دانشگاه تهران
دانشکده‌ی مهندسی برق و کامپیوتر

گزارش نهایی پروژه سیستم‌های سایبر- فیزیکی

سیستم مانیتورینگ انرژی و کارایی
(Energy and Performance Monitoring System)

گروه:

فاطمه جهانگیری
مائده داودزاده
سارا صفاری

تاریخ:
دوشنبه ۱۰–۴–۹۸

استاد:
دکتر مهدی کارگهی
دکتر مهدی مدرسی

1397-1398

فهرست مطالب

1- شرح کلی پروژه

اهداف پروژه

امروزه با فراگیر شدن برنامه های گوشی های همراه هوشمند و کاربردی بودن آنها در زندگی روزمره ما، علاوه بر نیاز به اینکه کار های مطلوب ما به صورت دلخواه ما انجام شود و خروجی مطلوبی برای ما به همراه داشته باشند و این سرویس ها در مدت زمان مورد قبولی آماده شوند و در اختیار کاربر قرار گیرند، لازم است از نظر مصرف انرژی نیز بهینه باشند زیرا این دستگاه ها که متکی به انرژِی باتری هایشان هستند باید بتوانند مدت زمان معقولی را بدون نیاز به شارژ مجدد و با کمترین اثر بر روی کارایی شان مشغول به کار باشند.

برای بهینه سازی در این حوزه ، که بعد از پیشرفت های بسیار در بعد کارایی و زمان اجرای نرم افزارها، نیاز به آن احساس می شود لازم است که برنامه ها از نظر مصرف انرژِی مورد بررسی قرار گیرند تا اثر اجرا شدنشان بر روی سطح انرژی دستگاه مشاهده، ثبت و تحلیل گردد تا به وسیله آن به نسل بهینه ای تری دست یافت.

همین انگیزه ای برای بررسی های بیشتری در این حوزه است که تا کنون به این حد نیاز به پژوهش در آن حس نشده بود و تا جایی که بررسی های عمیق در مورد انرژی و باتری به منظور بهینه سازی و ابزاری برای سنجش و مانیتور کردن اطلاعات در این زمینه که به یاری برنامه نویسان و تحلیل گران سیستم میاید، بسیار اندک بوده و این حوزه نیازمند پژوهش های بیشتری است.

این موضوع خود عامل اصلی شکل گیری پروژه ایست که هم اکنون در اختیار شما است.در این پروژه ما بر روی بستر سیستم عامل Android اقدام به جمع آوری اطلاعات مربوط به باتری و تحلیل آن با اجرای برنامه پرداختیم.

باشد که در ادامه، این پروژه کامل شده و تبدیل به ابزاری کاربردی برای سنجش برنامه های از نظر مصرف انرژی باشد که با استفاده از آن بتوان مصرف، دلیل این حجم از مصرف و در نهایت راهی برای کمینه کردن آن را تشخیص داد و گامی در جهت بهبود عملکرد باتری برداشت.

محدوده پروژه

با توضیحاتی که در بخش اهداف پروژه انجام شد، پروژه ما در همین زمینه و با تمرکز بر روی سیستم عامل Android نسخه 7.1.1 (Nougat 7.1.1) و با استفاده از برخی از امکانات kernel سیستم عامل Android و با استفاده از tracer های موجود در سطوح مختلف معماری این سیستم عامل، داده های مربوط به ولتاژ باتری و جریان ناشی از آن و انرژی باتری را دنبال کرده و در نهایت مقادیر را صورت میانگین نمایش میدهد که با داشتن زمان شروع trace کردن و تمام اتفاق های سطوح سیستم می توان بررسی و تحلیل نمود که چه سری اتفاقاتی در چه مدت زمانی ای چه مصرف انرژی را داشته است.

معرفی پلتفرم و ابزارهای استفاده شده در پروژه

پلتفرم مورد استفاده ما در این پروژه علاوه بر سیستم عامل Nougat Android نسخه 7.1.1، Kernel نسخه Shamu که kernel مربوط به گوشی هوشمندی است که به عنوان محیط تست و توسعه مورد استفاده قرار گرفته است می باشد.این دستگاه، گوشی Nexus 6 متعلق به کمپانی Motorola است.

دلیل استفاده از این گوشی این است که در بین گوشی های موجود، از حیث فرکانس به روز رسانی داده های باتری دارای فرکانس بالاتری است و دوره تناوب آن ms150 است که به نسبت باقی مدل ها سریعتر به روز رسانی میشود و میتواند داده های بیشتر و در بازه های کوچکتر را تشخیص دهد و ثبت کند.

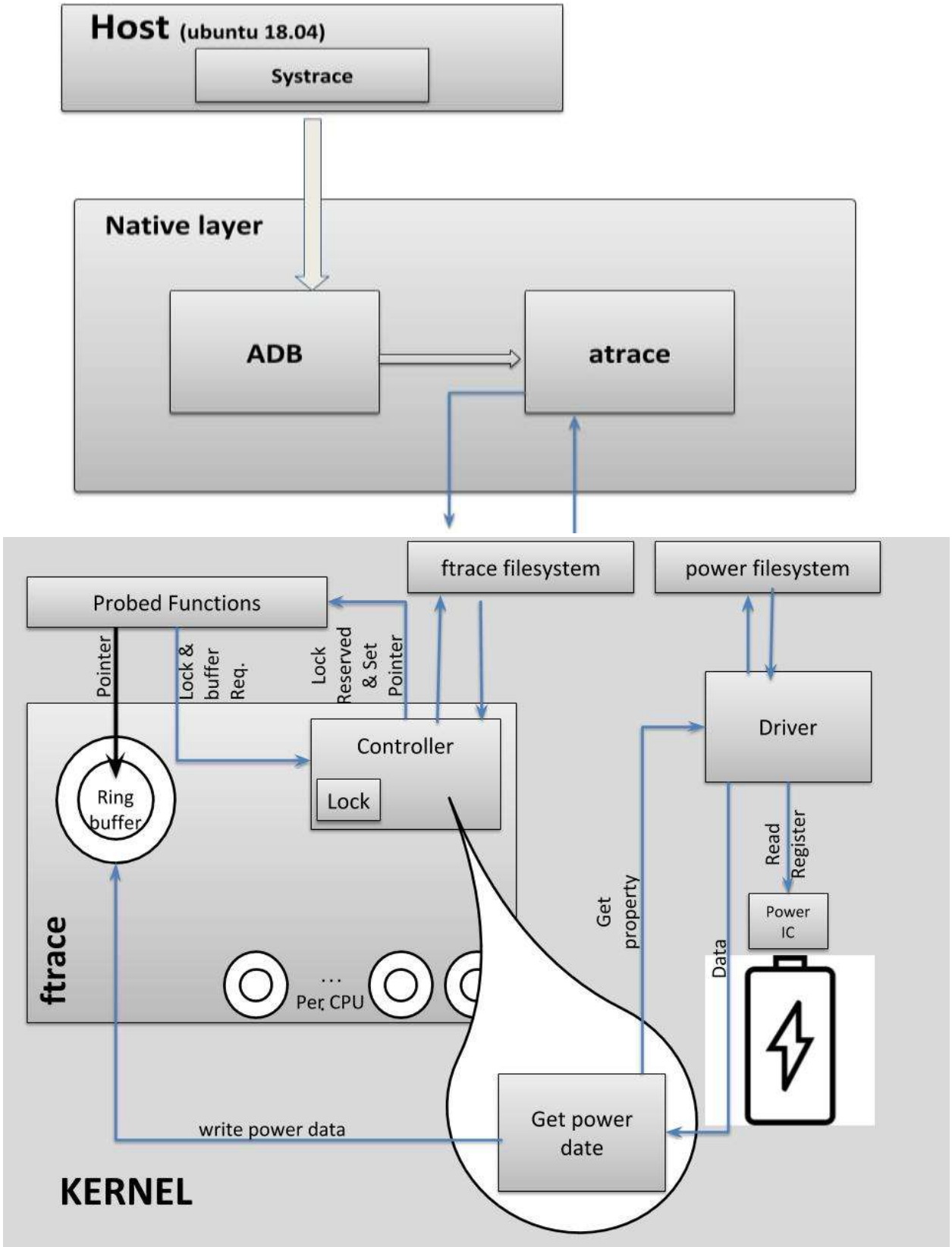
تمام داده های بدست آمده برای نمایش در اختیار ابزار مانیتورینگ و پروفایلینگ موجودی به نام systrace که خود توسط شرکت google طراحی و عرضه شده است، قرار میگیرد و سایر باقی اطلاعات نمایش میدهد.

ابزار systrace خود برای مانیتورینگ استفاده میگردد و داده های بدست آمده بنا به درخواست کاربر را به صورت نمودار نمایش میدهد و ما سعی کرده ایم که از امکانات این ابزار استفاده کرده و با اضافه نمودن داده های کاوش شده در پروژه ، آن را به این امکان جدید مجهز نماییم.

به منظور تست و تحلیل بهتر، برنامه ای به نوشته شد که برای ای کار مناسب باشد و مشخصه های لازم را داشته باشد. این مشخصه ها شامل ساده بودن برنامه و در عین حال داشتن task ای با مصرف انرژی به نسبت بالا است.به این منظور برنامه Android ای که کار با سنسور ژیروسکوپ و که اتصال WIFI دارد مورد استفاده قرار گرفت که اطلاعاتی را به سروری میفرستد و در سمت سرور آنها نمایش داده میشوند.

2- طراحی مفهومی

در این پروژه جهت بررسی performance از همان systrace استفاده می شود و داده های باتری به این ابزار اضافه شده است. systrace خود عملا wrapper برای ابزارهای دیگر محسوب می شود. به طور کلی systrace اطلاعات خود را از طریق atrace به دست می آورد که atrace ابزاری در سطح native framework در اندروید است که اطلاعات خودش را از ftrace بدست می آورد


Host (ubuntu 18.04)
Systrace
Native layer
ADB
atrace
ftrace filesystem
power filesystem
Probed Functions
Pointer
Lock & buffer Req.
Lock Reserved & Set Pointer
Controller
Lock
Ring buffer
Driver
Read Register
Power IC
Get property
Data
ftrace
... Per. CPU
write power data
Get power date
KERNEL

در شکل بالا روند زنجیره ای انتقال اطلاعات از ftrace به systrace تشریح شده است.
از آن جایی که داده های باتری در driver تولید می شوند هدف این است که این اطلاعات به داده های ftrace اضافه شوند تا در مسیر معکوس آنچه گفته شد از ftrace به atrace و در نهایت از طریق systrace در قالب گزارشی نمایش میدهیم. روند انتقال اطلاعات از driver ها به ftrace هم در قالب نمودار زیر توضیح داده شده است که البته در ادامه به تشریح جزئیات خواهیم پرداخت.

3- پیاده سازی

شکست کار بین اعضای تیم

در ابتدا پروژه به ۲ قسمت performance و power تقسیم شده بود که 1 نفر از اعضا بر روی performance و ابزار و تجهیزات آن و یک نفر بر روی power تمرکز داشتند.
در قسمت performance هم تقسیم کار بین اعضا بدین شکل بود که خانم داودزاده بر روی ابزارهای tracing سطح کرنل و نحوه پیاده سازی آن ها چه به صورت تئوری و چه به صورت کد در کرنل اندروید تمرکز داشتند. خانم صفاری مسئول تهیه یک اپلیکیشن جهت تست این برنامه بودند و همچنین بررسی ابزار systrace از منظر بررسی کدهای python این ابزار.
همچنین خانم جهانگیری در قسمت power مسئول شناسایی driver ها و امکانات مربوط به power بودند.
در ادامه پس از نزدیک شدن این فاز ها بهم خانم جهانگیری و داودزاده مسئول زدن کدهای کرنل برای انتقال اطلاعات power به ابزارهای موجود سطح کرنل بودند و در ادامه هم پس از با موفقیت به انجام رسیدن این کار، به اضافه کردن این داده ها به systrace و فهمیدن جزئیات فایل های مربوط به html پرداختند.
همچنین برای تست تغییرات کرنل، نیاز به build کرنل و AOSP اندروید بود که این فرآیند توسط خانم جهانگیری انجام می گرفت.

مشخصات محیط توسعه

از آن جایی که نیاز به تغییر کرنل وجود داشت سیستم عامل باید build می شد. نسخه کرنل مورد استفاده 1.7android-msm-shamu-3.10-nougat-mr است و شاخه aosp آن 55android-7.1.1_r است. از آن جایی که nexus 6 معماری ARM دارد برای کامپایل کرنل معماری آن ARM در نظر گرفته شده است و با کانفیگ shamu_defconfig کامپایل شده است گوشی Nexus 6 روت شده است . درایور باتری این گوشی Maxim MAX۱۷۰۵ است که از یک coulomb counter که از maxim Modelgauge جهت اصلاح و تنظیم داده های باتری در IC استفاده میکند . همچنین جهت اندازه گیری جریان یک مقاومت mohm۱۰ دارد.

تشریح پیاده‌سازی

از بین ابزار های در دسترس در اندروید، به این نتیجه رسیدیم که چون systrace ابزار پر کاربردتر و فراگیر تری نسبت به سایر ابزار ها است، نسبت به بقیه ارجحیت دارد. به این دلیل که خود این ابزار از ابزار های سطح پایین تر استفاده می کند، ما هم باید برای افزودن داده به این ابزار، از افزودن داده به ابزار های سطح پایین مانند , ftrace atrace شروع میکردیم.

یکی از قابلیت های ftrace ، داشتن چندین نوع tracer است که باتوجه به مورد استفاده ای که داریم میتوانیم آن ها را استفاده کنیم. در systrace و یا حتی atrace که از هر دو به نوعی از ftrace استفاده می کنند، از نوعی tracer به نام nop استفاده می کنند. از دلایل این انتخاب میتوان به اینکه parse داده ها در قالب این tracer ساده است و systrace هم از همین تریسر استفاده می کند , و سربار کمتری دارد اشاره کرد و همچنین tracer های دیگر در این ابزار به علت حجم بالای داده موجب crash در کرنل میشوند.به این دلایل که در ادامه جزئیات بیشتری به آن اضافه خواهد شد، تصمیم بر استفاده از ftrace nop گرفته شد.

جهت کاهش overhead خواندن داده های باتری مانند جریان و ولتاژ و انرژی این داده ها مستقیما از درایور باتری در کرنل خوانده می شوند همچنین این روش به لحاظ زمانی دقت بیشتری دارد . این خواندن به صورت پریودیک توسط یه thread اجرا می شود که این thread با فعال شدن tracing ساخته می شود و با پایان tracing از بین می رود . جهت ثبت این داده ها از رینگ بافر خود ftrace که همزمان در آن داده های perfomance ریخته می شود استفاده می شود و thread ثبت داده های باتری بخشی از بافر را رزرو کرده و lock آن را در اختیار میگیرد. بنابراین داده های انرژی در کنار داده های ftrace در فایل سیستم مربوط به آن قرار می گیرند.

در قسمت بعدی باید در ابزار سطح کرنل ftrace این داده ها را همراه بقیه داده ها در ring buffer می نوشتیم تا در نهایت توسط همین ابزار در file system ها به عنوان log داشته باشیم. ابزار سطح کرنل ftrace به این صورت است که در ابتدا تمام داده های خود را که به منظور trace کردن ثبت کرده است در ring buffer ای می نویسد و این ring buffer توسط توابع دیگری که در خود ftrace تعریف شده است این داده ها را با فرمت بهتر و قابل تحلیل تری در فایل سیستم های خودش میریزد. علت انتخاب بافر ftrace هماهنگی زمانی بیشتر داده های performance و داده های باتری می شود

پس از آنکه داده ها در filesystem های ftrace قرار گرفتند و همچنین توسط atrace هم قابل نمایش بودند، به فاز آخر، یعنی visual کردن این داده های خوانده شده رسیدیم.

در این فاز، ما داده های power را از بقیه داده ها که از فایل سیستم trace جدا کردیم و همچنین با ثبت اطلاعات دیگری همچون timestamp اولین داده atrace و کل داده های power توانستیم داده های جریان و ولتاژ و انرژی را به صورت نمودار در systrace نمایش دهیم.

تغییرات اعمال شده

تغییراتی که در این پروژه اعمال شد به طور مختصر به دو ابزار ftrace , systrace باز میگردد. در ابزار ftrace، تغییرات مربوط به اضافه کردن کدهایی برای گرفتن داده از بخش های مربوط به power است و همچنین نوشتن این داده ها در ring buffer.

در ابزار systrace این تغییرات شامل قسمت دریافت اطلاعات atrace است که در این قسمت داده های power که به atrace اضافه شده بودند را از بقیه داده ها جدا کردیم و به ازای هر داده جدا شده هم (یعنی هر یک log) قسمت های مختلف آن از جمله مقدار جریان و ولتاژ و انرژی و timestamp آن را هم parse کردیم و در متغیر هایی ذخیره کردیم به این منظور که بتوان از آنها برای کشیدن نمودار استفاده کرد.

در قسمت بعدی هم، همچنین یک تابع مربوط به کشیدن نمودار اضافه شده است که از داده های جدا شده قسمت قبل استفاده میکند.

4- تغییرات نسبت به فاز پروپوزال

چالش های پروژه

- **تجربیات منجر به تغییر در تصمیم گیری**

در ابتدا تصمیم بر آن بود که داده های systrace , strace در کنار هم قرار بگیرند. با گذر زمان و کامل شدن اطلاعات لازم و داشتن دید کافی نسبت به عمق پروژه، این تصمیم گرفته شد که در فاز اولیه طرح اضافه کردن داده به systrace در اولویت کار قرار گیرد.

- **تجارب ناموفق**
  - **روش های متعدد خواندن داده های باتری**

در قدم های اول پروژه هدف این بود که داده های باتری بدون نیاز به روت کردن یا تغییر کرنل خوانده شوند که تنها راه موجود در گوشی های که امکان دسترسی مستقیم به فایل سیستم های باتری در کرنل وجود ندارد (در بیشتر گوشی هایی که ورژن noguat روی آن های نصب شده است این قابلیت به دلایل امنیتی وجود ندارد) استفاده از batteryManager است که یک api در سطح framework که اطلاعات باتری را از سیستم سرویس مربوطه که خود این اطلاعات را از hal می خواند و hal نیز نهایتا از sysfs ها این اطلاعات را میخواند. این کلاس مجموعه از constant ها و string ها است که توسط Intent.ACTION_BATTERY_MANAGER استفاده می شوند تا اطلاعات مرتبط با باتری و حالت شارژ را به دست آورد (همان اطلاعاتی که در لایه کرنل است) . constantهای این کلاس اطلاعات در مورد وضعیت سلامت باتری، وضعیت شارژ، درصد شارژ باتری، ولتاژ و جریان باتری دارند. لیست کامل constant ها و method های این کلاس:

BatteryManager reference

از آن جایی که این کلاس از یک broadcast receiver جهت دریافت اطلاعات استفاده می کنند و این اطلاعات باید از hal تا سطح framework انتقال داده شوند overhead هایی دارد که برای ما ناشناخته است بنابراین در گام بعدی هدف استفاده از متدی بود که تنها نیاز به روت کردن گوشی داشت و کرنل در آن تغییری نمی کرد. که در این متد اطلاعات از sysfs خوانده میشد. دایرکتوری هایی که این sysfs در کرنل قرار دارند به شرح زیر است :

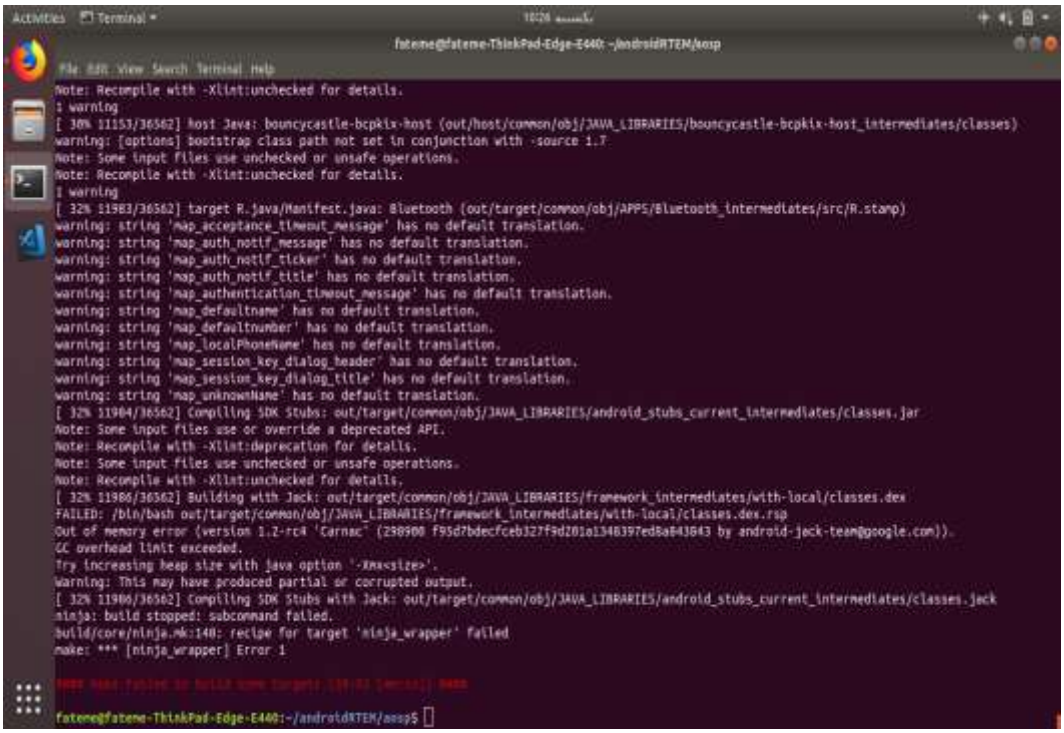
```
sys/class/power_supply/*/capacity
/sys/class/power_supply/*/charge_counter
/sys/class/power_supply/*/charge_full
/sys/class/power_supply/*/current_avg
/sys/class/power_supply/*/current_max
/sys/class/power_supply/*/current_now
/sys/class/power_supply/*/cycle_count
/sys/class/power_supply/*/health
/sys/class/power_supply/*/online
/sys/class/power_supply/*/present
/sys/class/power_supply/*/status
/sys/class/power_supply/*/technology
/sys/class/power_supply/*/temp
/sys/class/power_supply/*/type
/sys/class/power_supply/*/voltage_max
/sys/class/power_supply/*/voltage_now
```

که از طریق سیستم کال read بر هر یک از این مسیر ها در کرنل داده مورد خوانده می شود . که البته جهت استفاده از این روش در یک اپلیکیشن در سطح فریم ورک لازم که داخل اپلیکیشن root shell اجرا شود که این خود overhead قابل توجهی می تواند داشته باشد. هر چند از طریق تغییر دادن hal یا نوشتن یک system service جدید به این داده ها از به روش دلخواه می توان دسترسی داشت که در واقع خود اندروید battery service را دارد که به طور مستقیم اطلاعات را از hal می خواند و battery Manager هم از آن استفاده میکند. که این روش نیازمند build کردن aosp است.

