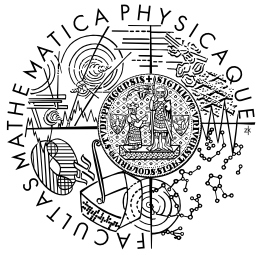


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

David Jaromír Šebánek

Vizualizace trasování procesů v Linuxu

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kára, Ph.D.

Studijní program: Informatika (B0613A140006)

Praha 2025

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Poděkování. iLoveImg upscaling, VS Code, ChatGPT, Qt dokumentace, hwloc dokumentace, Linux kernel dokumentace, WSL, Yordan K., J. Kára, atd.

Název práce: Vizualizace trasování procesů v Linuxu

Autor: David Jaromír Šebánek

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kára, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Abstrakt.

Klíčová slova: Linux, trasování

Title: Visualization of tracing of processes in Linux

Author: David Jaromír Šebánek

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Kára, Ph.D., Department of Distributed and Dependable Systems

Abstract: Abstract.

Keywords: Linux, tracing

Obsah

Úvod

Na světě je mnoho počítačů, mnoho operačních systémů a mnohem více programů pro tyto systémy. Moderní systémy hojně využívají přepínání mezi programy k zefektivnění využívání existujících zdrojů, snaží se tak být co nejefektivnější. Mnoho větších systémů je dnes distribuovaných, s mnoha procesory a technologiemi, které se snaží využít tyto výpočetní jednotky v maximální možné míře.

Ovšem systém není vždy schopen sám zvýšit výkon. V tom případě je nutné začít zkoumat, kde se práce zdržuje, na co se nejvíce čeká a proč se na to čeká. Nalezený problém se pak může hlouběji zanalyzovat a výkon tak s vyřešeným problémem navýšit.

Na hledání problémů existuje mnoho metod, nicméně tato práce se bude zabývat pouze jednou z nich - trasování systému. Trasování systému je ve zkratce úkon, při kterém se na systému spustí nějaká práce společně s programem, který zaznamenává, co přesně systém během práce dělá. Program pak zaznamenaná data uloží. Tato data jsou ovšem často zakódována tak, aby se šetřilo místem - uživatel z těchto dat mnohem více spíš nic nevyčte než naopak.

Na záchranu přicházejí interpreti těchto dat, často s grafickým prostředím, jejichž cílem je vizualizace trasování. Vizualizace pak představí data uživateli v takovém formátu, že v nich již lze hledat místa, kde výkon systému nebyl dostatečně vysoký. Jedním z takových vizualizačních nástrojů je program KernelShark pro Linux. Avšak žádný program není dokonalý, KernelShark nevyjímaje.

KernelShark dokáže efektivně zobrazit rozhodnutí systémového plánovače úloh, na jakém CPU proces pracoval, než šel spát, na jakém CPU práci obnoví, jak dlouho období nečinnosti trvá a podobně. KernelShark ale neumožňuje snadno získat důvod uspání procesu, na kterou událost nebo proces čeká. K tomuto je nutné využít další nástroje. Analýza problémů pak musí spoléhat na dalších několik nástrojů pro získání celé představy o systému a událostech v něm, což je v praxi obtížné, často až nemožné.

Cíle práce

Cíle práce jsou primárně dva: hlouběji představit trasování, jeho vizualizaci a KernelShark, než jak je pouze nastiňuje úvod, a vylepšit KernelShark tak, aby analýza trasovacích dat byla informativnější a uživatelsky příjemnější.

Struktura práce

Kapitoly práce lze rozdělit na dvě části. První část je teoretická a její součástí jsou kapitoly jedna až tři. V nich se hlouběji popisuje trasování v Linuxu, nástroje pro sběr a vizualizaci trasovacích dat a speciálně věnuje jednu kapitolu KernelSharku. Druhá část je zaměřená na vylepšení KernelSharku a pokrývá kapitoly čtyři až deset. Zde se analyzují požadavky na vylepšení, vytvářejí se technická rozhodnutí pro implementaci, součástí jsou i vývojové a uživatelské dokumentace,

spolu s rozšířeními pro každé vylepšení, příklady využití a zhodnocení splnění podmínek. Každé vylepšení má vlastní kapitolu, dodatečná vylepšení jsou seskupena v jedné kapitole a jejich popisy jsou stručnější. Závěr práci shrnuje.

1 Trasování v Linuxu

V této kapitole si představíme trasování systémů, specificky trasování v operačním systému Linux. Zároveň představíme i některé nástroje, díky kterým lze trasovací data sbírat. Po přečtení kapitoly by měl mít čtenář alespoň hrubou představu o konceptu trasování a o jeho podpoře trasování na Linuxu, hlavně o částech, které budou relevantní pro zbytek práce.

1.1 Definice trasování a použití

Trasování lze definovat jako „sběr událostí a jejich dat, které se staly během běhu trasovaného systému či trasovaného procesu“. Logování lze definovat podobně, nicméně trasování je většinou akce sběru nízkoúrovňových dat, která slouží hlavně vývojářům. Trasovací data pak slouží k pohledu na chování systému jako celku, jak spolu některé události souvisí a detailní informace o každé události. Tato data převážně slouží k diagnostice systémů - ať už se jedná o hledání anomálií, míst pro optimalizace, či jako způsob ladění.

1.2 Podpora v Linuxu

Linux široce podporuje trasování kernelu různými způsoby, některé na sebe mohou navazovat. Jedním ze způsobů je použití tracepointů v kernelu. Dalším je *tracefs* systém k trasování událostí. Zajímavé jsou kprobes. Pro tuto práci nejzajímavějšími technologiemi pak budou *ftrace* a právě *systém k trasování událostí*. Krom vyjmenovaných existují další způsoby a nástroje pro trasování, nicméně vyjmenování všech není ani nutné, ani zajímavé.

Následující sekce nepředstavují vyčerpávající seznam existujících technologií, nýbrž jen výběr takových, které se autorovi zdály relevantní k této práci či obecně zajímavé. Relevantní technologie jsou více rozepsané. Technologie zajímavé jsou pak zmíněny na konci a bez tolika detailů.

1.2.1 Linuxové Kernel Tracepoints

Pro trasování částí kernelu obsahuje Linux podporu skrze tzv. tracepoints, počeštěně „tracepointy“. Těmi se dají označit místa v kódu jádra a k těmto místům připojit sondovací funkce. Dokumentace kernelu rozlišuje tracepointy na zapnuté a vypnuté. Vypnuté tracepointy nemají na chod kernelu vliv, kromě nepodstatné časové (kontrola podmínky) a paměťové (existence v kódu) stopy. Tracepoint je vypnutý, pokud nedostal sondovací funkci. Opakem je pak zapnutý tracepoint - když kód přechází přes něj, zavolá se sonda tracepointu a po své práci se vrací na místo volání. Sondovací funkce mohou dělat různou práci, ale častým úkolem je vytvořit trasovací událost v místě tracepointu a zapsat ji na konec trasovacího bufferu. Trasovací buffer je v paměti, má omezenou velikost a je cyklický (po zapsání `N` událostí, kde `N` je kapacita bufferu, se před zapsáním další události vyhodí událost nejstarší).

Tracepointy lze deklarovat a definovat pomocí souboru `linux/tracepoint.h`. V tomto souboru jsou definována makra a funkce pro tyto účely. Makrům se dodají názvy tracepointu, prototyp sondy a názvy jejích parametrů. Hlavička také obsahuje soubor funkcí, s nimiž lze přiřazovat sondy tracepointům pomocí registrací a deregistrací. Lze také kontrolovat, zdali je tracepoint zapnutý, přičemž toto je implementováno tak, že není potřeba řešit podmíněné větvení výpočtu. Důležitou funkcí pak je i volání sondy daného tracepointu přes funkci `trace_<název tracepointu>(<argumenty>)`. Dokumentace kernelu doporučuje nazývat tracepointy ve stylu „subsystém_název-události“. Takto se standardně nazývají ostatní tracepointy v jádře. Tracepoint bude po definici globálně přístupný v celém kernelu a tento formát názvů zabraňuje kolizím. Sondy se předávají při registraci a jsou to nějaké funkce. Při kompilaci se pak kontroluje správný typ funkce.

Tracepointy pak lze vložit jak do regulárních funkcí, tak do inline funkcí, statických inline funkcí i do rozvinutých cyklů. Nedoporučuje se volat tracepointy či kontrolovat zdali jsou zapnuté uvnitř hlavičkových souborů z důvodů vedlejších efektů include direktiv se zapnutým makrem `CREATE_TRACE_POINTS`, které nakonec vyúsťují v nemalý inline a mohou kernel zpomalovat. Proto existuje navíc soubor `tracepoint-defs.h`, který obsahuje funkce, které se mají použít namísto přímého volání a kontroly.

1.2.2 Trasování událostí

S tracepointy 1.2.1 generujícími události můžeme pracovat i více. Základní práci s tracepointy, které vytváří trasovací události, umožňuje systém trasování událostí (dále také nazýván „trasovací systém pro události“). Události zde označují statické tracepointy, které jsou v kernelu k dispozici po kompilaci.

Adresář /sys/kernel/tracing/

Systém s podporou trasování obsahuje v adresáři `/sys/kernel/tracing/` (dále „trasovací adresář“) soubory a podadresáře, díky nimž lze komunikovat s trasovacím systémem pro události. Dohromady se tomuto souborovému systému říká *tracefs*. Předně lze zde zapnout či vypnout různé typy událostí různými způsoby, např. pro zapnutí události „sched/sched_waking“ lze buď napsat:

```
echo 1 > /sys/kernel/tracing/events/sched/sched_waking/enable
```

nebo

```
echo 'sched_waking' >> /sys/kernel/tracing/set\_event
```

Pro vypnutí stačí buď předat v prvním způsobu nulu, v druhém pak stačí před název události vložit vykřičník (stále v rozsahu jednoduchých uvozovek). Je možné zapnout či vypnout i všechny události daného podsystému, nebo pracovat s událostmi pouze nějakého modulu kernelu.

Formát událostí

Každá událost obsahuje i soubor se svým formátem. V něm je definována struktura dat události, název a identifikátor události a formát tisku této události.

Předepsaný formát je pak využíván při čtení dat z binárních trasovacích záznamů. Data události jsou strukturována do datových polí ve formátu `field:field-type field-name; offset:N; size:N;`. Offset značí offset daného pole v trasovacím záznamu a size je velikost v bajtech. Každá událost obsahuje pole označená prefixem `common_`, události pak mohou dále specifikovat vlastní pole, např. událost „sched_wakeup“ obsahuje pole s ID procesu probouzeným procesem, jemuž událost patří.

Filtrování událostí

Ne všechny události daného typu je nutné sbírat při trasování, někdy nás mohou zajímat jenom pokud splňují další podmínky. Trasování událostí dovoluje specifikovat filtry, tj. predikáty, které se vyhodnotí při každém záznamu dané události. Tyto filtry pak mohou cílit na jednotlivá datová pole. Pro číselná pole se dají použít klasické relační operátory na ekvivalenci (`==`), rozdílnost (`!=`), vztah menší/větší (nebo roven) (`<`, `>`, `<=`, `>=`), bitové „a zároveň“ (`&`). Pro pole s textem se dají použít operátory na ekvivalenci, rozdílnost a podobnost (`~`). Operátor pro podobnost pak očekává text ve složitých uvozovkách, který může obsahovat sekvenci abecedních znaků, wildcard znaky `?` a `*`, nebo třídu znaků (mezi hranatými závorkami `[]`). Například můžeme zkusit filtrovat pro jméno končící na „shell“ a písmena „s“, nebo „t“, nebo „u“: `some_text_field ~ "*shell[stu]"`. Pokud je v textovém poli ukazatel na string v uživatelském prostoru, musíme za název pole dodat `.ustring`. Pokud obsahuje pole ukazatel na funkci, pak musí být dle formátu typu `long` a za název pole se musí připsat `.function`. Pole typu `cpumask` nebo skalární pole obsahující číslo CPU lze filtrovat pomocí uživatelem zadané CPU masky ve formátu seznamu CPU. Například výraz `target_cpu & CPUS17-42` vyfiltruje události, jejichž hodnota v poli `.target_cpu` se nachází v rozsahu CPU 17 až 42. Predikáty pro různá pole se dají spojovat přes boolovské operátory a oddělovat závorkami.

Filtry lze přidat k dané události v trasovacím adresáři do souboru `filter` v adresáři události. Stačí přes `echo` zapsat do obsahu tohoto souboru daný predikát. Pokud chceme filtry vyčistit, stačí do souboru zapsat `0`.

Filtry lze nastavovat i pro celé subsystémy (resp. lze nastavit filtr všem událostem přes nastavení pro subsystém, čímž omezíme nutnost opakování stejných příkazů). V tom případě je nutno si dát pozor, aby všechny události v podsystému měly filtrovaná pole. Pokud tomu tak není, je daný filtr ignorován a událost používá filtr předchozí. Jistojistě budou vždy fungovat filtry na pole společná všem událostem.

Spouštěče událostí

Každá událost může navíc spouštět „příkazy“, když je zrovna vyvolána. Událost může mít vícero spouštěčů. Tyto příkazy mají vlastní filtry, podobně jako filtry pro události. Událost, která neprostoupí filtrem nějakého spouštěče tento spouštěč nevyvolá. Spouštěče bez filtrů se vždy spustí. Podstatnou vlastností spouštěčů je, že je lze spustit, aniž by byla samotná událost trasována. Tracepoint události se zavolá, ale pokud není zapnut, nebude událost trasována, nicméně spouštěče (a jejich filtry) spuštěny budou. Tohoto „polo-vypnutí“ se dá docílit skrze příkazy spouštěčů.

Podobně jako u filtrů se spouštěče přidávají do nějakého souboru události v trasovacím adresáři. Tímto souborem je `trigger` v adresáři události, pro kterou chceme spouštěč vytvořit. Formát výrazu přidání spouštěče:

```
echo 'command[:count] [if filter]' > trigger
```

Pro odstranění stačí před `command` přidat vykřičník. Část `if filter` vyžaduje napsání slova if a pak filtrovacího predikátu, jako pro filtr události. Část s `:count` pak říká, kolikrát bude spouštěč spuštěn. Zmíníme i chování operátoru `>`, který se chová jako operátor `>>` pro operace přidání a odebrání, tj. je nutno odebírat po jednom.

Spouštěcí příkazy jsou předem dány a jsou to právě tyto:

- *enable_event/disable_event* - tyto příkazy buď zapnou nebo „polo-vypnou“ specifikovanou událost při každém vyvolání spouštěcí události. Formát pro tyto příkazy je `enable_event/disable_event:<system>:<event>`, kde `<system>` je název systému, kterému patří událost jménem `<event>`. U těchto příkazů není povoleno mít více spouštěčů (buď zapínacích, nebo vypínacích) pro jeden typ události.

- *stacktrace* - tento příkaz zapíše záznam zásobníku do trasovacího bufferu. Může být jen jeden na spouštěcí událost.

- *snapshot* - tento příkaz vytvoří snapshot při spouštěcí události. Může být jen jeden na spouštěcí událost. Snapshot ve zkratce prohodí současný trasovací buffer s novým bufferem a původní buffer si uloží do souboru `snapshot`. Lze tak nasbírat zajímavé informace bez neustálých zápisů na disk.

- *traceon/traceoff* - tyto příkazy zapínají, nebo vypínají trasovací systém. Není povoleno mít více stejných příkazů na spouštěcí událost.

- *hist* - tento příkaz agreguje vyvolávání spouštěcí události. Data jsou pak využívána v histogramech událostí 1.2.6, další z trasovacích technologií.

1.2.3 Ftrace

Ftrace je interní nástroj kernelu pro vývojáře a návrháře systémů, díky kterému lze nahlédnout do chodu kernelu. Název pochází ze slov „function tracer“, nicméně tento nástroj má dnes funkcionality širší. Jeho hlavním účelem je analýza latencí a výkonu mimo uživatelský prostor. Jedním z nejčastějších využití je při trasování událostí, kde Ftrace využívá schopnosti tracefs. Pokud není tracefs aktivní v Linuxu, použitím Ftrace aktivován bude (tj. tracefs bude připojen přes `mount` příkaz). Ftrace pak má v trasovacím adresáři (připomeneme, že to je `/sys/kernel/tracing`) přístup ke svým řídícím a výstupním souborům. Souborů a adresářů je více, představíme si jen pár z nich, abychom se lépe s Ftrace seznámili.

- *available_tracers* - tento soubor má seznam tracerů, které byly zkompilovány do kernelu. Jeden ze zdejších tracerů lze skrze `echo` zapsat do souboru *current_tracer*, který značí, který tracer má Ftrace při trasování použít.

- *tracing_cpumask* - tento soubor je maska, přes kterou lze nastavit, která CPU mají být součástí trasování přes Ftrace.
- *trace_options* - soubor řídí chování tracerů a tištění výstupu. V souboru je seznam nastavení pro tyto účely. Na zakázání možnosti stačí přes `echo` zapsat do souboru `no<název>`, kde <název> bude jméno možnosti. Bez zakazujících prefixu bude možnost zapnuta. Vyjmenujeme si pár z možností:
  - print_parent - v trasovacích datech funkcí se zobrazí volající funkce vedle trasované funkce.
  - context-info - zobrazí se pouze data události, nebudou zobrazeny sloupce s PID, CPU či jiné zajímavé sloupce.
  - pause-on-trace - pokud otevřeme soubor trace, pak se trasování zastaví, dokud jej nezavřeme.
  - record-cmd - při trasování sched_switch události se zaznamenává jak PID, tak jména procesů. Pokud tuto možnost vypneme, nebudeme si ukládat názvy procesů a snížíme tak dopad trasování na výkon.
  - disable_on_free - při zavření souboru free_buffer se zastaví trasování.
  - stacktrace - po každé události se zaznamená zásobník kernelu
- *trace* - v tomto souboru jsou výsledky posledního Ftrace trasování ve formátu čitelném lidmi. Tento soubor nekonzumuje data po přečtení, ta přetrvávají. Pokud je trasování vypnuté, bude se zobrazovat stejný obsah. Soubor by neměl být čten během aktivního trasování, výsledky nemusí být konzistentní.
- *free_buffer* - tento soubor je využíván při uvolňování či zmenšování trasovacího bufferu - tato akce je provedena, když je tento soubor zavřen. Zároveň je možné tímto souborem trasování zastavit, pokud k tomu Ftrace nakonfigurujeme.
- *set_ftrace_pid* - Ftrace bude trasovat vlákna procesů jen a pouze s PID v tomto souboru. Pokud Ftrace nakonfigurujeme s možností `function-fork`, pak pokud proces s PID v tomto souboru udělá fork, pak PID podřízených procesů se sem přidají automaticky.
- *kprobe_events & kprobe_profile* - používají se u dynamických tracepointů, viz sekce 1.2.5.
- *stack_max_size* - pokud trasujeme zásobník (kernelu), tento soubor udržuje největší ze zaznamenaných velikostí zásobníků.
- *stack_trace* - tento soubor zobrazí největší zaznamenaný zásobník.
- *trace_clock* - záznamy událostí získávají časovou stopu, tedy kdy byly zaznamenány. V tomto souboru se nastavuje, který typ hodin je použit pro časové stopy, přičemž Ftrace má za výchozí hodiny lokální pro každé CPU. Typů hodin je několik, vyjmenujeme jich pouze pár:
  - local - Ftrace použije hodiny lokální pro každé CPU.

  - global - hodiny, které jsou synchronizovány pro všechna CPU, ale jsou trochu pomalejší
  - perf - pro synchronizaci s nástrojem Perf, který tento typ hodin používá. Nástroj má v budoucnu používat Ftrace buffery a tyto hodiny pomůžou s prokládáním dat.
  - counter - atomický čítač, nejsou to opravdové hodiny. Hodí se, pokud je nutné znát přesné pořadí událostí v celém systému (čítač je synchronizován pro všechna CPU).
  - x86-tsc - některé architektury specifikují vlastní hodiny, x86 používá hodiny tohoto typu.

- *per_cpu* - tento adresář obsahuje adresáře pro každé CPU na systému, v nichž jsou některé soubory jako v trasovacím adresáři, ale lokální pouze pro daný procesor. Například lze v `per_cpu/cpu0/trace` vidět obsah posledního trasování na tomto CPU.

Zmínili jsme pojem „tracer“. To je část kernelu, která má za úkol trasovat předem dané typy chování za běhu, zaměřuje se na nějaký aspekt systému. Ftrace má k dispozici několik tracerů, základně *function tracer*, který se sonduje při vstupu do kernelových funkcí. Je možné použít i *function graph tracer*, který sonduje jak při vstupu, tak při výstupu z funkce. Tím dokáže zobrazit alší informace, například dobu běhu funkce, nebo pro zkoumání volání funkcí. Dále zmíníme *wakeup tracer*, ten trasuje a zaznamenává maximální latenci mezi probuzením a naplánováním úlohy s nejvyšší prioritou, podobně pak existuje i *wakeup_rt tracer*, který se zajímá pouze o real-time úlohy. Zajímavým příkladem je i *nop tracer*. Pokud jej nastavíme do souboru current_tracer, pak ostatní tracery přestanou trasovat.

Chyby

Ftrace používá hlavně návratové kódy, které pak zpracuje jejich příjemce. Složitější chyby se pak ukládají do souboru *error_log* v trasovacím adresáři. Příkazy, které jej umějí číst, mohou z chyb vyčíst detailní informace, jsou-li k dispozici. Log je cyklický a pamatuje si nejvýše osm chyb, tj. chyby z posledních osmi příkazů.

Příklady použití

Ukážeme pár příkladů použití Ftrace. První příklad je velmi jednoduchý, použijeme nop tracer. Čtení souboru trace po trasování s nop tracerem vypadá takto:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 0/0   #P:8
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
```

```
#                              ||| / _-=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
```

Je zřejmé, že nop tracer opravdu nic netrasuje. Jediné, co nám zbylo, jsou vysvětlovací popisky a pár metadat o použitém traceru a záznamech. Nejvýše je název použitého traceru, pak počet záznamů v bufferu na počet zapsaných záznamů a vpravo počet procesorů na systému.

Druhý příklad je o trochu zajímavější. S použitím function traceru jsme získali výstup:

```
# tracer: function
#
# entries-in-buffer/entries-written: 410067/13139037   #P:8
#
#                      _-----=> irqs-off
#                     / _----=> need-resched
#                    | / _---=> hardirq/softirq
#                    || / _--=> preempt-depth
#                    ||| / _-=> migrate-disable
#                    |||| /     delay
# TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#    | |         |   |||||     |         |
  node-831     [007] ..... 30994.493939: get_futex_key <-futex_wake
  node-831     [007] ..... 30994.493940: hash_futex <-futex_wake
  node-831     [007] ..... 30994.493940: _raw_spin_lock <-futex_wake
  node-831     [007] ..... 30994.493940: mark_wake_futex <-futex_wake
  node-831     [007] ..... 30994.493940: __unqueue_futex <- \
                                         mark_wake_futex
  node-831     [007] ..... 30994.493941: wake_q_add_safe <-futex_wake
  node-831     [007] ..... 30994.493941: wake_up_q <-futex_wake
  node-831     [007] ..... 30994.493941: try_to_wake_up <-wake_up_q
  node-831     [007] ..... 30994.493941: _raw_spin_lock_irqsave <-\
                                         try_to_wake_up
  node-831     [007] d.... 30994.493941: ttwu_queue_wakelist <-\
                                         try_to_wake_up
...
```

Function tracer nám zaznamenal velmi mnoho informací, jak ukazuje hlavička. Výstup jsme zkrátili (a zmáčkli, aby se vešel do kódového bloku). Všechny viditelné funkce jsou z osmého CPU, jsou z jediného procesu a týkají se probouzení. Tento výpis je z autorova počítače na Linuxovém subsystému ve Windows 11 a trasování probíhalo pouze pár vteřin.

Trace-cmd

[TODO: Zdroj oficiální kernel a man docs + https://lwn.net/Articles/410200/]

Trace-cmd je front-endová aplikace pro Ftrace, která zjednodušuje používání tohoto nástroje. Tuto aplikaci vyvinul Steven Rostedt, o kterém ještě uslyšíme

v kapitole 3 o KernelSharku. Ftrace vyžaduje používání tracefs a zapisování či čtení ze souborů. S dokumentací je pak používání celkem jednoduché, což je velká výhoda u embedded systémů. Nicméně desktopové systémy mají prostředky ke složitějším nástrojům a způsobům interakce, práce se soubory pak může působit pomale. A právě nástrojem pro jednotnější použití je Trace-cmd. Další důležitou výhodou Trace-cmd je i to, že svá data ukládá v binárním formátu, resp. přímo kopíruje jejich binární reprezentaci z trasovacího bufferu, nikoliv jejich textovou verzi z Ftrace souboru `trace`. Při sběru mnoha událostí je tato vlastnost zásadní, jelikož není potřeba textový formát vytvářet a sběr je stále rychlý. Trace-cmd je důležitým nástrojem pro zbytek této práce, jelikož představuje zdroj dat pro KernelShark.

Trace-cmd dovoluje například vytvořit trasovací záznam pro nějaký proces přes svůj příkaz `trace-cmd record <cmd>`, kde <cmd> je jiný příkaz, jehož průběh (resp. průběh procesu, který spustí) má být trasován. Trasovací data jsou pak uložena do souboru, výchozím jménem `trace.dat`. Trace-cmd pak přes příkaz `trace-cmd report <input>`, kde <input> je název souboru vytvořeného přes Trace-cmd, dokáže vytvořit výstup, jako při čtení souboru trace v trasovacím adresáři. Trace-cmd dovoluje i trasovat přes síť s příkazem `trace-cmd listen`, což může být užitečné pro trasování embedded zařízení, ke kterým máme síťový přístup a nechceme trasování spouštět a číst přímo z nich. Pokud chceme pouze jednoduše začít trasovat přes Ftrace, pomohu příkazy `trace-cmd start` a `trace-cmd stop`. Trasovací data Ftrace lze pak extrahovat do souboru přes `trace-cmd extract`. Trace-cmd dovoluje i všechna trasování vypnout přes příkaz `trace-cmd restart`. Všechny příkazy navíc mají několik možností konfigurující jejich chování. Například, chceme-li trasovat zásobník kernelu po každé události, pak stačí zadat příkaz `trace-cmd record -T <cmd>`. Příkazů je více, tyto jsou ale jedny z nejzákladnějších.

1.2.4 Perf

[TODO: Zdroj https://perfwiki.github.io/main/ + https://www.brendangregg.com/perf.html]

Nemůžeme zmínit měření výkonu a trasování bez známého a dlouhodobě používaného programu k profilování, *Perf* (původním názvem Performance Counters for Linux/PCL). Profilování vytváří statistiky o běhu systému a často se používá na zjištění toho *kde* je největší výkonnová ztráta. Profilování nedokáže odpovědět na otázku *proč* je nějaká část software problémová. K tomu se pak většinou zkoumají právě trasovací data v pořadí výpočtu. Profilování je nicméně velmi důležitou částí analýzy a s trasováním tvoří ty nejzákladnější pilíře výkonostní analýzy.

Nástroj Perf je dostupný již od verze kernelu 2.6 a je součástí kernelové instalace. Perf je schopen pro svou analýzu použít „čítače výkonu“, což jsou hardwarové registry na CPU, které počítají různé údaje, například počet provedených instrukcí, počet přerušení, počet cache-misses, či počet špatně předpovězených větví výpočtu. Kromě toho dokáže Perf při analýze využít i různé typy událostí, mezi nimiž jsou i tracepointy od Ftrace, události software, například změna kontextu CPU. Události dány čítači na CPU, ale i události jako třeba načtení cache pro instrukce nebo naplnění cache TLB se pak označují za události hardware. Dalšími zdroji dat jsou i dynamické trasování přes sondování s kprobe1.2.5, či uprobe.

Perf dává k dispozici mnoho příkazů, my si představíme jen pár z nich.

- *perf list* - příkaz zobrazí podporované události na daném systému. Na většině systémů bude seznam podobný až na události hardware. Ty se mohou lišit dle použitého procesoru a jeho přítomných čítačů výkonu.
- *perf stat* - tímto příkazem se spustí počítání událostí v příkazu daném jako poslední argument Perf příkazu. Události se dají specifikovat, dokonce i na události uživatelské a kernelové. Perf pak vypíše statistiky událostí po skončení příkazu v argumentu.
- *perf record* - tímto příkazem se Perf připojí k nějakému vláknu procesu a periodicky ukládá své statistiky do nějakého datového souboru. Periodicky zde znamená, že po překročení daného čítače (který čítá vyvolání nějaké předem dané události, výchozí událostí jsou cykly na procesoru) Perf vytvoří statistiky. Tento příkaz dovoluje zaznamenávat i statistiky pro všechna vlákna na montorovaném CPU.
- *perf record* - tento příkaz interpretuje data v souboru vytvořeném při `perf record`.
- *perf top* - v reálném čase ukazuje procesy/funkce s nejvyšší spotřebou CPU.

Krom příkazů podporuje Perf i spouštění skriptů. Nativně je například podporován skript pro FlameGraphs (viz v druhé kapitole 2.3), které slouží k analýze zásobníku v čase. Dalším zajímavým skriptem je Gecko od organizace Mozilla, který slouží k analýze webových aplikací.

1.2.5 Kprobes

Dynamičtější přístup k trasovní se objevil u technologie Kprobes. Podobně jako tracepointy, i Kprobes se přidávají do kódu kernelu. Ale narozdíl od statických tracepointů, jež je nutné definovat před kompilací, lze Kprobes přidávat za běhu kernelu. Kprobes se ve skutečností dělí na `kprobe` a `kretprobe`. První mohou být přidány skoro všude v kernelu, druhé při návratu z funkcí. Aby šlo technologii využívat, je většinou zkompilována jako kernelový modul a jsou přidány registrační a deregistrační funkce. Registrační funkce pak dostává handler, který se spustí, když program narazí na kprobe.

Kprobe funguje na principu přerušení CPU. Při registraci si uloží sondovanou instrukci. Kprobe pak v originální sondované instrukci změní první bajt či bajty, aby se výpočet přerušil. Kprobe pak zavolá svůj handler, udělá nějakou práci, a nechá CPU běžet dále pomocí kopie sondované instrukce. Důležité je, že v moment práce má kprobe přístup ke všem registrům CPU, tedy i k registru z ukazatelem na vykonávanou instrukci. Proto je nutné při psaní handlerů pro Kprobes být maximálně opatrní, abychom nerozbili kernel zevnitř při použití této technologie.

Kretprobe vytvoří na vstupu funkce, jejímž je členem, kprobe (té uživatel může i nemusí dát vlastní handler). Kprobes si pak při vchodu do funkce a vyvolání vchodové sondy uloží návratovou adresu a originál nahradí adresou do libovolné části kódu, často k nějaké `nop` instrukci. Tím vytvoří, jak tomu říká dokumentace kernelu, jakousi „trampolínu“, v níž je další kprobe (v této je

handler, který uživatel musí pro kretprobe specifikovat). Když se pak sondovaná funkce (ta s vchodovou kprobe) vrátit, skočí namísto toho do trampolíny. Handler trampolíny zavolá uživatelem specifikovaný handler a Kprobes pak obnoví původní návratovou adresu z uložené kopie a výpočet pokračuje dál jako obyčejně. Kprobes má mechanismy i pro zvládání rekurzivních funkcí, které by kretprobe využívaly.

Kprobes očividně nutí CPU často zastavovat, proto lze kernel postavit s možností `CONFIG_OPTPROBES=y`, že namísto přerušení se při nalezení kprobe někam prostě skočí.

Kprobes si udržují černou listinu funkcí, ve kterých Kprobes operovat nesmí. Lze ji rozšířit pomocí makra `NOKPROBE_SYMBOL()` ze souboru `linux/kprobes.h`. Hlavními členy listiny jsou funkce od samotných Kprobes.

Podobnými technologiemi ke Kprobes jsou Uprobes, které se dají použít v uživatelském prostoru, a Fprobes, které fungují podobně jako function-graph tracer, ale lze přes ně trasovat pouze vybrané funkce namísto všech.

1.2.6 Histogramy událostí

Histogramy událostí jsou speciální spouštěče u událostí, které agregují vyvolání událostí a data události, pokud lze. Formát spouštěče je následující:

```
hist:keys=<field1[,field2,...]>[:values=<field1[,field2,...]>]
  [:sort=<field1[,field2,...]>][:size=#entries][:pause][:continue]
  [:clear][:name=histname1][:nohitcount][:<handler>.<action>]
  [if <filter>]
```

Po spuštění bude v podadresáři události v trasovacím adresáři (tj. v `[trasovací adresář]/events/[název události]`) nový soubor s názvem `hist`, tedy na stejném místě jako soubory pro definice filtrů a spouštěčů pro danou událost. Pomocí spouštěčů `disable/enable_hist:<system>:<event>[:count]` lze zapínat a vypínat spouštěče histogramů. Tyto spouštěče jsou podobné spouštěčům pro zapínání a vypínání událostí. Lze tak vytvořit další pravidla vedle filtrů, která určují, kdy sbírat data pro histogramy.

Vysvětlení formátu spouštěče

Klíče a hodnoty musí být datová pole z formátu události, hodnoty také musí být číselné, aby bylo možné je sčítat. Pokud nejsou hodnoty specifikovány, implicitně se použije „hitcount“ hodnota, které pouze sčítá, kolikrát trasování natrefilo na událost. Pokud zapíšeme jako klíče pole „common_stacktrace“, čímž jako klíč použijeme zásobník kernelu při této události. Klíče mohou být složené (více polí dohromady, maximálně tři) či jednoduché (jedno pole), přičemž různé permutace prvků ve složeném klíči vytvářejí různé klíče. Pro vytvoření složeného klíče stačí jednotlivé části klíče oddělit čárkou. Složené klíče lze pak setřídit pomocí parametru `sort` a specifikace až dvou polí. U dvou třídících polí pak budou setříděny nejprve podle prvního klíče a pak podle druhého klíče. Sort lze nicméně použít i na hodnoty a třídit podle nich. Pokud dáme histogramu jméno přes parametr `name` a toto jméno použijeme i u jiných histogramových spouštěčů, budou vytvořené histogramy tato data sdílet. Sdílení je možné pouze tehdy, pokud jsou spouštěče

kompatibilní, tj. mají stejný počet polí, pole jsou stejně nazvaná a pole mají stejný typ. Hodnoty oddělené čárkou pro `values` se sčítají.

Existují speciální pole, které lze použít pro klíče i hodnoty pro jakékoliv hodnoty, ačkoliv nejsou součástí formátu události. Jsou jimi `common_timestamp` typu nezáporného 64-bitového čísla a `common_cpu` typu 32-bitového znaménkového celého čísla. První pole je časová značka, kdy byla událost zaznamenána. Druhé pole říká, na kterém CPU se událost stala.

Je možné dodat i další parametry a modifikace do spouštěče. Parametr `nohitcount` nezobrazí hodnotu hitcount v histogramu. Pokud je toto použito, je nutné dodat nějakou hodnotu, která není „holý hitcount“, nelze tedy zobrazit `vals=hitcount:nohitcount`, ale lze zobrazit `vals=common_pid:nohitcount` nebo `vals=hitcount.percent:nohitcount`.

Modifikátor `.percent` je jedním z několika modifikátorů pro hodnoty, které se ve výchozím nastavení zobrazí jako zobrazí číslo jako celá čísla v desítkové soustavě. Dalšími modifikátory jsou třeba `.hex`, který zobrazí čísla v hexadecimální formě, `.log2` zobrazí dvojkový logaritmus čísla, `.execname` zobrazí PID jako jméno procesu (zde musí být hodnota z pole `common_pid`), `.stacktrace` dokáže zobrazit zásobník (zde musí hodnota být typu long[]) a nebo `.sym`, kdy se číslo interpretuje jako adresa nějakého symbolu.

Kromě části `if <filter>`, která funguje stejně jako u jiných spouštěčů, stačí vysvětlit parametr :handler.action. Action je funkce, která se zavolá při přidání či aktualizaci histogramu. Handler rozhoduje, zdali má být action zavolána, nebo ne.Výchozí nastavení tohoto parametru je prostá aktualizace dat v histogramu. Pokud chceme udělat více práce, třeba vyvolání další události, pak můžeme právě tento pár nastavit sami. Na výběr máme z několika předdefinovaných handlerů a action funkcí. Handlery jsou:

- onmatch(matching.event) - zavolá action funkci při přidání nebo změně týkající se události „matching.event“ (událost lze zapsat jako subsystém.událost, např. sched.sched_switch).
- onmax(var) - zavolá action pokud histogramová proměnná překročí nějakou hodnotu.
- onchange(var) - action zavolá při změně histogramové proměnné.

Dostupné action funkce jsou:

- trace(<název umělé událost>,seznam parametrů) - vygeneruje umělou událost, seznam parametrů pak určuje další datová pole v ní
- save(pole,...) - uloží aktuální data ve specifikovaných polích události, která spouštěč vlastní
- snapshot() - vytvoří snapshot trasovacího bufferu, který bude uložen do souboru `snapshot` v trasovacím adresáři

Ačkoliv teorie nikterak nebrání všem kombinacím handlerů a action funkcí, ne všechny jsou podporované. Pokud jsou použity, spouštěč se nespustí a vrátí chybový kód `-EINVAL`.

Histogramové proměnné

Histogramy dokážou pracovat s proměnnými, místy k ukládání dat z různých procesů a spouštěčů. Vytvářejí se přes syntax `název_proměnné=$datové_pole` a jsou přístupné globálně nebo lokálně pro nějaký klíč. K získání hodnoty v proměnné stačí napsat `$název_proměnné`. Do proměnných lze ukládat i trasovaný zásobník, nebo číselné literály. Hodnoty se samy od sebe nemění, kromě situace, kdy je proměnná použita v nějakém aritmetickém výrazu, poté svou hodnotu zahodí, dokud není znovu nastavena. Lze vytvořit i několik proměnných, stačí je oddělit dvojtečkami v definici spouštěče. Na pozici vytvoření proměnné ve spouštěči nezáleží, tj. lze použít proměnnou a definovat ji později. Příklad chování:

```
echo 'hist:keys=pid:ts=\$timestamp' > \
/sys/kernel/tracing/events/sched/sched\_waking/trigger

echo 'hist:keys=next_pid:wake_switch_lat=\$timestamp-\$ts' > \
/sys/kernel/tracing/events/sched/sched\_switch/trigger
```

První příklad uloží časovou značku při události sched_waking do proměnné *ts*. Proměnná je viditelná všem událostem se stejným PID jako v next_pid. Druhý příkaz vypočítá čas mezi sched_waking a sched_switch pro daný PID, určený polem next_pid. Po spočítání výrazu bude proměnná `$ts` prázdná a nelze ji použít, dokud není nastavená. Tento mechanismus zajistí, že hodnota v proměnné nebude použita se staršími daty v dalších sched_switch událostech.

Inter-event histogramy

Takto se označují histogramy, které kombinují data z několika různých událostí. Často jsou takovými histogramy zkoumány latence, například mezi probuzením procesu a přepnutím kontextu, nicméně systém pro podporu takových histogramů je obecnější a lze kombinovat více druhů dat. Tyto histogramy pak logicky nepatří žádné z událostí, patří všem naráz. Kvůli inter-event histogramům byly vytvořeny action funkce, podpora pro vytváření umělých událostí (ty pak sdružují data z několika událostí), histogramové proměnné, pole common_timestamp a podpora jednoduchých aritmetických operací.

1.2.7 Další trasovací technologie

Jak kapitola ukázala výše, Linux podporuje trasování různě a široce. Existuje mnoho technologií, některé jsme rozebrali výše. Technologie níže byly označeny za zajímavé, ale ne příliš relevantní ke zbytku práce. Jejich popis tedy bude krátký, vyznačující jejich záměry, některé charakteristiky a schopnosti. Pro detaily a další nástroje autor vřele doporučuje přečíst si oficiální dokumentaci kernelu.

Boot-time trasování

Trasování při bootování dovoluje uživateli trasovat procesy, které se dějí například při inicializaci zařízení. Navíc je plně podporován Ftrace a jeho funkcionality. K zapnutí je nutné modifikovat nastavení bootování. Ftrace je pak také konfigurován v souboru pro boot. Při tomto trasování lze mít i více instancí Ftrace běžících

naráz, tedy i víc tracerů, čímž lze získat více dat o spouštění systému, než pouze z jednoho traceru.

Uživatelské události

Tato technologie dovoluje uživatelům bez privilegovaného módu vytvářet vlastní trasovací události, které jsou zpracovatelné nástroji Ftrace i Perf. Aby k této možnosti byl přístup, musí být použita možnost `CONFIG_USER_EVENTS=y` při sestavování kernelu. Tracefs pak bude obsahovat další soubory pro tyto uživatelské události.

OSNOISE Tracer

OSNOISE je využíván v HPC (High Performance Computing) prostředí k měření interference, kterou pociťují aplikace kvůli aktivitám uvnitř operačního systému, tedy prodlevy způsobené chováním operačního systému či chováním hardwaru. Nejpřímějším příkladem v Linuxu jsou obyčejné žádosti o přerušení (IRQ), SoftIRQ, ale zdroji mohou být i přerušení vzniklá chybou hardware (NMI/Non-maskable interrupt) či jiné práce systémových vláken. OSNOISE pak aktivity, které zdržují, označuje za šum. Z tohoto označení vychází název traceru, OS (operační systém) + NOISE (šum), tj. tracer „šumu operačního systému“.

OSNOISE pracuje podobně jako hwlat_detector (Hardware Latency Detector) tracer. Periodicky spustí smyčku ve vlastním vlákně a v ní sbírá data o šumu. Narozdíl od hwlat_detector traceru nevypíná přerušení (tj. IRQ a SoftIRQ) a nechá vše běžet v preemptivním režimu - tím si zajišťuje detekci všech druhů interference. Při každém průchodu smyčkou detekuje vstupní události NMI, IRQ, SoftIRQ či plánování vláken a zvyšuje odpovídající čítače. Pokud prodleva nevznikne žádným z těchto zdrojů, zvýší se čítač hardwarového šumu. OSNOISE po ukončení smyčky zobrazí statistiky z trasování, např. nejvýraznější ze šumů nebo procentuální dostupnost CPU pro nějaké vlákno.

RV - Runtime Verification

RV je mechanismus založený na trasování, který slouží k monitorování běhu systému podle předem definovaných vlastností či pravidel. Ty jsou typicky zapsány ve formě automatů. Na rozdíl od klasického trasování, kde se zaznamenávají všechny události, RV kontroluje, zda sledovaný běh odpovídá očekávanému chování. Při porušení očekávání může vyvolat událost. Používá se například pro bezpečnostní monitoring, nebo formální ověření určitých vlastností v jádře. Tato metoda je rigorózní a zároveň nenáročná pro systém.

1.3 LTTng - Linux Trace Toolkit: next generation

[TODO: https://lttng.org]

Projekt oddělený od Linuxového jádra, ale také zaměřený na trasování, *LTTng* je kolekcí kernelových modulů, s nimiž lze trasovat kernel, a dynamických knihoven, které slouží k trasování uživatelských programů a knihoven. LTTng se snaží být alternativou k existujícím trasovacím nástrojům a jejich ekosystému. Software je ve vývoji již od roku 2005 a byl navržen pro minimální dopad na výkon ostatních

procesů při svém běhu. Projekt je open-source a dodnes velmi aktivní. Podporován je několika známými distribucemi, například ArchLinux, Ubuntu či Debian.

LTTng pak pracuje jako démon a uživatel s ním komunikuje přes jediný program s rozhraním pro terminál. LTTng dokáže trasovací data vytvářet, přičemž lze filtrovat události, které chceme zaznamenat, program dokáže i vytvářet spouštěče, a nebo dodává vlastní definice pro vytváření tracepointů. Trasovat lze přes síť, lokálně, paralelně (tj. lze zaznamenávat více různě nakonfigurovaných trasovacích dat naráz) a i v reálném čase. Zaznamenaná data poté přestavují unifikovaný log událostí, kde se mohou i míchat události z uživatelského prostoru s událostmi kernelu.

Data od LTTng pak může číst sada nástrojů *Babeltrace* nebo GUI program *TraceCompass*.

2 Vizualizace trasovacích dat

V této kapitole se zaměříme na to, proč je dobré mít vizualizátory dat a na nástroje sloužící k vizualizaci dat trasovacích. Jako vizualizátory bereme jakýkoliv typ software, který čte data a zobrazuje je přes nějaké grafické rozhraní. Mohou to být jednoduché skripty, které vytvoří soubory HTML a CSS pro zobrazení v internetovém prohlížeči, nebo samostatné aplikace s vlastními grafickými možnostmi.

2.1 Proč vizualizovat

Velmi rozšířeným typem systému je desktop. Počítač má dostatečné schopnosti na vytváření komplexnějších grafických aplikací. Toho spousta existujícího software využívá na zobrazení dat takovým způsobem, aby je lépe pochopili lidé. Ačkoliv rozhraní v terminálu je funkční a vhodné pro ostatní software, lidé více rozumí barvám, grafům a obrázkům, než pouhým seznamům a vypsaným číslům, hlavně pokud se jedná o mnoho dat. Terminál samotný má pak omezené schopnosti zobrazování grafiky, navíc ne všechny terminály jsou si rovny (např. terminály nemusí podporovat stejné barevné rozsahy). Trasovacích dat je mnoho, analyzovat v nich lze ledacos a (dobré) grafické rozhraní analýzu člověku usnadní, hlavně u dat o milionech událostí z velkých systémů.

Vizualizátory také mohou obsahovat funkcionality, které nejsou či nemohou být přítomny v nástrojích sloužících jako datový backend pro vizualizátory. Mohou to být například jednodušší rozhraní na vytváření filtrů či ukládání rozpracované analýzy, tj. například pozice v trasovacích datech a nějak vybrané události. Vizualizátory také mohou kombinovat data z více nástrojů, čímž se analýza zjednoduší, jelikož všechna potřebná data budou na jednom místě.

2.2 HPerf

[TODO: https://www.poirrier.ca/hperf/]

HPerf je frontendová aplikace pro nástroj Perf. Perf samotný se již snaží o pěkné zobrazení dat v terminálu, HPerf jde o krok dál a z dat vytvořených od `perf record` dokáže vytvořit stránku v HTML s JavaScriptem (viz obrázek 2.1), kterou lze navíc stylově upravovat pomocí vlastních CSS pravidel. HPerf dokáže vykonat práci `perf annotate` a `perf report` a navíc přidává vlastní funkcionality, to vše v GUI, k níž potřebujete pouze internetový prohlížeč s podporou JavaScriptu. Program má také minimální závislosti, jimiž jsou již zmíněný internetový prohlížeč, Perf, pak nástroje `objdump` a volitelně `highlight`. Na sestavení pak stačí použít buď GCC, nebo Clang, a program Make.

HPerf umí zobrazit assembly kód programu z dat a v okénku vedle i příslušnou část zdrojového kódu. Tato dvě okna jsou nezávislá. Okno se zdrojovým kódem navíc podporuje zvýrazňování syntaxe zkompilovaného jazyka.

HPerf dokáže pracovat s dlouhými trasovacími daty, ale velké binární soubory vytvoření HTML mohou zpomalit. Autor programu, Laurent Poirrier, toto vysvětluje nutností nahrát přes Objdump do paměti všechny assembly instrukce od

všech dynamických sdílených objektů v trasovacích datech.

2.3 Flame Graphs

[TODO: https://github.com/brendangregg/FlameGraph a https://www.brendangregg.com/flamegraphs.html]

Flame Graphs jsou jedním z výtvorů Brendana Gregga, známé osobnosti ve vizualizaci trasovacích dat, momentálně zaměstnaným ve společnosti Intel. Flame Graphs jsou grafy, které zvýrazňují hierarchická data, primárně vytvořené pro zobrazení zásobníku systému. Nejníže jsou nejširší, tedy nejvíce časté, prvky z dat, které jsou abecedně seřazeny. Ve spodní vrstvě najdeme bloky odpovídající funkcím nejčastěji se vyskytujícím v nasbíraných datech. Čím širší blok, tím častěji se tato funkce na zásobníku objevuje. Nad každým blokem jsou pak „potomci" (funkce volané z funkce v daném bloku). Tímto způsobem graf zobrazuje hierarchii volání.

Své jméno dostaly podle jejich tvaru a původnímu výběru barev, což byly hlavně teplé barvy. FlameGraph může vypadat například jako na obrázku 2.2 (příklad ze stránek Brendana Gregga). Tyto grafy se dají použít kromě zkoumání času na CPU i na průzkum času stráveném mimo CPU (např. když je proces blokován čekáním na data, nebo při preemptivním přepnutí), analýzu alokací paměti nebo celkem nově na vizualizaci profilování na GPU či AI akcelerátoru společně s celým zásobníkem, tzv. AI FlameGraphs. AI FlameGraphs mohou analyzovat výkonostní nedostatky běžící umělé inteligence a efektivně tak určit místa, kde by měly existovat optimalizace. Tím se může výrazně zlepšit spotřeba energie, pro kterou jsou umělé inteligence nechvalně známy.

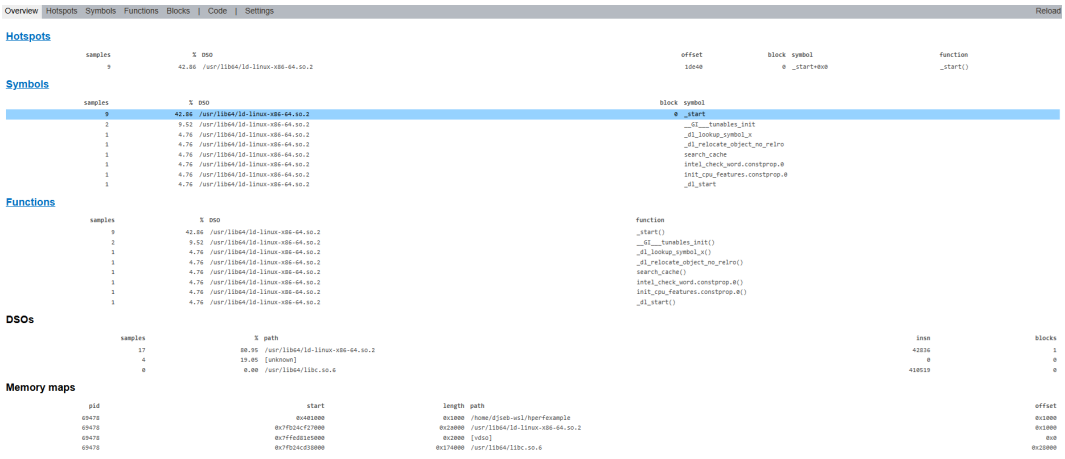
Tato vizualizace je velmi rozšířena a existuje mnoho implementací. Některé mění barvy prvků tak, aby se ukázala příslušnost v kódu, některé vytváří „rampouchové grafy", tj. FlameGraph otočený vzhůru nohama, s chladnými barvami. Tento typ vizualizace se pak hodí, pokud jsou zásobníky dlouhé a vrchní oblast by byla příliš řídká. Zajímavou změnou jsou „sunburst grafy", kdy se FlameGraph transformuje do koláčového grafu, jak ukazuje obrázek 2.3, získaný z GitHubu projektu s tímto cílem.

[TODO: Brát obrázky z internetu není úplně OK]

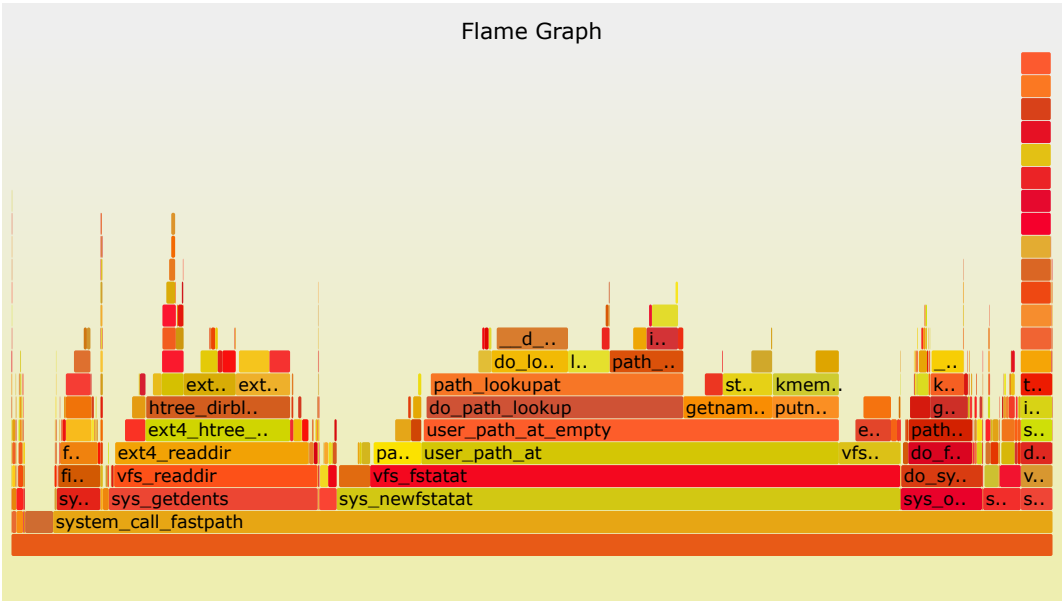
Brendan Gregg nabízí i Perl program na vytváření FlameGraphů z dat Perfu nebo DTrace (trasovací nástroj od Sun Microsystems, původně pro operační systém Solaris, dnes podporuje více operačních systémů, včetně Linuxu a Windows), ale i mnoha dalších profilovacích nástrojů, pokud jsou schopny zásobník zachytit. Program extrahuje pouze data zajímavá pro něj a poté vytvoří vizualizaci. Pro tento nástroj existuje i několik možností chování, jako například vytvoření rampouchových grafů, či změna různých vlastností v SVG, jako například velikost písma, název grafu, použité barvy, nebo i přidání poznámek. Program je open-source a dostupný na GitHubu Brendana Gregga.

2.4 TraceShark

[TODO: https://github.com/cunctator/traceshark]



Obrázek 2.1 Ukázka GUI HPerf



Obrázek 2.2 FlameGraph pro čas na CPU procesu programu Tar

Prvním „žraločím“ nástrojem který si představíme je je TraceShark. Tento vizualizátor se zaměřuje na vizualizaci událostí plánovače úloh v kernelu, specificky na události přepínání kontextu, probuzení procesu, vytváření a zavírání procesů, frekvenci CPU a neaktivitu CPU. Tyto události musí být sesbírány buď přes Perf nebo Ftrace (či frontendy pro Ftrace, jako Trace-cmd). Nástroj je open-source a stále ve fázi vývoje, seznam podporovaných událostí tedy může dále růst. Ukázku GUI lze vidět na obrázku 2.4, který je veřejně dostupný na GitHub repozitáři projektu. Mezi KernelSharkem a TraceSharkem neexistuje přímá spojitost, nicméně sdílí podobné rozhraní, některé cíle a podobné jméno.

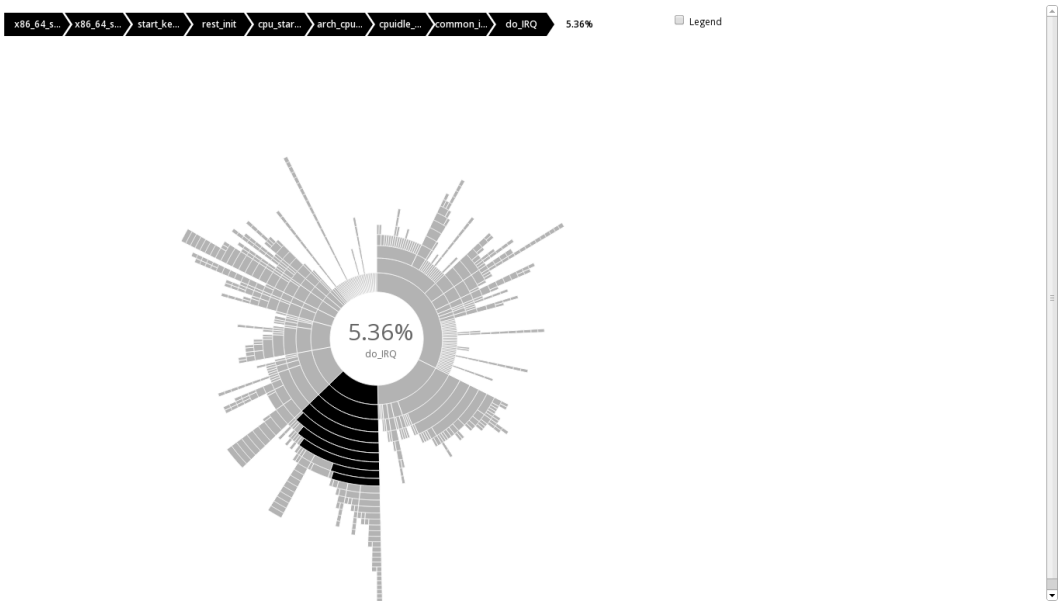
[TODO: Brát obrázky z internetu není úplně OK]

TraceShark v první grafové sekci ukazuje aktivity na každém z detekovaných CPU a jejich frekvence. V další sekci na CPU grafech ukazuje pro každý procesor, které procesy na něm běžely a to pomocí různobarevných obdélníčků. Zde se zobrazují i latence mezi označením procesu za připravený ke spuštění a opravdovým přepnutím procesoru na kontext tohoto procesu. Každá z použitých barev vyjadřuje jednu úlohu. Třetí sekce s CPU grafy pak ukazuje migrace úloh mezi procesory pomocí různobarevných šipek. Poslední sekce je vyhrazena pro grafy jednotlivých úloh, které si může uživatel sám přidat přes GUI. V grafech se dá pohybovat vpravo a vlevo rolováním kolečka myši, či, při zmáčknutí příslušného tlačítka, nahoru a dolů. Pod plochou pro grafy je i seznam událostí v trasovacích datech. Se seznamem lze interagovat, čímž se některé akce odrazí i v grafové ploše.

TraceShark dokáže pro události zobrazit jejich zásobník, filtrovat zobrazené informace podle identifikátoru vybrané události, nebo filtrovat podle PID procesu vlastnícího vybranou událost a nebo filtrovat podle CPU vybrané události. Program dokáže vytvářet snímky grafů, exportovat data ve formě vhodné pro vytváření FlameGraph vizualizací a lze i konfigurovat, které grafové sekce budou zobrazeny.

2.5 KernelShark

Tento nástroj, druhý ze „žraločích“ nástrojů, zde rozebírat nebudeme, má vlastní kapitolu 3. Nicméně jeho zmínka zde je nutná, jelikož je to další z více známých vizualizátorů, hlavně jako primární frontend pro Trace-cmd.



Obrázek 2.3 Ukázka sunburst grafu v akci

Obrázek 2.4 Ukázka GUI TraceSharku

3 KernelShark

Tato kapitola pojednává o programu KernelShark, což je GUI program pro vizualizaci a analýzu trasovacích dat z programu Trace-cmd. Kapitola nejprve krátce představí jeho autory, přispěvatele a účel. Dále se bude zabývat tím, jak KernelShark používat a jeho pluginy. Nakonec kapitola načrtne architekturu KernelSharku v modulech.

3.1 O autorech

KernelShark začal být vyvíjen v roce 2009 Stevenem Rostedtem jako GUI frontend pro data z Trace-cmd. Steven Rostedt je velkým přispěvatelem do Linuxového jádra, hlavně co se týče trasování, a udržuje repozitář pro Trace-cmd, jímž je také autorem. Dnes je zaměstnancem Googlu[1], dříve ale pracoval ve známých společnostech jako Red Hat nebo VMWare. V článku [**LWM-Kshark**] pak na této verzi KernelSharku ukázal analýzu plánovače úloh v reálném čase a zároveň představil funkcionality svého programu, například filtrování událostí podle procesu či plovoucí okénko s informacemi o vybrané události. KernelShark se nadále vyvíjel a k projektu se v roce 2017 (dle dat commitů v repozitáři) oficiálně přidal Yordan Karadzhov. Vystudovaný částicový fyzik, Yordan Karadzhov se později zaměřil na vývoj software a dnes pracuje u firmy Bosch jako softwarový inženýr[2] a je dnes nejvýraznějším přispěvatelem do KernelSharku.

Projekt je open-source a přispěvatelé pak žádají vlastníky repozitáře o publikaci svých commitů. Ke dni psaní je nejnovější dostupnou verzí `2.4.0`.

3.2 Instalace

Program nabízí někteří existující manažeři software na různých distribucích, například Zypper na openSUSE nebo Apt na Ubuntu. K dispozici je také balíček pro vývoj. Program si lze i stáhnout jako tarball a instalovat jej ručně. Poslední možností je klon gitového repozitáře, který si sami sestavíme. Manažeři závislosti a instalaci vyřeší za nás, proto zbytek této sekce se bude zajímat hlavně o ruční sestavení programu.

Předpoklady k instalaci

K sestavení je nutné mít na systém překladač pro jazyky C a C++ a sestavovací systémy Make a CMake. KernelShark je grafický program a k tomu využívá framework Qt, specificky Qt6. Dalšími grafickými nutnostmi jsou vývojové soubory pro FreeGLUT3 a vývojové soubory pro knihovny libXmu a libXi. KernelShark využívá pro některé své funkcionality soubory ve formátu JSON a je zapotřebí mít k dispozici knihovnu `libjson-c`. Potřebný je i textový font FreeSans, který může být nalezen v různých balíčcích dle distribuce (`fonts-freefont-ttf` na Ubuntu a `gnu-free-sans-fonts` na Fedoře). Jako frontend pro Trace-cmd nakonec vyžaduje

[1] **SR-LinkedIn** [**SR-LinkedIn**]
[2] **YK-LinkedIn** [**YK-LinkedIn**]

přítomnost tohoto programu a knihoven `libtracefs` a `libtraceevent`. Zbylými nutnými závislostmi jsou Flex a Bison. Volitelnými závislostmi jsou pak Doxygen a Graphviz, které slouží ke generování dokumentace.

Abychom byli schopni využít KernelShark naplno, je také nutné pracovat v prostředí, kde fungují Polkity. Těmi může KernelShark požádat o dočasný superuživatelský přístup pro spuštění Trace-cmd. Autorovi práce program fungoval perfektně v desktopovém prostředí KDE Plasma, nicméně v prostředí Qtile či při práci v Linuxovém subsystému ve Windows byly Polkity pouze napůl funkční, ačkoliv zbytek GUI fungoval bez problému. Problém byl vyřešen automatickým přijetím žádosti o zvýšení práv v pravidlech Polkitů. Tento způsob ale není doporučen, jelikož představuje bezpečnostní riziko.

Sestavení

Když máme všechny závislosti a zdrojový kód k dipozici, můžeme konečně vybudovat KernelShark. K dispozici máme z typů *Debug*, *Release*, *RelWithDebInfo* a *Package*. Poslední slouží hlavně pro vydavatele balíčků, pokud chtějí KernelShark nějak přesněji přeložit pro daný systém. S výběrem typu sestavení a, volitelně, vytvořením dokumentace, nám pak stačí přsunout se do `build` adresáře. Z něj pak zavoláme CMake, Make a nakonec jako superuživatel spustíme instalační skript pro GUI. Můžeme také spustit instalační skript pro vytvoření vývojových souborů. KernelShark je odteď sestaven a nainstalován. Knihovny nalezneme v podadresáři `libs` instalačního adresáře a vytvořené spustitelné binární soubory v podadresáři `bin`.

Odstranění programu

Pokud si přejeme KernelShark ze systému odstranit, jsou k dispozici skripty `cmake_clean.sh` a `cmake_uninstall.sh` v `build` adresáři.

3.3 Použití

Program lze spustit přes příkazovou řádku. KernelSharku lze při spouštění dodat různé argumenty, například cestu ke specifickému souboru k otevření, nebo pluginy, které se mají načíst.

KernelShark dokáže vizualizovat data z několika souborů naráz, přičemž každý vytváří vlastní datový proud, tj. *stream*, ve kterém jsou data uchována. Pokud uživatel otevře jeden stream, může pak připojovat další streamy. KernelShark zobrazí data ve vlastní skupině grafů pro tento stream a zařadí je na časovou osu, kterou případně rozšíří. Streamy není možné odebírat, jedinou možností pro odebrání streamů je buď restart KernelSharku, či otevření jednoho nového souboru s trasovacími daty. Streamy nemusí být něčím spojené, je možné i otevřít dva streamy z odlišných strojů s různými uálostmi a jinými časovými stopami. Na obrázku 3.1 je čerstvě spuštěný KernelShark.

Hlavní okno

Po spuštění se zobrazí hlavní grafické okno. To je rozdělené na 3 části: *menu*, *grafová oblast* a *seznamová oblast*. Části jsou zvýrazněny na obrázku 3.2, kde jsme KernelSharku dali nějaká data k zobrazení.

Menu

V menu je několik tlačítek pro práci se soubory, pro filtrování, tlačítko pro otevření dokumentace a tlačítko s různými nástroji. Zde se například zapínají a vypínají pluginy, nebo mění barvy používané KernelSharkem v grafech.

Grafová oblast

Grafová oblast obsahuje graf trasování, kontrolní řádek a informační řádek. Graf trasování se pak skládá z grafů pro jednotlivá CPU (zobrazeny jsou události jen z daného CPU) či procesy (zobrazeny jsou jen události daného procesu). Tyto grafy zobrazují události uspořádané v čase, který je reprezentován horziontální osou. Samotné grafy jsou v grafu trasování uspořádány svisle a lze použít posuvník k navigaci. V každém z grafů jsou události vyznačeny krátkými barevnými svislými čarami. Takovou čáru označuje KernelShark za *bin*. Pro velké objemy trasovacích dat se do binů seskupují záznamy událostí. Dokud jsou biny „stejné věci“ za sebou, jsou mezi nimi i, o trochu nižší, obdélníky, které souvislou práci vyznačují - výjimkou je zde „idle proces“, kdy CPU nepracuje a tyto obélníky se nekreslí. Termínem „stejná věc“ myslíme buď stejný proces v CPU grafech, nebo stejné CPU v grafech procesů. Pro zobrazení užších časových úseků lze v grafu přibližovat a oddalovat.

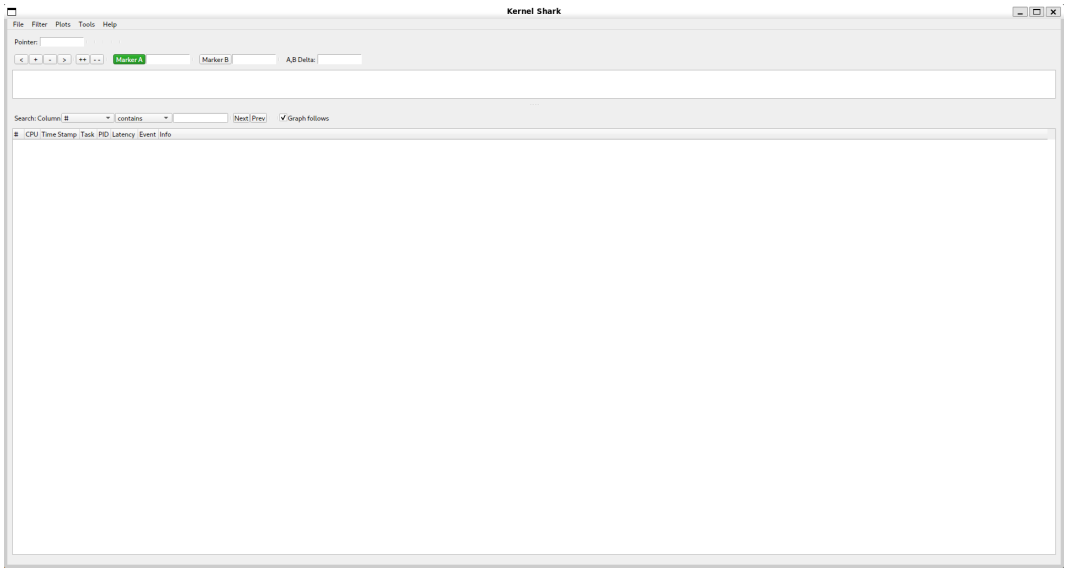
V kontrolním řádku se zobrazují ovládací prvky pro pohyb a přibližování v grafu a tlačítka s informacemi o (časových) pozicích dvou značkovacích čar, nazvanými „Marker A“ a „Marker B“. Pokud dvakrát klikneme na bin v grafu, můžeme tak jedním z markerů vyznačit danou časovou pozici a záznam, který se na ní udál. Tím se i zapíše pozice markeru. Vybranou pozici můžeme zrušit kliknutím pravého tlačítka myši s kurzorem nad tlačítkem daného markeru. Pro vybírání mezi markery A a B stačí s levým tlačítkem myši kliknout na tlačítko daného z markerů. Vedle tlačítek markerů je i informace o jejich časovém rozdílu. Pokud jsou markery aktivní, pak přibližování se bude snažit přiblížit oblast jimi vyznačenou (pokud je aktivní jen jeden, pak bude středem přiblížení bude událost, na kterou marker ukazuje).

V informačním řádku se zobrazuje (časová) pozice kurzoru myši a informace při najetí myši přes biny v grafu. Tyto informace jsou také v seznamu událostí.

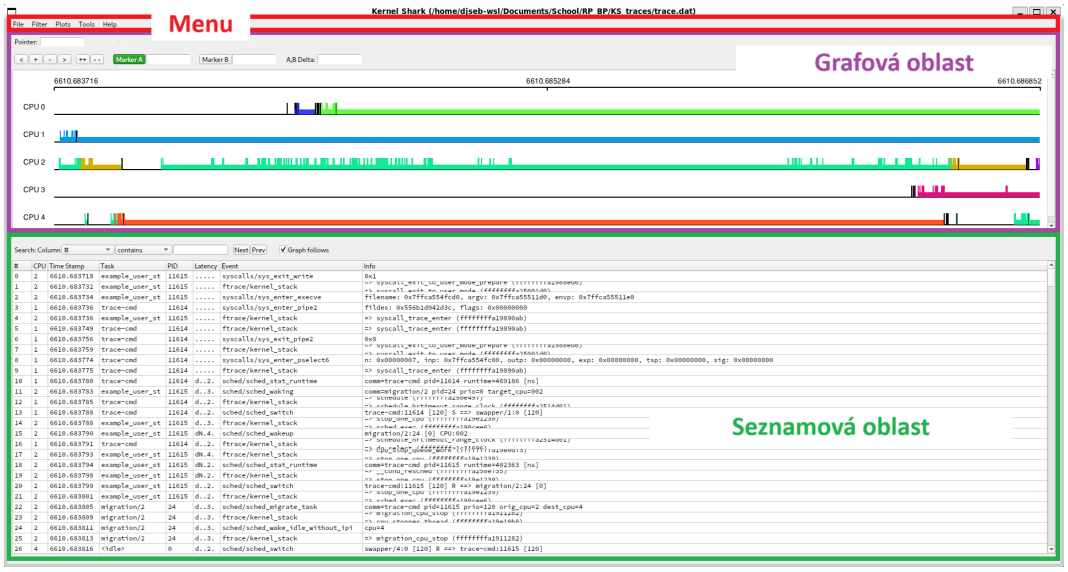
Seznamová oblast

Seznamová oblast zobrazuje trasovací data ve formě svislého seznamu s několika sloupci:

- Pozice události v datech
- Identifikátor CPU - číselný, používá označování jako operační systém.
- Časová značka události - v sekundách, s přesností na desetiny mikrosekund.



Obrázek 3.1 Čerstvě spuštěný KernelShark



Obrázek 3.2 KernelShark se zvýrazněným dělením hlavního okna

- Název procesu
- PID procesu
- Latence - jedná se o čtyři datová pole:
  - Zdali byla vypnuta přerušení, označeno písmenem „d“, jinak znakem „.“.
  - Zdali je potřeba přeplánování úloh, označeno písmenem „N“, jinak znakem „.“.
  - Událost při vyřizování přerušení, písmeno „h“ značí přerušení od hardwaru, písmeno „s“ značí přerušení vyvolané kernelem či pokud je tento typ přerušení vypnutý, písmeno „H“ značí přerušení od hardwaru, kde jsou přerušení od kernelu vypnuta, nebo se hardwarové přerušení událo při obsluze vyrušení od kernelu - jiné situace jsou značeny znakem „.“.
  - Čítač preempce - pokud je jiný než 0, tak kernel nepřepíná běžící úlohy, i kdyby nějaké událost o přeplánování zažádala. Pokud je čítač nulový, zobrazí se znak „.“.
- Název události
- Dodatečné informace - data zaznamenána v dané události.

V seznamu lze vyhledávat dle informací ve sloupcích, uživatel k tomu má dispozici textový vyhledávací řádek. V seznamu lze vyhledávat dopředu, či pozpátku. KernelShark podporuje jednoduchá vyhledávání typu „obsahuje“ (ve sloupci je někde vyhledávaný text), „přesná shoda“ (ve sloupci je pouze vyhledávaný text), „neobsahuje“ (ve sloupci není vyhledávaný text). Úspěšné vyhledávání v seznamu vyznačí událost splňující vyhledávací kritéria. Pokud je zaškrtnuté políčko „Graph follows“, bude tato událost vyznačena i v grafu.

Filtrování

Kvůli objemu dat získaných při trasování je často užitečné mít možnost ve vizualizaci filtrovat data, se kterými chceme při analýze pracovat. KernelShark proto dodává filtrovací nástroje - jednoduché filtry a pokročilé filtry. Jednoduché filtry filtrují podle typu události, zatímco pokročilé filtry dovolují vytvářet pokročilé filtrovací konstrukce a lze s nimi filtrovat i události dle jejich obsahu. KernelShark také dovoluje filtrovat události podle CPU nebo procesů. Pokud událost neprojde filtrem, pak není zobrazena v grafu ani v seznamu. Pokud na ni ukazoval nějaký marker, pak je vyznačen čárkovanou čarou. Jediné, co zůstane nepozměněno, jsou obdélníky mezi záznamy, které signalizují práci na CPU (graf CPU) nebo práci procesu (graf procesu).

K filtrům se lze dostat přes tlačítko `Filters` v menu. Všechny typy filtrování zobrazí uživateli dialog, přičemž filtrování podle typu události, CPU, nebo podle procesu zobrazí seznamy, kde se jednotlivé prvky k filtrování dají zaškrtnout a odškrtnout (tj. při filtrování podle CPU se zobrazí seznam CPU, při filtrování pole typu události se zobrazí seznam událostí). Speciiálně filtrování podle typu události ještě seskupuje události pod subsystémy, kterým patří, čímž je vytvořen dvouúrovňový seznam.

Pokročilé filtrování má dialog komplikovanější, přítomen je krátký pomocný text s příkladem použití, seznam aktivních pokročilých filtrů, soubor z něhož KernelShark čerpá data, pak tři řádky na sestavení filtru a nakonec filtr v textové podobě, kterou může uživatel upravovat. Zmíněné tři řádky na sestavení představují zjednodušení konstrukce filtru. První dovoluje vybrat událost z událostí přítomných v souboru s daty. Na druhé řáce může uživatel vybrat z „operátorů“, nicméně lze si vybrat i čárky, závorky, speciální znaky pro regex a i skutečné operátory, které mohou být relační nebo boolovské. Třetí řádek zobrazuje datová pole z formátu události, která lze použít k filtrování, například pole `comm_pid`, nebo speciálně pro událost sched_switch pole `prev_state`. Jednoduchý příklad pokročlého filtrování je na obrázku 3.3.

Relace

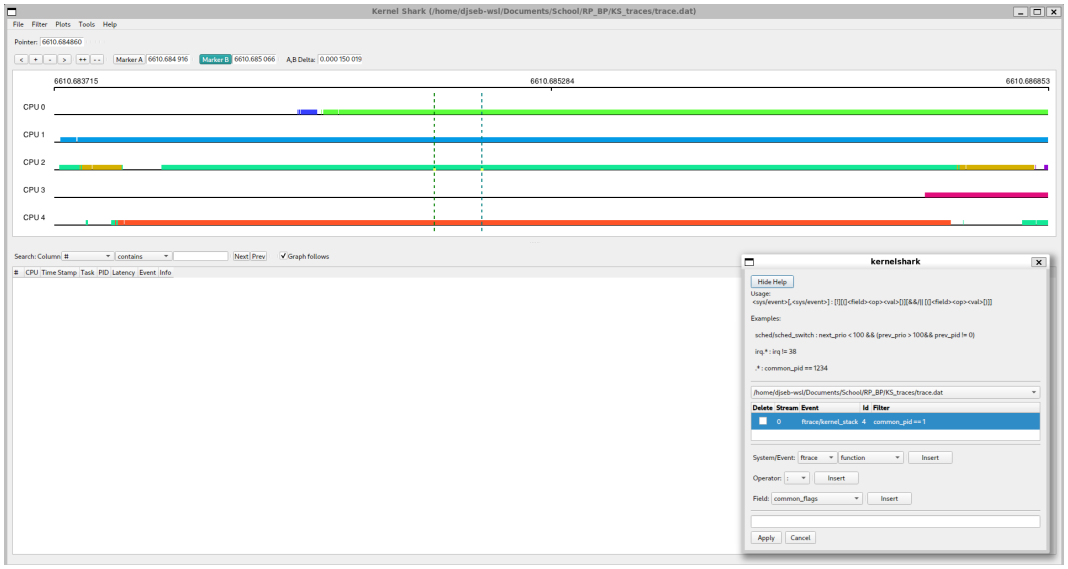
Při analýze si uživatel často vytvoří nějaký kontext k práci, například si vybere zajímavou oblast dat, vyznačí si nějakou událost Markerem B, načte si pluginy, nebo nějaké z nich povypíná a možná pracuje s více streamy naráz. Analýza také nemusí být záležitostí pár minut a uživatel může KernelShark vypnout během ní. Součástí KernelSharku jsou tedy i uživatelské relace, anglicky „sessions“, do kterých se právě tato data ukládají. Relace se uloží do souboru a uživatel může začít od bodu posledního uložení relace. Ukládání je prováděno ručně, nic se neukládá automaticky. Samotné soubory jsou ve formátu JSON a je možné je manuálně upravovat. Speciálně u pluginů KernelShark kontroluje jejich verzi a plugin o novější či starší verzi, než jaká je uložena, nenačte.

Trace-cmd GUI

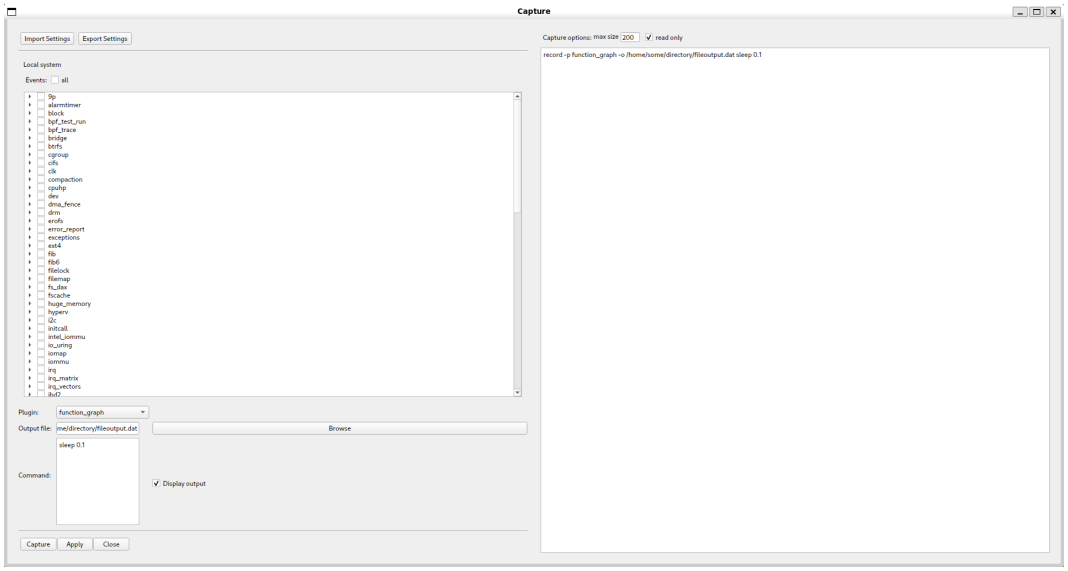
KernelShark dovoluje uživateli sbírat data od Trace-cmd skrze GUI. Uživatel musí pouze kliknout na tlačítko `Record` v menu `Tools`. Poté program zažádá o superuživatelská oprávnění přes nějaký aktivní Polkit. Při úspěšné elevaci oprávnění KernelShark zobrazí nové okénko (viz obrázek „Record okénka“ 3.4). V něm je napravo plocha pro kontrolu běžícího Trace-cmd a nalevo jsou konfigurační možnosti trasování. V nich si uživatel může vybrat, které události chce zaznamenávat, jaký tracer má Trace-cmd použít, cestu k výstupnímu souboru, zdali si uživatel přeje vidět výstup a nakonec textové pole s příkazem, který spustí trasovaný proces (ve výchozím nastavení je zde pouze `sleep 0.1`). Do tohoto pole lze přiat i další argumenty pro spuštění Trace-cmd. Nakonec jsou dole v konfigurační ploše tři tlačítka. Jedním se toto okno zavře, druhým se aplikuje konfgurace a třetím se trasování spustí. Uživatel tak teoreticky s Trace-cmd vůbec nemusí pracovat v terminálu a může používat pouze KernelShark.

3.4 Pluginy

KernelShark obsahuje rozhraní k vytváření pluginů, kterými pak může uživatel rozšířit chování programu, nejčastěji skrze kreslení do grafu či úpravu dat a nebo sběr statistik. KernelShark dovoluje vývojářům pluginů definovat menu pro pluginy, jejich chování při načítání trasovacích dat a kreslení aktivního pluginu. Uživatel může i nemusí využít všechny tyto možnosti. KernelShark také obsahuje některé



Obrázek 3.3 Aktivní pokročilý filtr - žádná událost nevyhovuje podmínkám



Obrázek 3.4 KernelShark GUI pro práci Trace-cmd

pluginy ve svojí základní instalaci, které označíme v této práci jako *oficální pluginy*. Mezi tyto pluginy patří například plugin sched_events, který kreslí červené a zelené rámečky do grafů procesů. Oficiální pluginy také představují inspiraci pro vývojáře dalších pluginů.

Obecně pluginy definují kontext, tj. jejich vlastní „globální úložiště“, jejich inicializační a deinicializační funkce, a handlery právě pro kreslení, či úpravu dat při načítání a nebo pro vytvoření menu. Vše ostatní je pak na vývojáři pluginu. KernelShark obsahuje speciální sestavovací instrukce pro pluginy, přičemž je možné dát zdrojové soubory vlastních pluginů do adresáře se zdrojovými soubory oficálních pluginů a upravit tyto instrukce. Pokud si ale vytvoříme sestavovací instrukce sami, KernelShark stále plugin přijme.

3.5 Architektura programu

Následující sekce bude autorův pokus o zachycení architektury KernelSharku. Architektura nemá vlastní dokumentaci, tato sekce tak nemůže zaručit, že architektura nastíněná zde je perfektním odrazem architektury ve vizi autorů KernelSharku. Nicméně se nám tento odhad bude hodit pro orientaci v kódu programu v této práci.

Obecně lze říci, že KernelShark je grafická aplikace, která používá klasický přístup zachytávání událostí uživatelské interakce ve smyčce a jejich následné zpracování. Má nějaký hlavní systém a k němu jsou definována místa pro práce pluginů.

Detailnější pohled na architekturu zachytíme pomocí modulů. Nebdeme se zaměřovat na konkrétní implementace.

- *Zpracovávání dat* - zde se KernelShark zaměřuje hlavně na čtení dat a jejich abstrakce, které KernelShark používá dále. Velkým zaměřením je transformace dat z Trace-cmd do lépe vizualizovatelných dat. Právě zde nejvíce žijí streamy a záznamy KernelSharku. Součástí jsou i dotazy na data v záznamech či ve streamech.

- *Datové modely* - KernelShark pracuje s několika datovými modely podle potřeby. Základním modelem jsou prvky uspořádané dle času do binů, tj. sdružení záznamů. S tímto modelem KernelShark pracuje nejvíce a dodává k němu i API na manipulaci a dotazování, například na index (do pole všech záznamů) prvního záznamu v daném binu. Existují ale i modely vhodnější pro seznamové zobrazení událostí a model pro filtrování takových seznamů.

- *Trace-cmd GUI* - modul starající se o vše týkající se okna pro práci s Trace-cmd skrze KernelShark.

- *Hlavní okno* - zde se seskupují hlavní grafické prvky, tj. graf trasování, seznam událostí a menu programu. Toto okno také představuje hlavní navigovatelný prvek v kódu, přes který se lze dostat k dalším (veřejným) prvkům, třeba právě ke grafu trasování a jeho prvkům.

- *Kreslení v grafu* - aby bylo možné kreslit do grafu, dodává KernelShark několik definic tvarů, jejich kreslení a interakce s myší (pouze dvojité kliknutí)

v tomto modulu. Tyto tvary jsou pak nejvíce využívány pluginy při vytváření tvarů vlastních.

- *Filtry* - modul se stará o vše ohledně filtrování, ať už filtrování samotné, vyhledávání správných událostí, nebo okna, se kterými uživatel pracuje při filtrování.

- *Vyhledávání* - modul se stará o vyhledávání v seznamu událostí, ale i vyhledávání uskutečněná nějakým kódem.

- *Relace* - zde najdeme „ukládací“ a „načítací“ funkce pro všechno, co KernelShark do relací ukládá.

- *Pluginy* - tento modul se stará hlavně o načítání, zapínání, inicializaci a působení pluginů (tj. kde se volají jejich handlery). Dodává i některé definice a deklarace, které mohou pluginy využít.

- *Pomocné nástroje* - modul s různými funkcemi a strukturami, kterými si KernelShark pomáhá. Patří sem definice některých vyskakovacích oken, funkce na získání užitečných dat, například všech CPU ve streamu, nebo operátory pro práci s barvami.

4 Obecná analýza a stanovení požadavků

V této kapitole se podíváme na možná vylepšení schopností KernelSharku. Vytvoříme pro ně požadavky a sepíšeme jejich cíle a nutné vlastnosti. Dále obecně analyzujeme, jak se dají vylepšení vytvořit. Hlubší, technická analýza každého z vylepšení pak bude součástí detailnějších pohledů na návrh a implementace vylepšení v jejich samostatných kapitolách.

4.1 Pluginy a modifikace

První možnou cestou rozšíření KernelSharku je přidání pluginů, které upraví zobrazované informace, nebo nějaké přidají. KernelShark již obsahuje oficiální pluginy a je možné se jejich strukturou inspirovat. Pluginy dokážou vykreslovat uživatelem definované tvary (s nimiž lze interagovat) do grafu, pozměňovat data záznamů událostí a přidávat nová menu tlačítka do hlavního okna. Pluginy tak zřejmě nejsou schopné všeho. Například nelze skrze pluginy vytvářet další záznamy, které by se pak zobrazily v grafu.

Abychom mohli docílit všech vylepšení, tak vylepšení nemohou být jen pluginy, ale i modifikace kódu KernelSharku. Přímo u zdroje je pak možné manipulovat s celým programem, nicméně na oplátku bude potřeba nerozbít to, co již funguje, zajistit podobný styl s předchozím kódem a vyznačovat místa změn, aby byla respektována licence. Obecně lze od modifikací požadovat následující:

- *Minimální vliv*, tj. vypnutá modifikace musí mít buď žádný, nebo minimální vliv na chod KernelSharku a jeho oficiálních pluginů. Pokud takový vliv má, musí být navržena tak, že se lze dostat ke starému chování.
- *Stylová podobnost*, tj. kód modifikací by měl být podobný ostatnímu kódu v KernelSharku pro zachování jednotného stylu.
- *Chovat se jako rozšíření*, tj. kód modifikací by se také měl snažit měnit existující kód co nejméně, chovat se jako rozšíření co možná nejvíce. Změny existujícího kódu musí být označeny a pokud nejsou triviální, popsány.

Od pluginů se bude obecně očekávat tento seznam:

- *Vlastní adresář*, tj. pluginy budou mít vždy vlastní adresář s vlastními intrukcemi pro sestavení, kódem a dokumentací, vše mimo adresář s KernelSharkem, jeho modifikacemi a oficiálními pluginy. Struktura repozitáře tímto požadavkem prospěje, pluginy budou lépe navigovatelné a kontrola nad sestavením a dokumentací pevnější. Samozřejmě je pak nutné napsat vlastní CMake instrukce k sestavení, ale to je přijatelná práce navíc.
- *Samostatnost*, tj. pluginy by se měly snažit být co nejsamostatnější, tj. nebýt závislé na ostatních pluginech, ať už vztahem „plugin A potřebuje k fungování plugin B“ nebo vztahem „plugin A zakazuje plugin B“. Zejména druhý ze vztahů by mohl být nepříjemný, jelikož by mohl snižovat efektivitu

analýzy, některé pluginy by prostě nemohly být aktivní. První ze vztahů je mírnější, existuje hlavně pro snížení potenciálního chaosu závislostí (tzv. „dependency hell“). Žádný z pluginů v této práci nebude právě z tohoto důvodu ani cílit na to být využitelný jinými pluginy jako knihovna.

4.1.1 Označení modifikací v kódu

Licence KernelSharku, LGPL-2.1, vynucuje u provedených změn v softwaru s touto licencí jasné vyznačení změněných míst společně s datem změny. Každá změna spojena s modifikací tak bude ohraničena dvojicí komentářů:

```
//NOTE: Changed here. ([TAG]) ([DATE])
...
// END of change
```

[TAG] je zkratkovité označení po typ modifikace, se kterou změna souvisí - každá modifikace musí mít takovou značku. [DATE] je datum napsání změny ve formátu YYYY-MM-DD. Pokud změna vznikne během vývoje nějaké modifikace, ale není na ni nutně vázána (například je to pomocná funkce s širším využitím), pak stačí ohraničit [TAG] uvozovkami.

4.2 Výběr vylepšení

Následující vylepšení umožňují v KernelSharku zobrazovat dodatečné informace, které usnadňují analýzu trasovacích událostí. Součástí jejich popisu budou i názvy těchto vylepšení, které se použijí v dalších kapitolách. Názvy budou anglické pro zachování jednotného jazyka programu. Vylepšení jsou buď pluginy nebo modifikace zdrojového kódu KernelSharku - tato informace bude také součástí jejich popisu.

Každé vylepšení, které není dodatečné, pak shrne do bodového seznamu ve vlastní kapitole hlavní myšlenky z textu své podsekce.

4.2.1 Lepší analýza zásobníku kernelu

Identifikace problémů k vyřešení

Zaznamenávací funkcionalita KernelSharku, tzv. Record okénko, dovoluje spustit program `trace-cmd record` pro nastartování trasování běžícího systému skrze GUI. Ačkoliv to není ihned zřejmé, tato funkcionalita dovoluje uživateli dodat argumenty Trace-cmd, kterými upraví chování zaznamenávání. Pokud uživatel nezná tyto argumenty, nebude je schopen využít. Jedním z argumentů je `-T`, který zapne zaznamenávání kernel zásobníku do události typu ftrace/kernel_stack. Zásobník je zaznamenán po každé události, kromě události zaznamenání zásobníku.

KernelShark již umí zobrazit události typu ftrace/kernel_stack. Bere je jako každou jinou událost v trasovacích datech a tak je zobrazí jak v grafu, tak v seznamu událostí. Nicméně nás většinou nezajímá, že se nějaký zásobník trasoval, nýbrž spíš co v něm během události bylo. Z grafu toto nevyčteme, informační řádek nám nedokáže dát celou informaci, jelikož na ní nemá prostor. Seznam událostí je na tom trochu lépe, ale informace zásobníku je v textové formě víceřádková a často

má řádků tolik, že většina prostoru seznamu je zabrána jen touto událostí. Celý zásobník je ovšem viditelný pouze, pokud je daná událost v seznamu vybrána, jinak se zobrazí pouze jedna řádka ze zásobníku. To nám analýzu zpomalí o vybrání zásobníkové události.

Extrakce požadavků

Zřejmě je zde co vylepšovat. Bylo by jistě příjemnější, kdyby bylo možné zobrazit obsah zásobníku přes záznam této události v grafu, nebo vidět nějakou podstatnou část v informačním řádku, namísto klikání mezi seznamem a grafem. Toto by nemělo být realizováno přes klikání na záznam samotný, jelikož takto uživatel záznamy zvýrazňuje. V tom případě by bylo vhodnější vytvořit tlačítko nad záznamy, jejichž zásobník si chceme zobrazit. To mohou být buď přímo záznamy se zásobníkem, nebo záznamy událostí po kterých byl zásobník zaznamenán. Abychom měli záznamy kde zobrazit, bylo by nejlepší vytvořit nějaké vyskakovací okénko. V tomto okénku by mohla být vedle zásobníku i další data, například po jakém typu události byl zásobník zaznamenán, nebo jakému procesu událost patřila. Přejetím kurzoru myši přes tlačítko bychom mohli donutit informační řádek k zobrazení nějaké zajímavé části zásobníku. Abychom toho byli schopni, bude nutno KernelShark dodatečně vylepšit o akce vyvolané při přejetí kurzorem myši přes objekty v graf. Také by bylo vhodné umět toto chování nějak konfigurovat, třeba nezobrazovat tlačítka nad nějakým typem události, nebo měnit barvu tlačítek. K tomu by mohlo stačit konfigurační okénko vyvolané nějakým tlačítkem v hlavním okně. Nakonec vybereme podporované události, tj. události, pro které plugin musí fungovat. Těmi budou sched_switch a sched_waking. Vybrány jsou proto, že budou podstatné i u dalších vylepšení. Tím se dá testovat izolace různých vylepšení, případně i jejich kompatibilita. Zároveň jsou často zajímavé k analýze. Sched_switch nám ukáže přepnutí z jednoho procesu na jiný. Zde lze zkoumat, proč byl proces přepnut, například zdali na něco začal čekat. Sched_waking nám pak ukazuje změnu spícího procesu na proces připravený k běhu. Tato událost a její záznam zásobníku mlže odhalit proč lze proces opět nechat běžet, například na co se přestalo čekat.

Co se zaznamenávání zásobníku týče, ačkoliv je možné vytvořit soubor s událostmi záznamů zásobníku, bylo by uživatelsky přívětivější, kdyby se dala tato možnost zapínat pomocí nějakého GUI prvku. Nejpřirozenějším výběrem by bylo zaškrtávací políčko, umístěné v Record okénku KernelSharku. Obsah okénka nemá se zbytkem programu další vazby, tedy není třeba řešit další kompatibilitu.

Vylepšení Record okénka je možné pouze jako modifikace, pluginy nemají k tomuto okénku přístup. Modifikaci budeme nazývat stručně jako *Record Kstack*. Ovšem tlačítka nad záznamy a vyskakovací okénka je možné definovat čistě v pluginu. Jelikož bude plugin zaměřen na analýzu zásobníku, bude se nazývat *Stacklook*.

4.2.2 Dělení vlastnictví událostí souvisejících se dvěma procesy

Identifikace problémů k vyřešení

Trasování ukládá mnoho různých událostí, přičemž některé jsou svázané se dvěma procesy. Tyto procesy jsou pak touto událostí spárované. Klasickým příkladem je událost sched_switch, tedy přepnutí se z kontextu jednoho procesu na kontext procesu jiného na stejném CPU. Trace-cmd dokáže tuto informaci uchovat, nicméně přepnutí je vyvoláno jen přepínajícím procesem. Zaznamená se tak jen jedna událost a to pro tento proces. KernelShark dokáže zobrazit ve svých CPU grafech tyto události v čase, dá se tedy zjistit, který proces byl přepnut do kterého na daném CPU. Nicméně tato informace není tak snadno zjistitelná v grafech procesů. Graf druhého procesu o přepnutí neví. Jediné, co ví, je že jeho proces začal pracovat. Pouze graf procesu přepínajícího bude o události přepnutí vědět a mít ji jako svou součást.

Tento systém, ač většinou funkční, nutí procesy se dělit o informaci, která se týká obou. Informaci pak jeden z nich ztrácí ve svém procesovém grafu. To nutí i některé pluginy upravovat vlastníky událostí, aby dosáhli svých funkcionalit. To je ovšem problém - některé pluginy mohou vyžadovat jiná data, než která jim KernelShark po upravení události dokáže poskytnout. Tedy pluginy měnící vlastníky musejí být načteny buď někdy jindy, nebo jiné pluginy zakazovat. Příkladem mohou být oficiální plugin sched_events a další z vylepšení, *Naps*. Naps plugin potřebuje mít sched_waking události v grafu procesu, který je probouzen. Vedle toho sched_events plugin potřebuje mít sched_switch události v grafech procesů, na které se přepíná. Dále pak zobrazuje své tvary mezi záznamy událostí typu sched_switch a sched_switch, nebo mezi záznamy událostí sched_waking a sched_switch. Tyto dva pluginy si pak vzájemně narušují očekávaná umístění těchto dvou typů událostí a ani jeden z pluginů nefunguje správně.

Zřejmým řešením problému je pak odstranění nutnosti měnit vlastníky událostí. Například umět události rozdělit a každému z procesů dát jednu polovinu. Ale rozdělování implikuje, že by pak bylo nutné rozdělit i data z jedné události do vzniklých polovin. To ovšem není vhodné, jelikož oba procesy nejspíše budou potřebovat data celé události k efektivní analýze. Lepší bude trochu odlišný přístup. Namísto dělení události dovolíme druhému procesu z páru číst stejná data, ať už kopií události nebo odkazem na ní. Takto nerozdělíme událost samotnou, ale její vlastnictví.

Extrakce požadavků

Podívejme se, co to znamená pro KernelShark. Záznamy událostí v KernelSharku obalují události z Trace-cmd, nebo se na ně dokážou odkázat. Jelikož KernelShark používá hlavně své záznamy, bude lepší pracovat s nimi. Z architektury KernelSharku vychází, že můžeme buď přidat umělé záznamy, které se automaticky zobrazí v grafu, nebo v grafu jenom vykreslovat objekty simulující záznam. Druhý přístup není vhodný. Simulace chování by byla zbytečně složitá na implementaci a záznamy v grafu by neodpovídaly záznamům v seznamu. Naopak první přístup jenom vytvoří umělý záznam a KernelShark se postará o zbytek. O co víc, umělé záznamy už KernelShark sám vytvářet umí během vytváření

záznamů z trasovacích dat. Přístup je tedy nejen jednodušší na implementaci, ale už i používán. Vytvořené umělé záznamy by měly umět odkázat na záznam původní. Ten totiž obsahuje i původní událost a její data. Zároveň by umělé záznamy měly obsahovat data, díky kterým budou přiřazeny do správných grafů a bude možné je rozlišit od ostatních záznamů, například jménem.

S vylepšením budou muset oficiální pluginy nebo ostatní části KernelSharku umět spolupracovat. Nové záznamy by jako ostatní záznamy měly podporovat rozhraní dotazů na záznamy. KernelShark by s nimi měl také umět alespoň základně pracovat, alespoň je umět kromě zobrazení i jednoduše filtrovat. Jelikož budeme měnit chování KernelSharku, bylo by dobré umět tuto změnu chování zapínat a vypínat. Toto nastavení by mělo být uložitelné do relací. Jediný oficiální plugin, který je nutno poupravit je sched_events.

Výše vypsané změny ukazují, že vylepšení nelze napsat jako plugin. Pouze modifikace bude schopná přidávat další záznamy do grafu a seznamu. Modifikaci nazveme *Couplebreak*, podle jejího účelu rozdělit párovou událost mezi dva procesy.

Podporovanými událostmi budou sched_switch a sched_waking. Vybrány jsou proto, že budou podstatné i u dalších vylepšení. Tím se dá testovat izolace různých vylepšení, případně i jejich kompatibilita. Zároveň jsou to události, které pro Naps a sched_events představují bez Couplebreaku problém a dá se na nich ukázat využitelnost vylepšení.

4.2.3 Vizualizace nečinnosti procesů

Identifikace problémů k vyřešení

Procesy se během běhu na CPU často střídají. Důvodů může být několik - proces třeba čeká na otevření souboru, nebo mu prostě vypršel čas na CPU a byl preemptivně přepnut. Vizualizace nečinnosti v KernelSharku chybí, ačkoliv data jsou dostupná v trasovacích událostech. Pokud uživatel potřebuje zjistit předchozí stav procesu (stav před přepnutím), musí si jej najít a vyčíst v seznamu procesů. Pokud uživatel chce zjstit, jak dlouho byl proces nečinný, než byl probuzen, nezbývá mu než manuálně najít událost přepnutí a následnou událost probuzení - tato událost ovšem patří nějakému jinému procesu.

Extrakce požadavků

Cílem vylepšení je vizualizovat pauzu mezi přepnutím procesu (sched_switch) a probouzením jiným procesem (sched_waking procesu, který první proces probouzí). Lze se inspirovat pluginem sched_events, který kreslí obdélníky mezi událostmi pro něj zajímavými. Nicméně samotné obdélníky by nedokázaly nést informaci o předchozím stavu procesu, tedy nějaké barevné či textové označení bude také požadováno. Bez vylepšení Couplebreak bude nutné přesunout záznamy pro sched_waking do grafu probouzeného procesu. S vylepšením bude stačit hledat záznamy cílových událostí probouzení. Neměli bychom se snažit kreslit obdélníky neustále. Velké množství by mohlo program zpomalovat. Mnoho záznamů na obrazovce také znamená, že budou v grafu blízko u sebe a obdělníky by pak pro sebe neměly prostor. Jelikož všechny systémy nejsou stejné, toto by mělo být konfigurovatelné v nějakém grafickém okénku. Všechny tyto změny se dají provést v pluginu.

Jelikož se bude zajímat o dobu, kdy jsou procesy nečinné, nazveme plugin *Naps*, kdy naps (česky zdřímnutí) označují právě dobu nečinnosti procesu.

4.2.4 Vizualizace NUMA topologie CPU vedle grafu

Identifikace problémů k vyřešení

KernelShark se primárně zabývá vizualizací trasovacích dat. Ovšem jenom z nich nelze vyčíst vše. Máme-li mnoho procesorů, je možné optimalizovat přístupy do paměti přes NUMA model. NUMA ve zkratce znamená, že některé procesory jsou blíže nějaké paměti či pamětem, což jejich přístupy zrychluje. NUMA tak vytváří nějakou topologii procesorů - CPU se seskupují do NUMA uzlů. NUMA uzly určují skupinu procesorů a pamětí, ke kterým mají tyto procesory blízko. Právě tu KernelShark nedokáže nijak zohlednit - procesory v grafu označuje pouze daty z trasování, pomocí indexů dodaných operačním systémem. Topologie ovšem ovlivňuje výkon aplikací s více komunikujícími procesy, nebo s procesy s pamětí na vzdáleném NUMA uzlu. Jak kernelový plánovač úloh, tak alokátor paměti se snaží NUMA lokalitu respektovat - zviditelněním v KernelSharku bude možné analyzovat, jak dobrá tato snaha je. Program *Hwloc* byl navržen právě pro zkoumání a vizualizaci topologií systémů využívajících NUMA technologii. Tento nástroj umí topologii systému i exportovat do souboru formátu XML a načíst data z takovýchto exportů. Co víc, Hwloc dokáže zachytit i zdali jsou některá CPU součástí jednoho jádra a jeví se jako samostatná CPU díky hyperthreadingu. Spojením schopností obou programů bychom mohli dodat KernelSharku možnost zobrazovat NUMA topologii procesorů systému a zároveň z grafu KernelSharku vyčíst, která část topologie byla více namáhána, než jiné.

Extrakce požadavků

KernelShark se zobrazováním topologií nijak nepočítá, není pro ně tedy žádná podpora. Topologie bude vytvořena pomocí programu Hwloc. Nutné tedy bude vytvořit/zabrat místo v hlavním okně, kde se bude vizualizace topologie zobrazovat. Toto místo by bylo dobré umět schovat, pokud bychom topologická data v daný moment nepotřebovali a vybrané místo by byl nevyužitý prostor. Topologii by bylo nejlepší zobrazit takovým způsobem, že CPU grafy by na zobrazení přirozeně navazovaly. Hwloc nečísluje CPU stejně jako operační systém, povolíme si tedy reorganizaci CPU grafů tak, aby byla respektována topologie. Části topologie by měly být rozlišitelné, alespoň nějakým popiskem, nebo barvou. Struktura topologie by měla být jednoduše pochopitelná, zobrazení ve stromovém stylu je zde přirozené, využívá jej Hwloc. Dále budeme muset dát uživateli nějaké okénko, ve kterém si vybere soubor s topologií, kterou chce zobrazit, a zdali topologii zobrazit chce. Soubor by vybíral pro každý otevřený stream zvlášť. KernelShark nijak nevynucuje stejný trasovaný systém v otevřených streamech, proto dává smysl toto respektovat a vybírat topologie pro každý stream odděleně. Vylepšení by také mělo být součástí ukládaných relací, už jen kvůli odstranění času stráveného hledáním a načítáním souboru topologie. Nakonec, pokud systém nevyužívá NUMA technologii, pak nemusíme zobrazovat NUMA uzly a zobrazíme pouze jádra. Ta jsou vytvořena v Hwlocu i kdyby byl hyperthreading vypnutý a jedno jádro by bylo ekvivalentní jednomu CPU.

Vylepšení bude jistě modifikací, pluginy nemají schopnosti drasticky měnit GUI. Modifikaci nazveme *NUMA Topology Views*, zkratkovitě *NUMA TV*.

4.2.5 Dodatečná vylepšení

Následující odstavce v rychlosti představí vylepšení KernelSharku, na která se práce nesoustředí, ovšem jejich existence KernelShark udělá trochu příjemnějším k použití a nebo jsou užitečná ve vícero jiných vylepšeních.

Aktualizace grafického kódu

KernelShark používá Qt6 v minimální verzi 6.3.0. Nicméně Qt6 verze 6.7.0 přináší nové rozhraní pro kontrolu statusu zaškrtávacího políčka a staré rozhraní označuje při kompilaci jako zastaralé. Tato verze bude již brzy součástí hlavních Linuxových distribucí a aktualizace těchto částí kódu v KernelSharku je na místě. Dodatečným vylepšením bude modifikace kódu KernelSharku, která bude nové rozhraní využívat. Název vylepšení bude samovysvětlující *Update Cbox States*.

Objekty v grafu reagují na najetí kurzoru myši

Grafová oblast KernelSharku dovoluje uživateli dodat další tvary ke kreslení a interakci. Takové tvary jsou velmi časté u pluginů, příkladem může být sched_events nebo Stacklook. Tyto tvary mají předdefinovaná rozhraní KernelSharkem, nicméně součástí těchto rozhraní není reakce na najetí myší přes grafický objekt. KernelShark již dokáže událost najetí myši zpracovat, využívá ji při najetí na záznamy. Jednou uživatelskou interakcí s tvary je dvojité kliknutí myši na objekt. Cílem tohoto vylepšení je dodat grafickým objektům pro grafovou oblast možnost reagovat na najetí myši a to podobným způsobem, jako je definováno dvojité kliknutí. Název tohoto vylepšení bude *Mouse Hover Plot Objects*.

Využití barevných tabulek KernelSharku

KernelShark nedovoluje využívat barvy, které používá pro procesy, CPU a streamy. Jejich využití by ale mohlo v některých případech vylepšit organizaci informací, například Stacklook může zabarvovat svá tlačítka barvou procesů, kterým události patří. Lze tak i v CPU grafu idenitifikovat vlastnící proces události, jejíž záznam zásobníku si budeme chtít zobrazit. Navíc, pokud využijeme barvy, které používá KernelShark, pak se využité barvy budou měnit společně s jinou hodnotou barveného slideru, který KernelShark uživateli zpřístupňuje. Vylepšení *Get Colors* dodá KernelSharku, ve formě modifikace, dodá funkce, kterými se získají barvy momentálně využívané KernelSharkem.

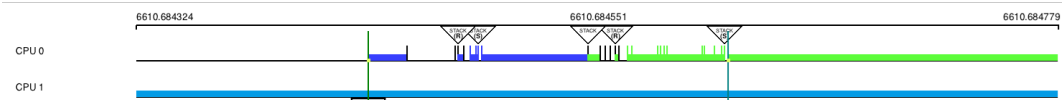
Možnost měnit text v informačním řádku

KernelShark nedovoluje měnit obsah informačního řádku. Možnost jej měnit se může hodit, pokud nějaký plugin, jako například Stacklook, by chtěl zobrazit nějaké rychlé informace uživateli. Dodatečným vylepšením bude modifikace kódu KernelSharku, která možnost měnit text v informačním řáku dodá. Vylepšení dostane název *Preview Labels Changable*.

Obdélníky mezi záznamy

KernelShark při vykreslování grafu dokresluje obdélníky mezi jednotlivé grafické reprezentace záznamů, které vyznačují aktivní proces na nějakém CPU. Tyto obdélníky jsou definovány a nakresleny během každého vykreslení na obrazovku. Ovšem ne všechny záznamy by se měly podílet na kreslení obdélníků, například záznamy zásobníku kernelu se dějí po přepnutí kontextu, kde by měl obdélník většinou skončit, a záznamy vytvořené Couplebreakem mohou chybně být začátky a konce jiných obdélníků. Vylepšení *NoBoxes* jako modifikace dodá masku viditelnosti záznamů, kterou se zakáže účast na kreslení zmíněných obdélníků, a plugin, který touto maskou označí vybrané záznamy událostí.

Příklad špatného zobrazování je na obrázku 4.1. Na tomto obrázku je ftrace/kernel_stack událost vyznačená velkou svislou čarou vpravo, couplebreak/sched_waking[target] je událost s velkou svislou čarou vlevo. Událost ftrace/kernel_stack vytváří velký obdélník až do konce grafu, ačkoliv se událo po přepnutí kontextu a procesor ve skutečnosti po zachycení dále nepracuje. Událost couplebreak/sched_waking[target], ačkoli neprovádí žádnou skutečnou práci na procesoru, vytváří dojem, že pracuje právě díky nakreslenému obdélníčku. V grafu CPU 1 je také velký obdélník, který začíná u události zachycení zásobníku kernelu a neměl by tedy být kreslen. Události zachycení zásobníku jsou časté a v grafu je více obdélníků spojených právě s nimi.



Obrázek 4.1 Ne všechny obdélníky mezi záznamy by se měly vykreslovat, některé tvoří iluzi opravdové práce.

5 Record Kstack

Tato kapitola se bude zabývat modifikací, která dodává uživateli přímější způsob, jak zapínat trasování zásobníku kernelu přes GUI KernelSharku. Modifikace je přímá a velmi jednoduchá, kapitola je tak krátká.

5.1 Cíl

Dát KernelSharku GUI prvek, kterým se zapne/vypne trasování zásobníku kernelu při trasování přes Record okénko.

5.2 Analýza

Jelikož chceme rozšířit GUI, dává smysl se porozhlédnout v kódu KernelSharku po GUI kódu pro Record okno. Jeho kód se nachází v souborech KsCaptureDialog.hpp/cpp - zde tedy budeme modifikovat.

5.2.1 Návrh

Modifikace bude nejspíše malá, proto nemá smysl vymýšlet složitý návrh. Nicméně si jako návrhové cíle vytyčíme jednoduchost použití, to nám zajistí Qt knihovna, a podobnost kódu modifikace s kódem KernelSharku. Tento cíl udělá kód příjemnějším ke čtení v budoucnosti.

5.2.2 Implementace

V hlavičkovém souboru najdeme třídu KsControlCapture, která sdružuje prvky konfigurující zachycování. Do jejích datových členů přidáme zašktávací políčko. V konstruktoru pak toto políčko nezapomeneme iniciovat a nastavit jako nezaškrtnuté jako výchozí stav. Poté najdeme místo, kde se stavy GUI prvků interpretují na konfiguraci zachycení. Zde přibude překlad zašktnutého políčka na přidání argumentu -T k ostatním argumentům pro zachytávací program trace-cmd. Tím bude modifikace dokončena.

5.3 Vývojová dokumentace

Vývojová dokumentace této modifikace slouží k orientaci ve vylepšení.

Modifikace používá značku RECORD KSTACK v ohraničeních změn. Jedinými změněnými soubory jsou *KsCaptureDialog.hpp/cpp* - v těchto souborech je přidáno zaškrtávací políčko do Record okénka KernelSharku, přidána je i inicializace tohoto políčka a význam zaškrtnutí.

Modifikace je pouze rozšíření GUI o políčko a přidání jeho sémantiky zaškrtnutí. Políčko musí být nutně v Record okénku, resp. ve třídě KsCaptureControl, jelikož nikde jinde nemá nastavení vliv.

5.4 Uživatelská dokumentace

Uživatelská dokumentace pojednává hlavně o tom, jak modifikaci instalovat a používat.

Stejně jako u jiných modifikací, pokud již máme instalovaný KernelShark s touto modifikací, není potřeba dělat nic. Jinak se musí KernelShark sestavit pomocí oficiálních instrukcí z kódu s touto modifikací. Tato modifikace nevyžaduje upravený soubor CMakeLists.txt.

Po otevření KernelSharku otevřeme okno Trace-cmd GUI přes tlačítko `Record`. Zde najdeme zaškrtávací políčko s popiskem „Enable kernel stack tracing“. Zaškrtnutím tohoto políčka povolíme trasování zásobníku kernelu. Poté spustíme trasování a otevřeme výsledná data v KernelSharku. Po každé události, vyjímaje události vytvořené KernelSharkem během načítání trasovacích dat, například Couplebreak události, se budou zobrazovat události ftrace/kernel_stack.

Změny v okénku lze pozorovat na následujícím obrázku 5.1 (obrázek byl zvětšen pomocí AI). Červené obdélníky zvýrazňují místa, kde se Record okno díky modifikaci změnilo. Černé obdélníky schovávají cesty na autorově stroji.

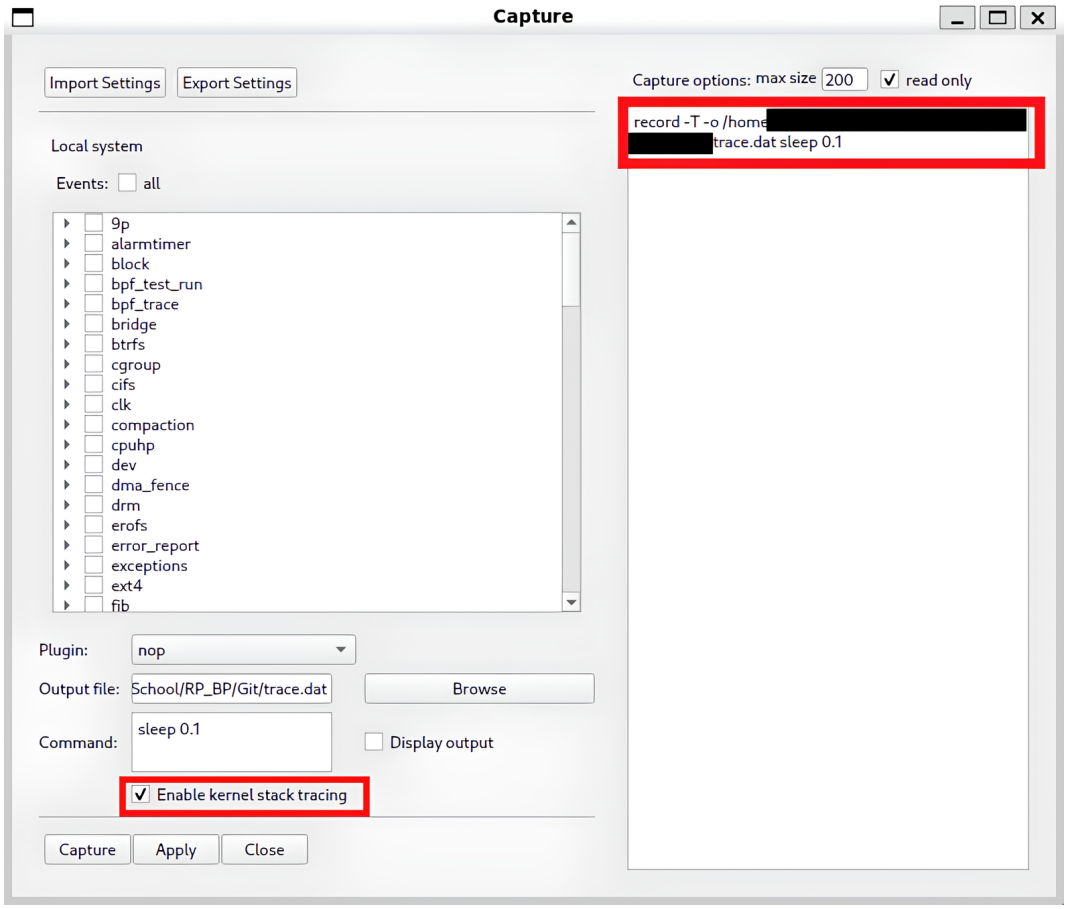
5.5 Rozšíření

Ne každá událost může mít zajímavý zásobník. Možnost zachytávat pouze po některých událostech by zde pomohla. Nicméně to je spíše rozšíření pro trace-cmd. KernelShark by mohl umět ignorovat záznamy zásobníku jádra, pokud navazují na nějakou událost, po které nás zásobník nezajímá. Data budou stále v souboru, ale KernelShark by je nenačetl.

5.6 Zhodnocení splnění požadavků

Jediný vlastní požadavek této modifikace bylo přidání zaškrtávacího tlačítka pro jasnější zapínání sběru zásobníků kernelu. Toto bylo splněno přidáním takového tlačítka do Record okénka a připsání interpretace zaškrtnutí jako dodání argumentu trace-cmd.

Obecné požadavky byly splněny také, GUI prvek byl přidán, kód byl rozšířen. Žádná modifikace existujícího chování se neudála. Tlačítko je ve výchozím stavu nezaškrtnuté a lze jej po zaškrtnutí odškrtnout. Tím jsou obecné požadavky splněny.



Obrázek 5.1 Zvýraznění změn této modifikace viditelných v GUI

6 Stacklook

Tato kapitola se zabývá pluginem pro KernelShark, díky kterému bude jednodušší analyzovat trasovací data, kde se zachytával zásobník kernelu. V kapitole se seznámíme s cíli, analýzou řešení, návrhem a použitím tohoto pluginu. Nakonec i řešení kriticky zhodnotíme a představíme návrhy pro rozšíření.

6.1 Cíle

- Nad podporovanými záznamy bude klikatelný objekt - po dvojitém kliknutí se zobrazí vyskakovací okénko.
- Tlačítka se budou zobrazovat nad záznamy jak v CPU grafech, tak v grafech procesů.
- Ve vyskakovacím okénku se zobrazí záznam zásobníku, název procesu, kterému událost záznamu patří a nějaká bližší informace specifická pro událost, nad kterou bylo tlačítko zobrazeno.
- Plugin bude mít konfigurační okénko, kde bude možné některé části chování upravit. Minimální součástí musí být zapínání a vypínání kreslení tlačítek nad podporovanými záznamy, úprava barev tlačítek a nastavování maximálního počtu viditelných záznamů v grafu, při jehož překročení se tlačítka nebudou zobrazovat.
- Informační řádek bude schopen zobrazit část zásobníku při přejetí kurzoru myši přes tlačítko. Která část zásobníku se zobrazí bude nastavitelné v konfiguraci pluginu. Jelikož informační řádek nemůže být změněn přes rozhraní pluginů a grafové objekty pluginů nereagují na přejetí myší, toto budou dodatečné modifikace, resp. modifikace *Preview Labels Changeable* a *Mouse Hover Plot Objects*.
- Tlačítka mohou být barevná stejně jako je zabarven proces, kterému záznam pod tlačítkem náleží. Získání barev procesů, CPU a streamů není v KernelSharku možné, KernelShark dovoluje barevné tabulky jenom vytvořit nanovo, které ovšem nebudou synchronizovány s tabulkami využívanými KernelSharkem. Zpřístupnění těchto tabulek bude dodatečná modifikace s názvem *Get Colors*.
- Je nutné podporovat alespoň události typu sched/sched_switch a sched/sched_waking.
- Plugin bude možné používat i pro KernelShark bez modifikací vytvořených jako součást této práce.

Poslední bod existuje pro případné uživatele, kteří nechtějí používat modifikovaný KernelShark, dokud nejsou modifikace součástí oficiálního vydání. Plugin se tak může využívat i v úplné izolaci od ostatních změn z této práce, což souhlasí s obecnými požadavky na pluginy. Ostatní cíle vychází z kapitoly *Obecná analýza a stanovení požadavků*. Jako samozřejmost se bere technická a uživatelská dokumentace a splnění obecných požadavků na pluginy.

6.2 Analýza

Tato sekce se pokusí zachytit postup, kterým se dostaneme k implementaci řešení. Jedná se o techničtější analýzu než analýza z kapitoly *Obecná analýza a stanovení požadavků*.

6.2.1 Propojení pluginu s KernelSharkem

