offwaketime - rev5

`offwaketime` è un profiler per Linux che sfrutta eBPF per catturare:

- lo stack trace dei momenti in cui un thread entra in attesa di un evento esterno alla CPU
- lo stack trace dei momenti in cui lo stesso thread termina l'attesa dell'evento esterno
- per ogni coppia diversa di stack trace di inizio-fine attesa, la durata cumulativa dell'attesa

È mantenuto dall'IO Visor Project.

Installazione

`offwaketime` è uno script del toolkit `bcc`, e viene installato assieme ad esso.

Inoltre, richiede che siano installati gli header del kernel in esecuzione sulla macchina che esegue il programma di cui si vuole effettuare il profiling.

Infine, per visualizzare graficamente l'output ottenuto, è necessario il renderer `FlameGraph`.

`bcc`

`bcc` è preparato downstream dalla grande maggioranza delle distribuzioni Linux:

- su ***Debian 12 Bookworm***, è possibile installarlo tramite il pacchetto `bpfcc-tools`
- su ***Ubuntu 24.10 Oracular Oriole***, è possibile installarlo tramite il pacchetto `bpfcc-tools`
- su ***Fedora 41***, è possibile installarlo tramite il pacchetto `bcc-tools`

- su ***Arch Linux 2025-02-27***, è possibile installarlo tramite il pacchetto `bcc-tools` in `extra`

IO Visor mantiene una lista completa di tutti i downstream che includono `bcc`.

Header del kernel

Per utilizzare `bcc`, è necessario che il kernel che si sta utilizzando abbia le seguenti funzionalità compilate[1]:

- `CONFIG_BPF_SYSCALL`
- `CONFIG_BPF_JIT`
- `CONFIG_HAVE_BPF_JIT`
- `CONFIG_HAVE_EBPF_JIT`
- `CONFIG_HAVE_CBPF_JIT`
- `CONFIG_MODULES`
- `CONFIG_BPF`
- `CONFIG_BPF_EVENTS`
- `CONFIG_PERF_EVENTS`
- `CONFIG_HAVE_PERF_EVENTS`
- `CONFIG_PROFILING`
- `CONFIG_DEBUG_INFO_BTF`
- `CONFIG_PAHOLE_HAS_SPLIT_BTF`
- `CONFIG_DEBUG_INFO_BTF_MODULES`
- `CONFIG_BPF_JIT_ALWAYS_ON`
- `CONFIG_BPF_UNPRIV_DEFAULT_OFF`
- `CONFIG_CGROUP_BPF`
- `CONFIG_BPFILTER`
- `CONFIG_BPFILTER_UMH`
- `CONFIG_NET_CLS_BPF`
- `CONFIG_NET_ACT_BPF`
- `CONFIG_BPF_STREAM_PARSER`
- `CONFIG_LWTUNNEL_BPF`
- `CONFIG_NETFILTER_XT_MATCH_BPF`

- `CONFIG_IPV6_SEG6_BPF`
- `CONFIG_KPROBE_EVENTS`
- `CONFIG_KPROBES`
- `CONFIG_HAVE_KPROBES`
- `CONFIG_HAVE_REGS_AND_STACK_ACCESS_API`
- `CONFIG_KPROBES_ON_FTRACE`
- `CONFIG_FPROBE`
- `CONFIG_BPF_KPROBE_OVERRIDE`
- `CONFIG_UPROBE_EVENTS`
- `CONFIG_ARCH_SUPPORTS_UPROBES`
- `CONFIG_UPROBES`
- `CONFIG_MMU`
- `CONFIG_TRACEPOINTS`
- `CONFIG_HAVE_SYSCALL_TRACEPOINTS`
- `CONFIG_BPF_LSM`
- `CONFIG_BPF_LIRC_MODE2`

Kernel distribuito

Se si sta usando il kernel Linux compilato dalla propria distribuzione, solitamente si possono installare i relativi header dal proprio package manager:

- su ***Debian 12 Bookworm***, è possibile installarli tramite il pacchetto `linux-headers-XXX`, dove `XXX` è la versione del proprio kernel ottenibile attraverso il comando `uname --kernel-release`
- su ***Ubuntu 24.10 Oracular Oriole***, è possibile installarli tramite il pacchetto `linux-headers-XXX`, dove `XXX` è la versione del proprio kernel ottenibile attraverso il comando `uname --kernel-release`
- su ***Fedora 41***, è possibile installarli tramite il pacchetto `kernel-devel`
- su ***Arch Linux 2025-02-27***, è possibile installarli tramite il pacchetto `linux-headers` in `core`