- **بیلد کردن AOSP**

```
. build/envsetup.sh
export USE_CCACHE=1
export CCACHE_DIR=~/.ccache
prebuilts/misc/linux-x86/ccache/ccache -M 50G
echo "USE_CCACHE=1" >> ~/.bashrc
lunch aosp_shamu-userdebug
make -j$(nproc --all)
```

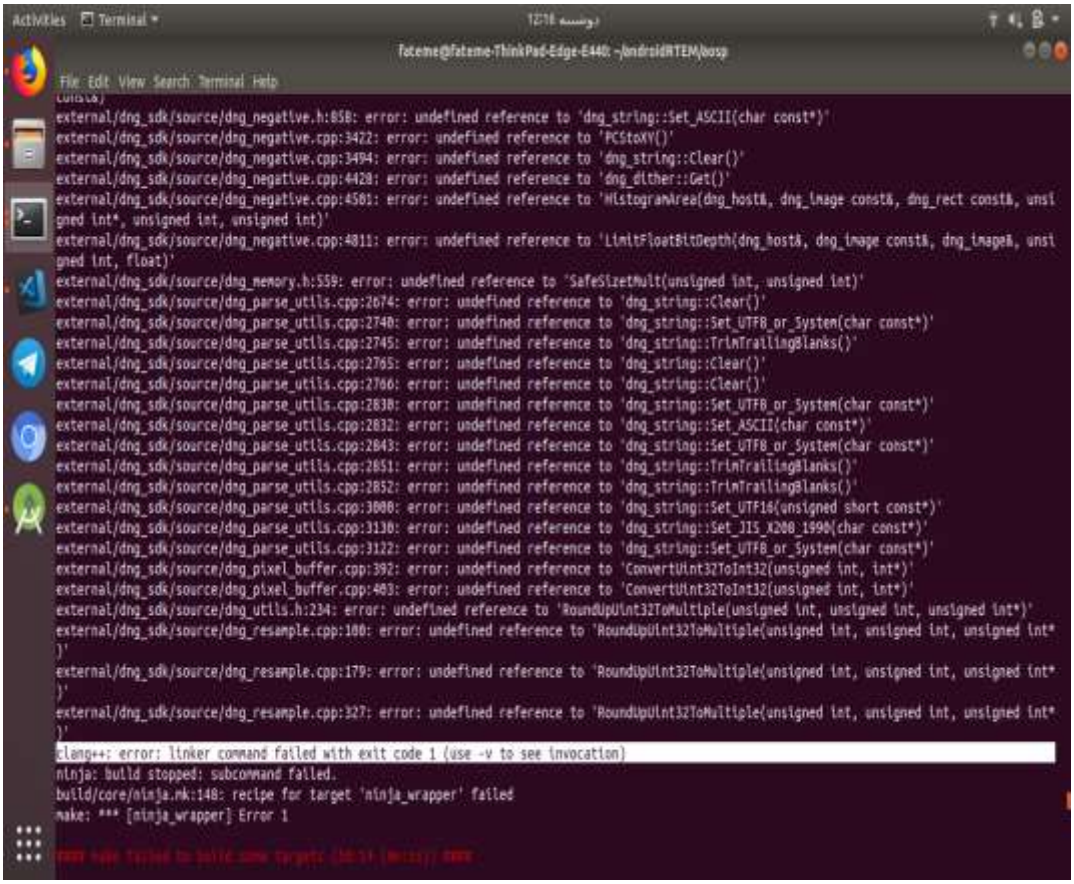
با افزودن export LC_ALL=C ادامه پیدا کرد اما باز هم موفقیت آمیز نبود و بعد از 50 دقیقه اروری که در ادامه نشان داده میشود ظاهر میشد:



که به نظر می رسید ارور به وجود آمده به دلیل مشکل در افزایش سایز heap باشد که با افزودن این دستور این ارور رفع شد:

```
export JACK_SERVER_VM_ARGUMENTS="-Dfile.encoding=UTF-8 -XX:+TieredCompilation -Xmx4096m"
prebuilts/sdk/tools/jack-admin kill-server
prebuilts/sdk/tools/jack-admin start-server
```

البته با وجود حل این مشکل باز بیلد دوباره ارور دیگری رخ داد:



که در نهایت با کم کردن حافظه jack server این مشکل شد.این مشکل معمولا در سیستم هایی که کمتر از ۱۶گیگ رم دارند اتفاق می افتدو. همچنین استفاده از cache موجب ارور میشد که حذف آن بخشی از ارور ها را حذف کرد. دستورات نهایی:

```
. build/envsetup.sh
export USE_HOST_LEX=yes
export LC_ALL=C
export JACK_SERVER_VM_ARGUMENTS="-Dfile.encoding=UTF-8 -
XX:+TieredCompilation -Xmx4096m"
prebuilts/sdk/tools/jack-admin kill-server
prebuilts/sdk/tools/jack-admin start-server
lunch aosp_shamu-userdebug
make -j$(nproc --all)
```

۵- نزدیکترین نمونه های مشابه

در طی مطالعاتی که در ابتدا و در حین انجام پروژه انجام شد به یک سری ابزار یا متد که هر کدام داده هایی مخصوص و با دقت های متفاوتی و متناسب با سطوح مختلف از معماری Android را ارائه می دادند که آن ها را در ادامه نام برده و برای هر کدام توضیح مختصری در مورد ماهیت و نحوه عملکردشان خواهیم داشت.

- **Trepn**

ابزار تست مصرف انرژی و کارایی است که به صورت اپلیکیشن روی دستگاه هدف نصب می شود. در حوزه ی اندازه گیری مصرف انرژی در برخی دستگاه ها این ابزار دقت لازم را دارد

Trepn اطلاعات خود را از power management IC و نرم افزار battery fuel gauge میخواند. هر چند اگر دستگاه هدف این اطلاعات مستقیم را در اختیار ابزار قرار ندهد از طریق آنالیز فرکانس و بار هر هسته CPU و GPU و روشنایی صفحه با استفاده مدل های انرژی که با استفاده از مدل های خاصی از پردازشگر های Snapdragon ساخته شده اند بهره می برد و انرژی را تخمین می زند. Apk این ابزار موجود است و می توان به طور دستی فرآیند شروع نمونه برداری آن را آغاز کرد و پایان داد. خروجی این نرم افزار یک فایل .csv است که نتایج نمونه برداری را شامل می شود. دقت و فرکانس نمونه برداری این ابزار تا حد زیادی وابسته به سخت افزار است که در بهترین حالت فرکانس آن 10 هرتز است. و با بررسی بایت کد این نرم افزار مشخص شد که جهت اندازه گیری مستقیم داده های باتری این ابزار هم از sysfs که در بخش تجارب ناموفق ذکر شد استفاده میکند . این ابزار بر یک گوشی HTC desire EYE تست شد که chipset گوشی آن Qualcomm Snappdragon 810 بود. خروجی تست مشابه زیر است .

System		Apr 19, 2019 @ 02:11:30 PM	50 sec		
Time [ms]	Battery Power [uW]	Time [ms]	Battery Remaining (%) [%]	Time [ms]	Battery Status
0	1041566	0	23	0	0
97	2222961	22	23	22	0
198	2057020	97	23	97	0
298	1943768	198	23	198	0
398	1140245	298	23	298	0
499	1322975	398	23	398	0
600	1274347	499	23	499	0
700	913019	600	23	600	0
800	918819	700	23	700	0
900	917104	800	23	800	0

- **powerTutor**

این ابزار اپلیکیشنی است که انرژی مصرف شده توسط کامپوننت های اصلی سیستم مانند CPU , GPS , network interface , display و اپ های مختلف را نشان می دهد . Power tutor از یک مدل مصرف انرژی که از طریق اندازه گیری های دقیق ساخته شده اند بهره می برد. به همین دلیل بر روی تعداد بسیار محدودی از گوشی ها نتایج دقیق دارد . به صورت اپلیکیشن نصب می شود و مصرف انرژی را روی نمودار برای کامپوننت های مختلف real time نشان می دهد(همانند Trepn داده ها را در اختیارمان نمی گذارد و بیشتر source code آن قابل استفاده است). در واقع این اپ برای مدل های جدید آپدیت نشده است.

- **batteryStat**

ابزار در سطح فریم ورک اندروید است و به جمع آوری داده های مربوط به انرژی در دستگاه اندروید می پردازد. این داده عمدتا توسط کلاس BatteryStatImpl.java جمع آوری می شود که با استفاده از PowerProfile و log های دیگر سیستم درباره کامپوننت های دیگر همانند wifi, cpu, camera , .. تخمینی درباره مصرف انرژی ارائه می دهد. این اطلاعات توسط batteryhistorian که توسط گوگل توسعه یافته است نمودار می شود. از طریق ADB و دستور dumpsys می توان به این اطلاعات لاگ شده دسترسی داشت با reset کردن این ابزار امکان لاگ کردن از مبدا زمانی خاصی وجود دارد.

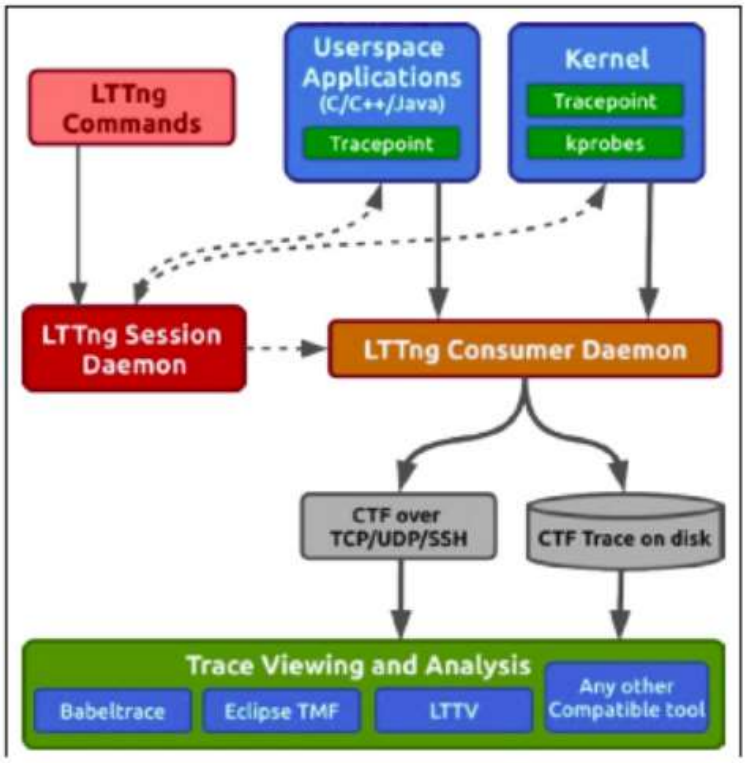
یک ابزار مشابه دیگر که البته در linux کاربرد دارد:

- **LTTNG:**

این ابزار می تواند:

  - analyze interprocess interactions in the system
  - analyze application-kernel interactions in the user space
  - measure the amount of time the kernel spends serving application requests
  - analyze how the system works under heavy workloads

تریس این ابزار از طریق کدهای static , dynamic است هنگامی که این کدها فراخوانده میشوند اطلاعات درباره اتفاقات در channels نوشته میشود. این session , channel ها روش تریس این ابزار هستند با استفاده از پارامتر هایی که کاربر مشخص میکند



این شکل تنها یک سشن است در حالی که این ابزار امکان هندل چندین سشن را نیز دارد

سشن ها از کانالها برای فرستادن داده ها استفاده میکنند کانال ها یک مجموعه ای از همراه ویژگی های تعریف شده هستند این ویژگی ها شامل : buffer size, trace mode, and the switch timer period.

این کانال ها برای ساپورت کردن بافر اشتراکی استفاده میشوند در این بافر هم اطلاعات ایونت ها نوشته میشود و همجایی که consumer daemon اطلاعات را از آن می خواند این بافر به اندازه های کوچک تر تقسیم میشود هنگامی که این ابزار شروع به نوشتن در یک قسمت می کند تا زمانی که آمپر شود و سپس به بافر بعدی میرود این زیر بافر هایی که اطلاعاتشان تکمیل شده است lock میشوند توسط consumer daemon و در دیسک ذخیره می شوند.اگر همه بافر ها پر شوند این ابزار ترجیح میدهد که مقداری از داده ها را ازدست بدهد تا اینکه سیستم را آهسته کند برای خواندن داده ها. و این دو انتخاب را پیش رو میگذارد:

- از اطلاعات جدید صرف نظر شود
- از اطلاعات قبلی (overwrite)

این ابزار همچنین به طور dynamic هم میتواند مورد استفاده قرار گیرد با استفاده از kprobe ها برای دنبال کردن ایونت ها در سطح کرنل. به طور خاص این ابزار امکان اضافه کردن یک پروب به کرنل را به ما میدهد در ابتدا باید یک script را آماده کنیم که در نهایت کامپایلر آن را به c ترجمه خواهد کرد که در این kernel module جدا کامپایل میشود.

Ftrace هم از kprobe پشتیبانی میکند که این امکان را فراهم میکند که بدون نوشتن یک اسکریپت بتوان پروب ها را اضافه نمود.

Lttng دو نوع پروب را پشتیبانی میکند native kprobe , kretprobes

native kprobe : یک basic probe که میتواند در همه جای کرنل قرار گیرد.

Kretprobes: تنها قبل از خروج از تابع و دسترسی به مقدار بازگشتی آن قابل استفاده است.

نحوه تریس کردن در سطح کرنل این ابزارهم با استفاده از kernel module هایی است که توسط این ابزار instrument شده اند از جمله:

Probe module ها که هر ماژول به یک subsystem در کرنل attach شده است با استفاده از tracepoint instrument point. همچنین این ماژولها به ورودی و مقدار بازگشتی سیستم کال ها نیز attach شده اند.

Ring buffer module کرنل تریسر این ابزار در رینگ بافر می نویسد که consumer daemon ز آن می خواند. زمانی که تریسینگ آغاز میشود session ها ماژول های مورد نیاز را load میکنند.

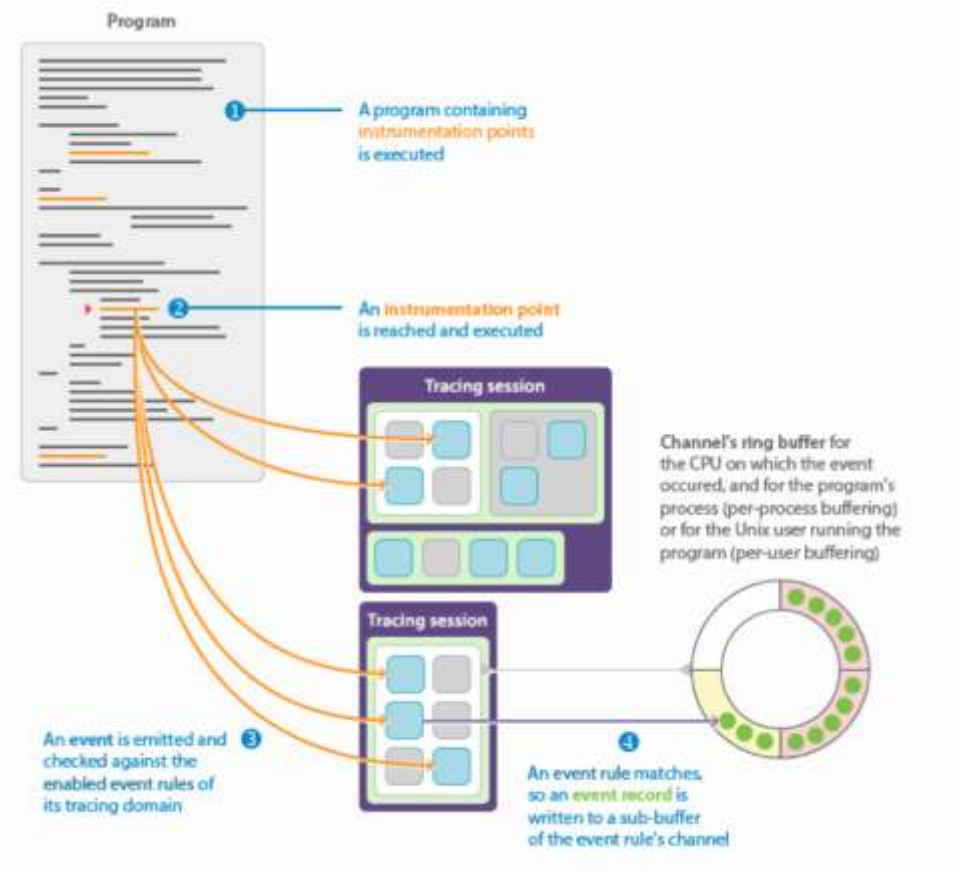
Consumer daemon , اشتراک گذارنده ring buffer بین user application و lttng kernel modules

ارتباط بین ما و session ها از طریق یک tracing session صورت میگیرد که این tracing session با استفاده از دستور

Lttng create

ساخته میشود که بطور خاص نام، فایل های تریس ، استیت فعالیت (channel، mode ،start or stop)

یک channel یک object است که مسئول چند ring buffer است. و events را در sub buffer ذخیره میکند. و ویژگی های channel هستند که مشخص میکنند چه کاری انجام شود وقتی همه زیر مجموعه های بافر پر هستند.



6- مبانی فنی پروژه

★ ارائه راه‌حل پیشنهادی بصورت کلی

برای داشتن داده های پاور همزمان با داده های ftrace، به این رسیدیم که باید توابع جدیدی که به منظور اضافه کردن power، میخواهیم داشته باشیم باید در کنار کدهای مربوط به شروع ftrace باشد تا این داده ها از timestamp های نزدیک تری برخوردار باشند.

همچنین برای اضافه کردن این داده ها به ftrace، از ایده ring buffer این ابزار کمک گرفتیم. به این دلیل که خود این ابزار در ابتدا log های خود را در ring buffer مینویسد ما هم اطلاعات خود را در ابتدا در ring buffer بنویسیم. در پی دنبال کردن این تصمیم، با توجه به نمونه های موجود در خود کرنل، تصمیم بر رزرو کردن قسمتی از ring buffer برای جلوگیری از deadlock های احتمالی شد.

همچنین در گام آخر، داده ها در systrace از بقیه داده های atrace جدا میشوند و در قسمت دیگری در همان فایل های systrace که در ادامه جزئیات آن ذکر خواهد شد، این داده ها visual میشوند.

