWER(proc.name) LIKE '%gfx%'
10            OR LOWER(proc.name) LIKE '%surfaceflinger%'
11            OR LOWER(proc.name) LIKE '%render%'
12            THEN s.dur ELSE 0 END) AS graphics_cpu_ns
13    FROM sched AS s
14   JOIN thread AS th ON s.utid = th.utid
15  JOIN process AS proc ON th.upid = proc.upid
16 )
17
18
19 SELECT
20     usage.sensor_cpu_ns / 1e6 AS sensor_cpu_ms,
21     usage.graphics_cpu_ns / 1e6 AS graphics_cpu_ms,
22     CASE
23         WHEN usage.sensor_cpu_ns > usage.graphics_cpu_ns THEN 'Sensors used more CPU'
24         ELSE 'Graphics used more CPU'
25     END AS comparison_result
26 FROM usage;
27

```

Query result (1 rows) - 80.7ms		
sensor_cpu_ms	graphics_cpu_ms	comparison_result
192.898172	4236.768251	Graphics used more CPU

Showing rows 1 to 1 of 1 |< Prev |> Next | Copy |

Query history (3 queries)

سوال 3. مدت زمانی طول می کشد تا تغییرات اسکن شده از سطح بر اساس مقداری که از سنسور خوانده شده است، روی صفحه نمایش ظاهر شود؟

حدود 88 میکرو ثانیه است. که از انتهای SensorDataProcessing تا ابتدای UIUpdate duration میباشد. که آن 16.83 میکرو ثانیه است.

بخش Event های مربوط به UI Update

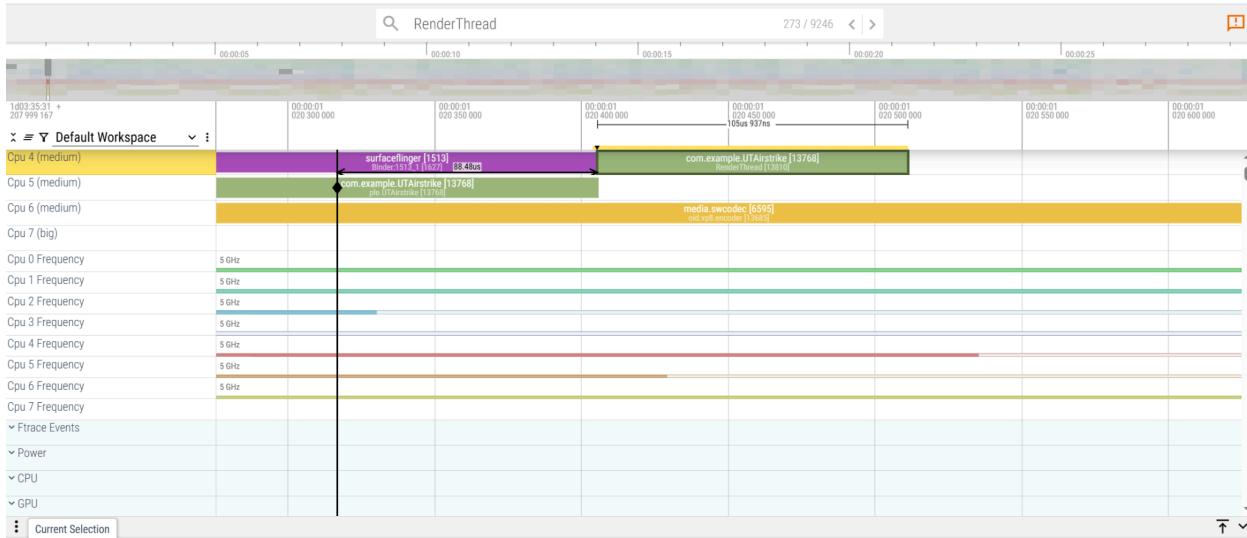
قسمتی که داده های پردازش شده توسط UI برای رندر صفحه استفاده می شوند.

معمولًاً با نام هایی مثل UIUpdate با RenderThread یا View.draw در تایملین مشخص است.