Zamysleme se nyní nad tím, jak bychom vytvořili plugin pro KernelShark s našimi cíli. Nejlepším začátkem by bylo nechat se inspirovat oficálními pluginy, od nich se lze naučit, jak plugin propojit s KernelSharkem. Oficiální pluginy definují hlavičkový soubor v jazyce C, kde se definuje kontext pluginu. Tento kontext je pak struktura s globálně přístupnými daty pro plugin. S kontextem se i deklarují kontextové funkce ovládající jeho inicializaci a destrukci přes KernelSharkem definované makro. Krom toho se zde deklarují i další globálně přístupné prvky, které nevyžadují funkcionality C++. Tento hlavičkový soubor může své části implementovat jak C kódu, tak v C++ kódu. Oficiální pluginy mají v C napsanou hlavně definici kontextových funkcí, registraci handlerů pro různé události během načítání trasovacích dat, registrace handlerů na kreslení do grafu, inicializaci dat pro textový font a inicializaci ukazatelů na hlavní okno. Handlery jsou buď implementovány v C++, nebo, hlavně u jednoduchých handlerů událostí, ještě v implementačním C souboru. Kód napsaný v C je tedy většinou použit na inicializaci pluginu a další části, většinou s komplikovanější business logikou, jsou implementovány v C++.

Postup oficiálních pluginů je dobrý, využijeme jej tedy též. Stacklook ovšem bude vyžadovat více C++ kódu. Jednotlivé soubory pak budou sloužit jednotlivým modulům Stacklooku, například modul konfigurace bude obsahovat datovou struktury či datové struktury, které konfiguraci tvoří. Pokud možno, hlavičkové soubory by pak měly reprezentovat každý jeden z modulů. Podobné postupy jsou hojně využívány, hlavně při dekompozicích souboru s kódem, který obsahuje vícero modulů najednou.

6.2.2 Analýza modulů řešení

Z našich cílů lze celkem přímo vymezit moduly, ze kterých se bude plugin skládat: *Propojující modul*, *Konfigurace*, *Tlačítka*, *Detailní pohledy*.

Propojující modul

Propojující modul máme zčásti navržen díky oficiálním pluginům. V „C části“ (napsaná v jazyce C) nastavíme textový font používaný v pluginu, vybereme podporované události při načítání dat a registrujeme handlery. V „C části“ zároveň inicializujeme pluginový kontext. Tato část bude mít za úkol tento kontext i správně odstranit. V „C++ části“ definujeme handlery pro kreslení a přístup k hlavnímu oknu. Kreslící handler bude vyžadovat funkci na vytvoření tlačítek, tedy zde se propojí tlačítkový modul. Funkce vytvoření tlačítek bude potřebovat data pro barvu tlačítek, ta je v konfiguraci, takže se zde objeví i konfigurační modul. Tento modul bude také obsahovat podporované události a bude je sbírat do dat v kontextu pluginu.

Konfigurace

Konfigurace bude rozdělená mezi GUI okno a datovou strukturu s konfiguračními daty, tzv. konfigurační objekt. Konfigurační objekt samotný implementujeme jako singleton. Důvodem je nutnost znát konfiguraci v různých částech programu a že konfigurace musí být vždy pouze jedna, což právě singleton splňuje. Prvky z cíle o konfiguraci můžeme jednoduše uložit jako čísla, pravdivostní hodnoty, textové řetězce a datovou strukturu pro barvy od KernelSharku. Aby bylo možné konfiguraci měnit pouze skrze konfigurační okno, tyto prvky schováme pomocí modifikátorů viditelnosti. Konfigurační objekt pak využije C++ mechanismus „zpřátelení“ přes klíčové slovo `friend` a označí konfigurační okno za svého přítele. Konfigurace pluginů KernelSharku nejsou persistentní, tak dodáme i nějaké výchozí hodnoty. Kvůli zobrazování zásobníku v informačním řádku, přidáme ke každému nastavení specifickému pro událost i offset od vrchu zásobníku. Tak budeme moci stanovit oblast zásobníku, ve které se většinou nachází ty nejzajímavější části zásobníku pro danou událost.

Konfigurační okno využije framework Qt (specificky Qt6), stejně jako ostatní grafické prvky KernelSharku. Na barevná nastavení vytvoříme tlačítko, které na kliknutí vyvolá nějaké okno na výběr barvy, například výchozí barevný dialog od Qt. Pro pohodlí uživatele i někde blízko tlačítka výběru barvy zobrazíme i aktuálně použitou barvu. Nastavení maximálního počtu viditelných záznamů lze reprezentovat jako spinbox. Konfigurace pro specifické události, tj. zdali nad danými událostmi zobrazovat tlačítka nebo offset použitý při zobrazování části zásobníku v informačním řádku, lze reprezentovat pomocí zaškrtávacího tlačítka a dalšího spinboxu. Mimo požadavky navíc dodáme zaškrtávací tlačítko pro barvení tlačítek barvami jejich procesů.

Tlačítka

Modul tlačítek se bude hlavně skládat z třídy pro tlačítka. Ta by měla být v grafu jasně viditelná a ihned rozpoznatelná uživatelem. Tvar tlačítek by měl nějak ukazovat, kterému záznamu patří. Zde je několik možností a tvarů na výběr, nicméně nejjednodušším tvarem bude trojúhelník. Jeden z vrcholů bude směřovat dolů, ostatní dva vrcholy budou nad tímto vrcholem. Rozpoznatelnost těchto tlačítek nakonec zajistíme přidáním nápisu `STACK` do každého z nich. Reakci na dvojité kliknutí a přejetí kurzorem myši implementujeme jako to dělají ostatní tvary definované KernelSharkem. U přejetí využijeme naše modifikaci na reakci na přejetí kurzorem myši a modifikaci na změnu informačního řádku, přesně jak říká jeden z cílů. Speciálně pro sched_switch budou tlačítka ještě obsahovat jedno písmeno symbolizující předchozí stav procesu před přepnutím. Takto nebude nutné se podívat do detailního pohledu, bude stačit se dívat do grafu. Nicméně uživatel bude muset znát zkratky pro možné předchozí stavy. Ty jsou součástí dokumentace Linuxu. [TODO: Sem se asi hodí zdroj.] Barva tlačítek bude určena hodnotami v konfiguračním objektu. Dodáme i možnost používat barvy procesů jako barvy tlačítek, to nám zajistí modifikace *Get Colors*. Aby nemohlo více tlačítek splývat díky stejné barvě, tlačítka budou mít obrys, jehož barva bude také dána konfigurací a bude oddělená od (vnitřní) barvy tlačítka.

Detailní pohledy

Modul detailních pohledů bude hlavně obsahovat třídu pro okna těchto pohledů. V okně pak zobrazíme záznam zásobníku ve formě seznamu. Tento formát se hodí na jednoduché zvýraznění položky jedním kliknutím. Dodáme ale i možnost si data zobrazit jako prostý text - tento formát je naopak lepší když se kopírují data po jných částech než po položkách. Na vrchol zásobníku dodáme značku `top`, aby bylo jasné, kde se vrch zásobníku nachází. Jeden z cílů nám ještě udává nutnost zobrazit název procesu a nějaké bližší informace o události. Pro sched_switch zde můžeme udat přechozí stav procesu před přepnutím, například zdali byl proces již ve stavu zombie, nebo zdali začal čekat na nějaká data a nebo zdali prostě běžel a byl preemptivně přepnut. Předchozí stav zapíšeme jak zkratkou, tak celým názvem (například „S - sleep“, nebo „Z - zombie“). Pro sched_waking pouze napíšeme, že byl proces probuzen. Nakonec se zamyslíme nad tím, jak dlouho budou okna žít a kolik jich může být najednou pro jednu událost. Naše pohledová okna můžeme vytvořit jako podřízená oknu hlavnímu, tedy pokud je ukončeno hlavní okno, budou ukončena i jeho podřízená okna. Toto zajistí Qt framework. Počet oken pro jednu událost nebudeme omezovat, nic se tím nerozbije a implementace bude jednodušší. Tato rozhodnutí by měla mít ten efekt, že uživatel bude schopen stream, nazvěme jej A, pak bude schopen začít zkoumat stream jiný, nazvěme tento B. Okna otevřená ve streamu A zůstanou otevřená i po přepnutí do streamu B, do té doby dokud je uživatel sám nezavře, nebo dokud uživatel nezavře hlavní okno. Lze tak porovnávat zásobníky z různých běhů trasování.

Předchozí stavy

Na získání informací o předchozím stavu ještě dodáme mini-modul *Předchozí stavy*. Nutně bude muset být schopen vytáhnout data o předchozím stavu ze záznamu události přepnutí v KernelSharku. Tato data pak bud muset umět dodat pluginu buď jako zkratka pro typ stavu (tlačítka a detailní pohledy), nebo jako celý název (detailní pohledy).

6.2.3 Údaje v informačním řádku

Postupujme dále, podívejme se na informační řádek. Informační řádek nabízí pouze pět míst pro text. Pokud bychom museli zobrazovat konec zásobníku, nebo i prvky za koncem (například kvůli vysokému offsetu v konfiguraci), v informačním řádku pak vyznačíme, že jsme na konci zásobníku a místa, kde by byly prvky za koncem označíme nějakou čárou, třeba mínusem. Pokud ale zobrazujeme oblast zásobníku a za ní zásobník pokračuje, označíme toto třemi tečkami v posledním místě pro text v informačním řádku. Nakonec, pro jasnou identifikaci procesu z jehož záznamu zásobníku čerpáme bude první místo v informačním řádku obsazeno názvem tohoto procesu. Prvky v informačním řádku tedy budou vždy v následujícím pořadí:

- Název procesu
- Věc na zásobníku na offsetu X od vrcholu zásobníku, nebo -
- Věc na offsetu X+1, nebo -

- Věc na offsetu X+2, nebo `-`
- `...`, nebo `(End of stack)`

Informační řádek bude vždy schopen zobrazit nejvýše tři prvky ze zásobníku.

6.2.4 Získání záznamů zásobníku

Nakonec se zamyslíme, jak získat data záznamů zásobníku kernelu. Tlačítka jsou nad záznamy, po kterých následuje záznam zásobníku. Tyto záznamy se vždy vytvoří na stejném CPU jako událost jim předcházející. Během načítání dat ale ještě nemáme události seřazené, ovšem při prvním kreslení už ano. Navíc neplatí, že všechny soubory trasovacích dat musí obsahovat záznamy zásobníku, lze systém trasovat i bez sbírání zásobníku. V tom případě nemá smysl, aby plugin cokoli kreslil. Do kontextu pluginu dodáme dvě proměnné, jednu jako indikátor „hledali jsme záznam trasování zásobníku kernelu“ a druhou jako „záznam trasování zásobníku existuje“, přičemž obě začnou na pravdivostní hodnotě „nepravda“. Při prvním kreslení prohledáme následníky všech záznamů námi podporovaných událostí. Pokud alespoň jeden záznam má za následníka záznam zásobníku, nastavíme druhou proměnnou v kontextu na „pravda“. Ukazatel na tento záznam si pak poznamenáme i u záznamu předcházející události. Po prohledání všech podporovaných záznamů nastavíme i první proměnnou na „pravdu“. Tak si zajistíme jediný chod při hledání těchto dat. Ačkoliv prohledávání může trvat dlouho, stane se tak pouze jednou a časová cena prohledávání tak není příliš výrazná.

6.2.5 Plugin pro nemodifikovaný KernelShark

Každá část implementace využívající nějakou z našich modifikací bude obalena podmínkovým `ifndef` makrem s názvem `_UNMODIFIED_KSHARK`. Toto makro bude aktivováno pokud uživatel při sestavování tuto proměnnou definuje argumentem pro CMake, více v uživatelské dokumentaci v sekci o instalaci. Pokud je makro definováno, pak zvolí alternativní implementaci z `else` větve, která nevyužívá žádnou z modifikací; jinak kód nebude součástí kompilace.

6.3 Vývojová dokumentace

Dokumentace pluginu

Dokumentace je napsána pro nástroj Doxygen. Dokumentovala se každá funkce i proměnná, nicméně vygenerovaná dokumentace obsahuje jenom prvky veřejné, pro skutečné implementační detaily je tedy doporučeno se podívat do zdrojového kódu. Dokumentace navíc obsahuje hlavní stránku a stránku s nástinem návrhu.

Struktura projektového adresáře

Adresář pluginu obsahuje další adresáře. Adresář „src“ je pro zdrojový kód a adresář „doc“ je pro dokumentaci uživatelskou a dokumentaci technickou. Očekává se, že na této úrovni jsou i adresáře pro sestavení. Na stejné úrovni žije i README, soubor s licencí a nejvyšší CMakeLists.txt. V těchto CMake instrukcích se nastaví

proměnné sestavení, například typ sestavení, a případně se zavolá generování dokumentace. CMake instrukce zodpovědné za vytvoření binárního souboru jsou v adresáři se zdrojovým kódem, stejně jako to dělá KernelShark.

Přehled modulů pluginu

- *Propojující modul* - Modul s kódem propojujícím KernelShark a plugin. Obsahem je hlavně kontext pluginu, funkce kontextu, handlery a implementačně pomocné funkce. Součástí tohoto modulu je část s C kódem a implementační část v C++ prvků z hlavičkového C souboru. Právě v „C++ části“ (napsaná v C++) se ostatní moduly propojují; zároveň tato část ukládá některá globální data s C++ typy. Soubory modulu jsou *stacklook.h/c* a *Stacklook.cpp*.

- *Konfigurace* - Modul se skládá ze dvou tříd, konfigurační objekt a konfigurační okénko. Konfigurační objekt obsahuje konfigurační data, která plugin zrovna využívá a je navržen jako singleton. Konfigurační okénko představuje GUI element, kterým se data v konfiguračním objektu manipulují. Soubory modulu jsou *SlConfig.hpp/cpp*.

- *Tlačítka* - Tlačítka jsou vykreslována v grafu nad záznamy podporovaných událostí. Dokáží reagovat na dvojité kliknutí myši, čímž vyvolají detailní pohled na zaznamenaný zásobník, nebo přejetí nad nimi, kdy zobrazí část zásobníku v informačním řádku. Soubory modulu jsou *SlButton.hpp/cpp*.

- *Detailní pohledy* - Detailní pohledy jsou grafická okna zobrazující zaznamenaný kernel zásobník, informace o události, která záznamu zásobníku předcházela, a informace o procesu vlastnící tuto událost. Soubory modulu jsou *SlDetailedView.hpp/cpp*.

- *Předchozí stavy* - Mini-modul, dodává data tlačítkům a detailním pohledům o stavu procesu před přepnutím kontextu na CPU. Soubory modulu jsou *SlPrevState.hpp/cpp*.

6.4 Uživatelská dokumentace

Tato sekce popíše jak instalovat a používat plugin Stacklook v KernelSharku a co od něj během běhu očekávat, či na co si dát pozor. Obrázek 6.1 ukazuje fungující plugin v akci. Styl a i obsah této dokumentace jsou velmi podobné uživatelské dokumentaci pluginu Naps, jelikož jsou efektivní a dají se jednoduše poupravit na jiný plugin.

6.4.1 Jak sestavit a instalovat Stacklook

Předpoklady

- CMake verze alespoň 3.1.2.

- Pokud chceme plugin využívající modifikovaný KernelShark, pak je nutná verze alespoň 2.4.0-couplebreak. Pokud chceme plugin pro nemodifikovaný KernelShark, pak je nutná verze alespoň 2.3.2.

- Závislosti KernelSharku (nalezneme v README repozitáře KernelSharku), zejména Qt6.
- Doxygen na technickou dokumentaci.

Sestavení a instalace pouze pluginu

1. V terminálu nastavme pracovní adresář na složku `build` (pokud ještě neexistuje, pak ji nejlépe vytvořme v kořenovém adresáři projektu).
2. Spusťme příkaz `cmake ...` Pokud hlavní soubor `CMakeLists.txt` není v nadřazené složce, předejme programu CMake platnou cestu k němu.
   - Používáte-li verzi KernelSharku bez modifikací, přidejme do příkazu argument `-D_UNMODIFIED_KSHARK`. Sestavení pro nemodifikovaný KernelShark odstraňuje tyto funkce:
     - Tlačítka mohou být stejné barvy, jako procesy, jimž patří záznamy s tlačítky.
     - Přejetí myši zobrazí část zásobníku v informačním řádku KernelSharku.
   - Pokud chceme generovat dokumentaci pomocí Doxygenu, přidejme do příkazu argument `-D_DOXYGEN_DOC=1`.
   - Výchozí typ sestavení je `RelWithDebInfo`. Chcete-li ho změnit (např. na `Release`), použijme argument `-DCMAKE_BUILD_TYPE=Release`.
   - Pokud se soubory Qt6 nenacházejí ve `/usr/include/qt6`, použijme argument `-D_QT6_INCLUDE_DIR=[PATH]`, kde `[PATH]` nahradíme cestou k souborům Qt6.
     - Pokyny pro sestavení předpokládají, že zadaný adresář má stejnou vnitřní strukturu jako výchozí možnost (tj. obsahuje složky QtCore, QtWidgets apod.).
   - Pokud se zdrojové soubory KernelSharku nenachází v `../KS_fork/src`, použijme argument `-D_KS_INCLUDE_DIR=[PATH]`, kde `[PATH]` nahradíme cestou ke zdrojovým souborům KernelSharku.
   - Pokud se sdílené knihovny KernelSharku (`.so` soubory) nenachází ve `/usr/local/lib64`, použijme `-D_KS_SHARED_LIBS_DIR=[PATH]` argument, kde `[PATH]` nahradíme cestou k sdíleným knihovnám KernelSharku.
3. Ve složce `build` spusťme příkaz `make`.
   - Pokud je třeba sestavit jen část pluginu, například pouze dokumentaci, můžeme vybrat konkrétní cíl.
   - Pouhé spuštění `make` vytvoří: *plugin* (cíl `stacklook`), *symlink* na sdílený objekt pluginu (cíl `stacklook_symlink`) a případně *Doxygen dokumentaci* (cíl `docs`), pokud tak bylo specifikováno v předchozím kroku.

4. (*Instalace*): Nahrajme plugin do KernelSharku, buď přes GUI, nebo při spouštění přes CLI s argumentem `-p` a cestou k symlinku nebo přímo k sdílenému objektu.

    - *DŮLEŽITÉ*: Vždy nainstalujte/nahrajme plugin před načtením relace, ve které byl aktivní! Jinak může dojít k neúplnému načtení konfiguračního rozhraní nebo k pádu celého programu.

K odstranění vytvořených binárních souborů použijme `make clean`.

Sestavení a instalace pomocí KernelSharku

1. Ujistěme se, že všechny zdrojové soubory (`.c`, `.cpp`, `.h`) Stacklooku se nacházejí ve složce `src/plugins` v adresáři projektu KernelShark.

2. Zkontrolujme, že soubor `CMakeLists.txt` v této podsložce obsahuje instrukce pro sestavení pluginu (inspirovat se můžeme podle jiných pluginů pro GUI). Pokud chceme sestavovat pro nemodifikovaný KernelShark, upravme tomu odpovídajícím způsobem build skript.

3. Sestavme KernelShark (pluginy se sestavují automaticky). Lze sestavit i pouze plugin, pokud jsme již předtím vytvořili instrukce sestavení.

4. (*Instalace*): Spusťme KernelShark. Pluginy sestavené tímto způsobem se načítají automaticky. Pokud by se z nějakého důvodu nenačetly, najděme sdílený objekt stejně jako u ostatních oficiálních pluginů, opět buď přes GUI, nebo přes CLI.

VAROVÁNÍ - načítání více verzí pluginu

Máte-li dvě nebo více sestavených verzí pluginu, *NE*načítejme je současně do KernelSharku. Pokud to uděláme, *DOJDE K PÁDU PROGRAMU*. Používejme vždy jen jednu z verzí, *NIKDY OBOJE NAJEDNOU*.

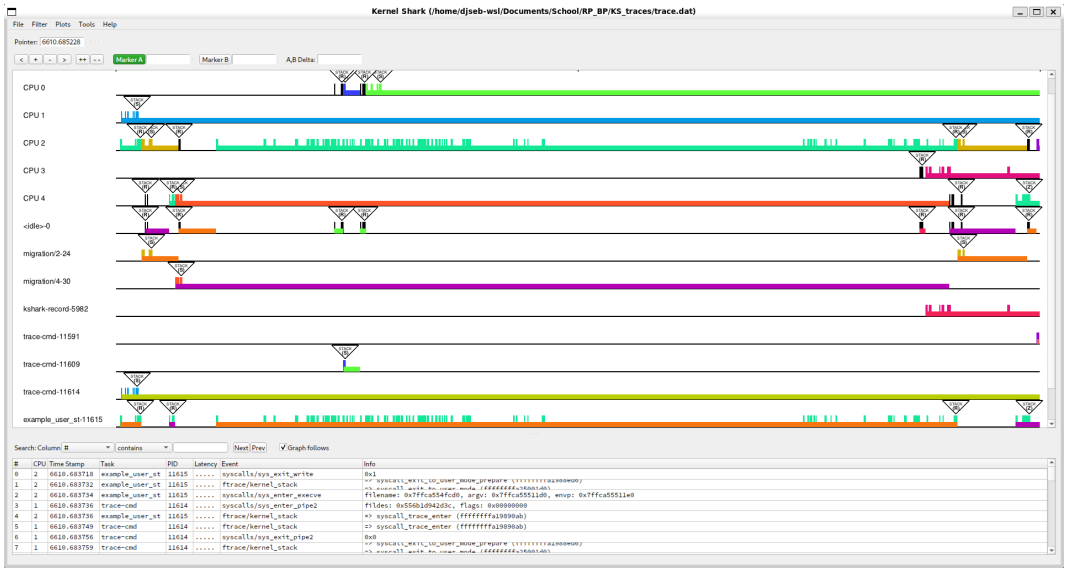
6.4.2 Jak zapnout/vypnout Stacklook

Zapnutí pluginu je velmi jednoduché. Stačí spustit KernelShark a přejít na položku v panelu nástrojů `Tools > Manage Plotting plugins`. Pokud byl plugin načten přes příkazový řádek, zobrazí se v seznamu pluginů jako zaškrtnuté políčko se svým názvem. Pokud ne, lze plugin dohledat pomocí tlačítka `Tools > Add plugin` - stačí nalézt symlink, ale je možné vybrat i samotný soubor sdíleného objektu. Jak je vidět, plugin využívá standardní mechanismus načítání pluginů v KernelSharku. Obrázek 6.2 ukazuje GUI pro zapínání a vypínání pluginů.

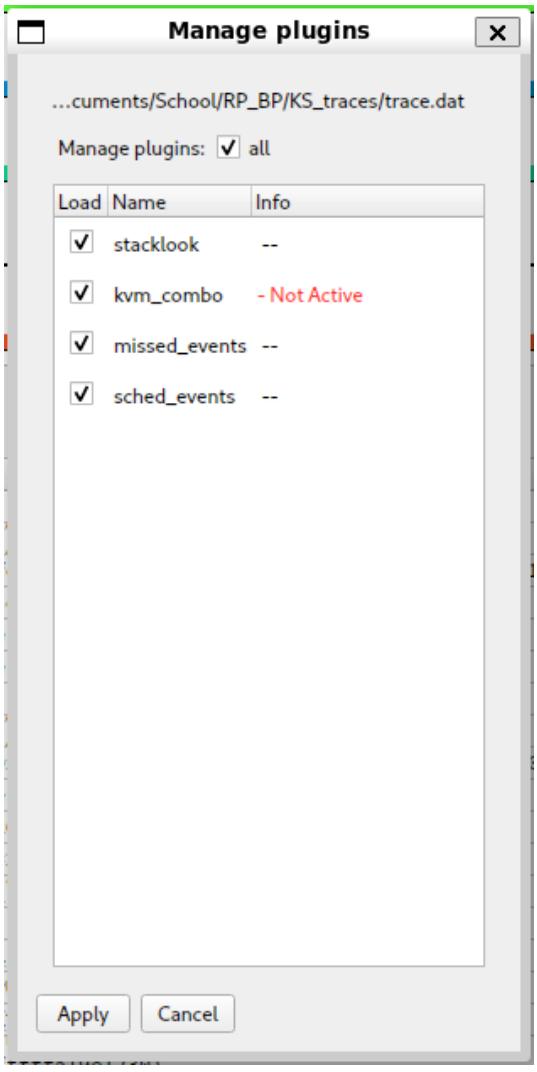
6.4.3 Jak používat Stacklook

Konfigurace

Konfigurace pluginu může být provedena kdykoliv, i před načtením jakýchkoliv trasovacích dat. Pro otevření konfiguračního okna (viz obrázek 6.3) stačí v hlavním



Obrázek 6.1 Fungující Stacklook



Obrázek 6.2 Okénko se správou pluginů

okně zvolit `Tools > Stacklook Configuration`. Vždy může být otevřeno jen jedno konfigurační okno.

Používáte-li verzi pluginu pro nemodifikovaný KernelShark, bude v konfiguračním okně chybět zaškrtávací políčko pro barvení tlačítek barvami procesů a nastavení offsetu, použitého při zobrazování prvků zásobníku v informačním řádku, viz obrázek 6.4.

Nyní si popíšeme jednotlivé možnosti konfigurace a jak je ovládat. Seřazeno shora dolů:

- *Limit záznamů v histogramu* - snížením této hodnoty omezíme, kdy se Stacklook aktivuje. Aktivace nastane pouze tehdy, pokud je viditelný počet záznamů menší nebo roven této hodnotě. Čím nižší číslo, tím větší přiblížení je třeba k aktivaci Stacklooku. Minimální hodnota je 0, maximální 1 000 000 000 (jedna miliarda), tato horní mez však bude pravděpodobně zřídka využita. Výchozí hodnota je 10 000 (deset tisíc).

- *Použít barvy procesů pro tlačítka Stacklooku* - zaškrtnutím tohoto políčka (pokud je k dispozici) bude výplň tlačítek Stacklooku zabarvena barvou procesu, ke kterému patřila událost, pro niž Stacklook našel záznam zásobníku. Pokud políčko necháme nezaškrtnuté, použijí se barvy v konfiguraci Stacklooku (výchozí barvy na obrázku 6.5). Tato volba je ve výchozím stavu vypnutá.

- *Barvy tlačítek Stacklooku a jejich obrysů* - tlačítka `Choose` otevřou dialog pro výběr barev, kde lze nastavit barvu výplně tlačítka nebo jeho obrys. Tato nastavení mají účinek pouze tehdy, pokud není dostupná nebo není aktivní volba barev podle procesů. Vedle tlačítek je zobrazen náhled aktuálně vybraných barev. Výchozí nastavení je bílá výplň a černý obrys. Příklad jiných barev si lze prohlédnout na obrázku 6.7.

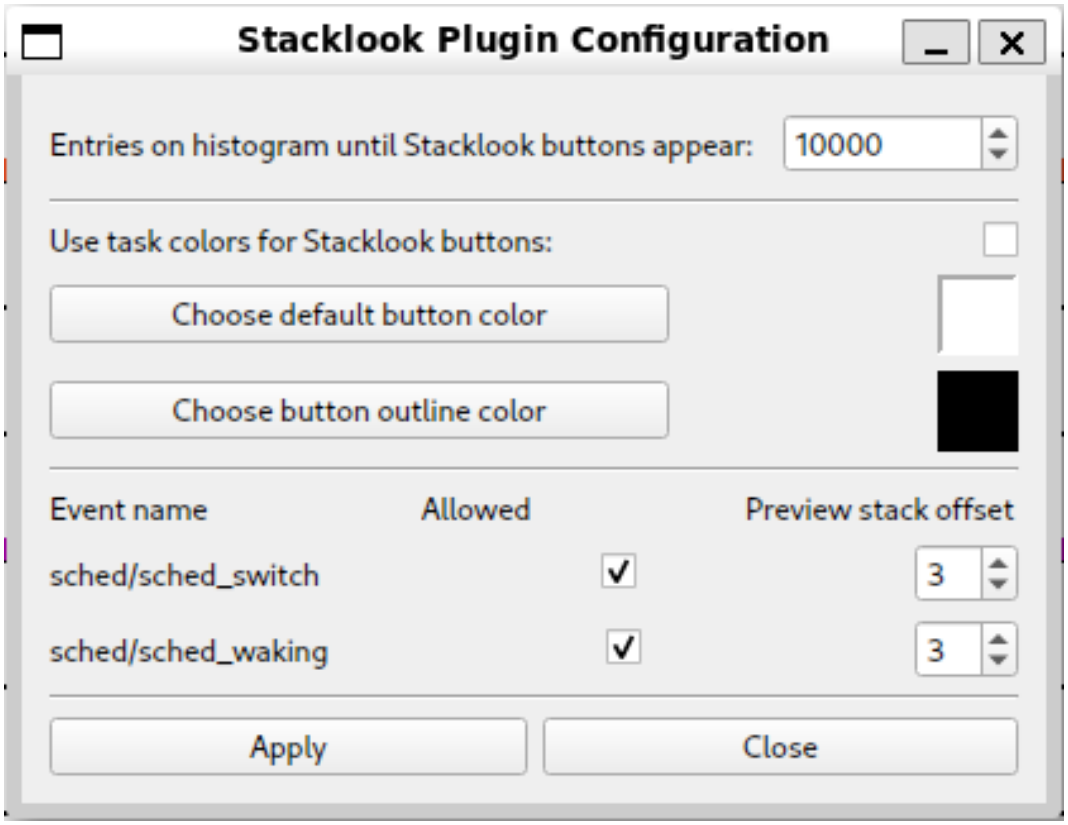
- *Nastavení pro jednotlivé události* - Každá událost má své vlastní zaškrtávací políčko, kterým lze Stacklook zapnout nebo vypnout pro daný typ záznamu, a (pokud je k dispozici) číselník s offsetem do zásobníku kernelu, který slouží k určení „nejzajímavější“ oblasti v zásobníku pro danou událost. Maximální hodnota offsetu je 100 000 000 (sto milionů), minimální 0. I zde je nepravděpodobné, že by bylo třeba využít maximální hodnotu. Ve výchozím stavu jsou všechny události povoleny (zaškrtnuty) a offset je nastaven na 3.

  - Připomenutí - číselník offsetu se nezobrazí, pokud je použit nemodifikovaný KernelShark.

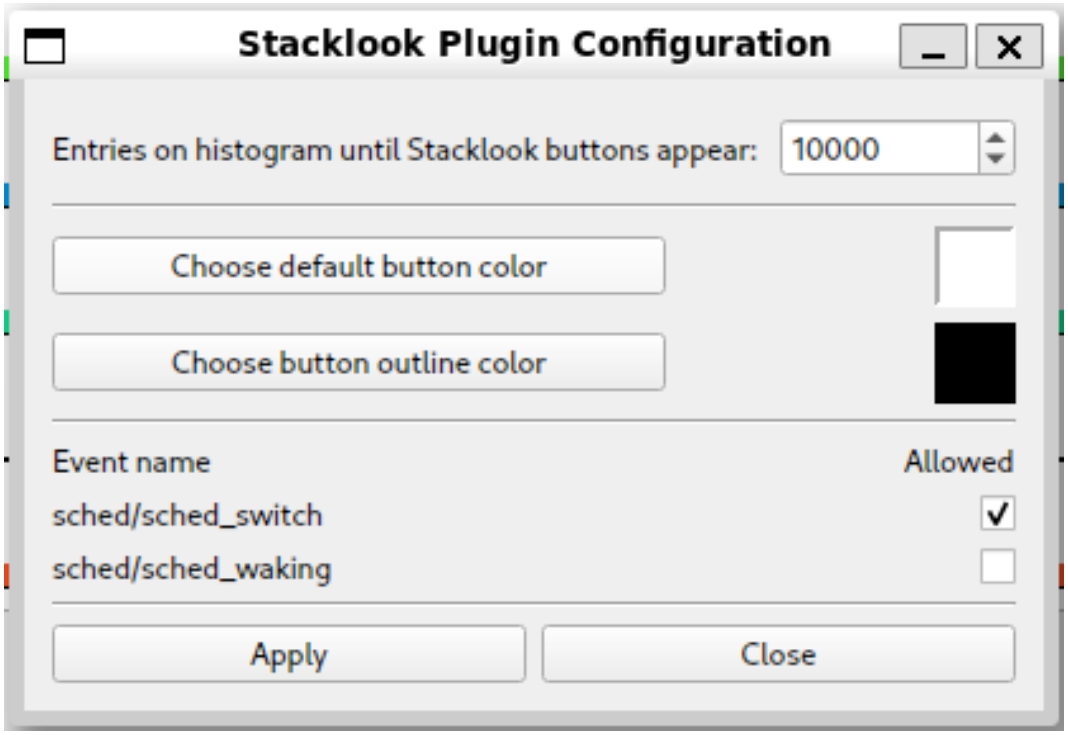
Tlačítko `Apply` uloží provedené změny a zavře konfigurační okno - pokud toto tlačítko nebude stisknuto, změny se neprojeví. V ovládacích prvcích konfiguračního okna se zobrazují pouze aktuálně platné hodnoty konfigurace, konfigurační okno si po zavření neaplikované změny nepamatuje. Tlačítko `Close` i tlačítko s křížkem v pravém horním rohu okna změny zahodí a okno zavřou.

U Stacklooku nelze konfigurovat:

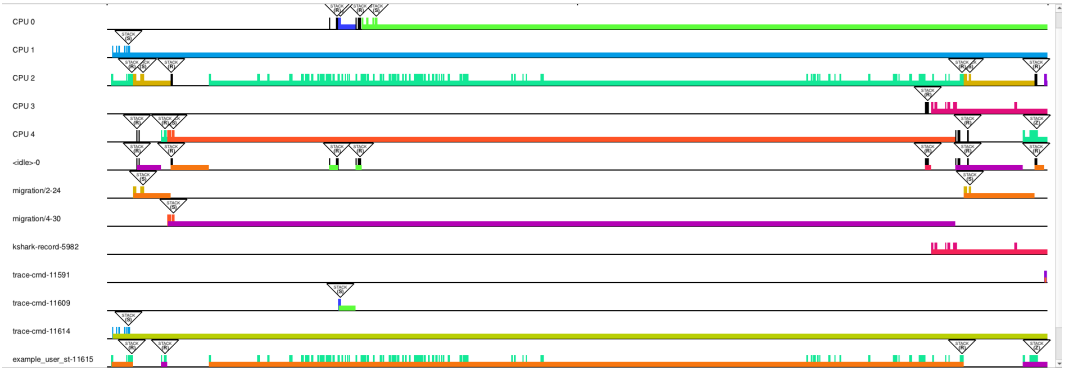
- Podporované události - plugin momentálně podporuje pouze události sched/sched_switch a sched/sched_waking.



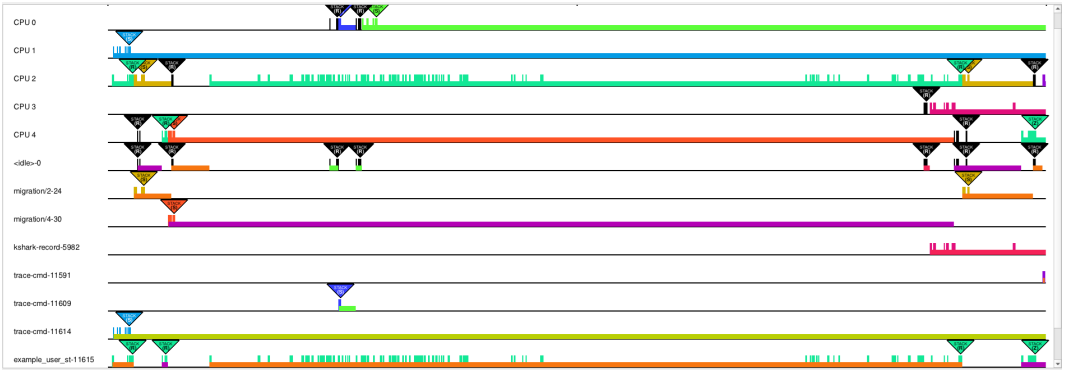
Obrázek 6.3 Konfigurační okno Stacklooku pro modifikovaný KernelShark



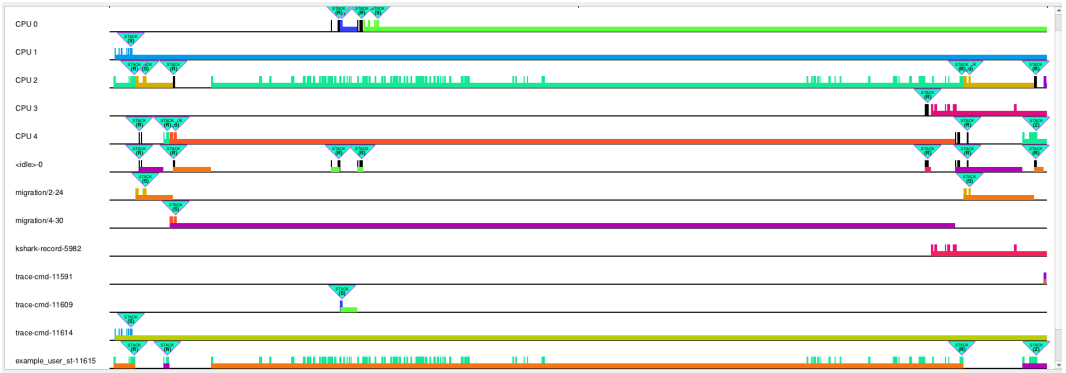
Obrázek 6.4 Konfigurační okno Stacklooku pro nemodifikovaný KernelShark



Obrázek 6.5 Tlačítka s výchozími barvami



Obrázek 6.6 Tlačítka využívající barvy procesů



Obrázek 6.7 Tlačítka s konfigurovanými barvami

- Text v oknech detailních pohledů.
- Text tlačítek.
- Velikost tlačítek.
- Pozice tlačítek.

Konfigurace není persistentní. Její momentální stav se nikam neukládá, ani do relací.

V grafu

Po načtení (a případné konfiguraci) pluginu přibližme zobrazení tak, aby bylo v grafu viditelných méně záznamů, než je nastavený limit. Nad každým podporovaným záznamem se objeví tlačítko - buď ve výchozí barvě z konfigurace, nebo, pokud používáme upravený KernelShark a máme zapnutou příslušnou část nastavení, bude tlačítko obarveno podle procesu. Používání barev procesů je plně kompatibilní s barevným posuvníkem KernelSharku.

Plugin nezobrazí tlačítka nad nepodporovanými událostmi nebo pokud podporovaná událost nenajde záznam zásobníku, ze kterého by bylo možné čerpat data.

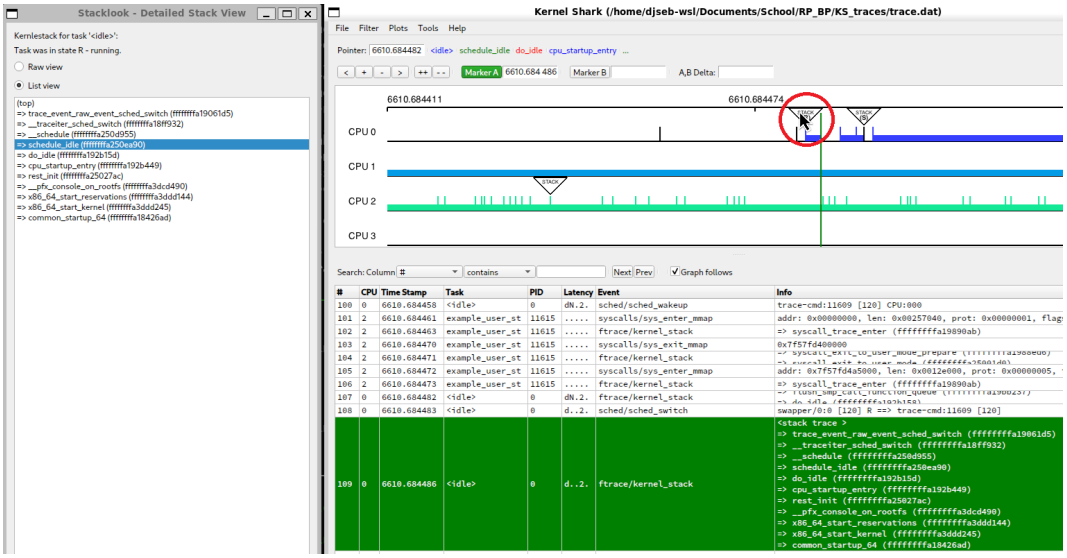
Pokud používáme modifikovaný KernelShark, přejeďme kurzorem nad libovolné tlačítko a informační řádek KernelSharku. Obsah řádku se změní a zobrazí:

- Název procesu jako první (nejlevější) položku.
- Položku v zásobníku kernelu na pozici danou konfigurovaným offsetem od vrcholu zásobníku.
- Položku v zásobníku kernelu následující po první.
- Položku v zásobníku kernelu následující po druhé.
- A nakonec buď tři tečky ..., pokud stack obsahuje další položky, nebo zprávu `(End of stack)`, pokud už žádné další položky nezbývají.

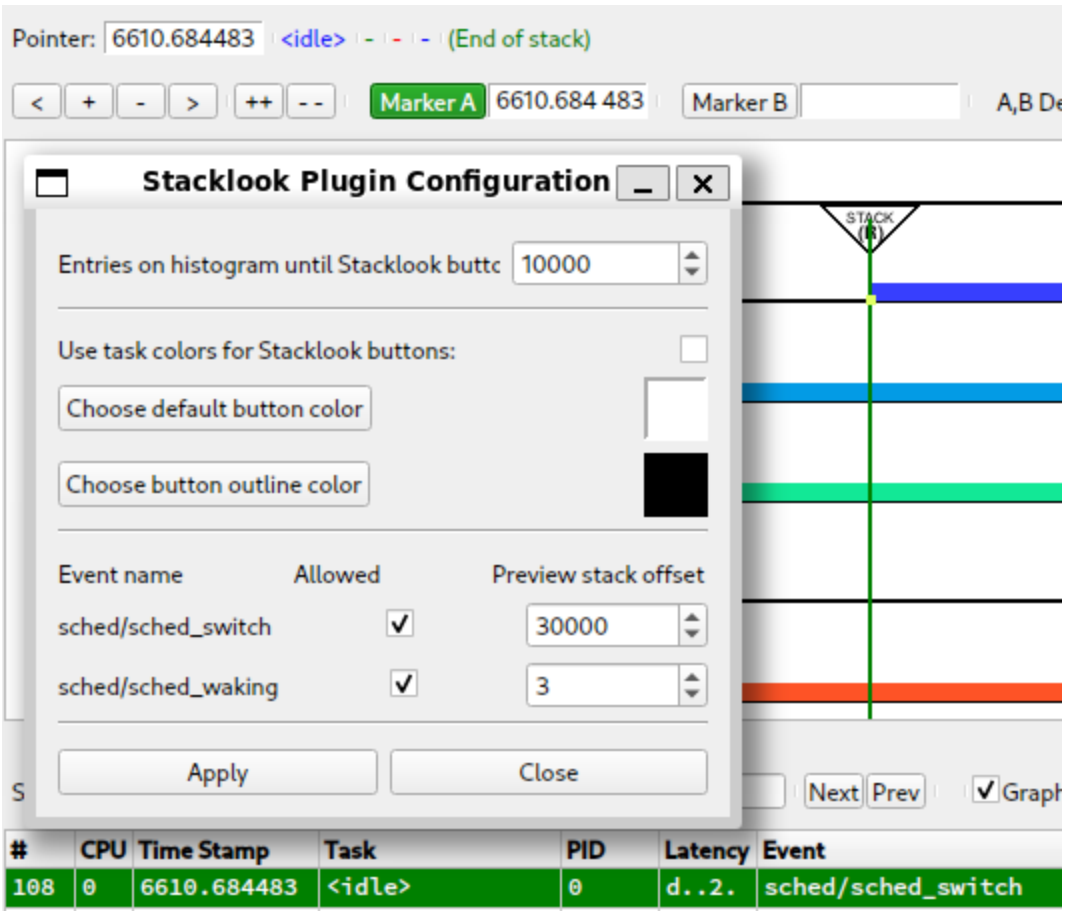
Pohleďme na obrázek 6.8 s malou ukázkou této funkcionality. Součástí je také okno Stacklooku (o nich více níže) a záznam zásobníku kernelu v hlavním okně v seznamu všech událostí. Vše je takto uspořádáné, aby bylo zřejmé, že položky v informačním řádku skutečně pocházejí ze zásobníku (použitý offset byl výchozí, tedy 3). Červený kruh zvýrazňuje záznam, nad kterým právě kurzor myši přejíždí.

Offset lze nastavit i tak vysoko, že se v náhledu zobrazí pouze poslední jedna, dvě nebo tři položky, případně žádná. V takovém případě Stacklook zobrazí pouze mínus a zprávu `(End of stack)` (viz obrázek 6.9).

Po dvojkliku na tlačítko Stacklooku se otevře nové okno detailního pohledu, také nazývané okno Stacklooku. Poku čteme odshora, tak v okně nejprve vidíme, že si prohlížíme zásobník kernelu nějakého procesu. Níže je napsáno, zda byl proces probuzen (zobrazuje se pouze u událostí sched/sched_waking) nebo o jejím předchozím stavu (pouze u událostí sched/sched_switch). Následují dvě rádiová tlačítka a zobrazení zásobníku kernelu k dané události. Rádiová tlačítka přepínají mezi různými způsoby zobrazení zásobníku:



Obrázek 6.8 Reakce tlačítek na přejetí kurzorem myši



Obrázek 6.9 Chování informačního řádku při velkém offsetu do zásobníku

- Ve výchozím stavu je zobrazení ve formě surového textu, tedy zásobník je pouze řetězec s konci řádků. Toto je užitečné pro zkopírování zásobníku jako jednoho textu nebo pro zvýraznění konkrétní části položky v zásobníku.
- Alternativně lze zásobník zobrazit jako seznam, což umožňuje jednodušší a rychlejší (namísto dvojitého kliknutí stačí jedno) zvýraznění jednotlivých položek.

Pro jeden záznam může být otevřeno více oken, zároveň může být otevřeno více oken pro různé záznamy, v libovolné kombinaci.

Vše popsáno výše si můžeme prohlédnout na obrázku 6.10.

Okno lze zavřít přes tlačítko `Close` v dolní části okna, nebo pomocí tlačítka s křížkem v hlavičce okna. Všechna okna se zavřou, pokud bude zavřeno hlavní okno KernelSharku.

Pro přehled předchozích stavů, které jsou napsány v detailních pohledech pro sched_switch události, je níže jejich seznam s krátkým vysvětlením každého z nich.

- Uninterruptible (disk) sleep - úroces čeká na dostupnost zdrojů a nereaguje na signály.
- Idle - pro speciální vlákno kernelu, nelze převést do stavu Running.
- Parked - pouze pro procesy kernelu, proces se dobrovolně vzdá CPU a označí se tímto stavem, aby mohl být spuštěn později.
- Running - proces běží na nějakém CPU.
- Sleeping - proces čeká na dostupnost zdrojů a reaguje na signály.
- Stopped - procesu byl zaslán STOP signál.
- Tracing stop - proces se zastavil, jelikož je právě trasován či laděn.
- Dead - přechodný stav těsně předtím, než bude proces dealokován.
- Zombie - proces skončil svou práci a čeká na uklizení rodičovským procesem.

6.4.4 Bugy a chyby

Pokud by se tlačítka Stacklook překrývala, bude tlačítko příslušící *starší* události vykresleno nad tlačítkem pro událost *pozdější*, avšak kliknutí nebo najetí myší na překryté tlačítko použije k interakci tlačítko pro *pozdější* událost. Podle autorových znalostí jde o interní chování KernelSharku, které nelze na straně pluginu opravit.

Načtení relace KernelSharku, kde byl Stacklook aktivní, bez předchozího načtení pluginu způsobí *segmentation fault* a *pád programu* při najetí kurzoru myši na Stacklook tlačítka.

6.4.5 Doporučení

- Vždy načtěme Stacklook před načtením relace. Může to programu ušetřit nepříjemná překvapení.
- Neotvírejme stovky a stovky oken Stacklooku, pokud nechceme zbytečně zatížit paměť.