★ **ارائه راه‌حل با جزییات**

- **خواندن داده های باتری از درایور**

چالش اول دسترسی به struct درایور باتری در کرنل در فایل trace.c است.در فایل trace.c بیشتر قسمت های کنترلی ftrace پیاده سازی شده است. در کرنل لینوکس یک device tree وجود دارد که لیست درایور ها و پوینتر به آن ها را نگهداری می کند و با اسم یک درایور و iterate کردن بر این لیست میتوانیم به struct مورد نظر خود دست یابیم.

```
power_supply_get_by_name("max170xx_battery");
```

پیاده سازی این تابع مطابق زیر است که در آن تابعی که باید برای match کردن و مقایسه استفاده می شود به تابع class_find_device پاس داده شود :

```
static int power_supply_match_device_by_name(struct device
*dev, const void *data){
    const char *name = data;
    struct power_supply *psy = dev_get_drvdata(dev);

    return strcmp(psy->name, name) == 0;
}

struct power_supply *power_supply_get_by_name(const char
*name){
    struct device *dev =
class_find_device(power_supply_class, NULL, name,
                      power_supply_match_device_by_name);

    return dev ? dev_get_drvdata(dev) : NULL;
}
```

خروجی تابع class_find_device یک struct از جنس max17042_chip است که ساختار آن با جزئیات در ضمیمه فنی موجود است. این struct حاوی یک instance از power_supply است که خروجی dev_get_drvdata و power_supply_get_by_name یک پوینتر به استراکت power_supply است که بخشی از ساختار آن این گونه است:

```
struct power_supply {
```

```
        const char *name;
        enum power_supply_type type;
        enum power_supply_property *properties;
        size_t num_properties;

        char **supplied_to;
        size_t num_supplicants;

        char **supplied_from;
        size_t num_supplies;
#ifdef CONFIG_OF
        struct device_node *of_node;
#endif

        int (*get_property)(struct power_supply *psy,
                    enum power_supply_property psp,
                    union power_supply_propval *val);
        int (*set_property)(struct power_supply *psy,
                    enum power_supply_property psp,
                    const union power_supply_propval *val);
        int (*property_is_writeable)(struct power_supply *psy,
                    enum power_supply_property psp);
        void (*external_power_changed)(struct power_supply *psy);
        void (*set_charged)(struct power_supply *psy);
...
...
...
```

با توجه به ساختار این استراکت برای به دست آوردن مقدار باتری مورد نظر کافی است که از متد get_property استفاده کنیم که در آن power_supply_property نوع دیتایی که میخواهیم مشخص می کند که این داده با توجه درایور می تواند شامل مقادیر متعددی باشد. این function pointer ها در فاز initialize کرنل با تابع مربوط به درایور مختص به دستگاه مقداردهی می شوند بخشی از تابع max17042_get_property در فایل max17042_battery.c که get_property در struct power_supply به این تابع اشاره می کند:

```
static int max17042_get_property(struct power_supply
*psy,
                 enum power_supply_property psp,
                 union power_supply_propval *val)
{
```

```
    struct max17042_chip *chip = container_of(psy,
                    struct max17042_chip, battery);
    int ret;
    int64_t ret64;
    if (!chip->init_complete)
       return -EAGAIN;

    switch (psp) {
    case POWER_SUPPLY_PROP_VOLTAGE_NOW:
       ret = max17042_read_reg(chip->client,
MAX17042_VCELL);
       if (ret < 0)
             return ret;

       val->intval = ret * 625 / 8;
       Break;
…
…
…
                    /* در ادامه برای هر property یک case و تابع خاص وجود دارد */
```

تابع read_reg رجیستر مربوط به ویژگی مورد نظر ما را از درایور از طریق I2bus می خواند . بنابراین در کل برای گرفتن مقادیر جریان و ولتاژ و انرژی داریم :

```
bt.psy-
>get_property(bt.psy,POWER_SUPPLY_PROP_CURRENT_NOW,&va
lueC);
bt.psy-
>get_property(bt.psy,POWER_SUPPLY_PROP_VOLTAGE_NOW,&val
ueV);   bt.psy-
>get_property(bt.psy,POWER_SUPPLY_PROP_CHARGE_COUNTER
_EXT,&valueE);
```

- **ساختار کنترلی** ftrace

فایل سیستم هایی که به ftrace اختصاص داده شده اند (debugfs) در واقع نوعی character device محسوب می شوند. که توسط struct به نام file_operation مدیریت می شوند. هر فایل سیستم ابتدا ایجاد می شود و این struct به آن نسبت داده می شود. در نمونه آورده شده فایل سیستم ایجاد شده در مسیر d/tracing/tracing_on در کرنل قرار دارد. d_tracer یک struct از جنس dentry است که توسط کرنل همراه با inode برای مدیریت virtual file system ها استفاده می شود.

```
trace_create_file("tracing_on", 0644, d_tracer,
                  tr, &rb_simple_fops);
```

در آرگومان های تابع فوق tr به متغیر گلوبال به نام global_trace در فایل trace.c اشاره می کند که که struct هایی که نقش بافر ftrace هنگام trace کردن را دارند اشاره می کند

```
static struct trace_array   global_trace;
```

ساختار rb_simple_fops :

```
static const struct file_operations rb_simple_fops = {
    .open       = tracing_open_generic_tr,
    .read       = rb_simple_read,
    .write      = rb_simple_write,
    .release    = tracing_release_generic_tr,
    .llseek     = default_llseek,
};
```

دیگر فایل های کنترلی مانند trace_clock و trace و buffer_size_kb و ... که مربوط به ftrace هستند هم ساختار مشابهی دارند . زمانی که سیستم کال خواندن یا نوشتن یا ... بر d/tracing/tracing_on فراخوانی شود تابع مربوط به هر یک صدا می شود. برای مثال زمانی که عدد ۱ در tracing_on نوشته یا echo می شود rb_simple_write فراخوانی می شود :

```
static ssize_t
rb_simple_write(struct file *filp, const char __user *ubuf,
        size_t cnt, loff_t *ppos)
{
    struct trace_array *tr = filp->private_data;
    struct ring_buffer *buffer = tr->trace_buffer.buffer;
    unsigned long val;
    int ret;
```

```
    ret = kstrtoul_from_user(ubuf, cnt, 10, &val);
    if (ret)
        return ret;

    if (buffer) {
        mutex_lock(&trace_types_lock);
        if (val) {
            tracer_tracing_on(tr);
            if (tr->current_trace->start)
                tr->current_trace->start(tr);
        } else {
            tracer_tracing_off(tr);
            if (tr->current_trace->stop)
                tr->current_trace->stop(tr);
        }
        mutex_unlock(&trace_types_lock);
    }

    (*ppos)++;

    return cnt;
}
```

در تابع بالا kstroul_from_user مقداری که کاربر وارد کرده است دریافت می کند.

- **ثبت داده های خوانده شده در** ring buffer:

همان طور که بیان شد struct trace_array به عنوان بافر استفاده می شود بخشی از این struct :

```
struct trace_array {
    struct list_head    list;
    char            *name;
    struct trace_buffer trace_buffer;
    int         stop_count;
    int         clock_id;
    struct tracer       *current_trace;
    unsigned int        flags;
    raw_spinlock_t      start_lock;
    struct dentry       *dir;
    struct dentry       *options;
    struct dentry       *percpu_dir;
```

```
    struct dentry       *event_dir;
    struct list_head    systems;
    struct list_head    events;
    struct task_struct  *waiter;
    int         ref;
};
```

دیتاهای باتری باید trace_buffer در استراکت بالا اضافه شوند که ساختار این struct به این شکل است و همان طور که قابل مشاهده است هر trace_buffer به یک trace_array دیگر اشاره می کند و ساختار کلی یک لینک لیست است :

```
struct trace_buffer {
    struct trace_array      *tr;
    struct ring_buffer      *buffer;
    struct trace_array_cpu __percpu *data;
    cycle_t             time_start;
    int             cpu;
};
```

دیتاهای موردنظر ما درباره انرژی در نهایت در ring_buffer نوشته می شوند. پوینتر به ring_buffer بدین شکل قابل دسترسی است :

```
global_trace.trace_buffer.buffer
```

همچنین جهت نوشتن در این رینگ بافر باید بافر رزرو شود که این تابع در در trace.c و ring_buffer.c پیاده سازی شده است (برای نوشتن در ring buffer با توجه به توابع tracer_bputs و tracing_mark_write پیش رفتیم و قسمتی از ring buffer را reserve کردیم به جای استفاده از :(lock

```
event = trace_buffer_lock_reserve(buffer, TRACE_PRINT,
alloc,irq_flags, preempt_count());
```

و دیتا را مطابق کد زیر در بافر مربوطه می ریزیم

```
entry = ring_buffer_event_data(event);
entry->ip = _THIS_IP_;

memcpy(&entry->buf, power_data, size);
```

- **کد نهایی اضافه شده به trace.c**

همان طور که در قسمت های بیان شد هنگامی که ۱ در فایل tracing_on نوشته می شود تابع rb_simple_write فراخوانی می شود که به تبع آن tracer_tracing_on فراخوانی می شود . برای این که خواندن داده های انرژی و وضعیت باتری همروند با تریس کردن انجام شود در تابع tracer_tracing_on یک thread ایجاد می شود که وظیفه آن خواندن داده ها به صورت مستقیم از درایور باتری و ثبت آن ها در بافر تریسر به صورت پریودیک است. دو نکته قبل از فعال کردن تریسر باید رعایت شوند تا داده ها دقیق تر باشند. اولا بایدusb charging باتری غیر فعال شود که این قابلیت بستگی به درایور شارژ دارد که خوشبختانه این امکان در nexus 6 وجود دارد و با اجرای این دستور در shell (در حالت root) :

```
Echo 0 > /sys/class/power_supply/battery/charging_enabled
```

همچنین نکته دومی که باید در نظر بگیریم این است که ftrace می تواند از کلاک های متعددی جهت ثبت timestamp استفاده کند که دقت و مطابقت آن ها با زمان واقعی متفاوت است . به صورت پیش فرض ftrace از local clock استفاده میکند که درواقع کلاک مختص به هر cpu است که این کلاک ها با هم sync نیستند بنابراین اگر اختلاف timestamp دو تریس را مقایسه کنیم این مقدار لزوما برابر با مقدار واقعی تایمی که بین این دو event گذشته است نیست. بنابراین اگر در این زمینه دقت بیشتری بخواهیم و فقط ترتیب تریس های ثبت شده برایمان مهم نباشد باید کلاک ftrace را تغییر دهیم. کلاک های دیگری که در ftrace در دسترس هستند شامل global (با همه cpu ها sync است که اندکی کندتر از local است) و uptime (که از شمارنده jiffies استفاده می کند و زمان به دست آمده از آن از زمان boot up است که دقت آن به مراتب بهتر است) و کلاک های دیگر . بنابراین جهت دقت بیشتر در فاصله های زمانی تریس ها نوع کلاک را تغییر می دهیم که این کار از طریق فایل سیستم ftrace به راحتی قابل انجام است :

```
Echo uptime > /d/tracing/trace_clock
```

کدهای نهایی اضافه شده به فایل trace.c:

```
struct battery_trace{
    struct power_supply *psy;
    struct trace_array *tr;
};
static bool stop_power_tracing;
static void read_power_data(void);
static struct battery_trace bt;
```

```
/* این تاب مسئول نوشتن داده ای است که در آرگومان دریافت میکند در رینگ بافر*/

int write_power_ringbuffer(const char* power_data){
    struct ring_buffer_event *event;
    struct ring_buffer *buffer;
    struct print_entry *entry;
    unsigned long irq_flags;
    int alloc;
    int size;
    size = sizeof(int)*12;

/*در خط بالا مقداری که میخواهیم در رینگ بافر نوشته شود را تنظیم کردیم*/

    if (unlikely(tracing_selftest_running || tracing_disabled))
        return 0;

/* در قسمت بالا این حالت که اجازه نوشتن در رینگ بافر داده نشده و یا ابزار تریس فعال باشد بررسی
شده است*/

    alloc = sizeof(*entry) + size + 2; /* possible \n added */
    local_save_flags(irq_flags);

/* در این قسمت فلگهای مربوطه برای نوشتن تنظیم شده اند زیرا رینگ بافر با توجه به این فلگ ها
نوشته میشود*/

    buffer = global_trace.trace_buffer.buffer;

/*رینگ بافری که میخواهیم در آن بنویسیم*/

    event = trace_buffer_lock_reserve(buffer, TRACE_PRINT, alloc,
                      irq_flags, preempt_count());

/*رزرو کردن قسمتی از همان رینگ بافری که در قسمت قبل داشتیم توسط این قسمت انجام میگیرد*/

    if (!event)
        return 0;
    entry = ring_buffer_event_data(event);
    entry->ip = _THIS_IP_;

    memcpy(&entry->buf, power_data, size);
```

```
/*در این قسمت یک پوینتر به قسمت رزرو شده از رینگ بافر از تابع قبل خروجی گرفتیم و با استفاده از
خط بالا مقدار دلخواه را در آن قسمت وارد میکنیم */

    /* Add a newline if necessary */
    if (entry->buf[size - 1] != '\n') {
        entry->buf[size] = '\n';
        entry->buf[size + 1] = '\0';
        stm_log(OST_ENTITY_TRACE_PRINTK, entry->buf, size + 2);
    } else {
        entry->buf[size] = '\0';
        stm_log(OST_ENTITY_TRACE_PRINTK, entry->buf, size + 1);
    }
    __buffer_unlock_commit(buffer, event);

/*در خط بالا داده هایی که نوشته شده در قسمت رزرو شده توسط این تابع ثبت شده و آن قسمت رینگ
بافر از حالت رزرو خارج میشود در حقیقت تمام لاک ها را آزاد میکند.*/

    return size;
}
static struct task_struct *power;

static void read_power_data(void){
    union power_supply_propval valueC,valueV,valueE;
    char power_buffer[40];
    int n;
   while (!stop_power_tracing){
                bt.psy-
>get_property(bt.psy,POWER_SUPPLY_PROP_CURRENT_NOW,&va
lueC);
        bt.psy-
>get_property(bt.psy,POWER_SUPPLY_PROP_VOLTAGE_NOW,&val
ueV);
        bt.psy-
>get_property(bt.psy,POWER_SUPPLY_PROP_CHARGE_COUNTER
,&valueE);
n=sprintf(power_buffer,"v:%d c:%d
e:%ld\n",valueV.intval,valueC.intval,(long)valueE.int64val);
        write_power_ringbuffer(power_buffer);
        msleep(80);
    }
}
static int start_power_tracing(void* tr){
```

```
    bt.psy=power_supply_get_by_name("max170xx_battery");
    if(bt.psy){
        stop_power_tracing=false;
        bt.tr=tr;
        read_power_data();
    }
    return 0;

}

        /*در تابع زیر تردی که باید به صورت متناوب در رینگ بافر بنویسد ساخته و شروع به کار میکند.*/

void periodic_update_power(struct trace_array *tr){
    int ret;
    power = kthread_run(start_power_tracing,
                        tr, "periodic_power_data_add");
    ret = PTR_ERR(power);
    if (IS_ERR(power))
        goto out_fail;
    return;
out_fail:
    kthread_stop(power);
}

        /*برای خارج شدن یا بسته شدن ابزار تریسینگ توسط تابع tracing_off ، ترد را kill میکند.*/

void stop_thread(void ){
    kthread_stop(power);
}
void tracer_tracing_on(struct trace_array *tr)
{
    if (tr->trace_buffer.buffer){
        ring_buffer_record_on(tr->trace_buffer.buffer);

    /*در این قسمت ابزار ftrace شروع به کار میکند پس با صدا زدن تابع زیر در این قسمت ما میتوانیم
                                                همزمان با شروع اطلاعات را داشته باشیم*/

        periodic_update_power(tr);

    }
    tr->buffer_disabled = 0;
    /* Make the flag seen by readers */
```

```
    smp_wmb();
}

/**
* tracing_on - enable tracing buffers
*
* This function enables tracing buffers that may have been
* disabled with tracing_off.
*/
void tracing_on(void)
{
    tracer_tracing_on(&global_trace);
}
EXPORT_SYMBOL_GPL(tracing_on);
```

- **افزودن داده های power از سطح کرنل به systrace:**

از آن جایی که داده های power مستقیما در بافر ftrace نوشته شده اند زمانی که systrace از atrace استفاده میکند این داده ها همراه با داده های performance در d/tracing/trace نوشته می شوند که atrace این مقادیر را میخواند و systrace آنها را از atrace دریافت می کند . درنهایت یک فایل html به این شیوه ساخته می شود(output_generator.py):

```
html_prefix = _ReadAsset(systrace_dir, 'prefix.html')
 html_suffix = _ReadAsset(systrace_dir, 'suffix.html')
 trace_viewer_html = _ReadAsset(systrace_dir,

'systrace_trace_viewer.html')

 # Open the file in binary mode to prevent python from
changing the
 # line endings, then write the prefix.
 html_file = open(output_file_name, 'wb')

html_file.write(html_prefix.replace('{{SYSTRACE_TRACE_VIEWER
_HTML}}',
                                    trace_viewer_html))

 # Write the trace data itself. There is a separate section
of the form
```

```
 # <script class="trace-data" type="application/text"> ...
</script>
 # for each tracing agent (including the controller tracing
agent).
 html_file.write('<!-- BEGIN TRACE -->\n')
 for result in trace_results:
   html_file.write('  <script class="trace-data"
type="application/text">\n')
   html_file.write(_ConvertToHtmlString(result.raw_data))
   html_file.write('  </script>\n')
 html_file.write('<!-- END TRACE -->\n')

 # Write the suffix and finish.
 html_file.write(html_suffix)
 html_file.close()
```

مطابق این کد متوجه می شویم که عملیات پارس trace های خوانده شده در کد پایتون انجام نمی شود و تریس های خوانده شده به صورت hard code مستقیما بین دو تگ script که class="trace-" data به آن نسبت داده می شود به فایل خروجی اضافه میشوند و کدی که جهت پارس این داده ها و تصویر کردن داده ها استفاده می شود در prefix.html و systrace_trace_viewer.html (کد ها در مسیر Sdk\platform-tools\systrace\catapult\systrace\systrace است)قرار دارند.

در perfix.html تابع onLoad که بلافاصله پس از لود کردن فایل html فراخوانی می شود فرآیند پارس را آغاز می کند. ابتدا در این قسمت تریس هایی که در تگ script با نام کلاس trace-data هستند خوانده می شوند. در این قسمت باید تریس هایی که مربوط به atrace هستند را تشخیص دهیم و داده های power را از تریس های دیگر جدا کنیم (systrace اطلاعات دیگری را هم غیر از داده های atrace جمع آوری می کند مانند اسم thread ها و پردازه ها که این اطلاعات را از دایرکتوری proc در کرنل می خواند) .

ابتدا در تگ script جدیدی در فایل perfix.html لیستی از متغیر هایی که لازم داریم اضافه می کنیم:

```
<script>
    var powerlogs = []; //برای لاگ های پاور جداشده از لاگ های تریس
    var powertimestampStart = [];//تایم اولین لاگ تریس
    var powertimestamps = [];//تایم استمپ های تمامی لاگهای پاور
    var minPowerData = [];//کوچکترین داده پاور
    var V = []//ولتاژ
    var I = []//جریان
    var E = []//انرژی
```

```
</script>
```

سپس در همین تابع onLoad در perfix.html داده های power را جدا کرده و داخل powerlogs می ریزیم:

```
for (var i = 0; i < traceDataEls.length; i++) {

   var traceText = traceDataEls[i].textContent;
   var tracelogs = [];
   traceText = traceText.substring(1);
   if (powerlogs.length == 0) {
       tracelogs = traceText.split(/\r\n|\r|\n/);
       powerlogs = tracelogs.filter(function(item) {
           var finder = "write_power_ringbuffer";
           return eval('/' + finder + '/').test(item);
       });
   }
}
```

سپس تایم استمپ اولین تریس را که مبدا نمودار محسوب می شود را از تریس اول پیدا می کنیم (این کد در همان حلقه بالا قرار می گیرد) :

```
if (powerlogs.length != 0) {
   var linenumber = 0;
   var index = 0;
   var startindex = 0;
   for (var i = 0; i < tracelogs.length; i++) {
       if (tracelogs[i].includes("[")) {
           linenumber = i;
           break;
       }
   }
   console.log("tracelogss")
   console.log(tracelogs[linenumber])
   for (var i = 0; i < tracelogs[linenumber].length; i++) {
       if (tracelogs[linenumber + 2][i] == ':') {
           index = i - 1;
           break;
       }
   }
   for (var i = index; i >= 0; i--) {
       if (tracelogs[linenumber][i] == ' ' &&
```

```
parseFloat(tracelogs[linenumber][i + 1]) > 0) {
            startindex = i + 1;
            break;
        }
    }

powertimestampStart.push(parseFloat(tracelogs[linenumber].substring(startindex,
index)));
}
```

کد نهایی فایل prefix.html که از خط ۳۳ به بعد در ذیل آمده است :

```
<tr-ui-timeline-view>
    <track-view-container id='track_view_container'>          </track-view-
container>
</tr-ui-timeline-view>
<script>
    var powerlogs = [];
    var powertimestampStart = [];
    var powertimestamps = [];
    var minPowerData = []
    var V = []
    var I = []
    var E = []
</script>
<script>
    'use strict';
    var timelineViewEl;

    function onLoad() {
        timelineViewEl =        document.querySelector('tr-ui-timeline-view');
        timelineViewEl.globalMode = true;

        var traceDataEls = document.body.querySelectorAll('.trace-data');
        var traces = [];

        for (var i = 0; i < traceDataEls.length; i++) {

            var traceText = traceDataEls[i].textContent;
            var tracelogs = [];
            traceText = traceText.substring(1);
            if (powerlogs.length == 0) {
                tracelogs = traceText.split(/\r\n|\r|\n/);
                powerlogs = tracelogs.filter(function(item) {
```

```
                    var finder = "write_power_ringbuffer";
                    return eval('/' + finder + '/').test(item);
                });
            }
            if (powerlogs.length != 0) {
                var linenumber = 0;
                var index = 0;
                var startindex = 0;
                for (var i = 0; i < tracelogs.length; i++) {
                    if (tracelogs[i].includes("[")) {
                        linenumber = i;
                        break;
                    }
                }
                console.log("tracelogss")
                console.log(tracelogs[linenumber])
                for (var i = 0; i < tracelogs[linenumber].length; i++) {
                    if (tracelogs[linenumber + 2][i] == ':') {
                        index = i - 1;
                        break;
                    }
                }
                for (var i = index; i >= 0; i--) {
                    if (tracelogs[linenumber][i] == ' ' &&
parseFloat(tracelogs[linenumber][i + 1]) > 0) {
                        startindex = i + 1;
                        break;
                    }
                }

powertimestampStart.push(parseFloat(tracelogs[linenumber].substring(startindex,
index)));
            }
            tracelogs = []
                // Remove the leading newline.
            traces.push(traceText);
        }

        var m = new tr.Model();
        var i = new tr.importer.Import(m);
        var p = i.importTracesWithProgressDialog(traces, powerlogs);
        p.then(
            function() {
                timelineViewEl.model = m;
                timelineViewEl.updateDocumentFavicon();
```

```
                timelineViewEl.globalMode = true;
                timelineViewEl.viewTitle = 'Android System Trace';
            },
            function(err) {
                var overlay = new tr.ui.b.Overlay();
                overlay.textContent = tr.b.normalizeException(err).message;
                overlay.title = 'Import error';
                overlay.visible = true;
            });
    }
    window.addEventListener('load', onLoad);
</script>
```