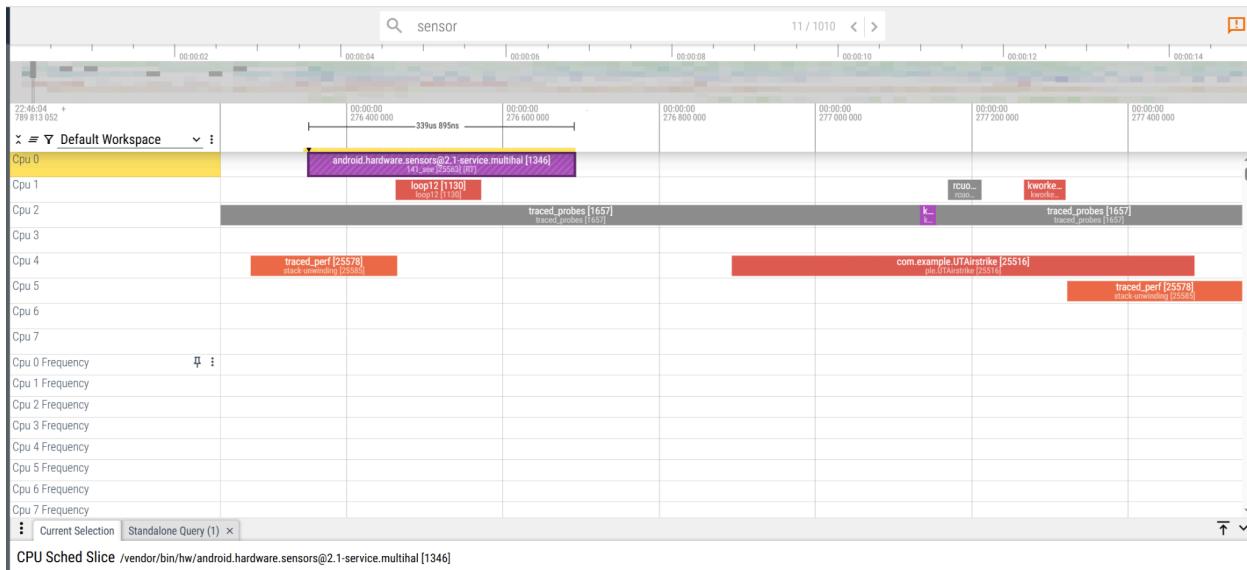
شروع بروزرسانی صفحه نمایش را نشان می دهد.

در شکل زیر نیز RenderThread را میبینیم که حدود 88.48 میکرو ثانیه طول کشیده است Scheduling latency یعنی تاخیر بین آماده شدن thread برای اجرا (مثل بعد از بیدار شدن از حالت خواب) و شروع واقعی اجرای آن روی CPU.

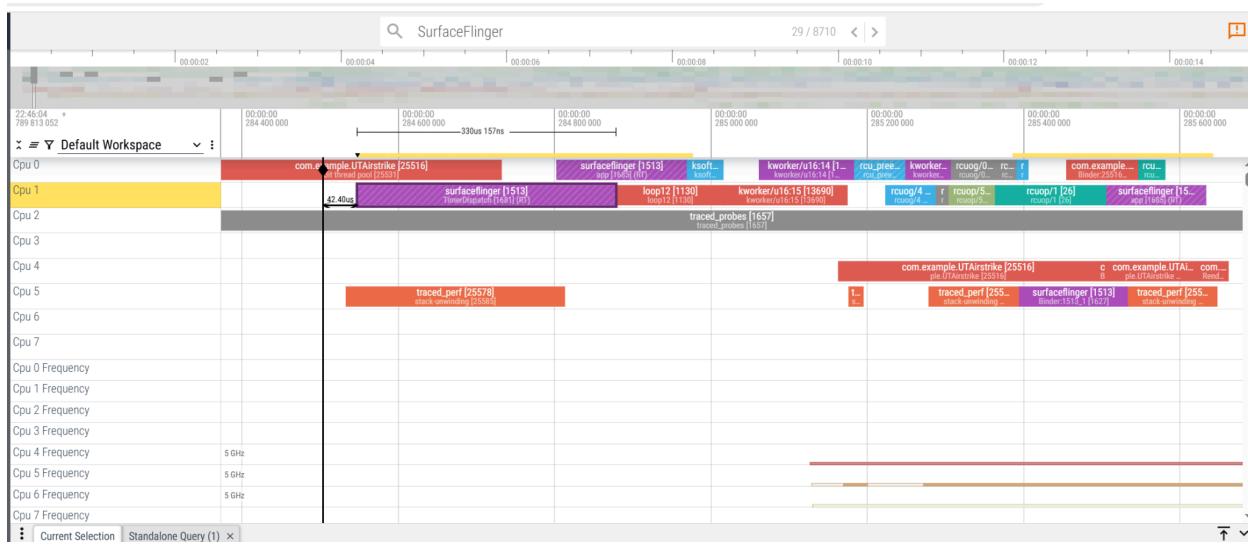
این تاخیر ۸۸ میکرو ثانیه نشان می دهد thread RenderThread حدود ۸۸ میکرو ثانیه منظر مانده تا نویتش برسد.