- Nedoporučuje se nastavovat příliš vysoký limit záznamů v histogramu v konfiguraci. Jinak by plugin mohl používat příliš mnoho paměti kvůli velkému množství naráz dostupných tlačítek Stacklooku.
- I když relace v KernelSharku fungují, jsou trochu nestabilní. Tento plugin se snaží jejich vnitřní logiku nenarušovat, ale varuje, že pokud plugin není načten předem, mohou nastat neočekávané problémy. Například načtení relace s aktivním pluginem nepřidá do menu `Tools` odpovídající položku pro vyvolání konfiguračního okna.

6.5 Rozšíření

Tato sekce se podívá na některá další vylepšení, která by pluginu prospěla.

Více událostí

Plugin lze rozšířit o další podporované události. Hlavními místy pro tato rozšíření je pak propojující modul, kde se podporované události sbírají a do kontextu se ukládají jejich identifikátory, a mapová konstanta s hodnotami informací specifických pro typ události, která je uvnitř funkci tlačítek, jež vytváří detailní pohledy.

Persistentní konfigurace

Konfigurace pluginu není persistentní, což je nepříjemné, pokud jsme u konfigurace pluginu měnili mnoho věcí. Pokud si je chceme zachovat, musíme nechat program běžet, což bude hlavně u velkých trasovacích dat zbytečně namáhat hardware. Jistě by pomohlo, kdyby bylo možné konfiguraci pluginu ukládat do relací KernelSharku, nebo do nějakých vlastních konfiguračních souborů.

6.6 Kritika

V této sekci jsou sepsány kritiky, které napadly autora pluginu. K nim jsou buď připsány obrany a nebo přiznání, že kritika má své místo a nejspíše bylo lepší vydat se jiným směrem.

Konfigurační singleton

Použití singletonu v tomto případě dává smysl, konfigurace musí být vždy jen jedna a hodí se, když je globálně dostupná. Návrhový vzor není ovšem bez chyb. Hlavními neduhy pak jsou obtížná testovatelnost a skryté závislosti v implementaci.

Návrh konfigurace se alespoň snaží zajistit, že konfigurační objekt nemůže být změněn jinak, než skrze GUI. Tím zakážeme alespoň skryté změny konfigurace a lze se spolehnout na jediný zdroj změn, zmíněnou GUI. To také znamená, že neexistuje jiný způsob změny konfigurace, třeba skrze kód. Není tak možné vytvořit nějaký jiný plugin, který by měnil konfiguraci Stacklooku. To není špatně a plugin počítá s tím, že na něm nebudou ostatní pluginy záviset.

Verze pluginu pro nemodifikovaný KernelShark

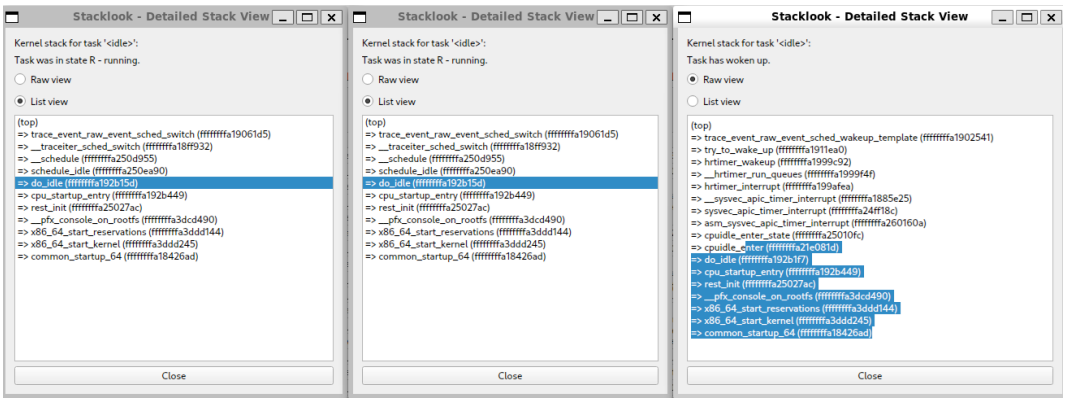
Je možné se dívat na tuto část pluginu jako zbytečnou práci navíc, která v kódu pluginu vytváří zmatky kvůli použití maker pro podmíněnou kompilaci. Cílem této verze byla přístupnost pluginu širší veřejnosti. Plugin, ačkoliv ne kompletní, plní své hlavní funkce i pokud pár drobností není k dispozici, jako například barvení tlačítek dle barev procesů. Jistě existují uživatelé, kteří jsou ochotni vyzkoušet plugin, ale nikoliv pozměněný KernelShark, třeba už jen ze strachu bezpečnosti provedených změn. Plugin se svou verzí pro nemodifikovaný KernelShark je právě pro tyto uživatele. Tak plugin může pomoci více uživatelům - což je hlavním cílem všech vylepšení.

Špatný záznam zásobníku pro událost

Stacklook se při prvním kreslení pokusí najít záznamy zásobníků. Pokud se mu toto podaří, dá odkaz na záznam události záznamu zásobníku záznamu události, která záznamu zásobníku předchází. Problémem je, že pokud se pro nějakou událost záznam zásobníku ztratí, nebo se posune k jinému záznamu zásobníku, pak je možné, že dvě události potom budou sdílet stejný záznam události záznamu zásobníku, přičemž jedné z nich nebude doopravdy patřit. Taková situace při testování pluginu nenastala, ale není ani nemožná (jak Trace-cmd, tak KernelShark dokážou alespoň detekovat, které události byly ztraceny).

6.7 Zhodnocení splnění požadavků

Plugin zřejmě splňuje obecné cíle pluginů, tj. vlastní adresář s instrukcemi k sestavení a nezávislost na jiných pluginech (nikde v analýze jsme se nerozhodli použít nějaký z dalších pluginů jako závislost a ani jsme nepřišli na konflikty s jinými pluginy). V analýze jsme pak postupovali tak, abychom každý z cílů splnili. Navíc jsme nepřišli na žádné problémy při vytváření řešení. Tím jsme splnili i požadavky vlastní.



Obrázek 6.10 Několik oken detailních pohledů představující svou strukturu

7 Couplebreak

V této kapitole se detailně seznámíme s modifikací pro KernelShark, která je schopna rozdělovat události a vytvářet pro ně záznamy KernelSharku. Tímto se zjemní dělení informací, což některým pluginům dovolí obejít jistá omezení od KernelSharku, kvůli kterým by bylo občas nutné některé pluginy vypnout. Představeny budou cíle, analýza řešení, návrh a použití tohoto vylepšení. Konečnou částí pak bude kritika, ve které modifikaci zhodnotíme a představíme některá rozšíření.

7.1 Cíle

- Podporované události dvou procesů dají vzniknout dvěma záznamům - původci a novému umělému záznamu.
- Modifikace bude navržena rozšiřitelně o další události.
- Nové záznamy budou patřit tomu procesu z páru, který předtím událost nevlastnil.
- Nové záznamy budou obsahovat odkaz na záznam s původní událostí.
- Nové záznamy budou splňovat rozhraní dotazů na záznamy KernelSharku. To znamená, že bude možné používat `kshark_get_*` funkce jako `kshark_get_pid` nebo `kshark_get_info` apod.
- Nové záznamy bude možné filtrovat jednoduchým filtrem.
- Vylepšení bude možno zapnout a vypnout. Toto nastavení bude možné uložit do relací KernelSharku a načíst je z nich.
- Součástí vylepšení bude i zajištění kompatibility s pluginem sched_events.
- Podporovány budou alespoň události sched_switch a sched_waking.

7.2 Terminologie

- *Couple/pár* je označení pro dva procesy, které sdílí nějakou událost. Například trasovací událost sched_waking, kdy nějaký proces rozhodne o probuzení jiného procesu, přirozeně obsahuje dva procesy - probouzejícího a probouzeného. Páry se dají často rozdělit na procesy cílové a počáteční.
- *Couplebreak událost* je fiktivní událost vytvořená Couplebreakem. Couplebreak momentálně vytváří pouze události cílové. Modifikace toto vyznačuje v sufixu jména události jako „[target]“. Navíc každá taková událost v KernelSharku obsahuje ve svém jméně prefix „couplebreak/“, podobně jako události plánovače úloh obsahují prefix „sched/“. Couplebreak události mají speciální ID začínající na hodnotě `-10 000`, které se postupně vždy o 1 snižuje.

- *Couplebreak záznam* je záznam vytvářený Couplebreakem pro Couplebreak událost. Významnou vlastností těchto záznamů je, že odkazují na záznam s událostí, kvůli které byla Couplebreak událost vytvořena.
- *Origin/počáteční proces* je proces z páru, pro který existuje nějaká událost, se kterou tento proces ovlivní druhý proces z páru.
- *Origin event/počáteční událost* je označení pro událost, která náleží počátečnímu procesu.
- *Původní událost* je událost, při které se Couplebreak aktivuje a vytvoří Couplebreak událost.
- *Target/cílový proces* je proces z páru, pro který existuje nějaká událost, která tento proces nějak ovlivní.
- *Target event/cílová událost* je označení pro událost, která náleží cílovému procesu.
- *Vlastník události* je proces, během kterého událost nastala, resp. proces, v jehož grafu se událost objeví.
- Termíny *(datový) stream, záznam, událost, relace* jsou převzaty z terminologie KernelSharku.

7.3 Analýza

Tato sekce se pokusí zachytit postup, kterým se dostaneme k implementaci řešení. Jedná se o techničtější analýzu než analýza z kapitoly *Obecná analýza a stanovení požadavků*.

7.3.1 Vytváření umělých záznamů

Začněme s jádrem modifikace, rozdělování událostí. Musíme si rozmyslet, kde by bylo možné takovou akci provést. Z logiky fungování programu lze očekávat, že někde se musí události z Trace-cmd přetvořit na záznamy událostí pro KernelShark. Hledáme tedy funkci, která má toto na starosti. Podíváme se na cestu volání programu, když otevřeme nějaký soubor s trasovacími daty. Pak stačí postupovat hlouběji a hlouběji, dokud nenajdeme naši funkci, která nese název `get_records`, která vytváří záznamy KernelSharku pro jeden stream. Právě zde KernelShark také vytváří umělé události pro situaci missed_events, tedy události, které se staly, ale nebyly při trasování zachyceny. Ty KernelShark vytváří před nějakou jinou událostí a tyto umělé události jsou pak umístěny v grafu trasování a seznamu událostí před záznamy událostí, kvůli kterým byly vytvořeny. Navíc má KernelShark jedno hlavní pravidlo pro umělé události - jejich identifikátory musí být záporné. Nyní víme, jak správně vytvářet záznamy pro umělé události a jaká pravidla pro takové události má. Pokud se na funkci `get_records` podíváme blíže, najdeme i místo, kde vytvářet umělé události tak, že budou v grafu i seznamu následovat po událostech, které způsobily jejich vznik, tj. původní události. Zde budeme vytvářet cílové události pro sched_waking a sched_switch. Druhá z událostí má

navíc speciální část ve funkci, z čehož nám plyne i jak vynucovat chování pokud se jedná o nějakou specifickou událost.

Pokračujme tedy návrhem samotných Couplebreak událostí a jejich záznamů v KernelSharku. Identifikátory Couplebreak událostí zvolíme jako negativní hodnoty od hodnoty -10 000 a budou růst po jedné do zápornějších hodnot (tj. budou tvořit sekvenci -10 000, -10 001, -10 002, ...). S identifikátory budou nejspíš chtít pracovat i další části KernelSharku, nebo pluginy, proto je dáme do nového hlavičkového souboru `libkshark-couplebreak.h`. Identifikátory můžeme vytvořit jako konstanty, vytvoříme je tedy jako makra s číselnou hodnotou. Dále, záznamy pak vždy obsahují offset do souboru trasovacích dat, nicméně taková hodnota není pro umělou událost užitečná. Namísto toho splníme jeden z cílů nyní a pole `offset` bude obsahovat ukazatel na záznam původní události. Tak bude možné i využívat data původních událostí. Jelikož Couplebreak rozbíjí události mezi dvěma procesy, nastavíme nynějšího „nevlastníka“ původní události na vlastníka nové události. Naše události musejí být i jednoduše rozlišitelné pro člověka, všechny cílové tedy budou mít názvy ve formátu `couplebreak/{původce}[target]`, kde {původce} bude název události bez prefixu subsystému, kterému by náležela (například jenom část „sched_switch“ z názvu „sched/sched_switch“). Specificky pro události sched_waking budeme chtít změnit CPU, na kterém se událost stala, na CPU, kde bude proces doopravdy probuzen - detaily pro způsob jak toho docílíme vyřešíme později. Záznamy událostí označíme i jako plně viditelné jak v seznamu událostí, tak v grafu trasování. Ostatní informace můžeme zkopírovat z původní události, jelikož Couplebreak události nevyžadují další rozdíly a druhá podporovaná událost, sched_switch, nemá nic, co by bylo zajímavé oddělené.

Nyní už víme, kde vytvářet události a co v nich má být. Vyřešme tedy jak je budeme vytvářet. Abychom mohli jednoduše rozšiřovat Couplebreak o další události v budoucnu, inspirujeme se tím, jak pluginy kreslí do grafu - v `get_records` bude funkce, která přijímá konstruktory (specifikované signatury) a data, která se při vytváření využijí. Pak nám bude stačit vytvořit funkce specifikované signatury a předat jim data pro tvorbu událostí. Později lze definovat funkce pro další podporované události, pokud se přidají nové.

Vytváření je skoro u konce. Vyřešíme už jen, jak nastavit správná CPU pro události sched_waking[target]. Funkce, které volají `get_records`, po práci této funkce setřídí vytvořené záznamy do jednoho pole, kde je vše uspořádáno podle času. Tímto se inspirujeme a vytvoříme setříděné pole, ve kterém následně najdeme správné CPU. Pole je nám poté k ničemu a zahodíme jej. Můžeme se zeptat, zdali nemáme nějaké dostupnější informace pro opravu CPU - odpověď je bohužel negativní. Události sched_waking sice obsahují pole `target_cpu`, ale to je pouze návrh. Operační systém a systémový plánovač úloh se může rozhodnout migrovat proces na nějaké jiné CPU a spustit ho až tam. Informaci o CPU, kde bude probouzená úloha opravdu spuštěna, je v události sched_switch, která přepíná kontext nějakého CPU na probouzený proces. Informace je tedy vázána na víc než na původní návrh a pro její správnost je nutné prohledat události budoucí od původní sched_waking.

7.3.2 Stav Couplebreaku

Couplebreak události je možné vytvářet, ovšem zatím se tomu tak děje vždy. To není v souladu s obecným požadavkem minimálního vlivu. Proto do datové struktury každého streamu přidáme proměnnou, která bude indikovat, zdali je Couplebreak ve streamu aktivní/zapnutý, či nikoliv. Zároveň si do streamu přidáme i počítadlo vytvořených typů Couplebreak událostí ve streamu a bitmasku, ve které budeme zaznamenávat, které Couplebreak události byly pro stream vytvořeny. Bitmasku implementujeme jako typ `int`. Bitmasku i počítadlo upravíme vždy pouze při prvním vytvoření Couplebreak událostí a nebo při inicializacích. Do funkce `get_records` tedy přidáme kontrolu, zda je Couplebreak aktivní. Pokud ano, budeme vytvářet Couplebreak události. Počet událostí a bitmasku budeme v této funkci vždy znovu inicializovat. Výchozí hodnoty všech nových proměnných jsou 0, pro indikátor to znamená `false`.

7.3.3 Úprava rozhraní datových streamů

Aby se záznamy Couplebreak událostí mohly chovat jako obyčejné události, musíme dotazy na ně ošetřit v implementacích funkcí API streamu. Implementace očekávají možné vlastní umělé události, jako missed_events události, takže i nastiňuje, kde by měl být kód, který řeší takové události. Tím se můžeme inspirovat. Rozhraní obsahuje několik funkcí, my si je dáme do seznamu a u každé si zapíšeme, zdali je něco potřeba měnit. Funkce mimo seznam nebyly pozměněny.

- Získání PID ze záznamu pro Couplebreak nalezne nový v datových polích původní události.
- Získání jména události pro Couplebreak nové jméno vytvoří.
- Získání latence událostí pro Couplebreak požádá původní událost o tuto informaci.
- Získání obsahu datového pole `info` pro Couplebreak požádá původní událost o tuto informaci.
- Získání identifikátoru události přes její název pro Couplebreak vrátí jednu z konstant pro identifikátory Couplebreak událostí.
- Získání všech identifikátorů událostí nebude pozměněno. Kdyby se změnila tato implementace, mělo by to příliš široké následky ve zbytku kódu. Proto bude vytvořena nová funkce na získání identifikátorů všech Couplebreak událostí. Na získání opravdu všech identifikátorů přítomných událostí ve streamu bude doporučeno zkontrolovat Couplebreak a pokud je aktivní, tak si zažádat i o seznam jak normálních identifikátorů, tak o seznam Couplebreak identifikátorů.
- Výpis záznamu události pro Couplebreak nejprve vypíše název a PID procesu původní události a poté vypíše to samé, jako by se vypisovala původní událost.
- Získání dat z formátu události, tj. názvu, typu a hodnoty datového pole formátu události ze záznamu události, nebo z události samotné pro Couplebreak požádá původní událost o tuto informaci.

7.3.4 Nové Couplebreak API

Aby se nám s Couplebreakem lépe pracovalo a aby mohli některé jeho koncepty používat i jiné části kódu, dodáme s modifikací i malé API. V té budou pomocné funkce na získání často potřebných dat, hodnoty identifikátorů událostí a hodnoty jejich pozic v bitmaskách streamů. Dodáme následující funkce:

- *couplebreak_get_origin* - vrátí ukazatel na záznam původní události z pole `offset` předaného Couplebreak záznamu události.
- *couplebreak_origin_id_to_flag_pos* - vrátí pozici indikátoru v bitmasce datového streamu podle předaného identifikátoru původní události (tj. pro pozici indikátoru pro sched_switch[target] předáme stream a identifikátor původní události v tomto streamu).
- *couplebreak_id_to_flag_pos* - vrátí pozici indikátoru v bitmasce datového streamu podle předaného identifikátoru Couplebreak události.
- *flag_pos_to_couplebreak_id* - vrátí identifikátor události podle předané pozice v bitmasce streamu.
- *is_couplebreak_event* - určí, zdali je předaný identifikátor události identfikátorem nějaké Couplebreak události.
- *get_couplebreak_event_name* - vrátí jméno Couplebreak události, kterou určíme předáním jejího identifikátoru. Jméno je C-string a volající má na starosti tento řetězec dealokovat.

Všechny prvky tohoto API dáme do souborů `libkshark-couplebreak.c/h`. Tento soubor pak bude součástí knihovny *libkshark*, k čemuž i upravíme CMake instrukce KernelSharku.

7.3.5 Konfigurace

Couplebreak už teoreticky naplno funguje, ale musíme být schopni jej ovládat. K tomu si pořídíme konfigurační dialog, který přidáme do kódu k pomocným okénkům KernelSharku. Pro plnou kontrolu nad jeho obsahem budeme dědit přímo od `QDialog` a zbytek si postavíme pomocí Qt frameworku sami. Přidáme krátký vysvětlující text, aby uživatel věděl, na co klikl a co to umí. Poté přidáme zaškrtávací políčko pro zapínání Couplebreaku. Jelikož streamů může být otevřeno v jedné relaci více, budeme stavět políčka dynamicky podle počtu streamů. Abychom věděli, kterému streamu patří nějaké políčko, přidáme blízko něj popis ve formátu `Stream #N`, kde N označuje identifikátor streamu. Naše rodičovská třída již zajišťuje tlačítka pro potvrzení, nebo zahození změn.

Co se interaktivity týče, tak propojíme potvrzující akci dialogu se změnou stavových proměnných Couplebreaku v příslušných streamech. Krom toho i donutíme KernelShark načíst všechna data znovu, aby se mohl Couplebreak projevit. Po datech také znovu načteme pluginy, aby mohly pracovat s novými daty. Tím dokážeme zapisovat provedené změny. Abychom ale také mohli zobrazovat aktuální konfiguraci, budeme pro každý stream kontrolovat, zdali je v něm Couplebreak zapnutý. Jestliže ano, tak zaškrtávací políčko předem zaškrtneme, jinak jej necháme nezaškrtnuté.

Konfigurační okénko se nezobrazí, pokud není načten alespoň jeden stream. Namísto toho se zobrazí chybová zpráva.

7.3.6 Filtry

KernelShark musí o Couplebreak událostech vědět alespoň v jednoduchých filtrech. Pokud by tomu tak nebylo, pak by při aplikaci jednoduchých filtrů nebyly Couplebreak události zahrnuty a KernelShark by je z grafu vyfiltroval. Přidáme tedy k dialogu pro jednoduché filtry funkci, která přidá do seznamu událostí i podsystém Couplebreak a události v něm. K tomu budeme pouze potřebovat získat všechny identifikátory Couplebreak událostí, což nám zajistí upravené stream API. Nakonec se stačí inspirovat kódem u obyčejného přidávání typů událostí do tohoto dialogu a poté můžeme přidat Couplebreak události.

Pokročilé filtry necháme stranou jako rozšíření modifikace. U těchto filtrů není žádný podstatný problém, jako u filtrů jednoduchých, tedy není tak nutné s nimi spolupracovat.

7.3.7 Propojení s relacemi

Ačkoliv není konfigurace Couplebreaku složitá, bylo by otravné ji při otevření relace vždy znovu nastavovat. Přidáme-li si čas navíc, který KernelShark musí strávit pro opětovné načtení dat, pak pro velká data bychom mohli čekat nepříjemně dlouho. Když si uložíme Couplebreak do relací, tak KernelShark bude potřebovat pouze jedno načtení a my se vyhneme opětovným konfiguracím.

Toto nejspíše bude ta nejjednodušší část implementace, jelikož bude stačit řídit se již existujícím ukládáním dat streamů do relačního souboru a poté uložit jeden bool, tj. indikátor zapnutého Couplebreaku. Nemusíme si ukládat nic dalšího, jelikož ostatní proměnné se vyplní při načtení dat. Při načítání pak budeme opět kopírovat to, co streamy obyčejně dělají, ale budeme jenom hledat hodnotu našeho indikátoru. Pro jednoduchost bude tato hodnota také boolovská, pod klíčem „couplebreak“.

7.3.8 Kompatibilita sched_events s Couplebreakem

Plugin sched_events potřebuje měnit vlastníky sched_switch událostí, aby mohl v grafech procesů správně vykreslovat rámečky pro latence. Se zapnutým Couplebreakem ve streamu, kde je plugin aktivní, tato potřeba odpadá. Plugin bude moci využívat ke kreslení události sched_switch[target], které mají správné vlastníky pro potřeby pluginu. Nám tedy stačí detekovat Couplebreak, pokud je detekován, tak říci pluginu, ať se zajímá o cílové switch události a nemění vlastníky Couplebreak událostí. Tím jsme s pluginem hotovi.

7.3.9 Zamítnutá alternativní řešení

Couplebreak události pouze vizuální

Jádrem tohoto přístupu bylo, že při vykreslování by se KernelShark pokusil zachytit podporovanou událost a nakreslit dva biny do grafu trasování. Nápad byl velmi rychle zamítnut, jelikož by toto byla operace pro každé vykreslení grafu,

což má být pokud možno rychlá operace. Krom toho by vytváření prostoru pro bin bylo dle odhadů zbytečně složité a se záznamy by nebylo možno pracovat v jiných částech kódu, protože nic takového KernelShark nepodporuje.

Dynamické identifikátory událostí

Kromě fixních identifikátorů pro Couplebreak události existoval i nápad, že budou negativními hodnotami identifikátorů původních událostí. Tento přístup byl ovšem zamítnut při návrhu dalších částí Couplebreak API, jelikož by bylo nutné vždy znát identifikátory původních událostí, což by byla práce navíc, která by i mohla obnášet přístup k souboru, což by byla další práce navíc.

7.4 Vývojová dokumentace

Tato sekce hodlá předat čtenáři strukturu modifikace, podle které se lze v modifikaci orientovat. Zde bude představen seznam změněných souborů, tj. místa, kde se lze s kódem vylepšení setkat, a rozdělení modifikace do modulů, tj. abstraktnější dělení modifikace než pouze na funkce a datové struktury.

Názvy

Krátkou část ještě věnujme názvům přidaných prvků v kódu. Ačkoliv je každá změna ohraničena komentáři, které ji vyznačují, jsou prvky pojmenovány tak, aby čtenář ihned poznal, že něco souvisí s touto modifikací. To je dosaženo tím, že součástí jména nových prvků je řetězec `couplebreak`, často jako prefix. Tak je tomu v novém API, ale i v čistě implementačních funkcích, pokud nebyla značka v ohraničujícím komentáři sama ohraničena uvozovkami (viz vysvětlení ohraničujících komentářů 4.1.1).

7.4.1 Modifikované a nové soubory

Modifikace používá značku `COUPLEBREAK` v ohraničeních změn. Níže je abecedně seřazený seznam souborů spojených s modifikací spolu s krátkým popiskem změn či novinek uvnitř souboru.

- *KsMainWindow.hpp/cpp* - v těchto souborech byly o hlavního okna přidány grafické elementy pro ovládání Couplebreaku přes GUI.
- *KsWidgetsLib.hpp/cpp* - v těchto souborech byl definován nový widget, přes který se Couplebreak zapíná a vypíná pro jednotlivé streamy; zároveň se zde upravil widget filtrující události (třída `KsEventCheckboxWidget`).
- *KsUtils.hpp/cpp* - v těchto souborech byla definována pomocná C++ funkce k získání identifikátorů všech Couplebreak událostí aktivních ve streamu.
- *libkshark.h/c* - v těchto souborech byla upravena datová struktura pro datové streamy, datová struktura definující rozhraní streamů, inicializace hodnot při alokaci a konstrukci nového streamu a nakonec byla přidána definice nové funkce v rozhraní streamů pro získání identifikátorů všech Couplebreak událostí aktivních ve streamu.

- *libkshark-configio.c* - do tohoto souboru se přidalo ukládání stavu Couplebreaku do relací.
- *libkshark-couplebreak.h/c* - v těchto nových souborech jsou definovány identifikátory pro jednotlivé Couplebreak události jako makra, pozice indikátorů v bitové masce aktivních Couplebreak událostí ve streamech a pomocné funkce s Couplebreakem spojené.
- *libkshark-tepdata.c* - zde se Couplebreak záznamy vytvářejí a upravují; zároveň jsou zde upravené implementace rozhraní streamů a nastavování přidaných datových polí streamů.
- *sched_events.c* - zde byl plugin upraven tak, aby při aktivním Couplebreaku dokázal tuto modifikaci správně využít.

7.4.2 Struktura modifikace

Modifikaci rozdělíme na moduly. Modifikace obsahuje několik částí, které ji různě integrují do existujících modulů KernelSharku. Jejich seskupením do „integračního modulu“ bychom přišli o lepší rozdělení, proto budou integrační části samostatnými moduly.

Couplebreak se dá rozdělit do pěti modulů: Couplebreak API, integrace s datovými streamy, spolupráce s relacemi, spolupráce sched_events s Couplebreakem a konfigurace Couplebreaku.

- *Couplebreak API* - tento modul dodává nové konstanty a funkce, hlavně pro získávání informací. Jeho hlavní úlohou je pak abstrakce jistých operací nad záznamy Couplebreak událostí a zviditelnění částí Couplebreaku, aby další kód mohl s Couplebreakem pracovat, například pluginy.
- *Integrace do datových streamů* - Couplebreak se v tomto modulu spojuje s datovými streamy KernelSharku a funkcemi, které je využívají. Dodává nové funkce, upravuje implementace funkcí starých, aniž by porušil jejich původní funkcionalitu, a přidává proměnné do datových struktur streamů kvůli svému stavu.
- *Tvůrce záznamů Couplebreak událostí* - modul má na starost vytvářet záznamy Couplebreak událostí, opravovat jejich data, a spojovat je s původními událostmi.
- *Integrace do jednoduchých filtrů* - zde jsou upraveny jednoduché filtry tak, aby věděly o Couplebreak událostech a ty mohly být filtrovány. Úpravy se týkají GUI pro jednoduché filtry.
- *Integrace do relací* - tento modul se stará o ukládání Couplebreaku do relací KernelSharku a načítání Couplebreaku z těchto relací.
- *Konfigurace* - modul se zajímá hlavně o GUI okénko pro konfiguraci Couplebreaku a aplikaci změn provedených v tomto okně na streamy, kterých se to týká. Kvůli této povinnosti tento modul hlavně komunikuje s modulem integrace do datových streamů.

- *Spolupráce sched_events s Couplebreakem* - rozšíření oficiálního pluginu sched_events, dovoluje pluginu používat Couplebreak události a otevírá tak kompatibilitu pro další pluginy.

7.5 Uživatelská dokumentace

7.5.1 Instalace

Couplebreak je součástí zdrojového kódu KernelSharku a je ve více souborech, některé z nich jsou nové. Proto byl upraven soubor sestavovacích instrukcí CMakeLists.txt. Pokud již máte nainstalovaný KernelShark, tak jej nainstalujte znovu, je nutné, aby se sestavil i s novými soubory. Instrukce k instalaci KernelShark dodává sám KernelShark a jsou také popsány v kapitole o něm 3.

7.5.2 Uživatel GUI

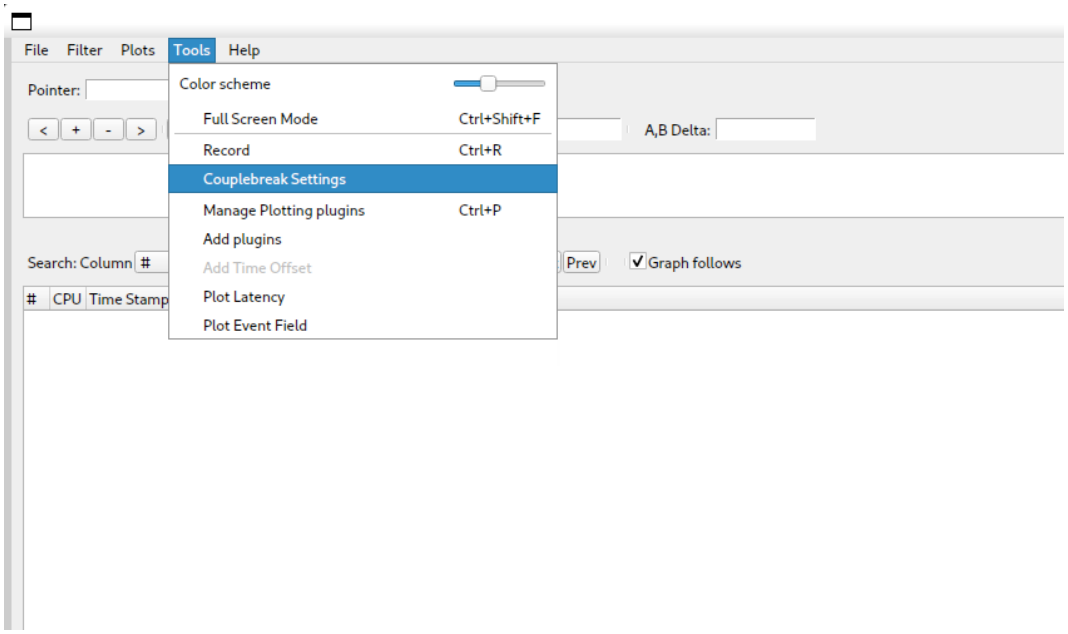