سپس به سراغ فایل systrace_trace_viewer.html می رویم تا نمودار داده های پاور را اضافه کنیم ابتدا لازم است که داده های جریان و انرژی و ولتاژ و timestamp ها را جدا کنیم و همچنین مقدار کمینه جریان و انرژی و ولتاژ را به دست می آوریم تا برای offset هر نمودار استفاده کنیم:

```
let minCurrent = 1000000000000;
let minVoltage = 1000000000000;
let minEnergy = 1000000000000;
let temp;
for (var i = 0; i < powerlogs.length; i++) {
    var startindex = 0;
    var endindex = 0;
    for (var j = 0; j < powerlogs[i].length; j++) {
        if (powerlogs[i][j] == 'V' && powerlogs[i][j + 1] == ':') {
            var spaceindex = 0;
            for (var k = j + 2; k < powerlogs[i].length; k++) {
                if (powerlogs[i][k] == ' ' || powerlogs[i][k] == '\r' ||
powerlogs[i][k] == '\n') {
                    spaceindex = k;
                    break;
                }
            }
            temp = parseFloat(powerlogs[i].substring(j + 2, k));
            if (temp < minVoltage) {
                minVoltage = temp;
            }
            V.push(temp);
        } else if (powerlogs[i][j] == 'I' && powerlogs[i][j + 1] == ':') {
            var spaceindex = 0;
            for (var k = j + 2; k < powerlogs[i].length; k++) {
                if (powerlogs[i][k] == ' ') {
```

```
                    spaceindex = k;
                    break;
                }
            }
            temp = parseFloat(powerlogs[i].substring(j + 2, k));
            if (temp < minCurrent) {
                minCurrent = temp;
            }
            I.push(temp);
        } else if (powerlogs[i][j] == 'E' && powerlogs[i][j + 1] == ':') {
            var spaceindex = 0;
            for (var k = j + 2; k < powerlogs[i].length; k++) {
                if (powerlogs[i][k] == ' ') {
                    spaceindex = k;
                    break;
                }
            }
            temp = parseFloat(powerlogs[i].substring(j + 2, k));
            if (temp < minEnergy) {
                minEnergy = temp;
            }
            E.push(temp);
        }
        for (var h = 0; h < powerlogs[i].length; h++) {
            if (powerlogs[i][h] == ':' && parseFloat(powerlogs[i][h - 1]) >= 0)
{
                endindex = h - 1;
                break;
            }
        }
        for (var h = endindex; h >= 0; h--) {
            if (powerlogs[i][h] == ' ' && parseFloat(powerlogs[i][h + 1]) >= 0)
{
                startindex = h + 1;
                break;
            }
        }

    }
    powertimestamps.push(parseFloat(powerlogs[i].substring(startindex,
endindex)));
}

for (var i = 0; i < powertimestamps.length; i++) {
    powertimestamps[i] = Math.floor((powertimestamps[i] -
```

```
powertimestampStart[0]) * 1000);
}
minPowerData.push(minVoltage);
minPowerData.push(minCurrent);
minPowerData.push(minEnergy);
```

مرحله بعدی اضافه کردن نمودار ها است برای نمونه جهت اضافه کردن نمودار انرژی از روشی تقریبا مشابه روشی که داده های cpu usage نمودار می شوند استفاده می کنیم . تابع buildChartSeries محور y جدید ایجاد می شود. جهت تصویر کردن این نمودار کامپوننتی که از قبل در این فایل موجود بود استفاده می کنیم که واحد محور افقی آن ۱ میلی ثانیه است و برای هر واحد یک مقدار باید به آن داده شود بنابراین به ازای timestamp هایی که داده ثبت کردیم داده را به نمودار میدهیم و در نقاط دیگر 0 به نمودار می دهیم در نمودار انرژی اختلاف انرژی بین دو نقطه را نمایش می دهیم تا داده ها معنا دار تر باشند :

```
'use strict';
tr.exportTo('tr.ui.tracks', function() {
   const ColorScheme = tr.b.ColorScheme;
   const ChartTrack = tr.ui.tracks.ChartTrack;
   const EnergyUsageTrack = tr.ui.b.define('energy-usage-track', ChartTrack);
   EnergyUsageTrack.prototype = {
       __proto__: ChartTrack.prototype,
       decorate(viewport) {
           ChartTrack.prototype.decorate.call(this, viewport);
           this.classList.add('energy-usage-track');
           this.heading = 'Energy remaining';
       },
       initialize(model) {
           this.series = this.buildChartSeries_();
           this.autoSetAllAxes({
               expandMax: true
           });
       },
       get hasVisibleContent() {
           return true;
       },
       addContainersToTrackMap(containerToTrackMap) {
           containerToTrackMap.addContainer(this.series_, this);
       },
       buildChartSeries_(yAxis, color) {
           if (!this.hasVisibleContent) return [];
           yAxis = new tr.ui.tracks.ChartSeriesYAxis(0, undefined);
           const pts = new Array(powertimestamps[powertimestamps.length - 1] +
```

```
10);
            var powerptr = 0;
            var pointvalue = 0;
            var prePointValue = 0;
            var x = false;
            for (let i = 0; i < powertimestamps[powertimestamps.length - 1] +
10; i++) {
                if (powertimestamps[powerptr] == i) {
                    if (prePointValue == 0) {
                        pointvalue = 0;
                    } else {
                        pointvalue = prePointValue - E[powerptr];
                    }
                    prePointValue = E[powerptr];
                    powerptr += 1;
                    while (true) {
                        if (powertimestamps[powerptr] !=
powertimestamps[powerptr - 1])
                            break
                        powerptr += 1;
                    }
                } else {
                    pointvalue = 0;
                }
                pts[i] = new tr.ui.tracks.ChartPoint(undefined, i, pointvalue);
            }
            const renderingConfig = {
                chartType: tr.ui.tracks.ChartSeriesType.AREA,
                colorId: color
            };
            return [new tr.ui.tracks.ChartSeries(pts, yAxis, renderingConfig)];
        },
    };
    return {
        EnergyUsageTrack,
    };
});
```

در نهایت EnergyUsageTrack را باید به لیست track ها اضافه کنیم برای این کار در ModelTrack تابع appendEnergyUsageTrack را اضافه میکنیم و آن را در تابع _updateContentsForLowerMode صدا می زنیم (در این قسمت صرفا بخش هایی از کد آورده شده است):

```
'use strict';
   tr.exportTo('tr.ui.tracks', function() {
    /*قسمت هایی از کد های این قسمت حذف شده است*/
    ModelTrack.prototype = {
           __proto__: tr.ui.tracks.ContainerTrack.prototype,

    updateContentsForLowerMode_() {
               if (this.model_.userModel.expectations.length > 1) {
                   const mrt = new
tr.ui.tracks.InteractionTrack(this.viewport_);
                   mrt.model = this.model_;
                   Polymer.dom(this).appendChild(mrt);
               }
               if (this.model_.alerts.length) {
                   const at = new tr.ui.tracks.AlertTrack(this.viewport_);
                   at.alerts = this.model_.alerts;
                   Polymer.dom(this).appendChild(at);
               }
               if (this.model_.globalMemoryDumps.length) {
                   const gmdt = new
tr.ui.tracks.GlobalMemoryDumpTrack(this.viewport_);
                   gmdt.memoryDumps = this.model_.globalMemoryDumps;
                   Polymer.dom(this).appendChild(gmdt);
               }
               this.appendDeviceTrack_();
               this.appendCpuUsageTrack_();
               this.appendCurrentUsageTrack_();
               this.appendVoltageUsageTrack_();
               this.appendEnergyUsageTrack_();


               this.appendMemoryTrack_();
               this.appendKernelTrack_();
               const processes = this.model_.getAllProcesses();
               processes.sort(tr.model.Process.compare);
               for (let i = 0; i < processes.length; ++i) {
                   const process = processes[i];
                   const track = new tr.ui.tracks.ProcessTrack(this.viewport);
                   track.process = process;
                   if (!track.hasVisibleContent) continue;
                   Polymer.dom(this).appendChild(track);
               }
               this.viewport_.rebuildEventToTrackMap();
               this.viewport_.rebuildContainerToTrackMap();
```

```
                this.vSyncTimes_ = this.model_.device.vSyncTimestamps;
                this.updateAnnotations_();
            },
/*  در ادامه تابع های دیگر این بین وجود دارد که آورده نشده است  */

            appendEnergyUsageTrack_() {
                const track = new tr.ui.tracks.EnergyUsageTrack(this.viewport);
                track.initialize(this.model);
                if (!track.hasVisibleContent) return;
                this.appendChild(track);
            },
                                /*در ادامه کد پیاده سازی این قسمت ادامه دارد که کد های مربوط به خود systrace هستند
                                                                                                                */


.....
.....
.....

            };
        return {
            ModelTrack,
        };
```

نحوه ی تحلیل راه‌حل و اثبات کارایی(مثلا زمان تاخیر و مصرف حافظه و ...)

- **تست داده های باتری**

ابتدا تستی در جهت خواندن مقادیر خوانده شده در کرنل اندازه گیری ها به طور مستقیم از sysfs با اجرای دستورات shell خوانده میشوند. ابزار تست که جهت اینکار استفاده شده است یک اپلیکیشن است که در نکسوس نصب شده است. قسمت هایی از کد این اپلیکیشن :

از آن جایی که اجازه دسترسی به فایل سیستم های پاور به طور مستقیم وجود ندارد این مقدار ها باید در یک shell در حالت root خوانده شوند بنابراین لازم است instance از shell ایجاد کنیم با استفاده از stream لازم نیست که به ازای هر بار خواندن پردازه shell را از اول ایجاد کنیم :

```
suProcess = Runtime.getRuntime().exec("su");
DataOutputStream os = new
DataOutputStream(suProcess.getOutputStream());
DataInputStream osRes = new
DataInputStream(suProcess.getInputStream());
```

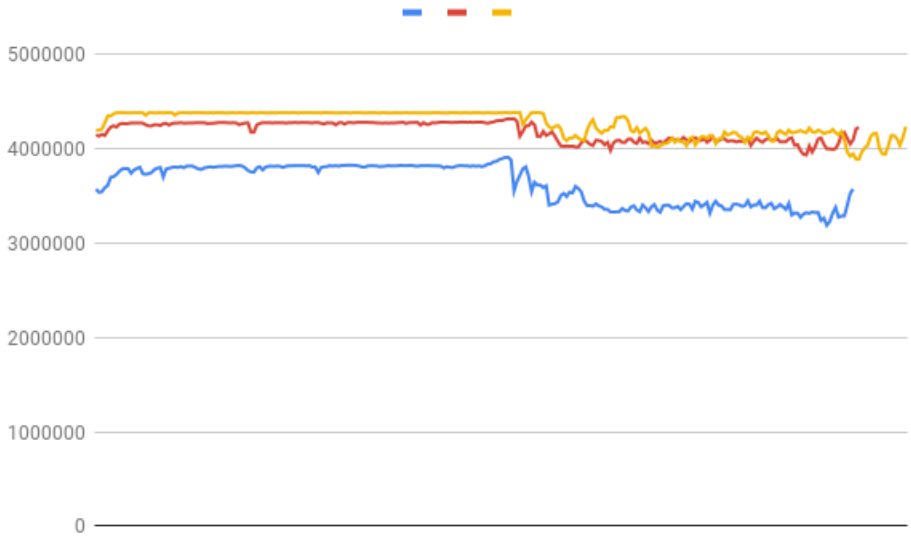
جهت اجرای پریودیک خواندن از باتری در این اپلیکیشن از handler در اندروید استفاده می کنیم که به دلیل این که در thread اصلی اجرا می شود سربار context switch ندارد . هر بار جهت خواندن داده ها :

```
os.writeBytes("cat
sys/class/power_supply/max17042xx_battery/current_now");
```

که خروجی دستور بالا در shell از DataInputStream قابل خواندن هستند. در نهایت داده های به دست آمده را در فایل روی sdcard ریخته می شود .

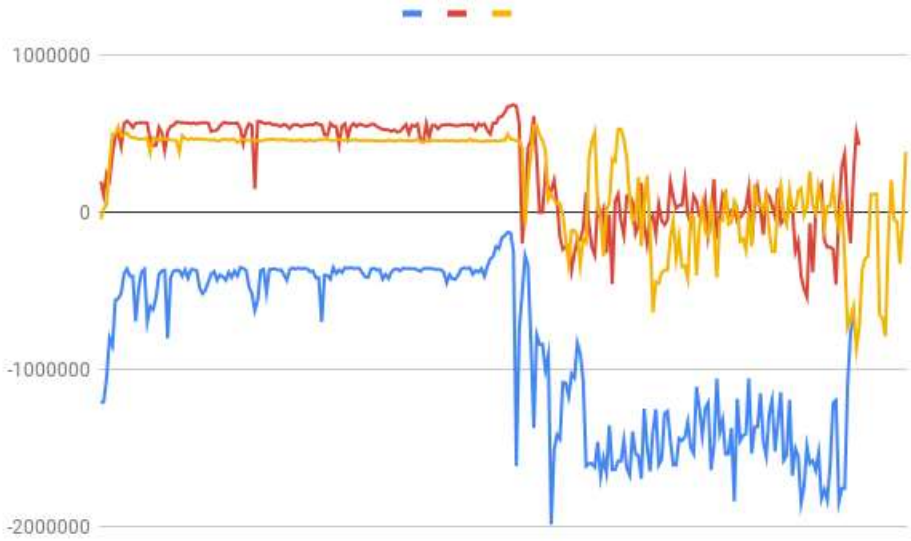
نمودار های زیر حاصل به تصویر کشیدن داده های حاصل از اپلیکیشن ذکر شده هستندکه شامل اندازه گیری جریان و ولتاژ و انرژی از طریق خواندن از فایل سیستم مربوط به آن است . این فایل سیستم ها درواقع interface به درایور باتری هستند و مقادیر از آن جا خوانده می شود (خطوط قرمز مربوط به اندازه گیری هنگام اتصال به usb و خطوط آبی بدون اتصال به usb و زرد مربوط به شارژر است) . در این تست در ۳۰ ثانیه ابتدایی عملیات خاصی انجام نمی شود و تنها صفحه نمایشگر در حالت پرنور در صفحه اپلیکیشن تست باتری باز هست و سپس از اپ خارج شده و ۱۰ ثانیه ویدیو ضبط شده است و در نهایت به اپ تست باتری بازگشته و تست متوقف شده است. همچنین در این تست ابتدا اندازه گیری بدون اتصالadb صورت گرفته است و سطح باتری در این حالت اندکی بیشتر است. همان طور که قابل مشاهده است اتصال به شارژر تاثیر زیادی دارد. بنابراین لازم است که حتما usb charging غیر فعال شود

Sys/class/power_supply/max170xx_battery/voltage_now



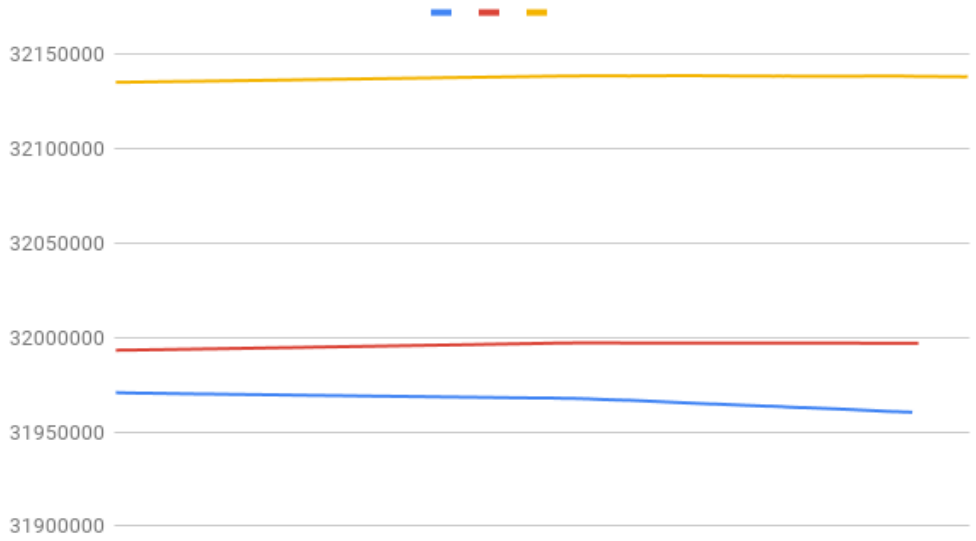
این مقادیر به میکرو ولت هستند

Sys/class/power_supply/max170xx_battery/current_now



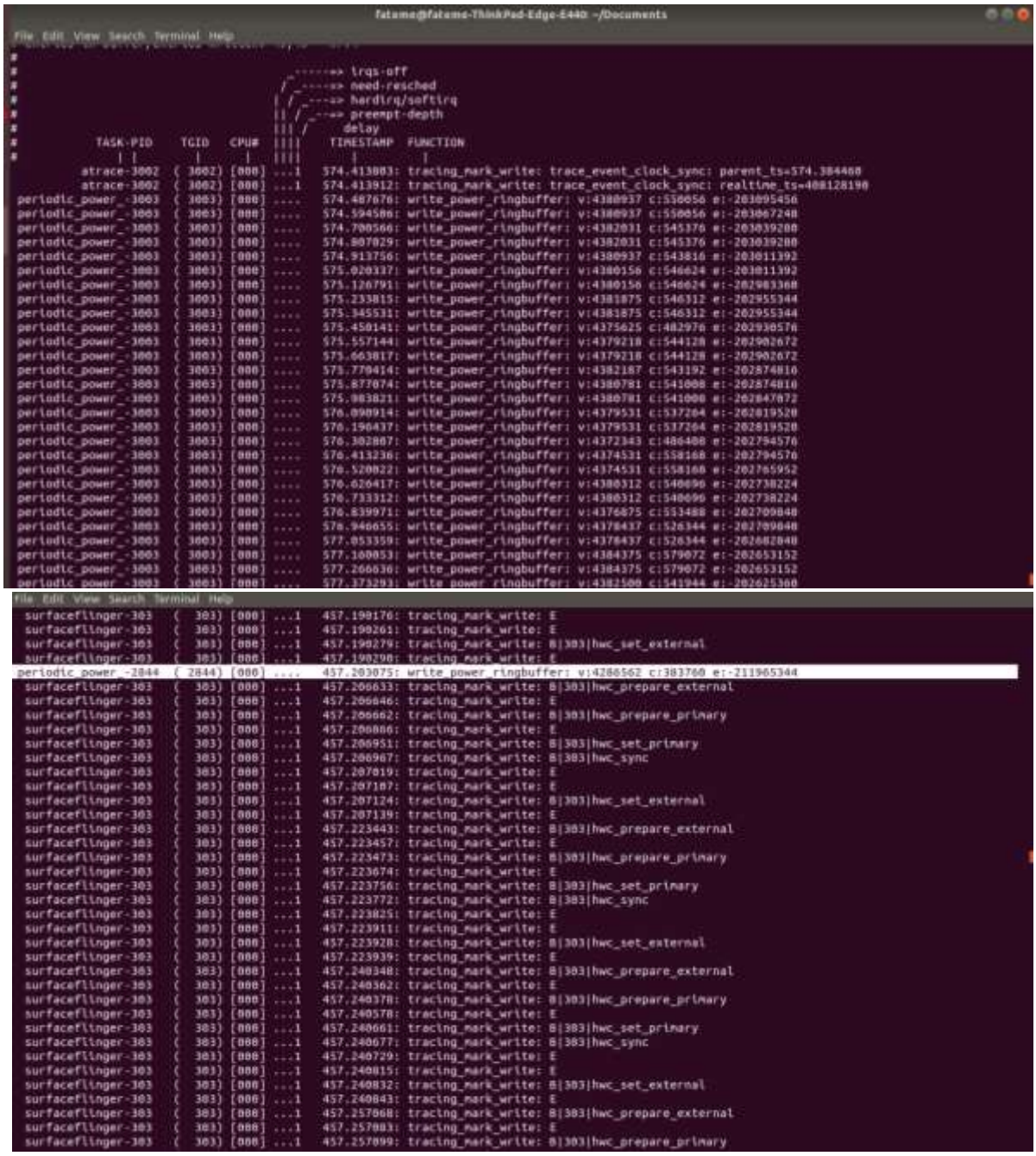
این اندازه گیری ها به میکروآمپر هستند و دقت اندازه گیری آن 156 میکرو آمپر است.

Sys/class/power_supply/max170xx_battery/charge_counter



این اندازه گیری ها به میکرو آمپر ساعت هستند و دقت آن ها ۵۰۰ میکرو آمپر ساعت است.

- تست داده های اضافه شده به ftrace



همان طور که در تصاویر بالا قابل مشاهده است پس از اجرای tracing این مقدار ها به درستی در بافر نوشته شده اند و در فایل d/tracing/trace قابل مشاهده هستند . با تکرار تست ها به نمونه قابل تاملی برخوردیم که در آن اختلاف دو تریس متوالی مربوط به انرژی اختلاف timestamp کمتر از ۱۰ میلی ثانیه داشتند که با توجه به شیوه پیاده سازی در کرنل این انتظار وجود داشت که این اختلاف کمتر از ۵۰ میلی ثانیه نباشد

```
periodic_power_-15464 (15464) [000] .... 18090.093080:
```

```
write_power_ringbuffer: V:3214687 I:-1340040 E:32347638

periodic_power_-12268 (12268) [000] .... 18090.093645:
write_power_ringbuffer: V:3214687 I:-1340040 E:32347638
```

که با بررسی بیشتر مشخص شد این خطا به دلیل استفاده ftrace از local clock هر cpu برای timestamp هر تریس است که همان طور که بیان شد نوع این کلاک می توان به global یا uptime تغییر داد تا این مشکل رفع شود

7- تست عملکرد

طرح تست

جهت سنجش ابزار از اپلیکیشن که به ازای آپدیت دیتای سنسور ژیروسکوپ داده ها را به سرور می فرستد استفاده شده است.
همان طور که بیان شد ابزار مورد نظر جریان و ولتاژ و انرژی کاسته شده را نشان می دهد.
علاوه بر این اپلیکیشن، یک اپلیکیشن دیگری که صرفا طی مدت زمان خاصی با نرخ زیادی reuqest و پس از آن مدت نرخ request کم یا ۰ میشود تست شده است تا تفاوت این دو حالت معلوم باشد.