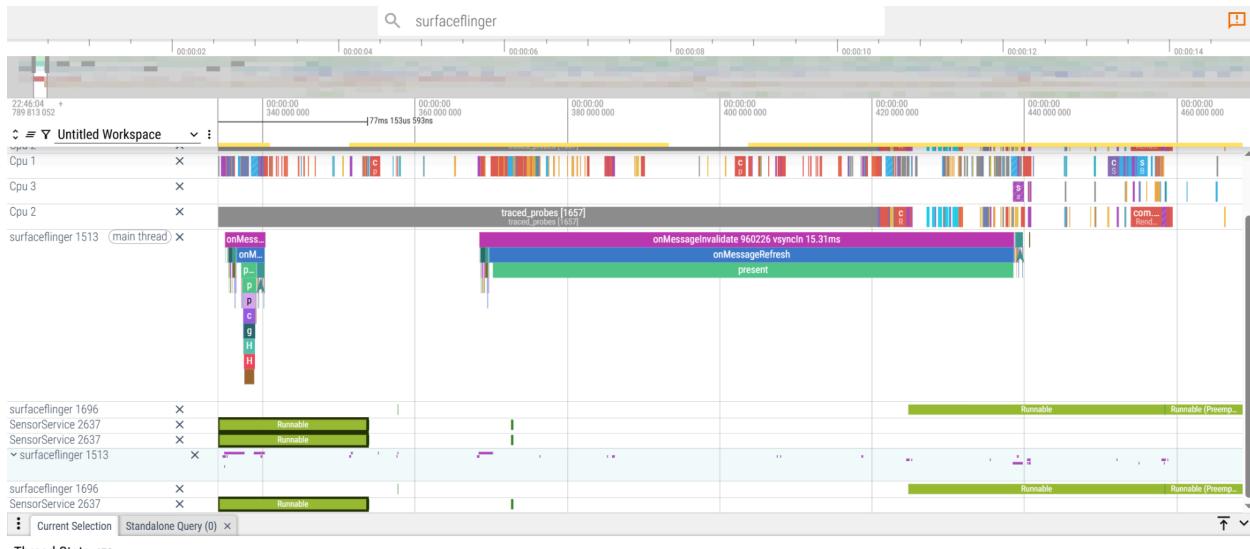


sensor:



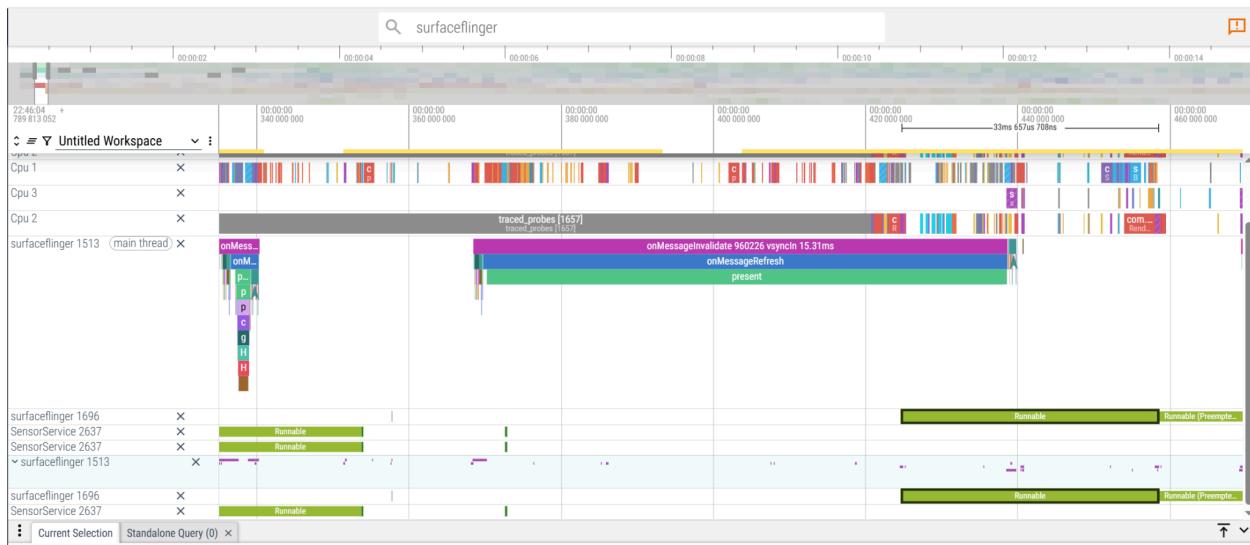
Surfaceflinger:





Thread State 670

Details		Related thread states	
Start time	00:00:00.276 630 703	Next state	Running for 201us 928ns ↗
Duration	77ms 153us 593ns	Woken by (maybe interrupt)	141_see [25563] /vendor/bin/hw/android.hardware.sensors@2.1-service.multihal [1346] ↗
State	Runnable	Critical path lite	
Process	system_server [2224]		
Thread	SensorService [2637]		
SQL ID	thread_state[670] -		



Thread State 2495

Details		Related thread states	
Start time	00:00:00.424 775 338	Previous state	Sleeping for 66ms 989us 896ns ↗
Duration	33ms 657us 708ns	Next state	Running for 54us 480ns ↗
State	Runnable	Woken by (maybe interrupt)	Binder:1513_1[1627] /system/bin/surfaceflinger [1513] ↗
Process	/system/bin/surfaceflinger [1513]	Critical path lite	
Thread	surfaceflinger [1696]		
SQL ID	thread_state[2495] -		



```

Enter query and press Cmd/Ctrl + Enter
1 WITH sensor_event AS (
2   SELECT ts AS sensor_ts
3   FROM slice
4   WHERE name LIKE '%sensor%'
5   ORDER BY ts LIMIT 1
6 ),
7
8
9 frame_present AS (
10  SELECT ts AS present_ts
11  FROM slice
12  WHERE name = 'DisplayFrame'
13  ORDER BY ts LIMIT 1
14
15 )
16
17
18 SELECT
19   (present_ts - sensor_ts) / 1e6 AS latency_ms
20 FROM sensor_event, frame_present
21

```