Uživatel GUI KernelSharku musí Couplebreak zapnout, modifikace je ve výchozím nastavení vypnutá. Stačí kliknout na nové tlačítko `Couplebreak Settings` v submenu `Tools` v menu KernelSharku, viz obrázek 7.1. Mělo by být umístěno pod tlačítkem `Record`. Po kliknutí se zobrazí konfigurační okénko, ve kterém se dá Couplebreak zapnout pro každý z otevřených streamů zvlášť, viz obrázek 7.2. Součástí okénka je i krátký vysvětlující text této modifikace, tlačítko `Apply` na použití vybrané konfigurace a tlačítko `Cancel` na zavření okénka bez provedení změn.

Zaškrtnutím políčka pro nějaký stream v konfiguračním okénku a následným použitím této konfigurace se data ve streamech načtou znovu. Toto také donutí pluginy aktivní v dotyčných streamech, aby se znovu načetly. Pluginy si tak mohou obnovit svůj kontext a použít nová data (tj. s novými záznamy Couplebreak událostí).

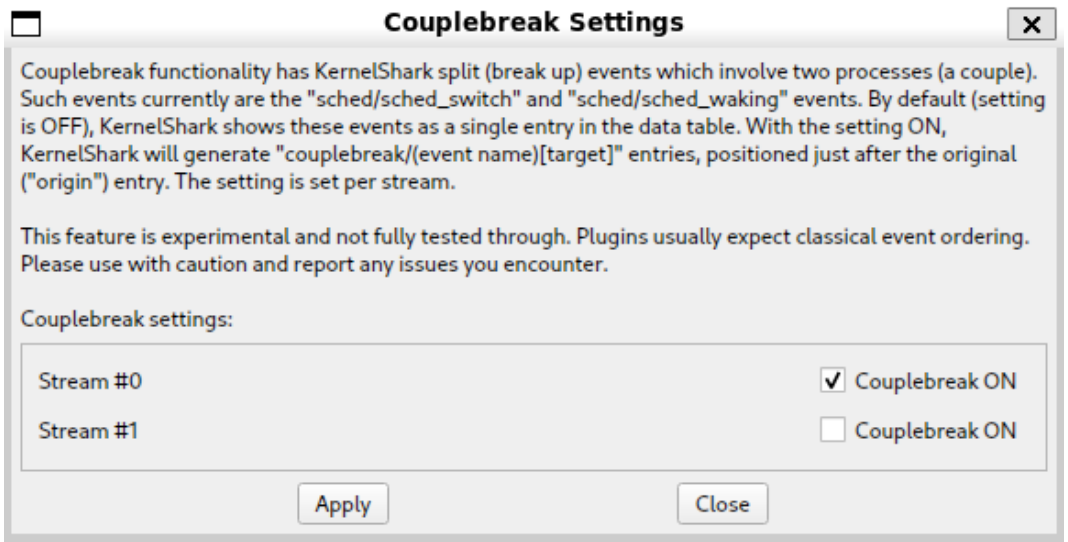
Záznamy Couplebreak událostí mohou být filtrovány jednoduchým filtrem stejně jako ostatní události, viz obrázek 7.3. Nelze je filtrovat pomocí pokročilých filtrů, ty nejsou podporovány. Záznamy jsou viditelné v seznamu událostí a i v grafu. Je možné, že budou zakrývat záznam své počáteční události, nebo budou naopak zakrývány takovým záznamem. Z hlediska rozlišení událostí to není velký problém, obě události jsou v podstatě to samé, jenom rozdělené. Pro vybrání zakrytého záznamu je pak nejjednodušší jej vybrat v seznamu událostí.

7.5.3 Vývojář

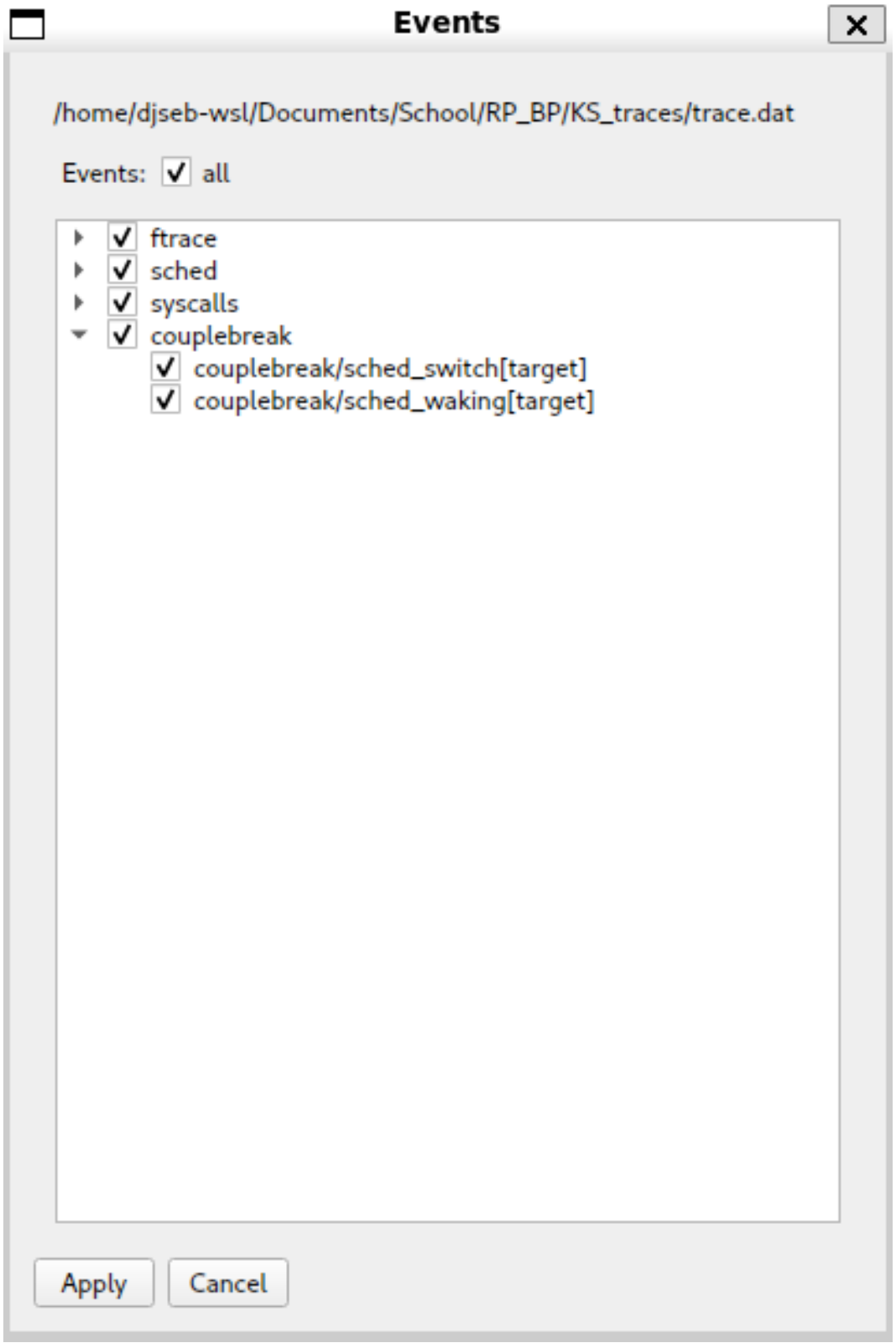
Modifikace se snaží být nerušivou součástí KernelSharku. Pokud budeme někdy chtít dále vyvíjet pro KernelShark, tak lze naše změny třeba nejprve navrhnout pro KernelShark bez aktivního Couplebreaku a až poté se pokoušet o integraci, pokud je nutná. Většinou by pak měl nový kód kontrolovat zda je Couplebreak aktivní přes proměnné ve streamu a měnit chování podle této informace. Samozřejmě existují i vylepšení, pro která nemá taková kontrola smysl, například rozšíření relací o nový datový formát - zde je nutné ukládat data Couplebreaku (alespoň zda byl ve streamu aktivní).



Obrázek 7.1 Submenu tools s modifikací tlačítkem pro konfiguraci Couplebreaku.



Obrázek 7.2 Dva streamy jsou otevřené, v jednom z nich je aktivní Couplebreak.



Obrázek 7.3 V jednoduchém filtru lze vybírat i Couplebreak události.

Vývojář pluginů pak může využít Couplebreak API a také kontrolovat přítomnost této modifikace ve streamu a dál tato data využívat. Modifikace by pak měla být respektována vždy, kdy se se očekává nějaký sled událostí a Couplebreak nějakou z nich dokáže dělit. Couplebreak ale může i pomoci s vytvořením kompatibility pro nějaké pluginy. Příkladem mohou být pluginy sched_events a *Naps*8. Oba z pluginů potřebují měnit procesy vlastnící nějaké události, aby mohly kreslit své tvary do grafu jednoho procesu. Pokud jsou oba pluginy aktivní bez zapnutého Couplebreaku, tak buď kreslí mezi špatnými záznamy, nebo na špatných místech. Oba totiž pohnuly se záznamy a narušili tak očekávání druhého z nich. S Couplebreakem nemusejí se záznamy hýbat, jelikož pro své potřeby si mohou vybrat záznamy od Couplebreaku.

Následující podsekce pak detailněji píší o hlavních prvcích, se kterými se nejspíše vývojář střetne, pokud bude s Couplebreakem pracovat.

Couplebreak API

K většině operací by měla posloužit nové API, kterou lze používat přes `#include`. S její pomocí lze kontrolovat identifikátor události u záznamu události, tj. zjistit, zdali se jedná o Couplebreak záznam. Také lze kontrolovat přítomné události v bitmasce ve streamu - pokud neznáme pozici indikátoru, API ji dokáže uživateli říci. Užitečnou funkcí je pak získání záznamu počáteční události z Couplebreak záznamu, nebo jen získání jména Couplebreak události.

Identifikátory Couplebreak událostí jsou pojmenované konstanty v makrech. Formát jména je `COUPLEBREAK_[zkratka]T_ID`, kde „[zkratka]“ je zkrácený název pro událost, pro kterou vytváříme novou Couplebreak událost, a T symbolizuje, že se jedná o cílovou/target událost. Příkladem, pro sched_waking událost je zkratka „SW“. Celý název konstanty identifikátoru je „COUPLEBREAK_SWT_ID“. Existují i makro konstanty pro pozice indikátorů v bitmasce, ale je doporučeno, aby se ke čtení bitmasky či její manipulaci používaly funkce k tomu určené, nikoliv tato makra.

Couplebreak ve streamech

Streamy KernelSharku nyní obsahují následující položky představující stav Couplebreaku:

- `couplebreak_on` - indikátor zapnutého či vypnutého Couplebreaku ve streamu.
- `n_couplebreak_evts` - počet vytvořených typů Couplebreak událostí, tj. pokud se ve streamu vytvořily pouze události pro sched_switch, bude počitadlo mít hodnotu 1, i kdyby bylo vytvořeno tisíc takových záznamů událostí.
- `couplebreak_evts_flags` - bitmaska, která zaznamenává, které typy událostí jsou ve streamu přítomné. Pozice indikátorových bitů lze najít v Couplebreak API. Maska zaznamenává od LSb.

Nedoporučuje se měnit jakoukoliv z položek výše, podobně jako se nedoporučuje měnit například původní položku ve streamu, počet opravdových zaznamenaných

událostí ve streamu `n_events`. Modfikace nemá definované chování, pokud se tyto položky změní. Čtení není nijak omezeno.

Každá funkce v rozhraní streamů může být používána jako dříve, i s Couplebreak událostmi. Pro Couplebreak většinou taková funkce bude pracovat s daty původního záznamu. Druhou možností je, že Couplebreak vytvoří požadované hodnoty na místě. To je hlavní rozdíl oproti normálním záznamům událostí, jelikož ty takovou hodnotu získávají ze souboru s trasovacími daty.

Záznamy Couplebreak událostí

Couplebreak události jsou přidány za běhu programu a nemohou nijak kontrolovat „původní data“, například původní název procesu, kterému událost patří. Dotazy na taková data jsou tak buď nasměrovány na záznamy událostí, kvůli kterým vznikly, nebo vytvořením dat nanovo (např. vytvoření jména události). Toto je způsobeno tím, co je v datech záznamů Couplebreak událostí, speciálně v jejich poli `offset`, kde není offset do trasovacího souboru, nýbrž ukazatel na původní záznam události. Z toho také vychází, že toto pole se nikdy nesmí pro záznamy Couplebreak událostí měnit. Kromě tohoto pole je zakázáno i jakkoliv měnit pole `event_id`, jelikož pro takový záznam je tato změna nevratná. Pokud jakékoliv z těchto dvou polí bude změněno, chování takového záznamu je nedefinováno, záznam lze považovat za rozbitý a již neobsahuje Couplebreak událost.

7.5.4 Bugy a chyby

Záznamy Couplebreak událostí by se neměly účastnit kreslení obdélníků mezi záznamy v grafu, jelikož nepředstavují opravdovou práci, nýbrž důsledek práce někde jinde. KernelShark ale nemá jak rozlišit tyto záznamy a obdélníky se kreslí i pro ně. Vylepšení *NoBoxes* se snaží toto chování opravit, nicméně neumí to dokonale.

Během vývoje KernelSharku dvakrát spadl při opravě CPU pro sched_waking[target] záznamy. Ačkoliv bug se již dlouho neobjevil, nebyl ani explicitně opraven. Pokud by KernelShark spadl při načítání trasovacích dat, je možné, že příčinou byl Couplebreak.

7.6 Rozšíření

Podpora více událostí

Přirozeným rozšířením této modifikace by bylo rozšíření Couplebreaku o více událostí, případně i o vytváření nějakých počátečních událostí. Existující kód Couplebreak událostí a všeho, co s jich týká, by měl dostatečně nastínit, jak postupovat při takových změnách.

Optimalizace průchodů daty při opravě CPU pro sched_waking

Couplebreak ve své stávající podobě donutí nasbíraná data, aby byla setříděna dle CPU a času do nějakého pole, ve kterém poté vyhledá konečná CPU pro probouzené události, tzv. opraví CPU. Stejné pole později vytváří ale i později další funkce, které jsou zavolány po naší opravě. Vytváříme tedy pole, které poté

zahodíme, ale po chvíli je vytvořeno znovu. Toto rozhodně představuje prostor pro optimalizace.

První možnou optimalizací je prosté neopravování CPU cílových událostí. Zkrátka volání opravy neprovedeme. Tím ale také přijdeme o informaci, kde nakonec byl proces probuzen. Je to ovšem nejjednodušší řešení.

Druhou možností je použít trik podobný hledání záznamů zásobníku v pluginu *Stacklook*. Vytvořili bychom plugin, který cílovým událostem CPU opraví v jednom průchodu při prvním pokusu o kreslení a dále se nevolá, pokud není plugin načten znovu. Nápad to není špatný, ale bylo by nutné pak řešit situace, kdy by některé jiné pluginy nebo části KernelSharku mohly očekávat původní CPU.

Třetí možností je ponořit se vnitřností KernelSharku a upravit místa, kde se pole vytváří a místa, kde zanikají. Pokud bychom opravili CPU, tak bychom si možná mohli na pole uschovat ukazatel a nějakým způsobem ho předat funkcím, které by pak nemusely vytvářet nové setříděné pole. Toto by byla čistá optimalizace a nebylo by nutné oddělovat opravy CPU od Couplebreaku.

Couplebreak v datovém streamu

Streamy mají v současnosti tři proměnné, které dohromady symbolizují stav Couplebreau ve streamu: počet typů vytvořených Couplebreak událostí, indikátor aktivního Couplebreaku a bitmasku vytvořených typů. Navrženým rozšířením je úprava, a to pouze na dvě proměnné. Jelikož máme omezeně mnoho bitů v bitmasce, lze v konstantním čase spočítat i počet vytvořených typů událostí z této proměnné. Tím bychom mohli z datové struktury odstranit počet Couplbreak událostí a místo toho toto číslo vždy rychle spočítat. Pokud bychom se chtěli opravdu omezit na minimum proměnných, bylo by možné použít pouze jednu bitmasku, kde jeden bit by speciálně označoval aktivní Couplebreak.

Toto rozšíření ale není moc užitečné v desktopovém prostředí, kde GUI aplikace obyčejně běží. Streamů KernelShark nevytváří mnoho a tak na paměti moc neušetříme. Není pro nás ani zajímavá časová stopa, protože desktopová prostředí většinou mají i dostatečné zdroje na rychlé výpočty, zvláště pokud se jedná o práci s bity.

Jediným dobrým důvodem pro rozšíření by bylo sjednocení proměnných streamu do jedné struktury. Tím bychom sice paměti nepomohli, ale ve streamu by bylo méně proměnných a stav Couplebreaku by byl více zapouzdřen, než doposud.

Funkce `get_*_event_ids` a širší podpora pro umělé události

Couplebreak přidal `get_couplebreak_evt_ids` („Couplebreak“ funkce) do rozhraní streamu. Vytvořena byla proto, že úprava `get_all_event_ids` („all“ funkce) pro spolupráci s Couplebreakem by způsobila příliš mnoho úprav chování v místech, kde byla použita. Tato použití někdy vůbec nepotřebují o umělých událostech vědět.

S „Couplebreak“ funkcí je ale funkcionalita „all“ funkce narušena - nyní už nevydá všechny události ve streamu, nýbrž jen všechny události, které nejsou vytvořeny Couplebreakem. Bylo by proto lepší, kdyby „all“ funkce byla alespoň přejmenována, aby lépe odrážela svou práci, například na `get_all_original_event_ids`. Poté bychom i mohli dodat funkci, která by vrátila opravdu všechny události.

Dobré by pak bylo projít celý kód KernelSharku a rozhodnout zdali jsou na místech použití „all“ funkce potřeba všechny události, nebo pouze originální události.

Celkově by bylo hezké dodat KernelSharku obecnou podporu pro práci s umělými událostmi, nejen pro události od Couplebreaku. Ten je sice jediným výrazným producentem umělých událostí, ale nemusí tomu tak být napořád, mohou přibýt další vylepšení, která budou umělé události využívat.

Rozdělování pouze některých událostí

Možným rozšířením je rozdělování pouze některých událostí, které bychom si vybrali v konfiguraci. Tím by bylo rozdělování jemnější. Nicméně by také bylo nutné refaktorovat části, které kontrolují aktivní Couplebreak, například pokud by nás zajímalo jenom to, zdali se rozdělují události typu sched_switch. Zároveň by toto znamenalo indikátory pro každou událost, na což by mohla být opět použita bitmaska.

Podpora pokročilých filtrů

Couplebreak události mohou být filtrovány jen pomocí jednoduchých filtrů. Ale bylo by možné (i příjemné) mít možnost je využít v pokročilých filtrech. S nimi by pak měl uživatel detailnější kontrolu nad zobrazováním záznamů a navíc by tak byla odstraněna tato nesrovnalost vůči ostatním událostem, které tyto filtry podporují.

7.7 Kritika

V této sekci autor kriticky zhodnotí své řešení. Každá kritika má nadpis a popis a dále buď obsahuje obranu proti ní, nebo souhlas s předneseným problémem.

Změny ve stream API

Tato podsekce očekává, že čtenář viděl kód Couplebreaku.

Změna funkcí ve stream API je trochu nepříjemná. Kvůli missed_events již KernelShark představuje nějaký způsob, jak zpracovávat vlastní umělé události. Couplebreak ale občas potřebuje speciální zpracování a tak je někdy cesta položená KernelSharkem opomíjena. Zpracovávání žádosti na stream API pak prochází přes sekci pro normální událost, pak sekci pro Couplebreak události a až poté se dostane k sekci pro obecné vlastní události, kde by dle původní implementace mělo probíhat zpracování. Couplebreak tak občas nerespektuje záměry kódu, což může čtenáři kódu působit tu trochu nepříjemnosti. Příkladem by mohla být funkce `tepdata_dump_entry`, kdy se zpracování pro Couplebreak stane před switch konstrukcí, ve které se očekávalo zpracování vlastních umělých událostí. Zde je přítomno i vysvětlení, proč je Couplebreak řešen jinak.

7.8 Zhodnocení splnění požadavků

Modifikace není v žádném z otevřených streamů aktivní, dokud není zapnuta skrze nové konfigurační okénko. KernelShark tak vůbec s Couplebreakem nemusí

pracovat. Plugin sched_events také pracuje jako dříve, pokud ve streamu, kde plugin působí, není Couplebreak zapnutý. Jedinou viditelnou změnou, která se koná vždy, je ukládání relací. Zde Couplebreak nutně ukládá, zdali je v nějakém streamu aktivní a každý soubor relace pak tuto informaci obsahuje. Ale načítání relací v původní implementaci nepracuje s Couplebreakem, a pokud je načtena starší relace, Couplebreak je vypnutý. Tím splňujeme obecný požadavek *minimálního vlivu* modifikací. Požadavek *stylové podobnosti* se musí ověřit čtením kódu provedených změn. Rychle lze ale například ověřit, že Couplebreak se snaží používat podobné formáty jmen pro prvky v C kódu a podobné formáty jako v C++ kódu. Požadavek o *chování se jako rozšíření* je též splněn - pokud je někde změna původního chování, je popsána v komentáři, například při opravách CPU u cílových sched_waking událostí. Navíc Couplebreak vždy kontroluje, zdali je zapnutý, pokud tomu tak není, pak se KernelShark chová jako dříve. Ostatní změny jsou triviálně pouze rozšiřující, jako nová funkce pro stream API, Couplebreak API s novými funkcemi, nebo nové proměnné v datové struktuře streamů.

Vlastní cíle Couplebreak plní, při analýze bylo řešení sestaveno tak, aby cíle splňovalo.

8 Naps

V této kapitole rozebereme plugin pro KernelShark, se kterým budeme schopni v grafu zobrazit dobu mezi přepnutím nějaké úlohy a jejím probouzením od nějakého jiného procesu. Termínem „nap“ pak označujeme právě období nečinnosti nějakého procesu. V kapitole se seznámíme s cíli, analýzou řešení, návrhem a použitím tohoto pluginu. Nakonec i řešení kriticky zhodnotíme a představíme návrhy pro rozšíření.

8.1 Cíle

- Mezi událostmi sched_switch procesu P1 a sched_waking procesu P2, který probouzí P1, se bude vykreslovat obdélník a text. Vykreslený text bude název předchozího stavu P1 před přepnutím. Vykreslený obdélník bude měnit svou barvu dle předchozího stavu.
- Plugin musí dostat sched_waking události do grafů procesů, které jsou těmito událostmi probouzeny, aby mohl své grafické objekty vykreslovat.
- Plugin bude spolupracovat s vylepšením Couplebreak. Namísto využívání sched_waking událostí se využijí cílové události probouzení. Spolupráce musí být automatická, tj. pokud je zapnut Couplebreak, plugin ho používá, a pokud je Couplebreak vypnutý, plugin jej nepoužívá - to vše bez vstupu od uživatele.
- Plugin bude možné konfigurovat skrze grafické okénko. Minimální konfigurovatelné nastavení bude maximální počet záznamů viditelných v grafu, než se plugin aktivuje.

8.2 Analýza

Tato sekce zachytí postup postup, kterým se dostaneme k implementaci řešení. Jedná se o techničtější a detailnější analýzu než analýza z kapitoly *Obecná analýza a stanovení požadavků*. Tato sekce bude často odkazovat na plugin Stacklook a jeho analýzu, abychom se vyhnuli zbytečnému opakování.

8.2.1 Propojení s KernelSharkem

Přeskočíme hlubší přemýšlení nad tím, jak připojit plugin do KernelSharku, to jsme již udělali u analýzy kapitoly pluginu Stacklook 6 a stejný postup můžeme jednoduše upravit pro naše účely. V rychlosti ale zopakujeme, že plugin s KernelSharkem propojíme pomocí handlerů pro kreslení, vytváření menu nebo načítání dat. Pluginy mají kontext, ve kterém jsou proměnné pro daný plugin globální, takový kontext si sami definujeme. Implementaci propojujícího modulu rozdělíme do C kódu, který se bude starat hlavně o propojení s KernelSharkem a správné přiřazování handlerů, a do C++ kódu, který se bude starat o složitější logiku fungování pluginu.

Jediným opravu podstatným rozdílem bude handler pro načítání událostí. Nyní nebudeme pouze vybírat pro nás zajímavé záznamy událostí, kterými jsou sched_switch a sched_waking. Pro události typu sched_waking je upravíme i tak, že proces, který událost vlastní, změníme z procesu probouzejícího na proces probouzený. Tím budeme pak schopni kreslit obdélníky mezi přepnutím kontextu z procesu až po jeho probouzení. Postup je velmi podobný pluginu sched_events, který takto přesouvá pro své účely události sched_switch.

Již zde je jasné, že pluginy si bez Couplebreaku neporadí a budou si vzájemně škodit, budou-li aktivní ve stejném streamu. V kapitole o Couplebreaku jsme sched_events vylepšili o možnost namísto sched_switch událostí vybírat události od Couplebreaku. Naps také bude obsahovat kód, který bude schopen Couplebreak využít. Takto bude i pěkně předvedena schopnost Couplebreaku být mostem pro kompatibilitu některých pluginů. Jediné, co my musíme přidat, je kontrola, zdali je ve streamu Couplebreak aktivován, která proběhne před určením událostí, které nás budou zajímat. Pokud je Couplebreak aktivní, tak se nebudeme zajímat o události sched_waking, nýbrž o události sched_waking[target]. Takto jsme stejně jako u pluginu sched_switch odstranili nutnost měnit data trasovaných událostí a plugin žádným dalším pluginům nemůže škodit reorganizací záznamů událostí.

8.2.2 Kreslení obdélníků do grafu

KernelShark dodává API na kreslení objektů, ovšem my bychom nyní chtěli kreslit pomocí dvou záznamů v grafu, nikoli pouze pomocí jednoho, jako u Stacklooku. KernelShark má i takovou situaci vyřešenou pomocí „intervalového kreslení“, tj. kreslení mezi dvěma záznamy, které přirozeně v grafu vytváří nějaký časový interval. Nám pak opět stačí jenom dodat nějakou kreslící funkci s danou signaturou, pomocí níž KernelShark obdélník nakreslí. Narozdíl od Stacklooku ale budeme kreslit obdélníky pouze v grafech procesů, jelikož doba nečinnosti procesu se zřejmě týká jenom tohoto procesu.

8.2.3 Nap obdélníky

Nyní nám už jen stačí obdélníky navrhnout. Název „nap obdélník“ označuje právě obdélník vytvořený tímto pluginem. Z cílů nám vychází, že musejí být dostatečně velké pro text a zároveň je dáno, že barva musí být spojena s předchozím stavem. Nejjednodušším řešením zde bude nějaká konstantní mapa předchozích stavů na barvy. Aby byly barvy jednoduše rozlišitelné, tak budeme pracovat s barvami světlými i tmavými. Přidáme ještě barvení vrchní a spodní strany obdélníku podle barvy procesu - k tomu nám pomůže dodatečné vylepšení *Get Colors*. Toto barvení nebude ve výchozím nastavení zapnuto, tj. obdélníky nebudou mít žádné hrany obarvené. Cílem tohoto vedlejšího vylepšení je jak ukázka *Get Colors*, tak o trochu hezčí prezentace obdélníků.

Obdélníky budou pak muset text zobrazovat dle světlosti barvy buď v černé nebo v bílé. Text umístíme někam doprostřed obdélníku. Je možné, že dva záznamy, mezi kterými má být obdélník, budou příliš blízko. Proto budeme text zobrazovat jenom pokud obdélníky budou dostatečně široké, to vypočteme pomocí délky textu k zobrazení. Podobně jako u Stacklooku získáme data pro text z informací v sched_switch události. Nebudeme ale vytvářet samostatný mini-modul jako u

Stacklooku, získání stavu bude pouze součást vytváření obdélníku.

Obdélníky nebudou mít definované interakce s uživatelem, není to součást našich cílů, tedy nebudou reagovat ani na dvojité kliknutí, ani na přejetí kurzorem myši.

8.2.4 Konfigurace

Zde využijeme dělení konfigurace ze Stacklooku na konfigurační objekt a GUI okénko, které je jediným místem pro manipulaci konfiguračního objektu. Také dáme uživateli možnost konfigurovat počet viditelných záznamů v grafu, než se plugin spustí. A podobně jako u Stacklooku přidáme zaškrtávací tlačítko pro použití funkcionalit od *Get Colors*. Žádné další konfigurační možnosti nedodáme, jelikož nejsou potřeba. Pokud sestavíme Naps pro nemodifikovaný KernelShark, tak zaškrtávací políčko nebude součástí konfigurace, nemělo by pak, co měnit.

8.2.5 Plugin pro nemodifikovaný KernelShark

Stejně jako Stacklook, i Naps využije podmíněnou kompilaci pomocí makra `ifndef` s názvem `_UNMODIFIED_KSHARK`. Zde se bude pouze týkat barvení hran nap obdélníků.

8.3 Vývojová dokumentace

Dokumentace pluginu

Dokumentace je napsána pro nástroj Doxygen. Dokumentována byla každá funkce i proměnná, ale vygenerovaná dokumentace obsahuje jen prvky veřejné. Pro implementační detaily je doporučeno se podívat do zdrojového kódu. Dokumentace obsahuje i hlavní stránku a stránku s nástinem návrhu.

Struktura projektového adresáře

Adresář pluginu obsahuje další adresáře. Adresář „src“ je pro zdrojový kód a adresář „doc“ je pro dokumentaci uživatelskou a dokumentaci technickou. Očekává se, že na této úrovni jsou i adresáře pro sestavení. Na stejné úrovni žije i README, soubor s licencí a nejvyšší CMakeLists.txt. V těchto CMake instrukcích se nastaví proměnné sestavení, například typ sestavení, a případně se zavolá generování dokumentace. CMake instrukce zodpovědné za vytvoření binárního souboru jsou v adresáři se zdrojovým kódem, stejně jako to dělá KernelShark.

Přehled modulů pluginu

- *Propojující modul* - Modul s kódem propojujícím KernelShark a plugin. Obsahem je hlavně kontext pluginu, funkce kontextu, handlery a implementačně pomocné funkce. Součástí tohoto modulu je část s C kódem a implementační část v C++ prvků z hlavičkového C souboru. Právě v „C++ části“ (napsaná v C++) se ostatní moduly propojují; zároveň tato část

ukládá některá globální data s C++ typy. Soubory modulu jsou *naps.h/c* a *Naps.cpp*.

- *Konfigurace* - Modul se skládá ze dvou tříd, konfigurační objekt a konfigurační okénko. Konfigurační objekt obsahuje konfigurační data, která plugin zrovna využívá a je navržen jako singleton. Konfigurační okénko představuje GUI element, kterým se data v konfiguračním objektu manipulují. Soubory modulu jsou *NapConfig.hpp/cpp*.

- *Nap obdélníky* - Modul se zajímá vzhled a kreslení nap obdélníků mezi záznamy. Součástí vzhledu je barva, barva vrchní a spodní hrany a text v obdélníku, tj. jeho barva, velikost a obsah. Soubory modulu jsou *NapRectangle.hpp/cpp*.

8.4 Uživatelská dokumentace

Tato sekce popíše jak instalovat a používat plugin Naps v KernelSharku a co od něj během běhu očekávat, či na co si dát pozor. Malá ochutnávka fungujícího pluginu je na obrázku 8.1. Styl a i obsah této dokumentace jsou velmi podobné uživatelské dokumentaci Stacklooku, jelikož jsou efektivní a dají se jednoduše poupravit na jiný plugin.

8.4.1 Instalace

Předpoklady

- CMake verze alespoň 3.1.2.

- Pokud chceme plugin využívající modifikovaný KernelShark, pak je nutná verze alespoň 2.4.0-couplebreak. Pokud chceme plugin pro nemodifikovaný KernelShark, pak je nutná verze alespoň 2.3.2.

- Závislosti KernelSharku (nalezneme v README repozitáře KernelSharku), zejména Qt6 a traceevent.

- Doxygen na technickou dokumentaci.

Tento plugin funguje výrazně lépe, pokud je KernelShark modifikovaný. V takovém KernelSharku je plugin plně kompatibilní s oficiálními pluginy, pokud je Couplebreak zapnutý. V případě nemodifikovaného KernelSharku pak tento plugin vyžaduje vypnutí pluginu sched_events, jinak nebude správně fungovat.

Sestavení a instalace pouze pluginu

1. V terminálu nastavme pracovní adresář na složku `build` (pokud ještě neexistuje, pak ji nejlépe vytvořme v kořenovém adresáři projektu).

2. Spusťme příkaz `cmake ...` Pokud hlavní soubor `CMakeLists.txt` není v nadřazené složce, předejme programu CMake platnou cestu k němu.

- Používáme-li verzi KernelSharku bez modifikací, přidejme do příkazu argument `-D_UNMODIFIED_KSHARK`. Sestavení pro nemodifikovaný KernelShark odstraňuje možnost barvit vrchní a spodní hranu obdélníků kreslených pluginem barvou úlohy, které obdélník patří.
- Pokud chceme generovat dokumentaci pomocí Doxygenu, přidejme do příkazu argument `-D_DOXYGEN_DOC=1`.
- Výchozí typ sestavení je `RelWithDebInfo`. Chceme-li ho změnit (např. na `Release`), použijme argument `-DCMAKE_BUILD_TYPE=Release`.
- Pokud se v `/usr/include` nenacházejí hlavičkové soubory knihovny traceevent, použijme argument `-D_TRACEEVENT_INCLUDE_DIR=[PATH]`, kde `[PATH]` nahradíme cestou k souborům knihovny.
- Pokud se v `/usr/lib64` nenacházejí sdílené objekty knihovny traceevent, použijme argument `-D_TRACEEVENT_LIBS_DIR=[PATH]`, kde `[PATH]` nahradíme cestou ke sdíleným objektům knihovny.
- Pokud se soubory Qt6 nenacházejí ve `/usr/include/qt6`, použijme argument `-D_QT6_INCLUDE_DIR=[PATH]`, kde `[PATH]` nahradíme cestou k souborům Qt6.
  - Pokyny pro sestavení předpokládají, že zadaný adresář má stejnou vnitřní strukturu jako výchozí možnost (tj. obsahuje složky QtCore, QtWidgets apod.).
- Pokud se zdrojové soubory KernelSharku nenachází v `../KS_fork/src`, použijme argument `-D_KS_INCLUDE_DIR=[PATH]`, kde `[PATH]` nahradíme cestou ke zdrojovým souborům KernelSharku.
- Pokud se sdílené knihovny KernelSharku (`.so` soubory) nenachází ve `/usr/local/lib64`, použijme `-D_KS_SHARED_LIBS_DIR=[PATH]` argument, kde `[PATH]` nahradíme cestou k sdíleným knihovnám KernelSharku.

3. Ve složce `build` spusťme příkaz `make`.