نحوه اجرای تست (پیاده سازی)

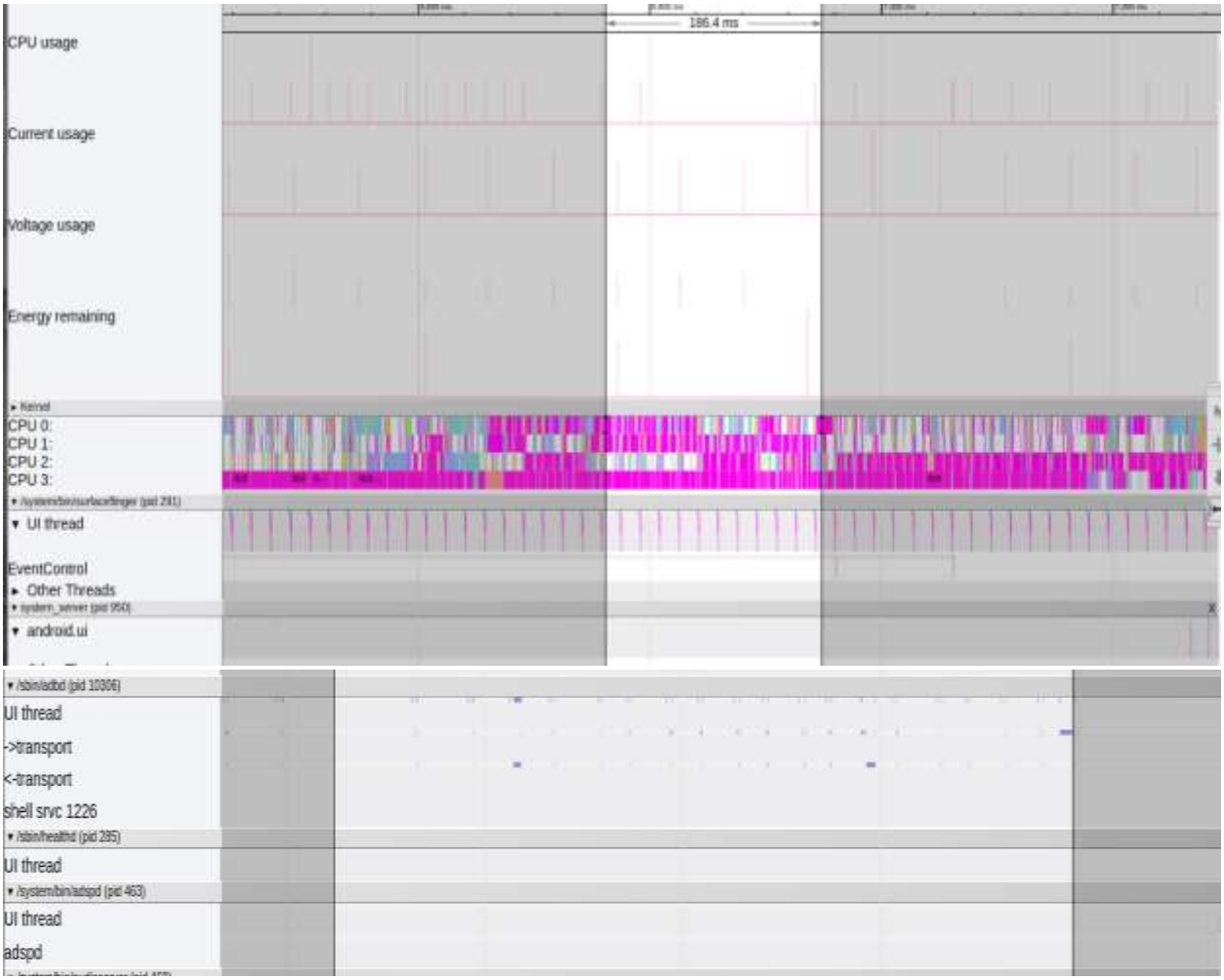
ابتدا لازم که که usb charging غیرفعال شود و سپس با چنین دستوری systrace اجرا می شود

```
python systrace.py -o mynewtrace.html -t 6 sched network hal
```

همچنین تمام اپ های دیگر باید force stop شوند تا باعث ایجاد تداخل نشوند.

نتایج تست‌های انجام شده

با بررسی تست متوجه می شویم که به علت فرکانس پایین اپدیت باتری نسبت به داده های performance امکان مقایسه مصرف سنسور و وای فای به خوبی وجود ندارد زیرا این کامپوننت ها به طور متناوب در هر دوره تناوب اپدیت باتری در حال استفاده شدن هستند . همچنین افزایش انرژی که در قسمت های پایانی تریس اتفاق افتاده است به دلیل استفاده atrace از shell جهت شروع فرآیند متوقف سازی تریس است همان طور که در تصویر قابل مشاهده است مصرف انرژی در بازه مشخص شده در بیشترین حالت است که با بررسی ترد ها و پراسس های مختلف متوجه می شویم که در این بازه shell و adbd فعال بوده اند.



8- هزینه نهایی

به علت ماهیت نرم افزاری پروژه، هزینه های اعم از خرید برد و ... این پروژه را شامل نمی شد ولی برای تست کد ها به علت اینکه نیاز بود اندروید روی گوشی دوباره از اول ریخته شود به یک گوشی جهت تست نیاز داشتیم.

9- پیوست‌های فنی

★ جزئیات ftrace و پیاده سازی آن:

مشخصات و جزئیات ابزار **ftrace:**

- معرفی:

به طور خلاصه یک ابزار tracing در سطح کرنل است که انواع متعددی دارد ولی چیزی که بین تمام انواع آن مشترک است ماهیت آن است که مربوط به trace کردن function call ها در سطح کرنل است، خال با توجه به نوع آن نحوه نمایش آن یا آپشن های موجود در آن متفاوت است. در ادامه به توضیحات بیشتری از این ابزار و انواع آن خواهیم پرداخت.

- اطلاعاتی که می توان از **Ftrace** گرفت:
  - PID
  - CPU number
  - Timestamp
  - Function name
  - Duration
- انواع **tracer** های **Ftrace:**

در واقع محتوای فایل سیستم available tracer تریسر های در دسترس را نشان میدهد. در گوشی های جدید به دلایل امنیتی خیلی از این tracer ها غیر فعال شده اند ولی با اضافه کردن کانفیگ های مورد نیاز برای هر tracer میتوان آن ها را در دسترس داشت که در ادامه نحوه فعال کردن دو مورد بیان خواهد شد.

- Function:

تمام functionهای در سطح کرنل را trace میکند.

```
# tracer: function
#
#           TASK-PID   CPU#    TIMESTAMP  FUNCTION
#              | |      |          |         |
            bash-4251  [01] 10152.583854: path_put <-path_walk
            bash-4251  [01] 10152.583855: dput <-path_put
            bash-4251  [01] 10152.583855: _atomic_dec_and_lock <-dput
```

- Function_graph:

در واقع نمونه کامل تر شده ای از function است با این تفاوت که function graoh علاوه بر ورودی تابع خروجی آن را هم میتواند trace کند در نتیجه میتوان اطلاعات درباره duration تایم اجرای یک تابع نیاز اطلاعات بدست آورد.

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |

 0)               |  sys_open() {
 0)               |    do_sys_open() {
 0)               |      getname() {
 0)               |        kmem_cache_alloc() {
 0)   1.382 us    |          __might_sleep();
 0)   2.478 us    |        }
 0)               |        strncpy_from_user() {
 0)               |          might_fault() {
 0)   1.389 us    |            __might_sleep();
 0)   2.553 us    |          }
 0)   3.807 us    |        }
 0)   7.876 us    |      }
 0)               |      alloc_fd() {
 0)   0.668 us    |        _spin_lock();
 0)   0.570 us    |        expand_files();
 0)   0.586 us    |        _spin_unlock();
```

- Wakeup:

بزرگترین latencyای را برای تسک های با اولویت بالا که پس از فعال شدن schedule میشوند، تریس و ذخیره میکند. در واقع همه تسک هایی که یک برنامه نویس انتظار دارد را تریس میکند.

```
# tracer: wakeup
#
wakeup latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 4 us, #2/2, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: sleep-4901 (uid:0 nice:0 policy:1 rt_prio:5)
    -----------------

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
  <idle>-0     1d.h4    0us+: try_to_wake_up (wake_up_process)
  <idle>-0     1d..4    4us : schedule (cpu_idle)
```

- Wakeup_rt:

به لحاظ عملکرد مشابه قبلی ولی بزرگترین latency را فقط برای RT taskها تریس و ذخیره میکند.

- Irqsoff:

درواقع قسمتی را تریس میکند که وقفه ها غیرفعال هستند و همچنین بیشترین latency را هم در فایل tracing_max_latency ذخیره میکند.هنگامی که داده های جدید حاوی مقدار بزرگتری باشند این مقدار با مقدار فعلی این فایل جابجا میشود.

```
# tracer: irqsoff
#
irqsoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 97 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: swapper-0 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: apic_timer_interrupt
 => ended at:   do_softirq

#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /
#                |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
  <idle>-0     0d..1    0us+: trace_hardirqs_off_thunk (apic_timer_interrupt)
  <idle>-0     0d.s.   97us : __do_softirq (do_softirq)
  <idle>-0     0d.s1   98us : trace_hardirqs_on (do_softirq)
```

- Preemptoff:

مشابه مورد قبل، با این تفاوت که مقدار زمانی غیر فعال بودن رابرای هر preemption تریس و ذخیره میکند.

```
# tracer: preemptoff
#
preemptoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 29 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: sshd-4261 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: do_IRQ
 => ended at:   __do_softirq

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
    sshd-4261  0d.h.    0us+: irq_enter (do_IRQ)
    sshd-4261  0d.s.   29us : _local_bh_enable (__do_softirq)
    sshd-4261  0d.s1   30us : trace_preempt_on (__do_softirq)
```

- Preemptirqsoff:

مشابه دو مورد قبل، با این تفاوت که بیشترین زمان غیر فعال بودن ر برای irqs و یا preemption تریس و ذخیره میکند.

```
# tracer: preemptirqsoff
#
preemptirqsoff latency trace v1.1.5 on 2.6.26-rc8
--------------------------------------------------------------------
 latency: 293 us, #3/3, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: ls-4860 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: apic_timer_interrupt
 => ended at:   __do_softirq

#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /
#              |||||     delay
#  cmd     pid ||||| time  |   caller
#     \   /    |||||   \   |   /
      ls-4860  0d...    0us!: trace_hardirqs_off_thunk (apic_timer_interrupt)
      ls-4860  0d.s.  294us : _local_bh_enable (__do_softirq)
      ls-4860  0d.s1  294us : trace_preempt_on (__do_softirq)
```

- Hw-branch-tracer:

از BTS CPU feature در cpuهای 86x استفاده میکند برای

اینکه بتواند تمام branch های درحال اجرا را trace کند.

```
# tracer: hw-branch-tracer
#
# CPU#        TO  <-  FROM
   0  scheduler_tick+0xb5/0x1bf   <-  task_tick_idle+0x5/0x6
   2  run_posix_cpu_timers+0x2b/0x72a     <-  run_posix_cpu_timers+0x25/0x72a
   0  scheduler_tick+0x139/0x1bf          <-  scheduler_tick+0xed/0x1bf
   0  scheduler_tick+0x17c/0x1bf          <-  scheduler_tick+0x148/0x1bf
   2  run_posix_cpu_timers+0x9e/0x72a     <-  run_posix_cpu_timers+0x5e/0x72a
   0  scheduler_tick+0x1b6/0x1bf          <-  scheduler_tick+0x1aa/0x1bf
```

- Nop:

در حالتی که هیچ یک از تریسر ها در دسترس نباشند nop در فایل current_tracer نوشته میشود درواقع به طور معمول در گوشی های جدید به دلایل امنیتی و غیره تنها tracer در دسترس غالبا همان nop میباشد که به این معناست که تمام تریسر ها غیر فعال هستند.

برای فعال سازی و استفاده از این ها بسته به نوع تریسر باید configهایی فعال شوند و کرنل دوباره build شود.

سپس باید نوع تریسر در فایل current_tracer نوشته شود و فایل tracing_on یک شود و سپس تست از گوشی مانند اجرای یک اپلیکیشن خاص یا ... انجام شود و درنهایت فایل tracing_on صفر شود.پس از انجام مراحل قبل، برای دیدن خروجی باید فایل trace در ترمینال cat شود. در قسمت بعد نوع دستورات هر مرحله در ترمینال بیان شده است.

- نحوه کارکرد:

با استفاده از این ابزار ما میتوانیم از اتفاقات سطح کرنل باخبرشویم ftrace توانایی این را دارد که نشان دهد چه events موجب fail شدن برنامه میشود و میتواند به برنامه نویس راهبرد بهتری بدهد برای debugging.

برای استفاده به عنوان ابزار debugging باید configuration های

CONFIG_FUNCTION_TRACER

CONFIG_FUNCTION_GRAPH_TRACER

CONFIG_STACK_TRACER

CONFIG_DYNAMIC_FTRACE

فعال شوند.

به طور خاص CONFIG_DYNAMIC_FTRACE امکان فیلتر کردن فانکشن های برای تریس را میدهد.

روش کارکرد این ابزار به این شیوه است که کدهایی که مخصوص به trace فانکشن ها هستند به این فانکشن ها اضافه میشوند ولی هنگامی که trace نخواهیم هیچ اثری بر فانکشن ندارند ولی در هنگام trace به کمک این کدهای اضافه شده میتوان از ftrace خروجی گرفت به بیان دیگر یک ابزار dynamic tracing میباشد به این صورت که کرنل، instructionهایی به کد اسمبلی برنامه اضافه میکند تا ابزار trace هر بار از اجرای این فانکشن با خبر شوند این مجموعه instruction ها در حالتی که استفاده از trace نداریم مانند nop در function هستند و تاثیری بر عملکرد آن ندارند اما هنگامی که شروع به trace میکنند overhead این instruction ها به اجرای فانکشن افزوده میشود.

هنگامی که کرنل function tracing را فعال میکند kernel build system پرچم -pg را اضافه میکند به کامپایلر توابع تا کدهای اضافه شده از حالت nop خارج شوند این فرآیند اسمبلی را در اصطلاح count میگویند.

لاگ های تریس در ring buffer نگهداری میشوند که یک ساختمان داده است که برای نگهداری trace data استفاده میشود.

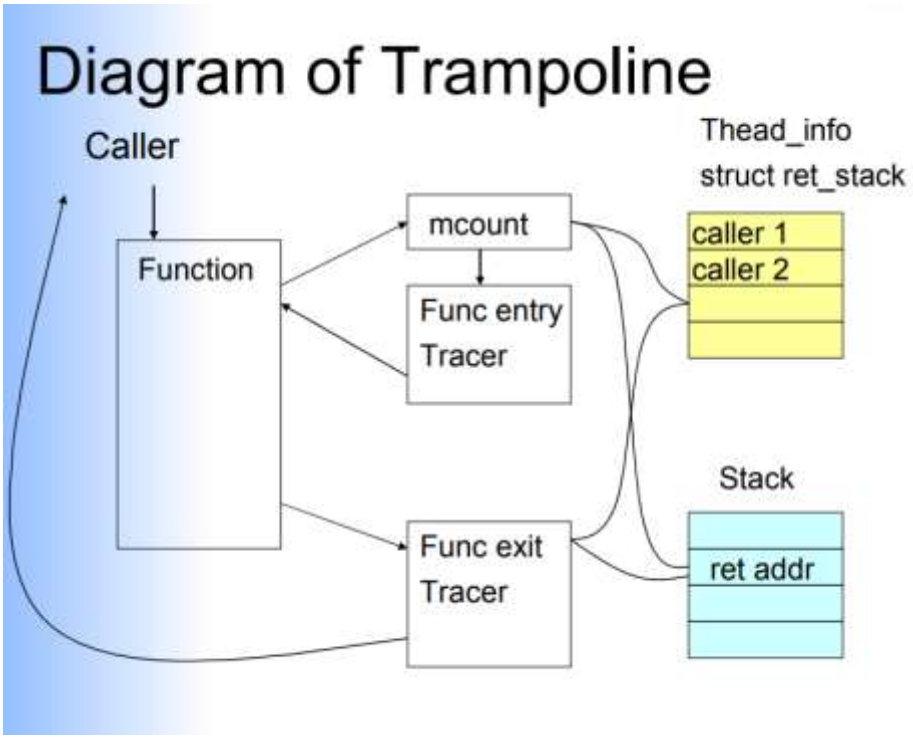
هنگامی که فانکشن در لیست set_ftrace_filter قرار میگیرد شروع به trace شدن میکند این کار کمک میکند تا از overhead بالایی که قبلا به آن اشاره شد جلوگیری گردد بدین شکل که تنها فانکشن های در این لیست trace خواهند شد و فانکشن هایی که در set_ftrace_notrace قرار دارند trace نمیشوند.

Function graph tracer هم داخل یک فانکشن و هم خارج از یک فانکشن را تریس میکند با استفاده از این ویژگی از داخل یک فانکشن هم میتوان با خبر شد.همچنین این ابزار تایمی که این فانکشن اجرا شده از لحظه فراخوانی تا بازگشت را ثبت میکند و در ستون duration نمایش میدهد البته این تایم overhead خود فانکشن های tracer را هم شامل میشود.

Ftrace به عنوان ابزار debugger از تابع trace_printk() استفاده میکند این تابع خروجی خود را در buffer مینویسد و در trace file ها قابل خواندن است. همچنین میتوان مانند کامنت در گراف function_graph نمایش داد.

این ابزار همچنین این امکان را هم میدهد که بتوانیم بفهمیم چه چیزی یک تابع خاص را فراخوانی میکند که این با استفاده از پیاده سازی یک backtracer برای آن فانکشن خاص است البته چون انجام این کار زمان زیادی میبرد و باعث lock سیستم میشود استفاده از back trace بطور مستقیم بهتر است. این کار با استفاده از function_stack_trace و فعال سازی این فیچر انجام میشود.

در ftrace , function و function_graph موجود هستند که function تنها ورودی توابع را trace میکند و برای اینکه exit را هم داشته باشیم از trampoline استفاده میشود.



دراین دیاگرام mcount درواقع ادرس بازگشت تابع زا ذخیره میکند و با ادرس trampoline جاگذاری میکند.

Function graph این اجازه را به ما میدهد که علاوه بر داشتن زمان ورود به تابع که توسط function tracer در دسترس است زمان خروج آن را نیز داشته باشیم که این امکان برای محاسبه مدت زمان اجرای فانکشن کاربرد دارد .

زمانی که پرچم -pg فعال میشود ftrace نیاز دارد تا رجیستر و شرایط استک را قبل از بازگشت به تابع منطبق کند و سپس از این طریق ftrace میتواند خروجی فانکشن را هم تریس کند. اینکار با استفاده از trampoline انجام میشود هنگامی که وارد یک فانکشن میشویم مقدار ادرسی که تابع instrument شده از انجا فراخوانی می شود را ذخیره میکند بعد از اینکه فانکشن گراف فراخوانی شد توسط ftrace ، مقدار ادرس را با ادرس یک ادرس routine ftrace برای هندل مردن مقدار بازگشت تریس جابجا میکند سپس ftrace اجرا میشود و هنگامی که به پایان رسید به جای اینکه به ادرس واقعی خود بازگردد به ادرسی که مربوط به ftrace است بازمیگردد سپس ftrace ، فانکشن گراف را برای استخراج داده ها دوباره فراخوانی میکند

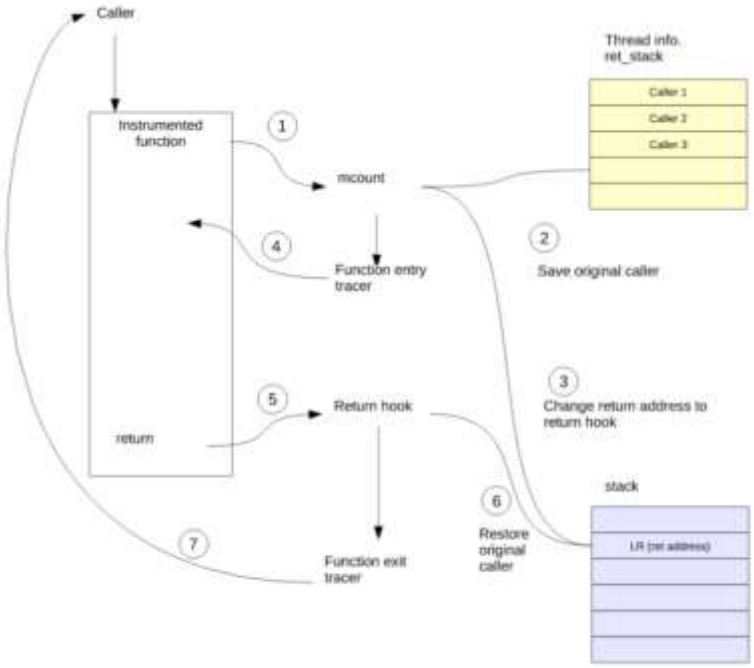


Figure 3: Mcount Handling and Return Trampoline

اطلاعات کاملتری از ساختار ftrace در این لینک قرار دارد.

- **Ring buffer:**

داده های ftrace قبل از نوشته شدن در فایل سیستم ها ابتدا در ring buffer ریخته میشوند که در ادامه به تشریح این قسمت خواهیم پرداخت:

از تعدادی page تشکیل شده است که هر cpu به طور جداگانه لیست صفحات خودش را دارد. یک writer تنها میتواند در بافری بنویسد که به cpu ایی وصل شده است که در حال حاضر اجرا میشود ولی برای خواندن از هر buffer ی میتوان خواند.

یکی از راه هایی که توسط آن میتوان این بافر را فعال یا غیر فعال کرد استفاده از tracing_on است به این صورت که در صورت ۱ بودن فایل ، اطلاعات میتوانند درون صفحات این بافر نوشته شوند البته همانطور که در توضیحات tracing_on آمده است این فایل تنها نوشتن در این بافر را تحت تاثیر قرار میدهد و روی overhead زمانی کدهای اسمبلی در فانکشن های سطح کرنل تاثیری ندارد.برای نوشتن در ring buffer ، سه لایه باید فعال باشند.

a. Global flag:این flag میتواند تمام ring buffer ها را فعال یا غیر فعال کند و دو بیت on , disable دارد.

On	Disable	
0	0	Ring buffers are off
1	0	Ring buffers are on
X	1	ring buffers are permanently disabled

b. Ring_buffer
c. Per cpu buffer

Ring buffer دو حالت overwrite , producer دارد.

در حالت اول producer در حال پر کردن بافر است قبل از آنکه consumer بتواند چیزی بخواند در این حالت ممکن است producer داده های قدیمی را overwrite کند که موجب از دست رفتن اطلاعات میشود.

در حالت دوم بافر بعد از پر شدن به producer امکان نوشتن نمیدهد در نتیجه ممکن است در این حالت داده های فعلی از دست بروند.

به علاوه هیچ دو writer نمیتوانند در یک زمان در بافر بنویسند اما امکان اینکه یک writer به دیگری interrupt بدهد وجود دارد ولی باید writer یی که interrupt داده کار نوشتنش را قبل از ادامه کار بعدی تمام کند این یک نکته کلیدی برای الگوریتم نوشتن است. در اینجا نوشتن در بافر مانند استک عمل میکند در واقع شبیه به روند زیر است:

writer1 start

<preempted> writer2 start

<preempted> writer3 start

writer3 finishes

writer2 finishes

writer1 finishes

مانند نوشتن، هیچ دو reader هم در یک زمان نمیتوانند بخوانند و در عین حال برخلاف writer ها حتی interupt هم نمیتوانند داشته باشند. یک reader نمیتواند یک writer را متوقف کند اما میتواند در حالی که نویسنده در حال نوشتن است از بافر بخواند اما این عمل نیازمند این است که خواننده در یک پراسس جداگانه باشد.

همانطور که قبلا اشاره شد ring buffer از مجموعه ای از صفحات است که به طور لینک لیستی به هم متصل هستند برای خواندن یک صفحه reader باید تخصیص داده شود به خواننده که این صفحه در لینک لیست بافر نیست به علاوه head_page, tail_page , commit_page هر سه به صفحات یکسانی اشاره دارند. یک reader_pageی که تخصیص داده میشود دارای یک next pointer به head page است و یک previous pointer به صفحه قبل از head page.

اطلاعات کاملتری درباره ساختار نوشتن و خواندن از ring_buffer دراین لینک قرار دارد.

- فایل سیستم های مورد نیاز برای **Ftrace**:
  - ➢ Current_tracer: در اکثر گوشی های غیر روت به طور دیفالت nop است ولی در کل تریسر مورد استفاده را نشان میدهد.
  - ➢ Available_tracer: لیست تریسر هایی که برای تریس کردن در دسترس هستند را نشان میدهد این تریسر ها از قبل در کرنل کامپایل شده اند و برای اضافه کردن تریسر جدید ما نیاز به کامپایل دوباره داریم برای اینکه از تریسر های موجود در این لیست استفاده کنیم کافی است که تریسر مورد نظر را در فایل current_tracer از طریق adb بگذاریم.