Query result (0 rows) - 20.7ms WITH sensor_event AS (SELECT ts AS sensor_ts FROM slice WHERE name LIKE '%sensor%' ORDER BY ts LIMIT 1), frame_present AS (SELECT ts AS pre... Showing rows 1 to 0 of 0 ⏪ Prev ⏩ Next ⏷ Copy ⏷

- مدت زمانی را که تابع فیلتر در سطح سی پی یو اجرا میشود را به صورت میانگین به دست آورید.

● FilterFunction به طور میانگین 10234.76 نانو ثانیه طول میکشد. و یک پراسس خوب را نشان میدهد.

```

Enter query and press Cmd/Ctrl + Enter
1 SELECT name, AVG(dur)/1e6 AS avg_duration_ms, COUNT(*) AS count
2 FROM slice
3 WHERE name LIKE '%filter%' OR name LIKE '%compute%' OR name LIKE '%process%'
4 GROUP BY name
5 ORDER BY avg_duration_ms DESC;
6

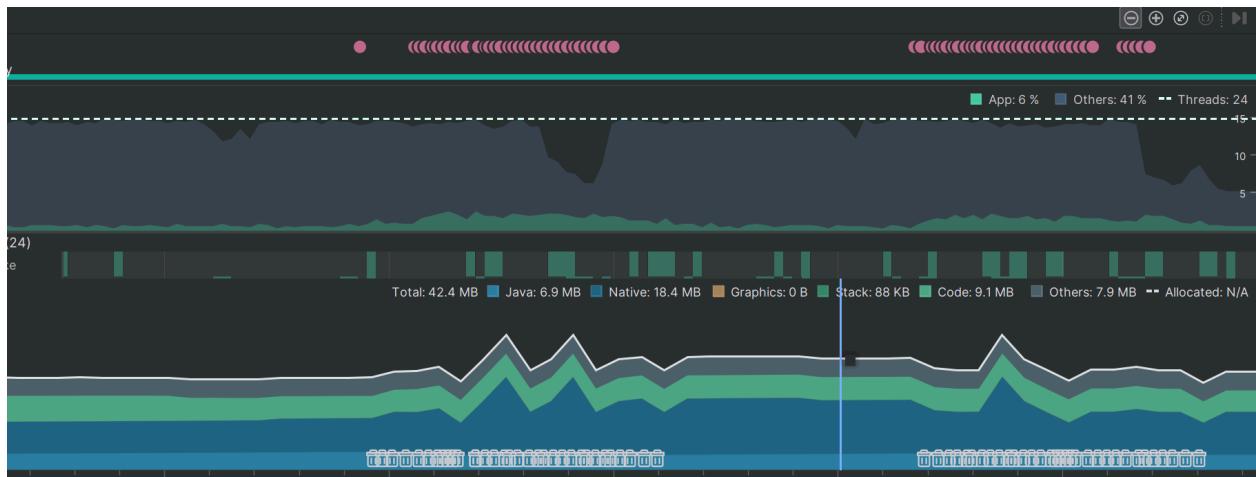
```

Query result (10 rows) - 113.4ms SELECT name, AVG(dur)/1e6 AS avg_duration_ms, COUNT(*) AS count FROM slice WHERE name LIKE '%filter%' OR name LIKE '%compute%' OR na... Showing rows 1 to 10 of 10 ⏪ Prev ⏩ Next ⏷ Copy ⏷

name	avg_duration_ms	count
Broadcast dispatched from (process unknown) android.intent.action.BATTERY_CHANGED	12.137084	1
setProcessGroup com.google.android.gms.persistent to -1	5.979532	1
JIT compiling int android.view.ViewRootImpl\$ViewPostTimeInputStage.processPointerEvent(android.view.ViewRootImpl\$QueuedInputEvent) (kind=Baseline)	2.284583	1
setProcessGroup com.sec.android.provider.badge to -1	1.820469	1
Process mark stacks and References	1.74157975	12