   - Pokud je třeba sestavit jen část pluginu, například pouze dokumentaci, můžeme vybrat konkrétní cíl.
   - Pouhé spuštění `make` vytvoří: *plugin* (cíl `naps`), *symlink* na sdílený objekt pluginu (cíl `naps_symlink`) a případně *Doxygen dokumentaci* (cíl `docs`), pokud tak bylo specifikováno v předchozím kroku.
4. (*Instalace*): Nahrajme plugin do KernelSharku, buď přes GUI, nebo při spouštění přes CLI s argumentem `-p` a cestou k symlinku nebo přímo k sdílenému objektu.
   - *DŮLEŽITÉ*: Vždy nainstalujme/nahrajme plugin před načtením relace, ve které byl aktivní! Jinak může dojít k neúplnému načtení konfiguračního rozhraní nebo k pádu celého programu.

K odstranění vytvořených binárních souborů použijme `make clean`.

Sestavení a instalace pomocí KernelSharku

1. Ujistěme se, že všechny zdrojové soubory (`.c`, `.cpp`, `.h`) pluginu Naps se nacházejí ve složce `src/plugins` v adresáři projektu KernelShark.

2. Zkontrolujme, že soubor `CMakeLists.txt` v této podsložce obsahuje instrukce pro sestavení pluginu (inspirovat se můžeme podle jiných pluginů pro GUI). Pokud chceme sestavovat pro nemodifikovaný KernelShark, upravme tomu odpovídajícím způsobem build skript.

3. Sestavme KernelShark (pluginy se sestavují automaticky). Lze sestavit i pouze plugin, pokud jsme již předtím vytvořili instrukce sestavení.

4. (*Instalace*): Spusťme KernelShark. Pluginy sestavené tímto způsobem se načítají automaticky. Pokud by se z nějakého důvodu nenačetly, najděme sdílený objekt stejně jako u ostatních oficiálních pluginů, opět buď přes GUI, nebo přes CLI.

VAROVÁNÍ - načítání více verzí pluginu

Máme-li dvě nebo více sestavených verzí pluginu, *NE*načítejme je současně do KernelSharku. Pokud to uděláme, *DOJDE K PÁDU PROGRAMU*. Používejme vždy jen jednu z verzí, *NIKDY OBOJE NAJEDNOU*.

8.4.2 Naps v GUI

Jak zapnout Naps

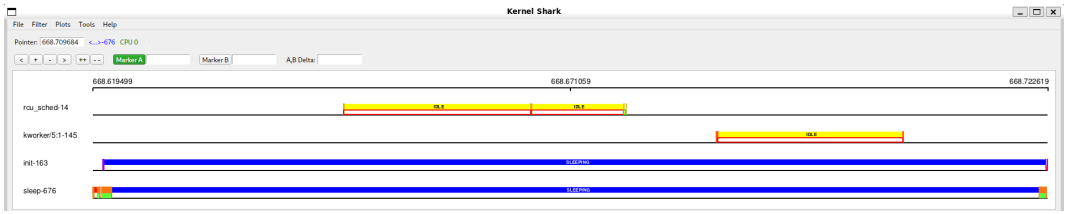
Plugin se zapíná velmi jednoduše. Stačí spustit KernelShark a přejít na položku v panelu nástrojů `Tools > Manage Plotting plugins`. Pokud byl plugin načten přes příkazový řádek, zobrazí se v seznamu pluginů jako zaškrtnuté políčko se svým názvem. Pokud ne, lze plugin dohledat pomocí tlačítka `Tools > Add plugin` - stačí nalézt symlink, ale je možné vybrat i samotný soubor sdíleného objektu. Obrázek 8.2 ukazuje GUI pro zapínání a vypínání pluginů.

Konfigurace

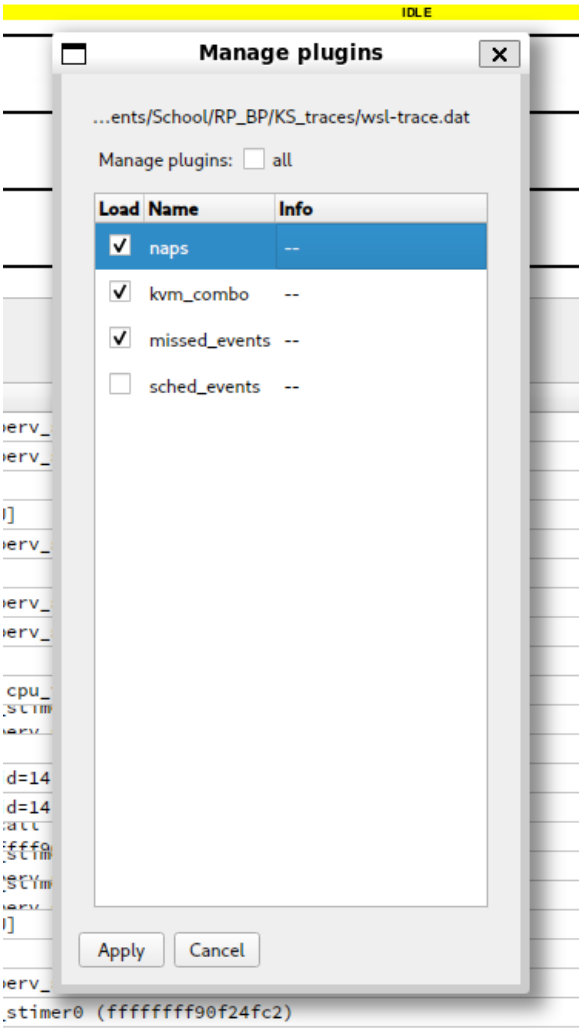
Konfigurace pluginu může být provedena kdykoliv, i před načtením jakýchkoliv trasovacích dat. Pro otevření konfiguračního okna (viz obrázek 8.3) stačí v hlavním okně zvolit `Tools > Naps Configuration`, viz obrázek 8.4. Vždy může být otevřeno jen jedno konfigurační okno.

V konfiguračním dialogu lze pluginu přenastavit dvě věci. První je maximální počet viditelných záznamů v grafu trasování KernelSharku, po jehož překročení plugin nebude nic do grafu kreslit. Pokud je hodnota moc vysoká, pak dovolujeme pluginu strávit pro více událostí více času kreslením. Hodnotu je dobré snížit i tehdy, pokud pociťujeme zpomalení výkonu při kreslení s mnoha záznamy. Výchozí hodnota tohoto nastavení je 10 000 (deset tisíc) záznamů.

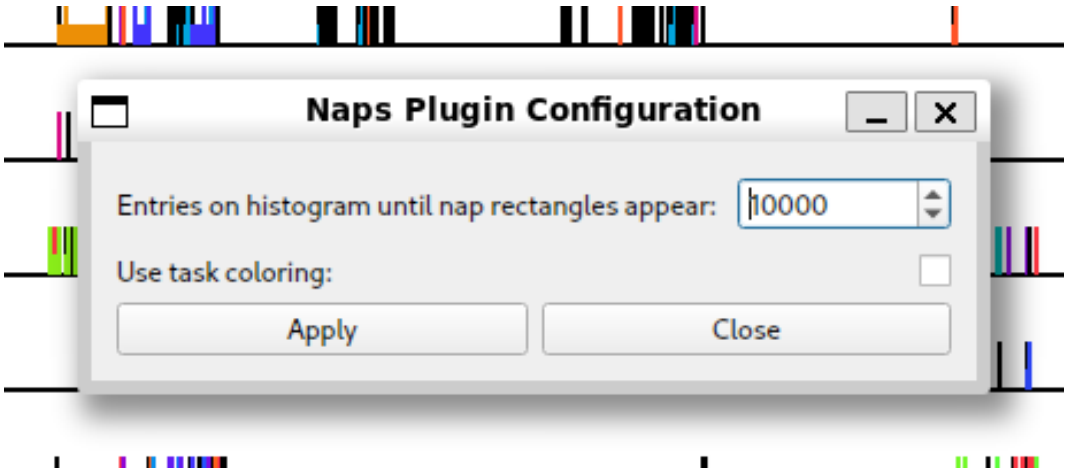
Druhou věcí, kterou lze konfigurovat, je barvení vrchní a spodní hrany obdélníků vykreslených pluginem barvou, kterou KernelShark využívá pro daný proces, v jehož grafu je obdélník kreslen (viz obrázek 8.6). Toto se dá řídit zaškrtávacím políčkem. Ve výchozím nastavení je možnost vypnutá a je použita stejná barva,



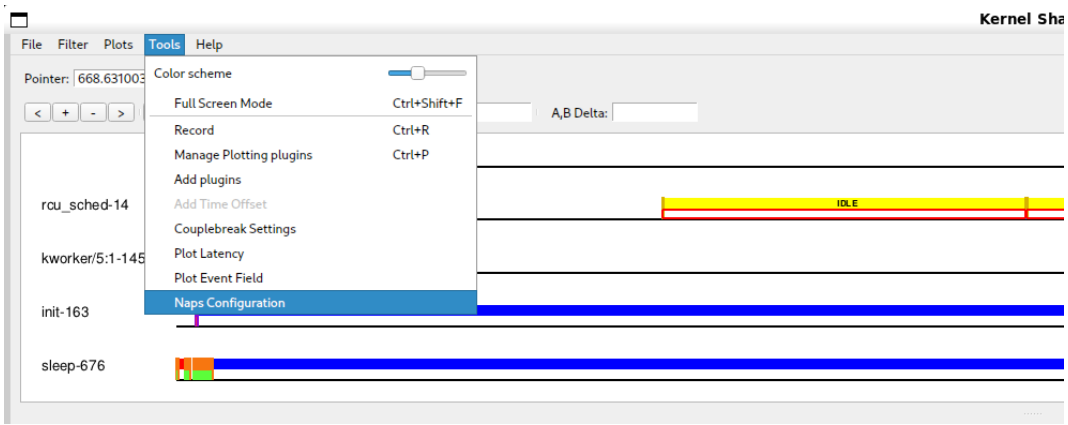
Obrázek 8.1 Fungující Naps



Obrázek 8.2 Okénko se správou pluginů se záznamem pro plugin Naps



Obrázek 8.3 Konfigurační dialog pro plugin Naps



Obrázek 8.4 Tlačítko na vyvolání konfiguračního dialogu pluginu Naps

jako pro vnitřek obdélníku, viz obrázek 8.5. Pokud byl ale Naps sestaven pro nemodifikovaný KernelShark, toto políčko není v konfiguraci přítomno.

Kliknutím na tlačítko `Apply` provedené konfigurační změny potvrdíme a zavřeme tak i dialog. Zobrazí se i zpráva, že změna byla úspěšná, viz obrázek 8.7. Kliknutím na tlačítko `Close` nebo na křížek v hlavičce dialogu naopak změny zahodíme, ale poté také zavřeme dialog. Změny takto zahozené se při znovuotevření dialogu nevrátí, hodnoty ke konfiguraci budou stejné, jako hodnoty aktuální konfigurace.

Plugin si konfiguraci nikam sám neukládá a nespolupracuje s relacemi KernelSharku. Změny provedené před zavřením programu budou muset být znovu nastaveny při dalším otevření.

Naps v grafech

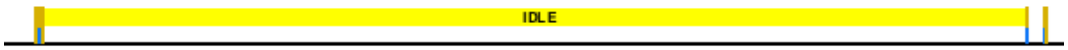
Zobrazená „zdřímnutí/naps“ procesů jsme již viděli na začátku dokumentace. Tato sekce je trochu více představí.

K zobrazení ob nečinnosti v grafu trasování je nutné mít v tomto grafu grafy procesů. Plugin pak automaticky nakreslí barvené obdélníky mezi událostmi sched_switch a sched_waking/sched_waking[target]. Barva je určena předchozím stavem procesu před přepnutím, například nepřerušitelný spánek je červený. Pokud jsme tak plugin nakonfigurovali, tak vrchní a spodní hrany obdélníků mají stejnou barvu jako proces a jeho záznamy. Pokud je obdélník dostatečně široký, tak je v něm i předchozí stav napsán velkým písmeny celým názvem. Pro příklad obdélníků s různými šířkami se podívejme na obrázek 8.8. S obdélníky nelze nijak interagovat. Obdélníky budou viditelné tak dlouho, dokud dovolí přiblížení dvou záznamů tvořících obdélník, aby byly také viditelné.

![Fig. 8](../images/NapsDifferentWidths.png) Figure 8.

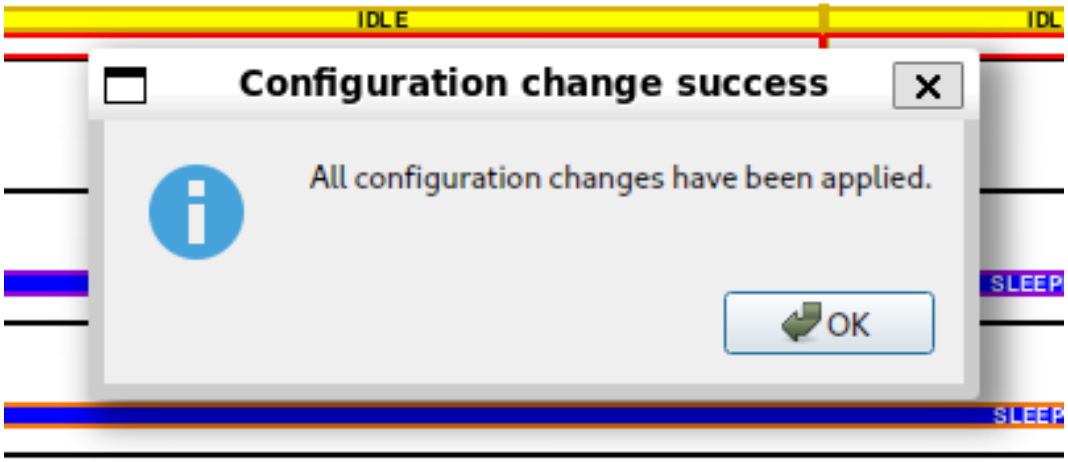
Níže je seznam předchozích stavů (s anglickými názvy) a barev s nimi spjatých. Přidána jsou i krátká vysvětlení těchto stavů.

- Uninterruptible (disk) sleep - červená. Proces čeká na dostupnost zdrojů a nereaguje na signály.
- Idle - žlutá. Pro speciální vlákno kernelu, nelze převést do stavu Running.
- Parked - oranžová. Pouze pro procesy kernelu, proces se dobrovolně vzdá CPU a označí se tímto stavem, aby mohl být spuštěn později.
- Running - zelená. Proces běží na nějakém CPU.
- Sleeping - modrá. Proces čeká na dostupnost zdrojů a reaguje na signály.
- Stopped - azurová. Procesu byl zaslán STOP signál.
- Tracing stop - hnědá. Proces se zastavil, jelikož je právě trasován či laděn.
- Dead - magenta/purpurová. Přechodný stav těsně předtím, než bude proces dealokován.
- Zombie - fialová. Proces skončil svou práci a čeká na uklizení rodičovským procesem.



Obrázek 8.5 Hrany obdélníku mají stejnou barvu, jako jeho vnitřek

Obrázek 8.6 Hrany obdélníku používají barvu procesu



Obrázek 8.7 Dialog signalizuje úspěšnou změnu konfigurace pluginu

Obrázek 8.8 Obdélníky, které nejsou dostatečně široké, nezobrazí název předchozího stavu

8.4.3 Bugy a chyby

Žádné nejsou známy.

8.4.4 Doporučení

Autor níže vypisuje pár doporučení při používání pluginu.

- Vždy načtěme Naps před načtením relace. Může to programu ušetřit nepříjemná překvapení.
- Pokud chceme dvě verze pluginu, sestavme je do různých adresářů.
- Opakujeme, že je doporučeno používat verzi pluginu s KernelSharkem, který obsahuje Couplebreak.
- Nedoporučuje se nastavovat příliš vysoký limit záznamů v grafu v konfiguraci. Jinak by plugin mohl používat příliš mnoho paměti kvůli velkému množství naráz kreslených nap obdélníků.
- I když relace v KernelSharku fungují, jsou trochu nestabilní. Tento plugin se snaží jejich vnitřní logiku nenarušovat, ale varuje, že pokud plugin není načten předem, mohou nastat neočekávané problémy. Například načtení relace s aktivním pluginem nepřidá do menu `Tools` odpovídající položku pro vyvolání konfiguračního okna.

8.5 Rozšíření

Interakce

Obdélníky fungují dobře, ale možná by bylo dobré přidat nějakou interakci, která by nám o době nečinnosti dala další informace. Například by při přejetí myši mohl informační řádek KernelSharku, nebo nějaká plovoucí vysvětlivka ukázat i rozdíl mezi časem přepnutím a probouzením. Nebo by mohlo být při dvojitém kliknutí zobrazeno okénko, ve kterém jsou nějaké statistiky a detaily o době nečinnosti. Přímé rozšíření tedy nedodáváme, ale tato nastínění snad ukazují, že rozšířit tento jednoduchý plugin lze.

Persistentní konfigurace

Stejně jako u Stacklooku, i zde by bylo pěkné mít schopnost ukládat si konfiguraci z nějaké relace KernelSharku. Zde je problém trochu mírnější, jelikož máme méně věcí, co lze konfigurovat. Ale i tak by bylo lepší nemuset při každém otevření KernelSharku a načtení pluginu manuálně konfiguraci měnit.

8.6 Kritika

Plugin je vcelku malý a autorovi se zdá, že funguje docela bezproblémově. Tu kritiku, kterou má, sdílí se Stacklookem, tj. *Konfigurační singleton* a *Verze pluginu pro nemodifikovaný KernelShark*. I zde platí ty stejné argumenty jako pro Stacklook.

8.7 Zhodnocení splnění požadavků

Plugin zřejmě splňuje první z obecných cíle pluginů, tj. vlastní adresář s instrukcemi k sestavení. Nezávislost na jiných pluginech není dokonale splněna, pokud plugin neběží v prostředí s Couplebreakem. Pokud tomu tak není, tak si s pluginem sched_events vzájemně škodí a jsou tak na sobě závislé tak, že se vzájemně zakazují. Tomu se bohužel nedá předejít, jelikož není jiný způsob, jak by každý z nich dokázal odvést svou práci.

Jinak jsme v analýze postupovali tak, abychom každý z našich cílů splnili. Navíc jsme nepřišli na žádné další problémy při vytváření řešení. Tím jsme splnili i požadavky vlastní.

9 NUMA Topology Views

Kapitola se bude zabývat posledním z hlavních vylepšení této práce, tj. přidáním informací o NUMA topologii CPU na systému do KernelSharku. Tato modifikace pak bude hlavně sloužit k vylepšení analýzy trasovacích dat, jelikož budeme jednodušeji schopni určit namáhanou část topologie. Kapitola postupně představí cíle, analýzu řešení, návrh, uživatelskou dokumentaci, návrhy pro rozšíření, kritiku řešení a konečné zhodnocení splnění cílů této modifikace.

9.1 Cíle

- Modifikace bude umět zpracovat topologická data z XML souboru vytvořeného programem Hwloc. Z tohoto souboru bude hlavně chtít vyčíst NUMA topologii procesorů.
- Zpracovaná topologická data budou zobrazena někde v hlavním okně. Místo zobrazení by mělo dovolovat přirozenou návaznost na CPU grafy. Ty mohou být přeuspořádány tak, aby respektovaly řazení v topologii.
- Pokud nemáme topologická data k dispozici pro nějaký stream, nebudeme topologii pro daný stream zobrazovat.
- Topologie budou zobrazovány jako stromy.
- Každý prvek stromu bude viditelně pojmenován. Pokud by jméno bylo příliš dlouhé, lze použít popisky při najetí myši a jinde zkratky.
- Topologické stromy nebudou zobrazovat NUMA uzly, pokud na systému existuje pouze jeden (a NUMA technologie je tedy nepřítomná/nevyužitá).
- Topologické stromy budou vždy zobrazovat alespoň jádra v topologii. Ta budou vždy obsahovat alespoň jeden procesor.
- Jádra budou zabarvena průměrnou barvou ze svých procesorů. NUMA uzly budou zabarveny průměrnou barvou jader, která jsou součástí NUMA uzlu.
- Místo s topologickými stromy bude možné schovat přes GUI prvek.
- Modifikace bude mít konfigurační dialog, ve kterém si bude uživatel pro každý otevřený stream schopen vybrat soubor s topologickými daty a typ zobrazení topologie, tzv. „pohled“ - buď výchozí, nebo se zobrazením NUMA topologie jako stromu.
- Pokud nebude vybrána topologie, ale bude vybrán stromový pohled, bude namísto toho použit výchozí pohled.
- Vybrání souboru topologie s odlišným počtem CPU, než jsou v daném streamu tuto topologii nezobrazí, použije se výchozí pohled a uživatel bude o nesrovnalosti informován.
- Modifikace bude uložitelná do relací.

9.2 Terminologie

Níže jsou termíny, které tato modifikace používá. Některé termíny jsou přímo inspirované terminologií Hwlocu, některé jsou specifické pouze pro NUMA TV, některé obsahují termíny KernelSharku (ty vysvětleny nebudou).

- *Blokový strom* - Bloky vedle sebe, které reprezentují strom. Blok je obdélník a reprezentuje uzel. Hrany jsou reprezentovány dotykem bloků. Nejlépe ukázáno na obrázku 9.1.
- *Zkrácená topologie* - Reprezentace topologie s pouze nejdůležitějšími částmi pro topologickou vizualizaci od NUMA TV. Jedná se o třívrstvou strukturu, kde v první vrstvě jsou logické indexy NUMA uzlů, v druhé vstvě jsou logické indexy jader, ve třetí vrstvě jsou logické indexy procesorů spárovány s OS indexy procesorů.
- *C* - Zkratka označující jádro v topologickém stromu, často zobrazováno se svým logickým indexem.
- *Jádro* - Topologická struktura v Hwlocu obsahující jeden či více procesorů a která je obsažena v NUMA uzlech. Na systémech bez hyperthreadingu je tento termín zaměnitelný s procesorem.
- *GL plocha* - kreslící OpenGL plocha používaná KernelSharkem na kreslení CPU grafů a grafů procesů.
- *Logický index* - Index dán části topologie během jejího zkoumání Hwlocem. Společně s typem objektu (PU, jádro, NUMA uzel) pak tvoří unikátní identifikátor komponenty v dané topologii.
- *NN* - Zkratka označující NUMA uzel v topologického stromu, často zobrazován se svým logickým indexem.
- *NUMA TV* - Zkratka pro „NUMA Topology Views“.
- *NUMA TV kontext* - Konfigurační objekt modifikace, který se stará o konfiguraci NUMA TV pro každý otevřený stream, tj. cestu k souboru topologie, kterou má stream načtenou (pokud vůbec), a který typ pohledu chce stream použít při načtené topologii.
- *OS index* - Index přidělen hardwarové komponentě operačním systémem. Nerespektuje topologii a pro vizualizace není moc užitečný. Tyto indexy používá KernelShark pro nápisy u CPU grafů, tj. pokud máme zobrazen graf pro CPU 0, tak „0“ je OS indexem daného procesoru.
- *(Procesorový) modul* - Fyzické místo, kde jsou instalovány procesory (dle Hwlocu). Také součást topologie v Hwlocu, může obsahovat jeden i více NUMA uzlů, zároveň jeden modul může být součástí jednoho i více NUMA uzlů.
- *PU/procesor* - Hwloc termín pro to samé, co jsou podle KernelSharku CPU. Mohou být seskupeny v jádru, každé jádro má aspoň jeden PU. PU je zkratka z anglického „processing unit“.

- *Topologie* - Struktura s detailní organizací paměťových modulů, procesorů, jiných zařízení na stroji/systému a obsahující nějaké dalších informace, jako například jméno stroje či celková paměť. Tuto strukturu dokáže zachytit Hwloc.
- *Úlohová výplň* - Dodatečný prázdný prostor pod topologickým stromem v topologické ploše. Je přítomen pokud KernelShark ukazuje grafy úloh pro nějaký stream.
- *Topologický strom* - Skupina Qt objektů, které dohromady tvoří blokový strom bez kořene, jehož první vrstvou jsou NUMA uzly, po nich je vrstva jader, které tvoří listy stromu. Topologické stromy jsou vždy součástí topologické plochy. Topologické stromy NUMA TV vždy vytváří z nějaké zkrácené topologie. Pokud zkrácená topologie nemá více než jeden NUMA uzel, pak se vrstva NUMA uzlů nezobrazuje.
- *Vizualizace topologie* - V NUMA TV zaměnitelné s topologickým stromem.
- *Topologická plocha* - Qt plocha s topologickým stromem a možnou úlohovou výplní. Je součástí wrapperu topologcké plochy. Lze přímo namapovat na třídu v kódu: `KsStreamTopoWidget`
- *Pohled/typ pohledu* - výčtová třída pro vybrání způsobu zobrazení topologie v KernelSharku. NUMA TV definuje dva pohledy:
  - Výchozí pohled, který ignoruje topologii. Pokud jso všechny pohledy nastaveny na výchozí, pak NUMA TV není v hlavním okně aktivní, tj. hlavní okno zobrazuje to samé, co bylo zobrazování před touto modifikací.
  - „NUMA tree“/stromový pohled, který pro topologii zobrazí topologický strom v topologické ploše.
- *Wrapper topologické plochy* - Nalevo od GL plochy, obsahuje topologickou plochu. Existuje z implementačních důvodů.

9.3 Analýza

Tato sekce se pokusí zachytit postup, kterým se dostaneme k implementaci řešení. Jedná se o techničtější analýzu než analýza z kapitoly *Obecná analýza a stanovení požadavků*. Zároveň se někdy odkážeme na Couplebreak a části jeho řešení, abychom se zbytečně neopakovali.

9.3.1 Hwloc

Chtěli bychom-li začít se zobrazováním dat topologie od Hwlocu, musíme si nejprve uvědomit, že se jedná o vylepšení, které přidá ke KernelSharku novou závislost, právě Hwloc. Toho jednoduše docílíme editací souborů sestavení pro CMake. Naštěstí pro nás dává dokumentace Hwlocu [**Hwloc-Docs**] přímý návod, jak dodání závislosti docílit. Nám tedy stačí řídit se dokumentací a CMake

instrukce překopírovat, zkontrolovat a drobně upravit pro KernelShark. Verzi Hwlocu vybereme co možná nejnvější, tj. verzi 2.11.

Na řadu přichází nahrání topologie z XML souboru do datové struktury, se kterou pak může Hwloc pracovat. Dokumentace nám opět pomůže a my můžeme přes Hwloc funkce a makra rovnou načíst XML soubor jako topologii. V té si nejprve získáme všechny OS indexy procesorů/PU a poté si přes každý z nich zjistíme i kterému jádru patří a kterému NUMA uzlu patří. Logické indexy PU, jader a NUMA uzlů si pak uložíme do zkrácené topologie. Abychom neztratili spojení mezi logickými indexy procesorů a jejich OS indexy, uložíme si do zkrácené topologie i OS indexy pro PU. Zkrácenou topologii můžeme implementovat pomocí neseřazených map a indexy můžeme reprezentovat pomocí celých čísel. Zkrácenou topologii použijeme proto, že Hwloc topologie obsahuje informace navíc, které nás u NUMA TV nezajímají, například uspoádání cache pamětí, nebo jméno stroje, na kterém byla topologie zachycena.

9.3.2 Hlášení chyb

Během nahrávání topologie se může stát nějaká chyba, například nebyl XML soubor ve správném formátu. I ve zbytku modifikace pak mohou nastat chvíle, kdy bude potřeba ošetřit a ohlásit nějakou chybovou či informační situaci. KernelShark sám výjimky nevyhazuje, namísto toho používá zprávy v terminálu nebo informační dialogy. NUMA TV bude používat návratové kódy a psaní do terminálu pro hlášení nějak zajímavých situací, jako třeba chyb.

9.3.3 Konfigurační dialog

Poté, co jsme schopni pracovat s Hwlocem a získat si pro nás zajímavá data, se můžeme začít soustředit vytvoření konfigurace NUMA TV. Začneme GUI pro uživatelskou interakci s konfigurací. Zde si můžeme i rozmyslet, co bude součástí konfigurace, kterou bude tento dialog měnit.

Všimněme si podobností s konfigurací pro Couplebreak **??**. I v této modifikaci chceme mít nastavení pro každý stream zvlášť. Můžeme se tedy konfiguračním dialogem Couplebreaku inspirovat a pozměnit jenom to, co pro každý ze streamů konfigurujeme. Naše cíle vyžadují alespoň konfigurovatelnou cestu k XML souboru topologie a typ pohledu, tj. „jak naložit s načtenou topologií“.

Pohledy navíc cíle definují dva, výchozí a stromový. Výchozí pohled už z názvu napovídá, že bude výchozí hodnotou konfigurace pro pohledy. Pohledy můžeme implementovat přes rádiová tlačítka, pak budeme schopni mít vybranou jen jednu možnost pro nějakou skupinu tlačítek.

Výběr XML souboru lze zjednodušit použitím souborového dialogu od Qt a filtrovat pro soubory s příponou `.xml`. My pak jenom musíme vytvořit nějaké vybírací `Select` tlačítko, kterým souborový dialog vyvoláme. Ze souborového dialogu pak uložíme cestu k souboru XML jako textový řetězec. Ten pak uživateli zobrazíme, aby bylo jasné, že nějaká topologie byla vybrána. Na vymazání vybrané cesty pak dodáme tlačítko `Clear`. Výchozí hodnotou cesty k souboru bude přirozeně prázdný text.

Speciální pozornost dáme tlačítku `Apply`. To totiž nejenže změny aplikuje, ale před aplikací i zkontroluje některé požadavky, které cíle požadují. Pokud je

vybrán stromový pohled, ale nebyl vybrán topologický soubor, pak konfigurace vynutí pohled výchozí, jelikož není co zobrazovat ve stromovém pohledu. Pokud vybereme soubor s topologií o N CPU, ale stream, kterému toto konfigurujeme má M CPU a zároveň M se nerovná N, pak konfigurace opět použije výchozí pohled a uživatel bude o nevhodném souboru topologie informován na chybovém výstupu. Tlačítko `Cancel` může fungovat jako dřív, tedy prostě dialog zavře bez použití změn.

Tlačítko vyvolávající konfigurační dialog umístíme vedle tlačítka pro konfigurační okéno Couplebreaku, tj. do submenu `Tools`.

9.3.4 Konfigurace NUMA TV

V předchozí podsekci jsme si už trochu seznámili s tím, jak se bude konfigurace chovat, hlavně z grafického pohledu. Nyní se zamyslíme nad tím, jak by se měla chovat mimo grafické prostředí.

Nejprve se zamyslíme nad tím, jak reprezentovat pohledy. My implementujeme jenom stromový pohled na NUMA topologii, ale teoreticky jich může být více, například pohled na procesorové moduly. Proto pohledy implementujeme jako výčtový typ. Tak se omezíme jen na nějakou konečnou množinu hodnot a budeme mít kódu zřetelněji popsané speciální číselné hodnoty.

Dokud uživatel nezmění konfiguraci v dialogu z výchozích hodnot, pak NUMA TV nebude ukládat žádnou konfiguraci pro stream, jelikož by to bylo zbytečné. Pokud ale uživatel vybere validní XML soubor s topologií, pak se situace mění. Aplikace konfigurace pak donutí NUMA TV vytvořit novou konfiguraci pro stream, včetně interpretace topologického XML sobuoru Hwlocem. Po ní nám zbyde zkrácená topologie, kterou také uložíme do konfigurace pro stream. Z toho nám vyplývá, že datová struktura konfigurace NUMA TV pro nějaký stream bude obsahovat prvky pro uložení cesty k XML souboru, uložení typu pohledu a uložení zkrácené topologie.

Pokud uživatel vymaže v konfiguračním dialogu cestu k topologickému souboru, pak není konfigurace pro stream potřeba. Pro nás to znamená návrat k výchozím hodnotám konfigurace (i kdyby byl vybrán stromový pohled, tak bez cesty k souboru topologie toto NUMA TV změní na výchozí pohled). Abychom zbytečně neukládali nepotřebnou konfiguraci, tak ji v této situaci smažeme.

Dalším problémem, co vyřešíme, je aktualizace topologie. Ta se stane, když uživatel změní cestu k topologickému souboru a nebo změní typ pohledu v konfiguračním dialogu a změny aplikuje. Pokud změní cestu, pak lze považovat starou konfiguraci topologie za zbytečnou a tak ji smažeme. Můžeme ji pak kompletně nahradit novou konfigurací. Pokud se změnil jen pohled, pak změníme jen ten. V každém případě ale požádáme NUMA TV o překreslení topologických vizualizací (vizualizační část je popsána v nižší sekci **??**). Pokud je topologie nějak nevalidní (případy popsány v sekci o konfiguračním dialogu výše **??**), pak návratovým kódem rozhodneme, jak se má KernelShark zachovat tj. jestli má do terminálu napsat nějakou chybovou zprávu.

Nakonec, namísto ukládání konfigurací ke každému streamu zvlášť si pořídíme manažera konfigurací, tzv. NUMA TV (konfigurační) kontext. Ten si bude pro každý stream pamatovat jeho specifickou konfiguraci. Krom toho bude mít i API pro přidávání, odebírání, a aktualizaci konfigurací pro každý stream. Jelikož mezi

streamy mohou být „díry“, tj. ne každý stream musí mít aktivní NUMA TV konfiguraci, bude nejlepší i toto vyřešit mapou, klidně i nesetříděnou.

9.3.5 Grafická reprezentace topologie/GRT

[TODO: tohle] Topostrom (barvy a interaktivita, vzhled topostromu celkově)...
CPU grafy a grafy úloh...
cpuReDraw + taskReDraw...
Reorganizace grafů dle topologie...
Místo pro GRT (+ velikost + nemožnost scrollovat + updateGeom)...
Spolupráce s filtry...
Schovávání topologie...

9.3.6 Životnosti

Vše patří KsTraceGraph...
Vše umírá v napsaných konstruktorech správně...

9.3.7 Relace

Relace implementujeme mnohem snadněji než u Couplebreaku. NUMA TV uloží pod identifikátor streamu vybraný typ pohledu a cestu k XML souboru (nebo žádnou cestu neuloží v případě nevybraného souboru v konfiguraci). Inspirovat se můžeme ostatním kódem pro ukládání relace, například pro ukládání Markerů A a B. A stejně se inspirujeme i pro načítání NUMA TV z relačního souboru.

Nyní zbývá vybrat momenty, kdy ukládat NUMA TV a hlavně kdy relační data pro modifikaci načítat. Nelze načítat kdykoliv, jelikož bychom jinak nezastihli vykreslení CPU grafů, tedy i jejich možnou reorganizaci, což by vizualizaci úplně pokazilo. Načítat relační data proto budeme před načtením grafu, abychom vykreslení CPU stihli. Ukládání není závislé na pořadí ukládání ostatních částí relace, ale kvůli symetrii s načítáním jej zařadíme před ukládání grafů.

9.3.8 Zamítnutá alternativní řešení

Tato část je souhrnem několika nápadů, které se objevily během vymýšlení řešení či během implementace, ale byly nakonec zavrženy z různých důvodů. Představen bude jak návrh řešení, tak důvod zamítnutí.

Konfigurační kontext jako singleton

Až moc dlouho jako prototyp, velmi nebezpečný vzor...

Kreslení stromu do OpenGL plochy

Příšerné, nikdy to tak nedělejme...

Graf vytvořen přes QTreeWidget

Nedostatečné schopnosti a layout nebyl vhodný, reimplementace by byla šílená

Zobrazení procesorů a stroje v topologickém stromě

Zbytečné, ačkoliv stream to aspoň ukotvoval jako strom - nicméně více místa = více radosti...

Vlastní zpracování XML souboru

Pro humor, tohle by byla prostě šílenost...

Topologie jako součást datové struktury streamu

Zbytečné omezení na jazyk C...

Topologické zobrazení přítomno neustále

Zbytečné plýtvání místem...

Skládání částí stromu

Zbytečně komplikované, zbytečná fnkcionalita, která se jen plete pod nohy existujícímu řešení přes menu, ale měla i svá plus...

9.4 Vývojová dokumentace

Cíle/úvod této sekce...

9.4.1 Modifikované a nové soubory

Modifikace používá značku `NUMA TV` v ohraničujících komentářích pro změny. Níže je abecedně seřazený seznam souborů spojených s modifikací spolu s krátkým popiskem změn či novinek uvnitř souboru.

- *(src/)CMakeLists.txt* -
- *KsMainWindow.cpp/hpp* -
- *KsNUMATopologyViews.cpp/hpp* -
- *KsPlotTools.cpp/hpp* -
- *KsSession.cpp/hpp* -
- *KsTraceGraph.cpp/hpp* -
- *KsWidgetsLib.cpp/hpp* -

9.4.2 Struktura modifikace

Velké dělení na *Konfiguraci* a *Vizualizaci*. Vše vlastí objekt KsTraceGraph, kromě konfiguračního dialogu, který je vázán jenom na hlavní okno KernelSharku hierarchií oken.

Dělení Konfigurace:

- *Konfigurační kontext* -
- *Konfigurační dialog* -
- *Vytváření zkrácených topologií z Hwlocu* -
- *Dotazy na topologii* -
- *Relace* -

Dělení Vizualizace (modul je i jeden grafický prvek):

- *Topologický strom* -
- *Prostor pro topologický strom* -
- *Skrývající tlačítko* -
- *Přerovnání CPU* -

9.5 Uživatelská dokumentace