```
echo "tracer_name" > current_tracer
```

که "tracer_name" درواقع تریسر دلخواه ما است که یکی از موارد available trace است.

  - ➢ Tracing_on: اطلاعات این فایل شامل ۰ و یا ۱ است. ۰ به معنی این است که ابزار تریسر اجازه نوشتن در بافر را ندارد و ۱ به معنی داشتن این اجازه است. البته این ابزار تنها نوشتن را فعال یا غیر فعال میکند ممکن است که این فایل ۰ باشد ولی همچنان ما overhead زمانی تریسینگ را داشته باشیم.
  - ➢ Trace: این فایل خروجی تریس را به فرمت قابل خواندنی ذخیره میکند.
  - ➢ Trace_pipe: این فایل شبیه فایل قبلی است با این تفاوت که این فایل همراه با tracing نوشته میشود البته خواندن از این فایل block است تا زمانی که داده جدید بازیابی شود. فایل قبلی تا زمانی که تریسی فعال نشود دوباره داده های تکراری خواهد داشت ولی این فایل چون به صورت لایو دیتای خود را تغییر میدهد اطلاعات آن با هر بار خواندن فرق دارد.
  - ➢ Tracer_options: اطلاعات این فایل در واقع به کاربر اجازه میدهد که دیتای فایل های قبل را کنترل کند و در کل تعیین کند که تریسر چگونه کار کند.
  - ➢ Tracing_max_latency: این فایل حاوی اطلاعات latency بزرگترین تایم است یعنی دیتای جدید تریسر از دیتای موجود در این فایل بزرگتر باشد در این فایل ذخیره خواهد شد.
  - ➢ Tracing_thresh: تنها زمانی که اطلاعات تریسر از latency ذخیره شده در این فایل بزرگ تر باشد ذخیره خواهد شد.

- Set_ftrace_pid: در این فایل میتوان با قرار دادن pid مورد نظر فقط فانکشن های مربوط به آن pid را تریس کرد. برای یافتن pidهای موجود میتوان از دستور ps لیست pidهای درحال اجرا و موجود را یافت.
- Max_graph_depth: توسط فانکشن گراف استفاده میشود و عدد۱ اولین فانکشن کرنلی که از userspace صدا زده شده را نشان میدهد.
- Trace_clock: مرجع تایمی که timestamp نمایش میدهد.
  - Local: کلاک پیش فرض که از کلاک هر cpu استفاده می کند که لزوما با هم هماهنگ نیستند.
  - Global : کلاکی که با همه cpu ها هماهنگ است و اندکی کندتر از local است
  - Counter : در واقع کلاک نیست و یک atomic counter است این شمارنده یکی یکی به صورت هماهنگ با همه cpu ها مقدار آن افزوده می شود.
  - Uptime : در واقع همان شمارنده jiffies است که و تایم استمپ به دست آمده نسبت به زمان boot است
- Uprobe_events: اضافه کردن dynamic tracepoints به برنامه برای اینکه یک pid خاص تریس نشود میتوان در فایل filter در دایرکتوری sys/kernel/debug/tracing/events/raw_syscalls/sys_enter با نوشتن common_pid != pid که pid شماره پراسس مورد نظر است حتی در این فایل میتوان چندین pid را ban کرد که اینکار با && کردن همه common_pid != pid ها صورت میگیرد برای اینکه این محدودیت ها برداشته شود میتوان 0 را در این فایل نوشت تا همه pid ها تریس شوند.
- Trace_marker: برای همزمان کردن اتفاقات درون کرنل و سطح کاربر این فایل ایجاد شده است.یک راه برای نوشتن در ring buffer از سطح کاربر است.
- Available_filter_functions:لیست فانکشنهایی که قابل تریس کردن هستند در کرنل را نشان میدهد و میتوان آنها را با استفاده از فایل سیستم های set_ftrace_filter و set_ftrace_notrace فعال یا غیر فعال کرد برای تریس کردن.برای مشاهده محتوای این فایل از دستور زیر استفاده میشود:

```
cat available_filter_functions
```

- Set_ftrace_filter: برای اینکه تنها فانکشنهای خاصی را بتوانیم تریس کنیم از این فایل استفاده میشود به این صورت که از بین فانکشن هایی که در available_filter_functions هستند، فانکشن هایی را که میخواهیم در این فایل مینویسیم.

```
Echo function_name > set_ftrace_filter
```

که function_name نام تابعی است که میخواهیم آن را تریس کنیم.

- Set_ftrace_notrace:برعکس فایل قبل، زمانی که نمیخواهیم فانکشنی تریس شود آن را در این فایل قرار میدهیم دستورات مانند قبل تنها نام فایل سیستم، set_ftrace_notrace است.
- Buffer_size_kb:مقدار سایز بافری که هر cpu buffer میتواند داشته باشد را تعیین میکند.برای هر cpu این مقدار یکسان است و مقدار موجود در این فایل درواقع برای هر cpu را نشان میدهد نه مقدار کل.
- Trace_option: آپشن های زیادی برای ftrace وجود دارد ولی optionیی که در این پروژه مورد نیاز ما است، overwrite است. این option اگر ۱ باشد، این امکان را به ما میدهد که در هنگامی که ring buffer پر شد، بتوان از داده های قدیمی صرف نظر و داده های جدید را ذخیره کرد و اگر ۰ باشد داده های جدید تا زمانی که داده های قدیمی خوانده نشده اند از دست خواهد رفت. محتوای این فایل بدین صورت است که اگر در ابتدای optionیی no وجود داشت یعنی این option غیر فعال است، در غیر اینصورت فعال است. برای دیدن محتوای این فایل:

```
cat trace_option
```

و برای تغییر در فعال بودن یا نبودن option ها:

```
echo no+"option_name" > trace_option
```

دستور بالا برای غیر فعال کردن option است برای مثال:

```
echo noprint-parent > trace_options
```

و برای فعال کردن هم:

```
echo print-parent > trace_options
```

توضیحات چند نمونه از optionها:

- Print_parent: در function tracer، عدم نمایش یا نمایش functionیی که به عنوان parent برای فانکشن دیگر تعریف شده است در داده های تابع فرزند. منظور از parent در واقع تابعی است که آن را فراخوانی کرده است.
- sym-offset:علاوه بر نام تابع، offset آن را هم نمایش دهد. بطور مثال:

```
ktime_get+0xb/0x20
```

- sym-addr: مانند قبلی، علاوه بر نام تابع آدرس آنرا هم نمایش دهد.
- Latency_format:این option، تریس را تغییر میدهد به این صورت که داده های بیشتری از latency را تریس کرده و نمایش میدهد.
- Sched-tree: این option تمام تسک های موجود در runqueue را هم تریس میکند ولی overhead زیادی خواهد داشت اگر runqueue تسک های زیادی را شامل شود.

البته بعضی از این فایل ها مربوط به tracer خاصی در اندروید است که برای مثال set_ftrace_pid برای function , function_graph می باشد این فایل ها تا زمانی که این قابلیت در گوشی فعال نباشند وجود ندارند ولی پس از فعال سازی توسط قطعه کدی که در فایل ftrace.c و در خط ۴۶۸۲ وجود دارد این فایل ها ساخته میشوند.

فایل سیستم های متعددی در کرنل وجود دارند که از طریق این لینک به صورت کامل تری قابل مشاهده هستند که طبق توضیحات بالا شاید بسیاری از این قابلیت های اشاره شده و فایل ها در دسترس نباشند به صورت عادی و برای دسترسی به آنها باید کانفیگ های مختص خودشان فعال و کرنل دوباره build شود.

برای دسترسی به محتوای این فایل ها باید در ترمینالی که توسط دستور adb shell به کرنل اندروید متصل شده است و رفتن به مسیر فایل ها که در اکثر گوشی ها در مسیر d/tracing قرار دارد

دستور زیر را وارد کرد:

```
cat "filename"
```

و یا برای تغییر آن باید دستور زیر را وارد کرد:

```
echo "value" > "filename"
```

که در واقع value مقداری است که میخواهیم در فایل نوشته شود و filename نام فایلی است که میخواهیم مقدار را در آن بنویسیم. مثال هایی از این دستور در بالا ذکر شده است.

Ftrace از debugfs به منظور کنترل داشتن بروی فایلها تا زمانی که فایلها خروجی خود را نشان دهد استفاده میشود.هر تریسری که در ftrace انتخاب میشود یک دایرکتوری در debugfs میسازد که به آن tracing میگویند.این دایرکتوری خروجی فایل های ftrace است.

- **نحوه نگهداری اطلاعات در** ring_buffer

```
struct buffer_data_page  {
u64 time_stamp; // page time stamp
local_t commit; // write committed index
unsigned char data[] RB_ALIGN_DATA; // data of buffer page
};
```

در این struct اطلاعات داده های هر page نگه داشته میشود که این struct خود درون یک struct دیگر است:

```
struct buffer_page {
struct list_head list; // list of buffer pages
local_t write; // index for next write
unsigned read; // index for next read
local_t entries; // entries on this page
unsigned long real_end; // real end of data
```

```
struct buffer_data_page *page; // Actual data page
};
```

که در این استراکت اطلاعات هر صفحه نگهداری میشود. برای اینکه از اندازه داده و اطلاعات هر صفحه مطلع باشیم از تابع (size_t ring_buffer_page_len(void *page میتوان استفاده کرد.

اطلاعات کلی ring_buffer از آنجا که هر ring_buffer , cpu خودش را دارد در استراکت دیگری ذخیره میشود.

```
struct ring_buffer_per_cpu {
    int             cpu;
    atomic_t            record_disabled;
    struct ring_buffer      *buffer;
    raw_spinlock_t          reader_lock;    /* serialize readers */
    arch_spinlock_t         lock;
    struct lock_class_key       lock_key;
    unsigned int            nr_pages;
    struct list_head        *pages;
    struct buffer_page      *head_page; /* read from head */
    struct buffer_page      *tail_page; /* write to tail */
    struct buffer_page      *commit_page;   /* committed pages */
    struct buffer_page      *reader_page;
    unsigned long           lost_events;
    unsigned long           last_overrun;
    local_t             entries_bytes;
    local_t             entries;
    local_t             overrun;
    local_t             commit_overrun;
    local_t             dropped_events;
    local_t             committing;
    local_t             commits;
    unsigned long           read;
    unsigned long           read_bytes;
    u64             write_stamp;
    u64             read_stamp;
    /* ring buffer pages to update, > 0 to add, < 0 to remove */
    int             nr_pages_to_update;
    struct list_head        new_pages; /* new pages to add */
```

```
    struct work_struct      update_pages_work;
    struct completion       update_done;

    struct rb_irq_work      irq_work;
};
```

این استراکت به طور کلی تر همراه با دیتاهای دیگری از جمله شماره cpu و lock و ... در استراکت ring_buffer ذخیره شده اند.

```
struct ring_buffer {
    unsigned            flags;
    int             cpus;
    atomic_t            record_disabled;
    atomic_t            resize_disabled;
    cpumask_var_t           cpumask;

    struct lock_class_key       *reader_lock_key;

    struct mutex            mutex;

    struct ring_buffer_per_cpu  **buffers;

#ifdef CONFIG_HOTPLUG_CPU
    struct notifier_block       cpu_notify;
#endif
    u64             (*clock)(void);

    struct rb_irq_work      irq_work;
};
```

این استراکت در واقع کامل ترین استراکت موجود است که استراکت های ابزار tracer مانند trace_array از این ساختار داده(ring_buffer) بطور مستقیم استفاده میکنند.

در فایل trace.c یک متغیر global داریم به نام global_trace که در واقع یک توصیف کننده است که tracing buffer را در حالت live نگه میدارد و برای هر cpu یک لینک لیست از صفحاتی دارد که قرار است ذخیره شوند به عنوان دیتای trace. این متغیر از جنس struct trace_array می باشد که تعریف این استراکت در trace.h خط 179 موجود است. بزرگترین ساختار داده ای است که tracer ها از آن استفاده میکنند که current_tracer را هم نگه داری میکند علاوه بر برخی اطلاعات دیگر.

```
struct trace_array {
    struct list_head    list;
    char            *name;
    struct trace_buffer trace_buffer;
#ifdef CONFIG_TRACER_MAX_TRACE
    struct trace_buffer max_buffer;
    bool            allocated_snapshot;
#endif
    int         buffer_disabled;
    struct trace_cpu    trace_cpu;  /* place holder */
#ifdef CONFIG_FTRACE_SYSCALLS
    int         sys_refcount_enter;
    int         sys_refcount_exit;
    DECLARE_BITMAP(enabled_enter_syscalls, NR_syscalls);
    DECLARE_BITMAP(enabled_exit_syscalls, NR_syscalls);
#endif
    int         stop_count;
    int         clock_id;
    struct tracer       *current_trace;
    unsigned int        flags;
    raw_spinlock_t      start_lock;
    struct dentry       *dir;
    struct dentry       *options;
    struct dentry       *percpu_dir;
    struct dentry       *event_dir;
    struct list_head    systems;
    struct list_head    events;
    struct task_struct  *waiter;
    int         ref;
};
```

یکی از فیلدهای این استراکت trace_buffer است که این متغیر هم از جنس استراکت است که ring_buffer در این استراکت قرار دارد.

```
struct trace_buffer {
    struct trace_array      *tr;
    struct ring_buffer      *buffer;
    struct trace_array_cpu __percpu *data;
    cycle_t             time_start;
    int             cpu;
};
```

بیشترین سایز ring_buffer 2*BUF_PAGE_SIZE می باشد و توسط تابع ring_buffer_resize میتوان آنرا resize کرد.

```
int ring_buffer_resize(struct ring_buffer *buffer, unsigned long size,
           int cpu_id)
{
    struct ring_buffer_per_cpu *cpu_buffer;
    unsigned nr_pages;
    int cpu, err = 0;

    /*
           * Always succeed at resizing a non-existent buffer:
           */
    if (!buffer)
        return size;

    /* Make sure the requested buffer exists */
    if (cpu_id != RING_BUFFER_ALL_CPUS &&
              !cpumask_test_cpu(cpu_id, buffer->cpumask))
        return size;

    nr_pages = DIV_ROUND_UP(size, BUF_PAGE_SIZE);

    /* we need a minimum of two pages */
    if (nr_pages < 2)
        nr_pages = 2;

    size = nr_pages * BUF_PAGE_SIZE;

    /*
           * Don't succeed if resizing is disabled, as a reader might be
           * manipulating the ring buffer and is expecting a sane state while
           * this is true.
           */
    if (atomic_read(&buffer->resize_disabled))
        return -EBUSY;

    /* prevent another thread from changing buffer sizes */
    mutex_lock(&buffer->mutex);
```

```
if (cpu_id == RING_BUFFER_ALL_CPUS) {
    /* calculate the pages to update */
    for_each_buffer_cpu(buffer, cpu) {
        cpu_buffer = buffer->buffers[cpu];

        cpu_buffer->nr_pages_to_update = nr_pages -
                    cpu_buffer->nr_pages;
        /*
        * nothing more to do for removing pages or no update
        */
        if (cpu_buffer->nr_pages_to_update <= 0)
            continue;
        /*
        * to add pages, make sure all new pages can be
        * allocated without receiving ENOMEM
        */
        /**continue in ring_buffer.c 1677 - 1807**/
```

تغییر سایز ring buffer در دو حالت فعال بودن یا نبودن آن امکان دارد ولی در حالت فعال امکان ایجاد اختلال را دارد و بهتر است در حالت غیر فعال این تغییر صورت گیرد.حتی در کد هم توسط قسمت

```
if (atomic_read(&buffer->record_disabled)) {
        atomic_inc(&buffer->record_disabled);
```

از غیرفعال بودن ring_buffer اطمینان حاصل شده است برای خواندن و چک کردن page های ring_buffer.

البته در این حد جزئیات مربوط به ساختار ring buffer برای اضافه کردن داده های پاور به این ابزار نیازی نیست چون توسط فانکشن هایی که در فایل ring_buffer.c قرار دارند تا حدخوبی از ارتباط مستقیم با این جزئیات و struct ها میتوان پرهیز کرد.

برای مثال از توابعی که مورد استفاده ما هست تابع ring_buffer_lock_reserve است که در تابع trace_buffer_lock_reserve در فایل trace.c استفاده شده است و ما برای ارتباط با ring buffer از فانکشن موجود در trace.c استفاده میکنیم در واقع ارتباط غیر مستقیم و توسط فانکشن ها با ring buffer ایجاد میشود.

- **دسترسی به ring buffer:**

Global_trace یک descriptor است که tracing buffer ها را نگه می دارد:

```
static struct trace_array global_trace;
```

ساختار این struct شامل یک فیلد struct trace_buffer است که این struct هم خود شامل ring_buffer و trace_array_cpu است که با امکان دسترسی به global_trace امکان دسترسی به فیلد های داخلی آن وجود دارد. Lock این struct از طریق تابع trace_buffer_lock_reserve در دسترس است.

با توجه به کد های موجود در کرنل، همچنین ما اجازه تصمیم برای overwrite کردن یا نکردن را هم داریم با توجه به کد موجود در فایل ring_buffer.c خط ۱۸۱۰

```
void ring_buffer_change_overwrite(struct ring_buffer *buffer, int val)
{
    mutex_lock(&buffer->mutex);
    if (val)
        buffer->flags |= RB_FL_OVERWRITE;
    else
        buffer->flags &= ~RB_FL_OVERWRITE;
    mutex_unlock(&buffer->mutex);
}
```

نمونه هایی از استفاده این تابع در فایل trace.c و توسط تابع set_tracer_flag در خط ۳۵۰۲

```
int set_tracer_flag(struct trace_array *tr, unsigned int mask, int enabled)
{
    /* do nothing if flag is already set */
    if (!!(trace_flags & mask) == !!enabled)
        return 0;

    /* Give the tracer a chance to approve the change */
    if (tr->current_trace->flag_changed)
        if (tr->current_trace->flag_changed(tr->current_trace, mask, !!enabled))
            return -EINVAL;

    if (enabled)
        trace_flags |= mask;
    else
        trace_flags &= ~mask;

    if (mask == TRACE_ITER_RECORD_CMD)
        trace_event_enable_cmd_record(enabled);
```

```
    if (mask == TRACE_ITER_OVERWRITE) {
        ring_buffer_change_overwrite(tr->trace_buffer.buffer, enabled);
#ifdef CONFIG_TRACER_MAX_TRACE
        ring_buffer_change_overwrite(tr->max_buffer.buffer, enabled);
#endif
    }

    if (mask == TRACE_ITER_PRINTK)
        trace_printk_start_stop_comm(enabled);

    return 0;
}
```

در واقع overwrite بودن یا نبودن یک بافر را با استفاده از آیتم flags موجود در struct آن میتوان میفهمید و یا تغییر داد.

- نحوه تشکیل فایل سیستم ها و ارتباط آن ها با کد:

همانطور که اشاره کردیم، فایل سیستم های متعددی در اندروید وجود دارند ولی به دلایل مختلفی تعداد کثیری از آن ها قابل استفاده نیستند و در ظاهر وجود ندارند ولی در واقع، پس از فعال شدن config های لازم برای هر file system، توسط تابع ساخته میشوند. (خط ۵۸۸۸ فایل trace.c)

```
struct dentry *trace_create_file(const char *name,
                 umode_t mode,
                 struct dentry *parent,
                 void *data,
                 const struct file_operations *fops)
{
    struct dentry *ret;

    ret = debugfs_create_file(name, mode, parent, data, fops);
    if (!ret)
        pr_warning("Could not create debugfs '%s' entry\n", name);

    return ret;
}
```

در واقع این تابع وظیفه ساخت فایل ها را داراست و بطور مثال پس از فعال نمودن function، این تابع توسط تابع دیگری (تکه کد زیر) که در فایل ftrace.c خط ۴۶۵۳ قرار دارد فراخوانی میشود و فایل سیستم set_ftrace_pid که مخصوص این نوع tracer است، ساخته می شود.