processNextBufferLocked	0.4958116125401929	1244
Broadcast in queue from (process unknown) android.intent.action.BATTERY_CHANGED	0.201927	1

بقیه نمودار ها:

View Live Telemetry:

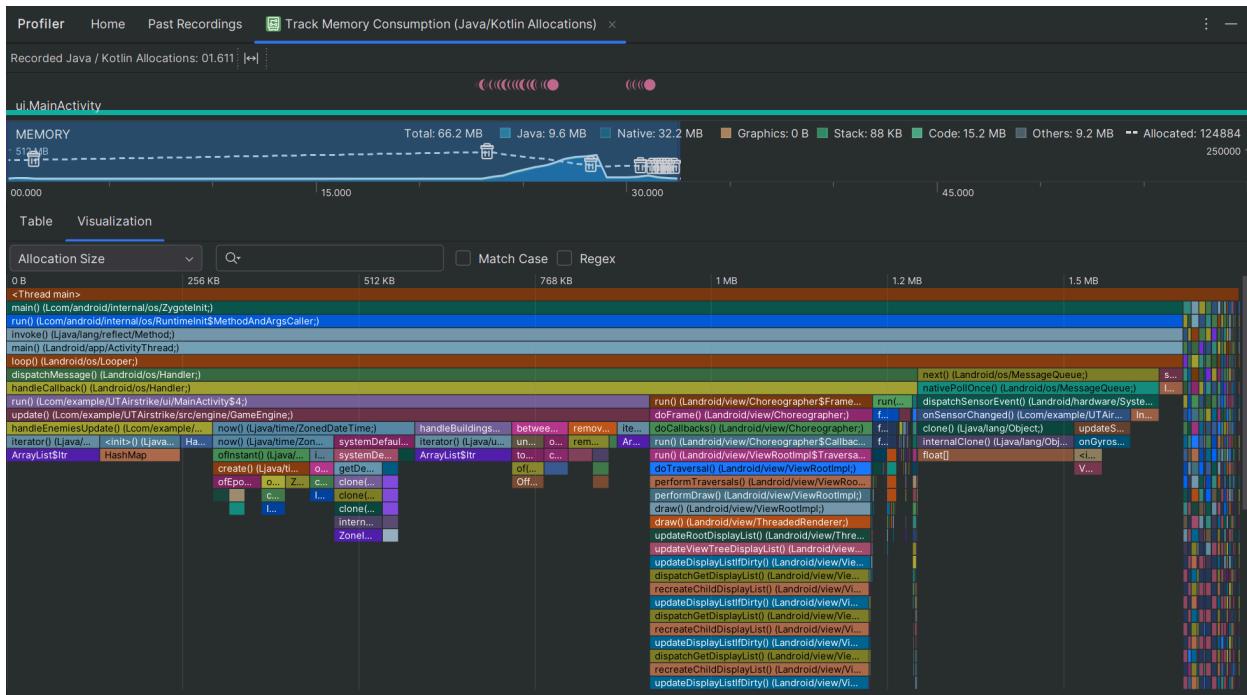


Analyze Memory Usage:

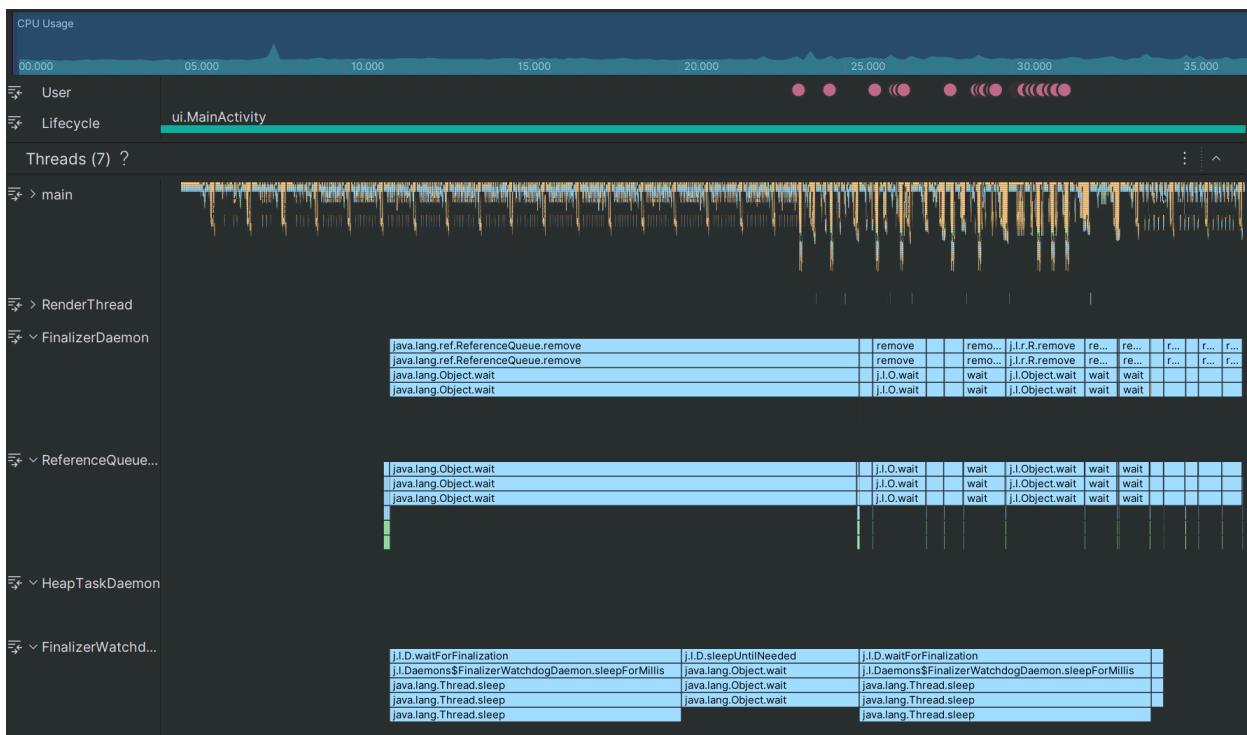
Analyze Memory Usage (Heap Dump)

Class Name	Allocations	Native Size	Shallow Size	Retained Size
app heap	34,618	3,494,831	978,925	563,555
EmojiCompat (androidx.emoji2.text)	1	0	51	294,240
MaterialButton (com.google.android.material.button)	2	0	1,836	5,337
MaterialTextView (com.google.android.material.textview)	1	0	881	5,085
View (android.view)	1	0	481	4,953
FitWindowsLinearLayout (androidx.constraintlayout.widget)	1	0	708	4,632
ConstraintLayout (androidx.constraintlayout.widget)	1	0	740	4,590
ContentFrameLayout (androidx.appcompat.widget)	1	0	696	4,520
DecorView (com.android.internal.policy)	1	0	896	4,379
LinearLayout (android.widget)	1	0	701	4,369
RelativeLayout (android.widget)	1	0	681	4,305
FrameLayout (android.widget)	1	0	664	4,198
ViewStubCompat (androidx.appcompat.widget)	1	0	504	3,624
AircraftView (com.example.UTAirstrike.ui)	1	0	508	3,603
BuildingView (com.example.UTAirstrike.ui)	1	0	508	3,603
EnemyView (com.example.UTAirstrike.ui)	1	0	512	3,600
ViewStub (android.view)	1	0	504	3,557
MainActivity (com.example.UTAirstrike.ui)	1	0	365	3,263