```
\subsection{Konfigurace}

To access NUMA TV's confguration, navigate to `Tools > NUMA Topology Views`
button in the toolbar. If no stream was loaded, an error pop-up will be shown
and nothing will happen otherwise.

With a loaded stream, the NUMA TV configuration dialog will be shown,
with a brief explanation of what to do in this window and in the lower half will be
a list of opened streams' NUMA TV configurations, namely what topology file to
load data from and what type of topology view should be used for this stream.
There are also two buttons, "Clear" and "Select...". Clear clears current
configuration's selected topology file (just the filepath to it, not the file
itself). Select button will open up a file dialog, which either starts in the user's
Documents directory or in the directory of the currently applied topology file
(applied = program already loaded this topology file and uses it for a stream).

At the very bottom are two buttons, "Apply" and "Close". Close will ignore any
changes to the configuration done in the window and close the dialog. Apply will
attempt to either create or update chosen stream's topology configuration with
the chosen values.

Beware that if the chosen topology in a file has a different amount of PUs than
KernelShark detected CPUs in its trace file, creation or update of this topology
```

```
configuration is ignored (old values remain or the default choice of no topology
file and DEFAULT view). Topology configuration changes will also be ignored if
no actual topology file was chosen (cleared the path or it has been deleted in
between choosing it and applying the changes), yet the NUMA tree view was set.

Applying an empty filepath current configuration of a stream and program will
default to using DEFAULT view with no topology file.

Currently, only NUMA topology tree view is supported, along with the DEFAULT view,
which does what KernelShark always has.

Reopening the configuration dialog will show applied configuration values
(currently in use by the program).

See figure 2 for a session where no topologies are loaded and figure 3 for
a session where one topology is loaded.

![Figure 2](./images/numatv-2.png)
Figure 2.

![Figure 3](./images/numatv-3.png).
Figure 3.

\subsection{GUI}

\subsubsection{Kde najít NUMA TV}

NUMA Topology Views are at first glance not present in the program. This is by
design, as the modification is supposed to be hidden until needed. To see the
modification in action, first open a stream. Then, open and use the configuration
dialog as outlined above.

The wrapper topology widget will be shown if at least one stream requested a
non-DEFAULT topology view. If any CPUs are shown on the GL widget, there will
also be a topology tree present for the streams that requested a NUMA tree view.
Streams configured to use the DEFAULT view only have blank spaces in as their
topology widget.

\subsubsection{Topologický strom}
The topology tree is a block tree, which is vertically starts at the top of the
start of CPU graphs in the GL widget and vertically ends at the last of those
graphs. If there are only tasks being drawn, there will be blank white space
instead. The topology tree will include only nodes relevant to the CPUs shown in
the GL widget (so if out of 8 CPUs, only 3 are marked as to show their graphs,
then the topology tree will be constructed with cores and NUMA nodes of those
three CPUs). If the topology has only single NUMA node detected (in which case
there is no actual NUMA topology present), the node will be hidden (only Cores
will be visible).

If more streams are opened, each tree starts at its respective stream's CPU
graphs.
```

```
Core nodes of the tree will be colored the average color of its CPUs (these colors
are defined by KernelShark), NUMA nodes are colored from averages of core node
colors. If a color is deemed too dark, white text is used for their labels,
otherwise the nodes use black text color.

\subsubsection{Synchronizace rolování s plochou grafu}

While the wrapper topology widget sits in a scrollable area, to retain consistency with
KernelShark's method of scrolling vertically through a graph, no mouse wheel events
are processed by it. Instead, the scrolling area moves the same amount as GL widget's
scrolling area, when its vertical scrollbar's value changes.

If KernelShark's main window is too narrow, a horizontal scroll bar also appears, but
at that point, it is recommended to resize the main window of the program.

\subsubsection{Přeuspořádání CPU grafů}

If a topology determined that CPUs adhere to a different ordering than OS indices indica
NUMA TV will rearrange the CPU graphs in the GL widget to properly connect to the topolo
of a stream. In DEFAULT view, KernelShark orders CPUs by their OS indices, with a NUMATI
they are ordered by their NUMA node's logical index, then their core's logical index and
by their logical index (i.e. a sorted sequence could look like (0, 0, 20), (0, 0, 60),
(1, 5, 2), (1, 5, 3), where first number in the vector is NUMA node's logical index, fol
by the core's logical index, ending with a PUs logical index).

KernelShark graph will still label CPUs with their OS indices.

Rearrangements may happen only if a non-DEFAULT view is chosen for a stream and rearrang
graphs only of that stream. A rearrangament may happen any time the CPU graphs are redra

See figure 4 for a stream with CPU graphs ordered by OS indices, then compare with figu
reordered CPU graphs.

![Figure 4](./images/numatv-4.png)
Figure 4.

![Figure 5](./images/numatv-5.png)
Figure 5.

\subsubsection{Skrývající tlačítko}

Hide button is a green button on the left of the wrapper topology widget. Clicking on it
hides the wrapper topology widget along with all topology trees (figure 6). Clicking on
it again makes the wrapper topology widget and its trees visible again (figure 7).
Accompanied with hidden/shown states are characters on the button: ">" for hidden
and "<" for shown.

This button is shown only if there's at least one stream requesting a non-DEFAULT topolo
Upon any load of topology requesting a non-DEFAULT view, the wrapper topology widget is
hidden (figure 7).
```

```
![Figure 6](./images/numatv-6.png)
Figure 6.

![Figure 7](./images/numatv-7.png)
Figure 7.

\subsubsection{Plovoucí popisky uzlů stromu}

Hovering with the mouse cursor over a topology tree node and letting it stay there for a
shows a tooltip with the current node's full name, e.g. hovering over a node labeled 'N
"NUMA Node 1" (figure 8).

Purpose of this is to allow short labels for the nodes, while preserving a quick way fo
understand what a certain node is.

![Figure 8](./images/numatv-8.png)
Figure 8.

\subsection{Popora relací}

NUMA Topology Views configuration can be saved in a session using new API of KsSession.
topology file path saved.

Importing a session automatically draws topology widgets, if any are desired.

\subsection{Nová API}

NUMA TV presents among new classes also four new global functions:
- `numatv_count_PUs`: This function counts the number of PUs in a brief topology.
- `numatv_count_cores`: This function counts the number of cores in a brief topology.
- `numatv_filter_by_PUs`: This functions returns a brief topology, where its members co
  are PUs given in a vector. This function is very useful when some CPU graphs are hidd
  filtering the brief topology, we get a brief topology where only the visible CPUs are
  same for their cores and NUMA nodes.
- `numatv_stream_wants_topology_widget`: This function checks whether a stream is reque
  non-DEFAULT view in its configuration in the given NUMA TV context

Any other API that got introduced is either explicitly labeled with "numatv" ot "topolo
somewhere (letters can be uppercase). If not, it belongs to a type with that label or a
file with that label.

Use carefully.

\subsection{Bugy a chyby}

Qt may sometimes complain about recursive call when clicking the Load button in the con
on actual work of the program, as Qt makes sure to detect these calls and stop them. Th
is clicked too fast.
```

```
No others are known to the author. But the modification was rather significant and there
went wrong in secret.
```

9.6 Rozšíření

Cíle/úvod této sekce...

Více pohledů

Package view, něco spešl, ALE nutno vyřešit 9.7...

Konfigurační dialogy by mohly být zobecněny

Couplebreak a NUMA TV mají v podstatě stejnou šablonu, bylo by lepší ji zobecnit...

9.7 Kritika

V této sekci autor kriticky zhodnotí své řešení. Každá kritika má nadpis a popis a dále buď obsahuje obranu proti ní, nebo souhlas s předneseným problémem.

Nerozšiřitelnost

Primárně pro NUMA topologii, nedostatečný vhled dopředu pro další viewtypes, ať už v GUI nebo v konfiguraci...

Nevhodné umístění třídy `KsStreamNUMATopology`

Mělo to mít vlastní soubor nebo do NUMATV souboru...

Kostrbaté vložení závislosti do `CMakeLists.txt` KernelSharku

Ale je to oficiální...

Nemožnost upravovat aktivní topologie

Místo toho create + delete only...

Výjimky místo návratových kódů

KernelShark to tak nedělá...

Porušení SRP

Zas tak moc ne, furt to zlepšuje analýzu, nicméně chápu, odkud míříme...

Orientace a velikost popisků topologického stromu

Mohlo by být větší/tučné/vertikální/rotované...

Barva uzlů topologického stromu

Avg z toho většinou udělá šedou/hnědou... mohlo by být jinak...

Konfigurace a widget v oddělených mapách

Nemuselo by být, ale dává smysl...

Malá interaktivita topologického zobrazení

A to vůbec nevadí...ale samozřejmě by šlo něco takového dodat...

Používání C ukazatelů

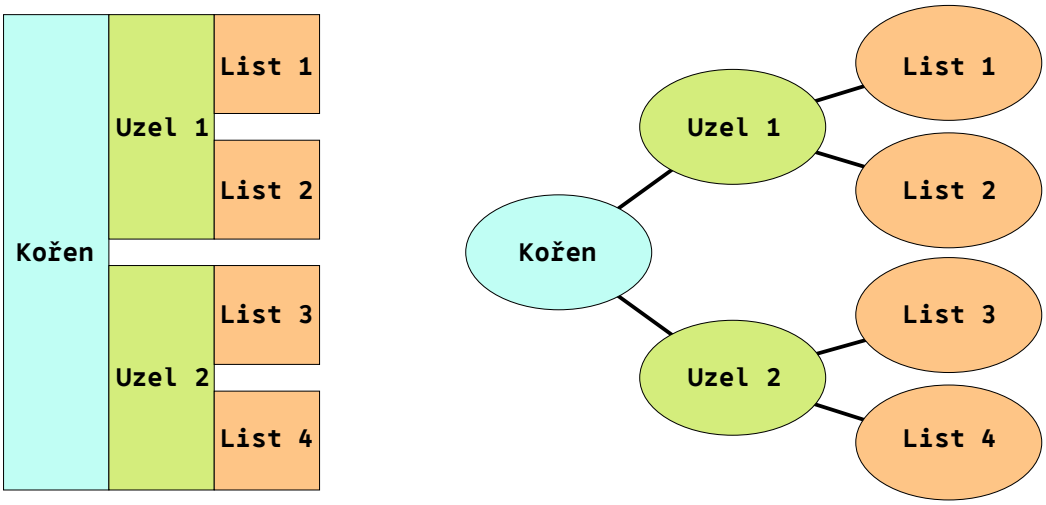
Pravda, nejsou moc bezpečné, ale KernelShark to často dělá a Qt si s nimi umí poradit...

Dialogy namísto standardních výstupů

Grafická aplikace, ne nutně bude vždy uživatel mít přístup k jejímu terminálovému výstupu...

9.8 Zhodnocení splnění požadavků

Obecné požadavky hlavně + možná nějaká porušení + splnění vlastních cílů...



Obrázek 9.1 Blokový strom (vlevo) a klasická grafová reprezentace stromu (vpravo)

10 Dodatečná vylepšení

Jedná se o vylepšení, která vznikla kvůli ostatním vylepšením, nebo jako rychlá zlepšení chování KernelSharku. Pro každé z nich bude stručně popsán návrh, řešení, použití a případná varování.

10.1 Aktualizace kódu zaškrtávacích políček

Nejpřímějším zlepšením byla aktualizace grafického kódu, aby využíval novější Qt API pro zaškrtávací políčka. V nové API se nahrazuje funkce QCheckBox::stateChanged za QCheckBox::checkStateChanged. Hlavní změna se týká v argumentu značící zaškrtnutí - předtím stačilo dodat argument typu int, od Qt 6.7 ale nová funkce vyžaduje jednu z enumerovaných možností specifických pro QCheckBox.

Aktualizace nakonec spočívala pouze v nahrazení signálu stateChanged na checkStateChanged a použití výčtových hodnot (dle původního čísla) namísto pouze celých čísel.

KernelShark původně vynucoval v sestavení Qt verze 6.3, nyní vyžaduje alespoň verzi 6.7, aby mohl využít novější signál pro zaškrtávací políčka.

Změněné soubory: nejvyšší *CMakeLists.txt* KernelSharku (v KS_fork adresáři), *KsTraceViewer.hpp/cpp*, *KsCaptureDialog.hpp/cpp*, *KsMainWindow.hpp/cpp*. Značkou modifikace v C++ kódu a CMake instrukcích je UPDATE CBOX STATES.

Pokud mají v plánu vývojáři KernelSharku či jeho pluginů dále pracovat s KernelSharkem, musejí tedy použít alespoň Qt6 verze 6.7.0.

10.2 Zpřístupnění barev užívaných v grafu

Modifikace je pouze zpřístupnění barevných tabulek, které KernelShark používá pro streamy, CPU a procesy. Díky zpřístupnění pak mohou tyto tabulky využívat i jiné části programu, nebo i pluginy. Jedná se o pouhé získání const reference na objekty využívané uvnitř KsGLWidget objektu. Přirozeně se jedná o malé úpravy, lokalizované v souboru *KsGLWidget.hpp*, se kódovou značkou GET COLORS. K využití této modifikace stačí mít přístup k GL objektu a zavolat nové metody.

K využití přes pluginy se váže varování - pokud si KernelShark uloží do relace plugin, který využívá tuto modifikaci, ale načtení pluginu není dokonalé, program při první snaze o získání hodnoty z jakékoliv z tabulek spadne. Plugin je nedokonale načten vždy, pokud KernelShark načte relaci s daným pluginem, ale ten není předem explicitně načten uživatelem, tedy skrze argumenty při spouštění, nebo přes GUI. Další možnou podmínkou je, že plugin není postaven jako oficiální pluginy KernelSharku, tedy během sestavování KernelSharku samotného. Takto lze postavit například plugin Stacklook. Tato podmínka ovšem nebyla rigorózně otestována a je to spíše pouhá domněnka. Chybě se lze vyhnout třemi způsoby: buď uživatel vždy explicitně načte plugin, nebo plugin nebude tyto tabulky využívat, nebo bude obsahovat výchozí hodnotu, kterou použije namísto tabulek při načtení z relace - barevné tabulky se využijí až později, například až pokud si je uživatel zapne v konfiguraci pluginu.

Tuto modifikaci využívají pluginy Stacklook a Naps pro barvy procesů, a modifikace NUMA Topology Views pro barvy procesorů.

Příklad: Stacklook nabízí možnost barvit tlačítka dle barev procesů, kterým patří, viz obrázek 10.1.

10.3 Reakce objektů v grafu na přejetí myší

Tato modifikace dodala všem potomkům třídy KsPlot::PlotObject, jednodušeji plot-objekty, veřejnou metodu pro reakci na přejetí myší a redefinovatelnou privátní virtuální metodu, kterou veřejná metoda volá pro viditelný objekt. Privátní metoda má výchozí definici prázdnou. Krom toho je dodána detekce přejetí přes plot-objekt v grafu a reakce na přejetí. Toto chování bylo vsunuto na konec zpracování události pohybu myši a funguje podobně jako detekce dvojitého kliknutí. Soubory s modifikací: *KsPlotTools.hpp* s novými metodami a *KsGLWidget.cpp* pro vsunutou detekci a reakci. Kódová značka modifikace je MOUSE HOVER PLOT OBJECTS.

Nabízí se zde otázka, zdali je toto dostatečně výkonná implementace - myš se pohybuje často, objektů může být mnoho. Řešení musí vždy projít všechny plot-objekty a u každého se rozhodnout, zdali reagovat na myš či nikoliv. Při praktickém použití s rozumnými limity pro zobrazované plot-objekty (nedává smysl neustále zobrazovat tlačítka Stacklooku nad každým prvkem grafu, vedlo by to k přemíře informací) nenastaly problémy a nejvíc program zpomaloval objem dat a náhlé vykreslování objektů, nikoliv práce této modifikace. Pokud bude ale objektů příliš mnoho, výkon může být ovlivněn. Proto se optimalizace výkonu nechává jako *rozšíření* této práce. Inspirací může být nahrazení lineárního prohledávání for-cyklem vyhledáváním přes souřadnice, tedy přes nějaké mapování souřadnic na objekty na těchto souřadnicích.

Tuto modifikaci používá hlavně plugin Stacklook.

10.4 Měnitelné nápisy v hlavičce grafu

Tato modifikace přidává do souborů *KsTraceGraph.hpp/cpp* veřejnou funkci, s níž lze přepsat obsah informačního řádku KernelSharku. Jedná se o funkci typu setter, pouze nastaví hodnoty nápisů v informačním řádku.

K použití stačí mít k dispozici KsTraceGraph objekt. Pak se dá s modifikací pracovat i v pluginech a jiných částech KernelSharku.

V kódu lze tuto modifikaci nalézt pod značkou PREVIEW LABELS CHANGEABLE.

I zde se objevuje bug ze sekce o zpřístupnění barev využívaných v grafu, jenom se tentokrát netýká tabulek, nýbrž informačního řádku. Zde ale nelze spoléhat na nějaké výchozí hodnoty, tedy uživatel musí buď plugin vždy explicitně načíst, nebo nepoužívat plugin využívající tuto modifikaci.

Příklad použití: tlačítka Stacklooku (v červeném kroužku) žádají na přejetí myší o zobrazení několika prvků zásobníku kernelu, viz obrázek 10.2.

10.5 NoBoxes

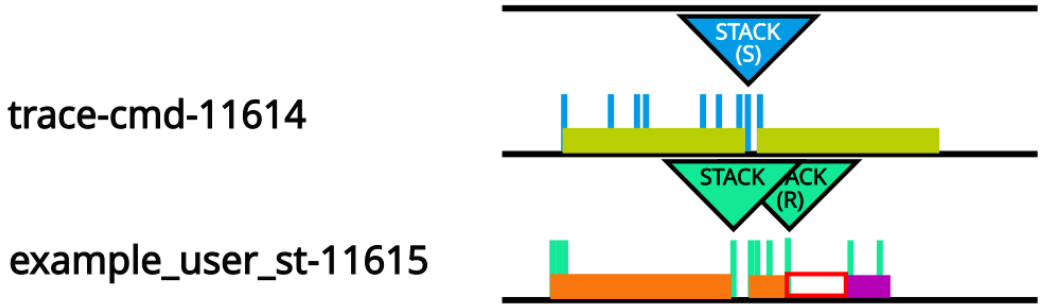
Jak popsala čtvrtá kapitola, chtěli bychom některé záznamy při vykreslování ignorovat, jinak se nám může zobrazovat falešná práce v grafu. Vykreslování obdélníků mezi záznamy se děje při každém vykreslení grafu. Ovlivnit obdélníky dokáže v této chvíli máloco. Pracuje se zde s biny, tedy sdruženími jednoho či více záznamů. V binech se pak nelze na mnoho spolehnout, ale můžeme použít data viditelnosti, u kterých KernelShark ukládá i nastavení speciálních chování. Viditelnost binu lze ovlivnit viditelností záznamů, které sdružuje.

Vylepšení vytvořilo novou masku viditelnosti pro záznamy, KS_DRAW_TASK-BOX_MASK. Tato maska značí, zdali se záznam účastní kreslení mezi-záznamových obdélníčků. Vykreslování pak bylo upraveno tak, že pokud bin využije viditelnost záznamu, který se nemá účastnit kreslení obdélníků, pak obdélníky budou při kreslení tento bin ignorovat.

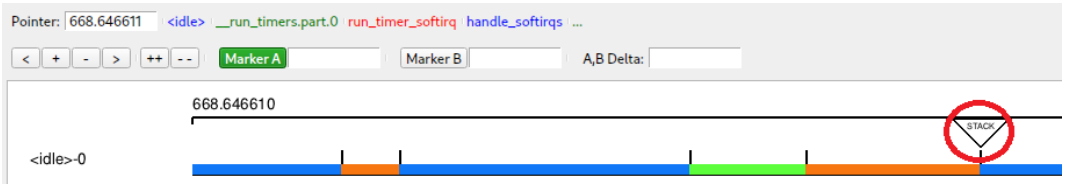
Na výběr záznamů, kteří budou masku používat, byl vytvořen plugin pro tuto modifikaci, nese název NoBoxes. V jeho kódu jsou zapsány události, na jejichž záznamy má být maska použita, plugin pak lze zapínat a vypínat jako každý jiný. Pokud plugin není načten, modifikace nemá na chod KernelSharku vliv.

Příklad fungujícího vylepšení je na obrázku 10.3. Oproti minulému obrázku (obrázek 4.1) ubylo několik obdélníků a graf nyní odpovídá skutečné práci.

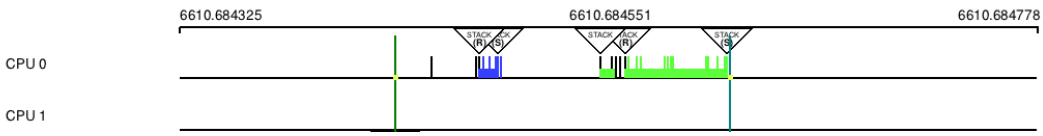
Problém tohoto vylepšení je ale právě vykreslování obdélníků. Vykreslování se děje často a použitá implementace tohoto vylepšení při grafických změnách v hlavním okně KernelSharku obdélník vykreslí i pokud by neměl projít novým filtrováním. Při častých změnách pak nastává problikávání obdélníků. Oprava byla označena za rozšíření, jelikož objem práce k tomuto dodatku byl odhadnut na příliš velký a vytvořený best-effort přístup funguje, ačkoliv ne dokonale.



Obrázek 10.1 Stacklook tlačítko využívá barvu, kterou KernelShark udělil procesu.



Obrázek 10.2 Při přejetí kurzorem myši se vlevo nahoře informační řádek změní.



Obrázek 10.3 Vylepšení donutí některé obdélníky k tomu, aby nebyli nakresleny.

Závěr

Cíle/úvod této kapitoly...

10.6 Shrnutí práce

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

A Přílohy

A.1 První příloha