```
static __init int ftrace_init_debugfs(void)
{
    struct dentry *d_tracer;

    d_tracer = tracing_init_dentry();
    if (!d_tracer)
        return 0;

    ftrace_init_dyn_debugfs(d_tracer);

    trace_create_file("set_ftrace_pid", 0644, d_tracer,
            NULL, &ftrace_pid_fops);

    ftrace_profile_debugfs(d_tracer);

    return 0;
}
```

بطور کلی تر فایل سیستم های general برای انواع tracer ها در فایل trace.c و توسط همان تابع trace_create_file ساخته میشوند. این توابع اعم از:
فایل trace.c خط ۶۳۲۷

```
static void
init_tracer_debugfs(struct trace_array *tr, struct dentry *d_tracer)
{
    int cpu;

    trace_create_file("trace_options", 0644, d_tracer,
            tr, &tracing_iter_fops);

    trace_create_file("trace", 0644, d_tracer,
            tr, &tracing_fops);

    trace_create_file("trace_pipe", 0444, d_tracer,
            tr, &tracing_pipe_fops);

    trace_create_file("buffer_size_kb", 0644, d_tracer,
```

```
            tr, &tracing_entries_fops);

    trace_create_file("buffer_total_size_kb", 0444, d_tracer,
            tr, &tracing_total_entries_fops);

    trace_create_file("free_buffer", 0644, d_tracer,
            tr, &tracing_free_buffer_fops);

    trace_create_file("trace_marker", 0220, d_tracer,
            tr, &tracing_mark_fops);

    trace_create_file("saved_tgids", 0444, d_tracer,
            tr, &tracing_saved_tgids_fops);

    trace_create_file("trace_clock", 0644, d_tracer, tr,
            &trace_clock_fops);

    trace_create_file("tracing_on", 0644, d_tracer,
            tr, &rb_simple_fops);

#ifdef CONFIG_TRACER_SNAPSHOT
    trace_create_file("snapshot", 0644, d_tracer,
            tr, &snapshot_fops);
#endif

    for_each_tracing_cpu(cpu)
        tracing_init_debugfs_percpu(tr, cpu);

}
```

و همچنین تابع در همان فایل خط ۶۳۷۰

```
static __init int tracer_init_debugfs(void)
{
    struct dentry *d_tracer;

    trace_access_lock_init();

    d_tracer = tracing_init_dentry();
    if (!d_tracer)
        return 0;

    init_tracer_debugfs(&global_trace, d_tracer);
```

```
    trace_create_file("tracing_cpumask", 0644, d_tracer,
            &global_trace, &tracing_cpumask_fops);

    trace_create_file("available_tracers", 0444, d_tracer,
            &global_trace, &show_traces_fops);

    trace_create_file("current_tracer", 0644, d_tracer,
            &global_trace, &set_tracer_fops);

#ifdef CONFIG_TRACER_MAX_TRACE
    trace_create_file("tracing_max_latency", 0644, d_tracer,
            &tracing_max_latency, &tracing_max_lat_fops);
#endif

    trace_create_file("tracing_thresh", 0644, d_tracer,
            &tracing_thresh, &tracing_max_lat_fops);

    trace_create_file("README", 0444, d_tracer,
            NULL, &tracing_readme_fops);

    trace_create_file("saved_cmdlines", 0444, d_tracer,
            NULL, &tracing_saved_cmdlines_fops);

#ifdef CONFIG_DYNAMIC_FTRACE
    trace_create_file("dyn_ftrace_total_info", 0444, d_tracer,
            &ftrace_update_tot_cnt, &tracing_dyn_info_fops);
#endif

    create_trace_instances(d_tracer);

    create_trace_options_dir(&global_trace);

    return 0;
}
```

همچنین توابعی نیز وجود دارند که برای یک cpu میتوانند فایل ها را بسازند. این توابع همچنین برای ساختن فایل هایی که اطلاعات درون آن ها per cpu می باشد نیز کاربرد دارند.

فایل trace.c خط ۵۷۲۱

```
static struct dentry *
trace_create_cpu_file(const char *name, umode_t mode, struct dentry *parent,
                      void *data, long cpu, const struct file_operations *fops)
{
    struct dentry *ret = trace_create_file(name, mode, parent, data, fops);

    if (ret) /* See tracing_get_cpu() */
        ret->d_inode->i_cdev = (void *)(cpu + 1);
    return ret;
}
```

تابع زیر برای ساختن فایل هایی که اطلاعات آن ها برای هر cpu جداست کاربرد دارد:

همان فایل خط ۵۷۳۱

```
static void
tracing_init_debugfs_percpu(struct trace_array *tr, long cpu)
{
    struct dentry *d_percpu = tracing_dentry_percpu(tr, cpu);
    struct dentry *d_cpu;
    char cpu_dir[30]; /* 30 characters should be more than enough */

    if (!d_percpu)
        return;

    snprintf(cpu_dir, 30, "cpu%ld", cpu);
    d_cpu = debugfs_create_dir(cpu_dir, d_percpu);
    if (!d_cpu) {
        pr_warning("Could not create debugfs '%s' entry\n", cpu_dir);
        return;
    }

    /* per cpu trace_pipe */
    trace_create_cpu_file("trace_pipe", 0444, d_cpu,
                tr, cpu, &tracing_pipe_fops);

    /* per cpu trace */
    trace_create_cpu_file("trace", 0644, d_cpu,
                tr, cpu, &tracing_fops);

    trace_create_cpu_file("trace_pipe_raw", 0444, d_cpu,
                tr, cpu, &tracing_buffers_fops);
```

```
    trace_create_cpu_file("stats", 0444, d_cpu,
                tr, cpu, &tracing_stats_fops);

    trace_create_cpu_file("buffer_size_kb", 0444, d_cpu,
                tr, cpu, &tracing_entries_fops);

#ifdef CONFIG_TRACER_SNAPSHOT
    trace_create_cpu_file("snapshot", 0644, d_cpu,
                tr, cpu, &snapshot_fops);

    trace_create_cpu_file("snapshot_raw", 0444, d_cpu,
                tr, cpu, &snapshot_raw_fops);
#endif
}
```

همچنین option های ftrace توسط تابع trace_set_options تنظیم میشوند که در فایل trace.c خط ۳۵۳۴ قرار دارد.

```
static int trace_set_options(struct trace_array *tr, char *option)
{
    char *cmp;
    int neg = 0;
    int ret = -ENODEV;
    int i;

    cmp = strstrip(option);

    if (strncmp(cmp, "no", 2) == 0) {
        neg = 1;
        cmp += 2;
    }

    mutex_lock(&trace_types_lock);

    for (i = 0; trace_options[i]; i++) {
        if (strcmp(cmp, trace_options[i]) == 0) {
            ret = set_tracer_flag(tr, 1 << i, !neg);
            break;
        }
    }
```

```
    /* If no option could be set, test the specific tracer options */
    if (!trace_options[i])
        ret = set_tracer_option(tr->current_trace, cmp, neg);

    mutex_unlock(&trace_types_lock);

    return ret;
}
```

که در واقع تابع بالا از طریق تابعی که در اثر نوشته شدن در آن فعال میشود صدا زده میشود یعنی تابع tracing_trace_options_write در فایل trace.c خط ۳۵۶۷

```
static ssize_t
tracing_trace_options_write(struct file *filp, const char __user *ubuf,
            size_t cnt, loff_t *ppos)
{
    struct seq_file *m = filp->private_data;
    struct trace_array *tr = m->private;
    char buf[64];
    int ret;

    if (cnt >= sizeof(buf))
        return -EINVAL;

    if (copy_from_user(&buf, ubuf, cnt))
        return -EFAULT;

    buf[cnt] = 0;

    ret = trace_set_options(tr, buf);
    if (ret < 0)
        return ret;

    *ppos += cnt;

    return cnt;
}
```

در واقع option ها توسط یک آرایه ای از char* ها ساخته شده اند که در همان فایل و خط ۸۱۲ قرار دارد.

```
static const char *trace_options[] = {
    "print-parent",
    "sym-offset",
    "sym-addr",
    "verbose",
    "raw",
    "hex",
    "bin",
    "block",
    "stacktrace",
    "trace_printk",
    "ftrace_preempt",
    "branch",
    "annotate",
    "userstacktrace",
    "sym-userobj",
    "printk-msg-only",
    "context-info",
    "latency-format",
    "sleep-time",
    "graph-time",
    "record-cmd",
    "overwrite",
    "disable_on_free",
    "irq-info",
    "markers",
    "function-trace",
    "print-tgid",
    NULL
};
```

در واقع تابعی که به عنوان نمونه استفاده از تغییر overwrite بودن یا نبودن ring_buffer در قسمت های قبل بود مربوط به overwrite در لیست بالا است.

- **جزئیات عملکرد** tracefs file system:

به طور کلی device driver ها در یونیکس به دو دسته کلی block و character دسته بندی می شوند. Devise هایی که از نوع کاراکتر هستند توسط struct dev در کرنل رجیستر می شوند .بیشتر عملیات های درایورها از سه struct به نام های file_operation و file و inode جهت پیاده سازی استفاده می کنند می کنند.

همان طور که گفته شد برای کنترل و اجرای ftrace از tracefs file systems استفاده میشود که به هر کدام از این فایل ها در کرنل یک struct file_operation نسبت داده می شود. ساختار این struct به طور کلی این گونه است:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t
*);
        [...]
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
long);
        [...]
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        [...]
```

که زمانی که کاربر system call هایی مانند write و read را بر هر فایل اجرا می کند توابعی که این درابتدا برای مقداردهی این function pointer ها استفاده شده اند صدا زده می شوند . پیاده سازی این توابع که ftrace را کنترل می کنند عمدتا در مسیر /kernel/tracing/tracer.c قرار دارند.

برای مثال tracing_on توسط instance ای از این struct به نام rb_simple_fops کنترل می شود که function pointer write توسط rb_simple_write پیاده سازی شده است که در آن توسط تابع kstrtoul_from_user مقداری که یوزر در این فایل نوشته است خوانده می شود و اگر ۱ باشد توابع مربوط فعال سازی تریسر ها صدا زده می شوند.

- **نحوه نوشتن در ring_buffer با استفاده از** :ring_buffer_lock_reserve

با توجه به فایل سیستم tracer_marker و ماهیت آن و با توجه به تابع tracing_mark_write در فایل trace.c این روند بدین ترتیب است که برخلاف نوشتن توسط فانکشن ها که نیازمند لاک گرفتن است این روند حالت رزرو کردن قسمتی از بافر را دارد.

Trace_buffer_lock_reserve توسط فانکشن trcing_mark_write صدا زده میشود که خود این تابع ring_buffer_lock_reserve را فراخوانی میکند که این تابع در فایل ring_buffer.c قرار دارد و در واقع بخشی از ring buffer را رزرو میکند و یک اشاره گر به event برای اینکه مستقیما در ring buffer بتوان نوشت بازمیگرداند.

در ادامه داده های event را توسط تابع ring_buffer_event_data خارج میکند این تابع داده های event مورد نظر را که به عنوان ورودی گرفته است را خروجی میدهد.

سپس با داده هایی که میخواهد در ring buffer ذخیره کند را با memcpy در یک struct از جنس print_entry ذخیره میکند که entry در واقع منظور داده های ورودی است. در نهایت توسط تابع __buffer_unlock_commit داده ها در ring buffer ذخیره میشوند.

روند تابع __buffer_unlock_commit بدین صورت است که در ابتدا باید با استفاده از تابع this_cpu_write درواقع به نوعی flag write برای cpu را set کند.

در ادامه ring_buffer_unlock_commit داده هایی را که میخواهیم در ring buffer داشته باشیم را commit میکند و قسمت رزرو شده را برای نوشتن بقیه فانکشن ها آزاد میکند.

روش های دیگر نوشتن در ring buffer استفاده از lock به طور مستقیم است که نمونه آن در تابع tracing_cpumask_write و یا tracing_resize_ring_buffer که در فایل trace.c قرار دارند، موجود است.

با بررسی روش های متعدد نوشتن در ring buffer استفاده از روش reserve ترجیح دارد به دلیل اینکه با توجه به اینکه حجم داده ای که میخواهیم در ring buffer وارد کنیم کم و فرکانس آپدیت این داده بسیار کمتر از آپدیت شدن ring buffer است. در نتیجه می توان روند trace_marker را تقلید کرد.

برای استفاده از ftrace لازم است تا فایل سیستم tracing_on ۱ شود پس در کد، فانکشن tracer_tracing_on فراخوانی میشود(توسط tracing_on).پس میتوان در اینجا فانکشنی که قرار است دیتای پاور را اضافه کند قرار داد تا از شروع عملیات tracing ما دیتای پاور را هم نمونه برداری کنیم.

- **بررسی ساختار درایور های باتری و دسترسی به آن ها در کرنل:**

با بررسی def_cofing لیست درایورهای باتری که توسط nexus 6 استفاده می شوند عبارتند از:
battery_max17042 , android_battery, bq28400, battery_current_limit
که از این بین 17042max درایوری است که مستقیما دیتا را از ic مربوطه می خواند.
کرنل جهت مدیریت درایورهای متعدد از توابع موجود در فایل های power_supply_sysfs.c و power_supply_core.c استفاده می کند.
هر یک از attribute های مختلف مثل current یا voltage از طریق device_attribute که در ساختار sysfs اکسپورت می شود.

```
#define POWER_SUPPLY_ATTR(_name)
{
    .attr = { .name = #_name },
    .show = power_supply_show_property,
    .store = power_supply_store_property,
}

static struct device_attribute power_supply_attrs[] = {
    /* Properties of type `int' */
    POWER_SUPPLY_ATTR(status),
    POWER_SUPPLY_ATTR(charge_type),
    POWER_SUPPLY_ATTR(health),
    POWER_SUPPLY_ATTR(present),
    POWER_SUPPLY_ATTR(online),
    POWER_SUPPLY_ATTR(authentic),
.
.
}
```

زمانی که سیستم کال خواندن بر نود های مربوط صدا می شوند از طریق تابع power_supply_show_property مقدار attribute از درایور خوانده می شود و در فایل سیستم نوشته می شود.

ساختاری که در درایور با باتری ارتباط برقرار می کند:

```
struct power_supply {
    const char *name;
    enum power_supply_type type;
    enum power_supply_property *properties;
    size_t num_properties;

    char **supplied_to;
    size_t num_supplicants;

    char **supplied_from;
    size_t num_supplies;
#ifdef CONFIG_OF
    struct device_node *of_node;
```

```
#endif

    int (*get_property)(struct power_supply *psy,
            enum power_supply_property psp,
            union power_supply_propval *val);
    int (*set_property)(struct power_supply *psy,
            enum power_supply_property psp,
            const union power_supply_propval *val);
    int (*property_is_writeable)(struct power_supply *psy,
                    enum power_supply_property psp);
    void (*external_power_changed)(struct power_supply *psy);
    void (*set_charged)(struct power_supply *psy);
…
…
…
```

این struct اینستنس از آن در همه درایور های قرار دارد برای مثال در 17042max :

```
struct max17042_chip {
    struct i2c_client *client;
    struct power_supply battery;
    enum max170xx_chip_type chip_type;
    struct max17042_platform_data *pdata;
    struct work_struct work;
    struct work_struct check_temp_work;
    struct mutex check_temp_lock;
    int    init_complete;
    bool batt_undervoltage;
#ifdef CONFIG_BATTERY_MAX17042_DEBUGFS
    struct dentry *debugfs_root;
    u8 debugfs_addr;
#endif
    struct power_supply *batt_psy;
    int temp_state;
    int hotspot_temp;
    struct delayed_work iterm_work;
    struct max17042_wakeup_source max17042_wake_source;
    int charge_full_des;
    int taper_reached;
    bool factory_mode;
    int last_fullcap;
};
```

شیوه دسترسی به این استراکت در trace.c از طریق مراجعه به kernel tree است.

- **بررسی دقیق ابزار** systrace:

systrace در فایلی به نام systrace.py قرار دارد که به طور کلی این فایل حاوی کار های سطح بالاتر لازم برای نمایش نتایج است و قسمت اصلی کد های آن از جایی که sys.exit(run_systrace.main())، در فایل systrace.py فراخوانی میشود آغاز میگردد.

برای همین منظور به فایل run_systrace.py میرویم که در آن تابع main() ، main_impl(sys.argv) را فراخوانی میکند.
قبل از بررسی این تابع لازم به ذکر است که چندین نوع agent از فایل های مربوطه و لازمه import شده اند که عبارتند از :

```
ALL_MODULES = [atrace_agent, atrace_from_file_agent,
atrace_process_dump,battor_trace_agent, ftrace_agent,
walt_agent]
```

در main_impl توسط تابع parse_options اطلاعات وارد شده در commandline جزءبه جزء parse شده و options , categories را میسازند که دارای فیلد هایی متفاوت و حاوی اطلاعاتی هستند که در ادامه کار در بخش های مختلف به هرکدام نیاز است.
Category ها مواردی هستند که ما به systrace اعلام میداریم که نیاز به trace شدن دارند برای مثال sched,freqو ...
با توجه به تعریف پروژه نیاز خواهد بود category جدیدی اضافه شده و در این قسمت parse و شناسایی شود.
در ادامه بعد از set شدن options و categories شاهد چنین قطعه کدی هستیم :

```
if categories:
   if options.target == 'android':
     options.atrace_categories = categories
   elif options.target == 'linux':
     options.ftrace_categories = categories
```

در این قسمت برای target لینوکس ftrace و اندروید atrace کاتگوری هایشان ست میشود. که البته این target به صورت دیفالت android است مگر اینکه توسط کاربر در commandline ذکر شود که در این

صورت parser ایجاد شده توسط تابع parse_options به عنوان ورودی به util.get_main_options داده میشود که در فایل util.py با –target این مقدار ست شده یا در حالت دیفالت خود باقی میماند. با توجه به توضیحات فوق سیستم خود قادر به تشخیص و ست کردن این مقدار نیست و برای حالتی جز حالت دیفالت باید دستی ست شود. علاوه بر آن میتواند نشانگر این هم باشد که میتوان بر روی محیط linux نیز از امکانات systrace استفاده نمود.

سپس به initialize_devil میرسیم که ابتدا دستگاه adb را شناسایی کرده و path آنرا در اختیار قرار میدهد به این صورت که تابع find_adb() فراخوانی شده و

```
paths = os.environ['PATH'].split(os.pathsep)
```

و برای این path ها :

```
for p in paths:
  f = os.path.join(p, executable)
  if os.path.isfile(f):
      return f
return None
```

که باز در ادامه این f در همان تابع initialize_devil() در local_paths استفاده میگردد.
آخرین کار در این تابع این است که devil_env.config با این configuration ها initialize() شود
برای دانستن چگونگی آن باید به فایل devil_env.py در devil/devil نگاهی بیندازیم. در این تابع تمام location هایی که وابستگی های devil وجود دارد را با همه configuration هایی که قبلا ساخته شده و به آن پاس داده ایم را مشخص مینماید
در کامنت ها ذکر شده که فقط فرمت توضیح داده شده توسط
py_utils.dependency_manager.BaseConfig را میپذیرد.
(چنین عبارتی در systrace\catapult\dependency_manager\dependency_manager در فایل dependancy_manager_unittest.py پیدا شد)
لذا اگر نیاز باشد برای محیط devil وابستگی تعریف کنیم که از آن استفاده کند باید به فرمت گفته شده باشد.
برای ایجاد تغییرات نیاز به بررسی دقیق تر devil_env.py است.

در ادامه یک controller ایجاد میشود که به استناد از یکی از کامنت های داخل کد خود یک agent به حساب می آید.

```
controller =
systrace_runner.SystraceRunner(os.path.dirname(os.path.abspath(__file__)),
options)
```

در اینجا به فایل systrace_runner.py میرویم که تابع فوق پیاده سازی شده است.
در این فایل کلاسی به همین اسم وجود دارد که توسط تابع زیر agent های مورد نیاز ما برحسب options ساخته میشود.

```
agents_with_config = tracing_controller.CreateAgentsWithConfig(options,
AGENT_MODULES)
```

تابع استفاده شده در کد بالا در فایل tracing_controller.py قرار دارد که به ازای تمامی modules , توسط get_config(options) , config را مقدار دهی میکند سپس به وسیله آن در module.try_create_agent(config) , agent میسازد و در نهایت تمامیشان را به فرم کلاسی به نام AgentsWithConfig که یک فیلد agant و یک فیلد config دارد در لیستی به نام result نگه میدارد.