Intent (android.content)	1	0	64	2,620
AppOpsManager (android.app)	1	0	28	1,340
Message (android.os)	19	0	1,216	1,125
ApplicationPackageManager (android.app)	3	0	147	1,121
MotionEvent (android.view)	1	0	22	1,095

Track Memory Consumption:



Find CPU HotSpots:



Analysis All threads **RenderThread X**

Summary Top Down Flame Chart Bottom Up Events

Time Range 00:00.000 - 00:37.147
Duration 37.15 s
Data Type Thread
ID 13136

Longest running events (top 10)

Start Time	Name	Wall Duration	Wall Self Time	CPU Duration	CPU Self Time
00:23.418	callOnFinished	7.2 ms	84 µs	869 µs	55 µs
00:23.419	post	7.02 ms	60 µs	716 µs	40 µs
00:23.419	sendMessageDe...	6.89 ms	83 µs	616 µs	60 µs
00:23.419	sendMessageAt...	6.77 ms	34 µs	508 µs	35 µs
00:23.419	enqueueMessage	6.74 ms	91 µs	473 µs	15 µs
00:23.419	getUid	6.56 ms	53 µs	458 µs	60 µs
00:23.419	get	6.49 ms	93 µs	398 µs	91 µs
00:23.419	getMap	5.61 ms	5.61 ms	126 µs	126 µs
00:25.703	callOnFinished	2.74 ms	1.05 ms	2.75 ms	130 µs
00:31.827	callOnFinished	2.12 ms	935 µs	1.23 ms	49 µs

Profiler Home Past Recordings Find CPU Hotspots (Java/Kotlin Method Recording) ×

Analysis All threads **RenderThread X**

Summary Top Down Flame Chart Bottom Up Events

CPU Usage

Interaction

- User ui.MainActivity
- Lifecycle
- Threads (7) ?
- > main
- > RenderThread
- > FinalizerDaemon
- > ReferenceQueue...
- > HeapTaskDaemon
- > FinalizerWatchd...
- > Binder:13105_3

Clear thread/event selection

Name	Total (µs)	%	Self (µs)	%	Children (...)	%
RenderThread() ()	9,411,562	100.00	9,315,833	98.98	95,729	1.02
callOnFinished() (android.view.RenderNodeAnir	95,729	1.02	9,182	0.10	86,547	0.92
post() (android.os.Handler)	79,654	0.85	6,519	0.07	73,135	0.78
sendMessageDelayed() (android.os.Hand	66,271	0.70	5,587	0.06	60,684	0.64
sendMessageAtTime() (android.os.Har	57,638	0.61	3,586	0.04	54,052	0.57
enqueueMessage() (android.os.Han	54,052	0.57	6,676	0.07	47,376	0.50
getUid() (android.os.ThreadLoca	36,688	0.39	5,551	0.06	31,137	0.33
get() (java.lang.ThreadLocal)	29,656	0.32	6,052	0.06	23,604	0.25
access\$000() (java.lang.Th	15,046	0.16	3,096	0.03	11,950	0.13
getMap() (java.lang.Threa	6,994	0.07	6,994	0.07	0	0.00
currentThread() (java.lang.	1,564	0.02	1,564	0.02	0	0.00
intValue() (java.lang.Integer)	1,481	0.02	1,481	0.02	0	0.00
enqueueMessage() (android.os.N	9,302	0.10	5,707	0.06	3,595	0.04
setAsynchronous() (android.os.N	1,386	0.01	1,386	0.01	0	0.00
uptimeMillis() (android.os.SystemClock	3,046	0.03	3,046	0.03	0	0.00
getPostMessage() (android.os.Handler)	6,864	0.07	3,587	0.04	3,277	0.03
<init>() (android.view.-\$\$Lambda\$1kvF4Juy	5,220	0.06	3,488	0.04	1,732	0.02
requireNonNull() (java.util.Objects)	1,673	0.02	1,673	0.02	0	0.00