جز مواردی که ذکر شد نیاز است که controller ای هم وجود داشته باشد که خود یک agent است.

```
controller_config = tracing_controller.GetControllerConfig(options)
```

که به صورت زیر در tracing_controller.py پیاده سازی شده است و با داشتن options و پارامتر های مورد نیازش, یک instance از آنرا برمیگرداند.

```
return TracingControllerConfig(options.output_file, options.trace_time,
                               options.write_json,
                               options.link_assets, options.asset_dir,
                               options.timeout, options.collection_timeout,
                               options.device_serial_number, options.target)
```

درنهایت با داشتن موارد فوق کنترلر setup میشود.

```
self._tracing_controller =
tracing_controller.TracingController(agents_with_config, controller_config)
```

با داشتن controller , تابع StartTracing() را برای controller مان در run_systrace.py فراخوانی میکنیم.
این تابع در tracing_controller.py تعریف شده است که در آن هم برای agent هایی که قبل تر در agents_with_config ساخته شده بودند و هم برای controller_config, تابع StartAgentTracing(...) صدازده میشود.
اما ابتدا تابع باید برای controller_agant صدا زده شود و در صورت موفق بودن باقی به عنوان child_agents_with_config با فراخوانی همان تابع شروع به کار مینمایند.

به همان ترتیب با رسیدن به stopTracing() در فایل run_systrace.py برای controller, این تابع در tracing_controller.py صدا زده میشود.
در آن ابتدا StopAgentTracing() باید برای child_agent ها فراخوانی شود و بعد برای controller_agent و در قسمت جمع کردن اطلاعات از agent های خاتمه یافته , GetResault(…) فراخوانی شده و تمامی نتایج در لیستی به نام all_results نگهداری میشود.
تابع GetResult() در فایل tracing_controller.py به این صورت عمل مبکند که _log_path را که در مرحله شروع یعنی در تابع StartAgentTracing() ست شده است,

```
self._log_path = controller_log_file.name
```

را با مد read باز میکند و Data از آن میخواند

```
data = ast.literal_eval(outfile.read() + ']')
```

و در نهایت تابع TraceResult(..) در فرم کلاسی که در trace_result.py است منبع این اطلاعات یعنی اینکه مربوط به چه نوع tracer ای بوده و داده خام را نگه میدارد.

Source_name
Raw_data

در tracing_controller.py از حساسیت های تایمر های زیر

```
@py_utils.Timeout(tracing_agents.START_STOP_TIMEOUT)
@py_utils.Timeout(tracing_agents.GET_RESULTS_TIMEOUT
```

برای توابع استفاده شده که به نظر میرسید نیاز به دانستن مقادیر این تایمر ها داشته باشیم که در فایل
systrace\catapult\systrace\systrace\tracing_agents__init__.py
مقدار دهی شده اند.

```
START_STOP_TIMEOUT = 10.0
GET_RESULTS_TIMEOUT = 30.0
```

در نهایت امر با

```
controller.OutputSystraceResults(write_json=options.write_json)
```

که در پیاده سازی اش در systrace_runner.py دوتابع , یکی برای JSON ودیگری برای HTML خروجی مطلوب را تولید میکنند.

این دو تابع در output_generator.py تعریف شده اند و در تابع مربوط به خروجی JSON , بعد از تبدیل به dic فایل output_file_name را که قبلا در زمان ساخته شدن systraceRunner به

```
self._out_filename = options.output_file
```

ست شده بود با مد write باز میکند و json.dump(results, json_file) جرا میشود. در نهایت هم از این تابع یک path برگردانده میشود که به عنوان محل داده ها در OutputSystraceResults() چاپ میشود

برای فایل HTML در systrace_runner.py همانند حالت JSON روی output_generator این بار تابع GenerateHTMLOutput(…) فراخوانی میشود و در تعریف این تابع در فایل output_generator.py تابع دیگری به نام _ReadAssert() به صورت بازگشتی فراخوانی میشود و در آن تابع update برای update_systrace_trace_viewer که در فایلی به همین نام قرار دارد فراخوانی میشود و این تابع بر اساس حالت هایی که برای update کردن دارد شروع به انجام اقدامات لازم مینماید

بعد از چک کردن git و فایل تابعی به نام vulcanize_trace_viewer.WriteTraceViewer روی vulcanize_trace_viewer که از import , tracing_build شده است فراخوانی میشود ولی چنین فایلی یافت نشد و نزدیک ترین نتایج فایل
systrace\catapult\common\py_vulcanize\py_vulcanize
است که حاوی چندین فایل است که مربوط به مراحل ساخت خروجی HTML است.

در ادامه فایل های مربوط به هر کدام از این agent ها را بررسی میکنیم تا با ماهیت و کارکرد شان اشنا شویم

ftrace_agent.py بعد از مشخص کردن path های بدست آوردن اطلاعات کاتگوری های متفاوت, از مهم ترین توابع میتوان به StartAgentTracing() و StopAgentTracing() اشاره کرد که با فراخوانی StartTracing() و به طبع آن StartAgentTracing() برای این agent به صورتی که در این فایل پیاده سازی شده اجرا میشود.

Battor_trace_agent.py طبق کامنت های داخل کد , battor یک مانیتور کننده ی فرکانس بالای پاور است که برای تست باتری استفاده میشود و اطلاعاتی را به فرمت زیر در اختیار قرار میدهد:

```
time  current  voltage  <sync_id>

time = time since start of trace (ms)
current = current through battery (mA) - this can be negative if the battery is
charging
voltage = voltage of battery (mV)
```

درمورد <sync_id> :

در فایل tracing_controller.py تابع _IssueClockSyncMarker() در StopTracing() صدازده میشودکه در آن برای هر agent ای که StopTracing() شده باشد(یعنی در _child_agents برود) چک میکند که SupportExplicitClockSync فعال است یا نه . اگر فعال باشد sync_id را با GetUniqSyncID() پر میکند و RecordClockSyncMarker فراخوانی میشود که در فایل battor_trace_agent.py همان تابع برای _battor_wrapper صدا خواهد شد پس به این ترتیب sync_id برای اولین بار از controller میاید و اگر ست شود بعد از مراحل بالا فیلد فوق را خواهیم داشت.

atrace_agent.py همانند agent های دیگر به وجود امدن آن با کانفیگر های مد نظر و حساس به کاتگوری های ذکر شده و وجود توابع Start/Stop_agent_Tracing() و GetResult() و همچنین RecordClockSyncMarker

زمانی که برای StartAgentTracing , battor_agent() فراخوانی میشود یک wrapper ایجاد میشود برای اینکه کد پایتون با battor device ارتباط برقرار کند. و برای این wapper ابتدا تابع StartShell() و سپس StartTracing() فراخوانی میشود.

در فایل battor_wrapper.py تابع اول battor binary shell را آغاز میکند و در StartTracing اتفاقات مربوط به shell پیاده سازی شده است.

قبل از کد هایی که بالا ذکر شد, در battor_trace_agent , وجود دارد که زمانی به پیاده سازی آن در battery_utils نگاه میکنیم در آن اطلاعان پروفایل مدل های متفاوت گوشی ها ذکر شده که profile 6NEXUS به صورت زیر است

```
{
'name': 'Nexus 6',
'witness_file': None,
'enable_command': (
'echo 1 > /sys/class/power_supply/battery/charging_enabled && '
 'dumpsys battery reset'),
'disable_command': (
'echo 0 > /sys/class/power_supply/battery/charging_enabled && '
'dumpsys battery set ac 0 && dumpsys battery set usb 0'),
 'charge_counter': (
 '/sys/class/power_supply/max170xx_battery/charge_counter_ext'),
 'voltage': '/sys/class/power_supply/max170xx_battery/voltage_now',
'current': '/sys/class/power_supply/max170xx_battery/current_now',
},
```

- مرحله build کردن kernel و AOSP :

Build kernel

نسخه کرنل مورد استفاده: android-msm-shamu-3.10-nougat-mr1.7

لیست دستورات استفاده شده جهت build کردن کرنل:

```
export CROSS_COMPILE=$(path_to_aosp)/aosp/prebuilts/gcc/linux-x86/arm/arm-eabi-4.8/bin/arm-eabi-
export ARCH=arm && export SUBARCH=arm
make clean
make defconfig_shamu
make -j$(nproc --all)
```

که در نهایت کامپایل با موفقیت انجام شد.

Build AOSP

ابتدا نیازمند که enviroment را setup کنیم و ابزارهای لازم را نصب کنیم (تمام ابزارهای نصب شده تاکنون):

```
sudo apt-get install git ccache lzop bison gperf build-essential zip curl zlib1g-dev g++-multilib python-networkx libxml2-utils bzip2 libbz2-dev libghc-bzlib-dev squashfs-tools pngcrush liblz4-tool optipng libc6-dev-i386 gcc-multilib libssl-dev gnupg flex lib32ncurses5-dev x11proto-core-dev libx11-dev lib32z1-dev libgl1-mesa-dev xsltproc unzip python-pip python-dev libffi-dev libxml2-dev libxslt1-dev libjpeg8-dev openjdk-8-jdk libc++-dev
```

با توجه به این که درایور ها و نسخه های کرنل موجود برای Nexus 6 تا ورژن ۷,۱,۱ اندروید ساپورت می شدند لازم بود که branch را تغییر دهیم:

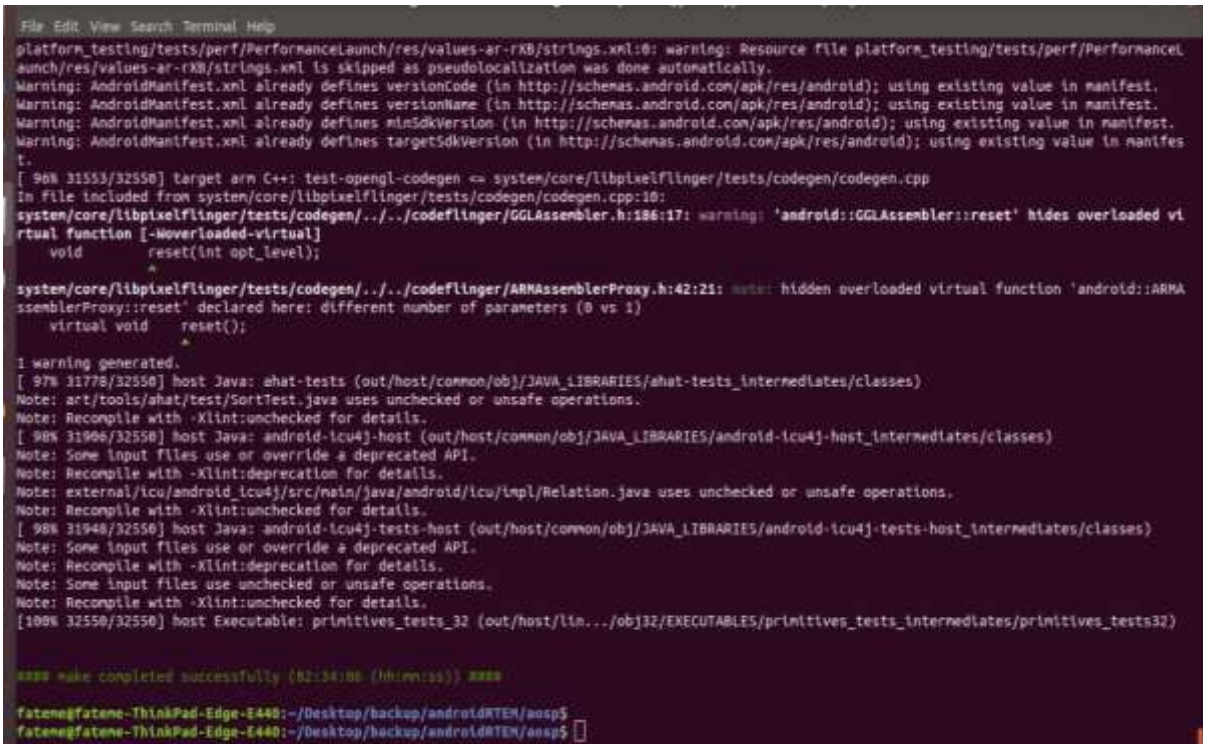
```
repo init -u https://android.googlesource.com/platform/manifest -b android-7.1.1_r55
```

که در نهایت ۲۰ گیگ فایل دانلود شد.

سپس نصب درایور های مربوطه

```
wget https://dl.google.com/dl/android/aosp/moto-shamu-n6f26r-d48980a4.tgz
tar xvzf moto-shamu-n6f26r-d48980a4.tgz
./extract-moto-shamu.sh
```

```
. build/envsetup.sh
export LC_ALL=C
export JACK_SERVER_VM_ARGUMENTS="-Dfile.encoding=UTF-8 -
XX:+TieredCompilation -Xmx4096m"
prebuilts/sdk/tools/jack-admin kill-server
prebuilts/sdk/tools/jack-admin start-server
lunch aosp_shamu-userdebug
make -j$(nproc --all)
```



پس از روت کردن و ریختن فایل های **img**، فایل **tracing_on** با دستور **echo** فعال شد.

اگر مشکلی از قبیل کرش کردن کرنل به وجود امد باز به بخش تجربه های ناموفق مراجعه شود. با استفاده از **set_ftrace_pid** و تریس کردن پراسسی برای مثال **camera** تست شد.

مرحله اضافه شدن داده های باتری جمع آوری شده از سطح کرنل به ابزار systrace :

پس از افزودن داده های power که شامل جریان، ولتاژ و انرژی هستند به ابزار سطح کرنل ftrace و همچنین atrace، این داده ها باید با parse شدن و جدا سازی از بقیه اطلاعات به systrace منتقل شوند.
برای این تغییر در systrace، پس از بررسی کد های systrace که در مسیر sdk اندروید قرار دارند (Sdk\platform-tools\systrace\catapult\systrace\systrace) دریافتیم که در بین این فایل ها باید

prefix.html و systrace_trace_viewer.html تغییر داده شوند زیرا در این فایل ها داده ها از atrace و بقیه filesystem یا logها خوانده شده و parse میشود و توسط توابعی در این فایل ها به صورت یک html در می آیند (-که البته تنها با ابزار chrome قابل دسترسی است) و داده ها را به صورت دیاگرام نمایش میدهد.
در فایل prefix.html داده های power از بین بقیه داده ها جدا شده و در آرایه powerlogs نگهداری میشوند از طرفی داده های ولتاژ و جریان و انرژی هم که در فایل systrace_trace_viewer.html استفاده و مقدار دهی شده اند به عنوان متغیر global در این فایل تعریف شده اند.همچنین timestamp اولین داده atrace هم برای داشتن timestamp مبدا برای کشیدن نمودار مربوط به داده های power جدا شده است.علاوه براین timestamp، برای کشیدن نمودار ناچار به داشتن timestamp های ولتاژ و جریان و انرژی هم بودیم بنابراین به طور جداگانه متغیری global دیگری از جنس آرایه تعریف کردیم تا timestamp های مربوط به power را نگه داریم.
در فایل systrace_trace_viewer.html، از داده های powerlogs، مقدار های ولتاژ و جریان و انرژی را جدا کردیم و آنها را در متغیری که قبلا در فایل prefix.html تعریف کرده بودیم نگه داشتیم. همچنین timestamp های داده های power را جدا کردیم و در powertimestamps که
همان متغیر global مربوط به timestamp باطری در فایل قبل بود، ذخیره کردیم.

10- ابزار سطح سیستم مورد استفاده برای :performance

.1.1 Strace:

معرفی:

در دو عنوان debugger و profiller میتواند مورد استفاده قرار گیرد. که در عنوان debugger برای نشان دادن سیستم کال های فراخوانی شده، آرگومانها، مقدار بازگشتی به کار میرود. در این عنوان علاوه بر اینکه میتوان فهمید یک برنامه Fail شده است علت آن را هم میتوان دریافت.

در عنوان profiller میتوان زمان اجرای هر سیستم کال را بطور خاص بیان کند.

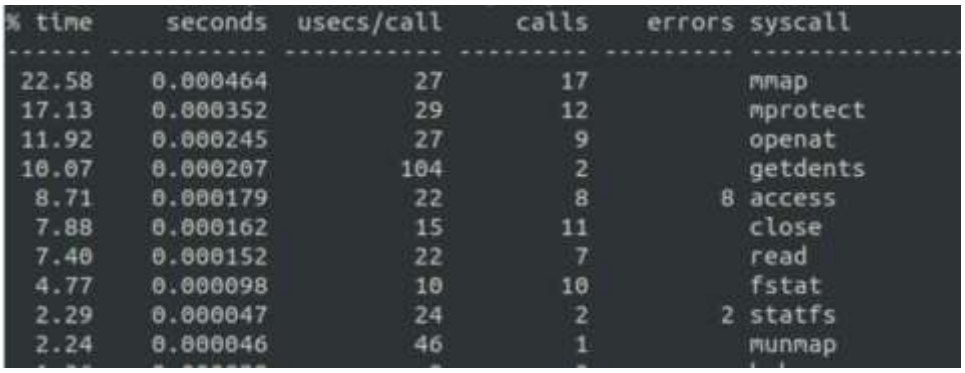
این ابزار مستقیما از ptrace بهره میگیرد به این صورت که هر گاه یک process سیستم کالی را فراخوانی کرد strace از آن مطلع می شود وبا stop کردن process به بررسی رجیسترها می پردازد تا اطلاعاتی که میخواهد از سیستم کال ها داشته باشد را استخراج کند سپس process را resume میکند و هنگامی که مقدار بازگشتی سیستم کال آماده شد با تکرار روند قبل آن را استخراج میکند همچنین زمان اتفاق افتادن سیستم call ها را هم print میکند.

- روش استفاده:

این ابزار از طریق adb و ترمینال لینوکس در دسترس است. با توجه به آپشن های موجود در این ابزار اطلاعات مختلفی را میتواند monitor کند از جمله:

  - تایم فراخوانی
  - زمان اجرا

- خطاهای سیستم کال در صورت وجود
- نام سیستم کال
- آرگومان ها
- مقدار بازگشتی
- Instruction point



% time	seconds	usecs/call	calls	errors	syscall
22.58	0.000464	27	17		mmap
17.13	0.000352	29	12		mprotect
11.92	0.000245	27	9		openat
10.07	0.000207	104	2		getdents
8.71	0.000179	22	8	8	access
7.88	0.000162	15	11		close
7.40	0.000152	22	7		read
4.77	0.000098	10	10		fstat
2.29	0.000047	24	2	2	statfs
2.24	0.000046	46	1		munmap

شکل بالا نمونه ای از اجرای strace میباشد.

البته این ابزار باید در کرنل اندروید فعال شود و به صورت مستقیم قابل استفاده نیست.

1.2. **Ltrace:**

خیلی شبیه به strace است با این تفاوت که این ابزار ،library calls را مانیتور میکند که strace ، به بررسی syscalls ها میپردازد.همچنین trace کردن library function call ها رویکردی متفاوت از system call ها دارد.

استفاده از این دو ابزار بسته به نوع برنامه است که library heavy باشد یا syscall heavy.

شرح استفاده ltrace از ptrace به صورت زیر میباشد:

- به یک برنامه باید attach شود توسط ptrace
- یافتن PLT در برنامه

PLT مخفف عبارت Procedure Linkage Table است که یک مجموعه assembly instruction است در library function که هنگامی که library function فراخوانی می شود اجرا میشود که به آن trampolines هم میگویند.

- استفاده از PTRACE_POKETEXT برای overwrite کردن کدهای اسمبلی در plt برنامه
- و در نهایت resume کردن اجرا برنامه

با استفاده از option های strace , ltrace میتوان خروجی های Strace را در ltrace هم داشت بنابراین در خروجی این دو ابزار تفاوت چندانی نیست.

البته این ابزار هم مانند strace دسترس نیست ولی می توان این را هم مشابه آن فعال کرد.به دلیل موجود راهنمایی های بیشتری و مرسوم تر بودن استفاده از strace و همچنین داشتن داده های مشابه، در این پروژه استفاده از strace به این ابزار ارجحیت یافته است.

.1.3 Ptrace:

این ابزار که پایه بسیاری از ابزار های دیگر همانند sysdig , ltrace , strace , .. است توانایی انجام task 3 عمده را دارد:

- Trace system call
- Read and write in memory and register
- manipulate signal delivery to the traced process

استفاده تنها از این ابزار خیلی مرسوم نیست و تنها زمانی که یک پراسس خاصی را بخواهیم مورد بررسی قرار دهیم استفاده میشود ولی در عین حال در strace هم میتوان با استفاده از option هایی که در command line در اختیار است تنها یک process را trace کرد. مشابها این ابزار نیز نیاز به فعال کردن دارد و در دسترس نیست.

11- مقالات و مراجع مورد استفاده

https://android.googlesource.com/
https://developer.qualcomm.com/software/trepn-power-profiler
https://developer.qualcomm.com/forum/qdn-forums/software/trepn-power-profiler/28349
https://developer.android.com/studio/command-line/dumpsys
https://developer.android.com/studio/profile/battery-historian
https://source.android.com/devices/tech/health/implementation
https://blog.packagecloud.io/eng/2016/03/14/how-does-ltrace-work/
https://stackoverflow.com/questions/5494316/how-does-strace-work
https://blog.packagecloud.io/eng/2016/02/29/how-does-strace-work/
https://jvns.ca/blog/2017/03/19/getting-started-with-ftrace/
https://www.osadl.org/fileadmin/dam/presentations/RTLWS11/rostedt-ftrace.pdf
https://source.android.com/devices/tech/debug/ftrace
https://android.googlesource.com/kernel/msm/+/android-5.1.0_r0.6/Documentation/trace/ftrace.txt
https://android.googlesource.com/kernel/msm/+/android-5.1.0_r0.6/Documentation/trace/ftrace-design.txt
https://android.googlesource.com/kernel/msm/+/android-5.1.0_r0.6/Documentation/trace/ring-buffer-design.txt
https://en.wikipedia.org/wiki/LTTng
https://blog.selectel.com/deep-kernel-introduction-lttng/
https://jvns.ca/blog/2017/07/05/linux-tracing-systems/
https://lwn.net/Articles/491510/
http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html
https://blog.selectel.com/kernel-tracing-ftrace/

http://www.kernelmsg.com/15

مرجع اصلی واسه این پروژه بعد از مرحله آشنایی با ابزار ها، خواندن کد های کرنل و systrace بود.