Kernel da sorgente

Se si sta usando un proprio kernel compilato da sorgente, è possibile installare i relativi header attraverso `make` con il comando:

```
make headers_install
```

Eventualmente è possibile modificarne la posizione di installazione, specificando la variabile `INSTALL_HDR_PATH`:

```
make headers_install INSTALL_HDR_PATH="/usr/lib/modules/1.2.3-mykernel"
```

`FlameGraph`

`FlameGraph` è un insieme di script awk e `perl` che non sono preparati downstream da nessuna distribuzione, rendendo quindi necessario scaricarli da sorgente:

```
git checkout --depth=1 'https://github.com/brendangregg/FlameGraph.git'
```

Utilizzo

Per utilizzare `offwaketime`, è prima necessario assicurarsi che il programma che si vuole profilare sia in esecuzione, poi eseguire il tool installato come superuser.

In base a come si è installato il toolkit `bcc`, il profiler `offwaketime` si troverà in directory diverse:

- su ***Debian 12 Bookworm***, in`/bin/offwaketime-bpfcc`
- su ***Ubuntu 24.10 Oracular Oracle***, in `/bin/offwaketime-bpfcc`
- su ***Fedora 41***, in `/usr/share/bcc/tools/offwaketime`
- su ***Arch Linux 2025-02-27***, in `/usr/share/bcc/tools/offwaketime`

Per convenienza, in questa guida viene usato `offwaketime` per indicare l'eseguibile installato:

```
offwaketime
```

Allo stesso modo, lo script Perl `flamegraph.pl` si troverà in directory diverse in base a dove si è clonato il repository.

Per convenienza, in questa guida viene usato `flamegraph` per indicare quell'eseguibile:

```
flamegraph
```

Selezione dell'oggetto del profiling

La profilazione di `offwaketime` può essere impostata per includere o escludere diversi programmi in esecuzione sul proprio sistema operativo.

Se non viene specificato nulla, la profilazione di `offwaketime` si applica a tutto il sistema operativo:

```
offwaketime
```

L'opzione `-k` filtra la profilazione a solamente i kernel thread, come ad esempio i thread di `[kworker]`:

```
offwaketime -k
```

Viceversa, l'opzione `-u` filtra la profilazione a solamente gli user thread, cioè quelli delle applicazioni avviate in user space:

```
offwaketime -u
```

L'opzione `-p` permette di specificare i pid di uno o più processi da includere, escludendo tutto il resto:

```
offwaketime -p 150000,150001,150002
```

L'opzione `-t` è più granulare, e permette di specificare i tid di uno o più *thread* da includere, escludendo il resto:

```
offwaketime -t
```

È possibile abilitare la visualizzazione del tid dei thread specificando a `ps` le opzioni `-m` e `-L`, e guardando la colonna `LWP`, light-weight process:

```
ps -m -L
```

Selezione della durata del profiling

La durata del profiling di `offwaketime` può essere regolata per avere una certa durata, o per terminare quando richiesto dall'utente con `SIGTERM`.

Se non viene specificato nulla, la profilazione di `offwaketime` dura fino alla ricezione di `SIGTERM` (solitamente una pressione di `Ctrl+C`):

```
offwaketime
```

È possibile far terminare la profilazione dopo un certo numero di secondi specificandolo come primo argomento:

```
offwaketime 2
```

Relativamente ai programmi userspace, `offwaketime` è in grado di risolverne i simboli e quindi di visualizzare il relativo stack **solo se i programmi stessi sono ancora in esecuzione** quando la profilazione ha termine.[2]

Selezione formato di output

`offwaketime` può emettere output in due diversi formati, ognuno con diversi usecase.

Se non viene specificato nulla, `offwaketime` emette output in un formato human-friendly utilizzando testo pre-formattato:

```
offwaketime
```

Un esempio di esso:

```
waker:             0
--                --
```

```
finish_task_switch.isra.0
__schedule
schedule
syscall_exit_to_user_mode
do_syscall_64
entry_SYSCALL_64_after_hwframe
[unknown]
[unknown]
__close
_$LT$std..os..fd..owned..OwnedFd$u20$as$u20$core..ops..drop..Drop$GT$::drop::h5178fe097455d6a9
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
9

waker:           lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
ptrace_request
--               --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
irqentry_exit_to_user_mode
```

```
exc_debug_user
asm_exc_debug
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
671

waker:           lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
ptrace_request
--               --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
irqentry_exit_to_user_mode
asm_exc_int3
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
697
```

```
waker:           lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
ptrace_request
--               --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
irqentry_exit_to_user_mode
exc_debug_user
asm_exc_debug
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
881

waker:           lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
```

```
ptrace_request
--                --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
irqentry_exit_to_user_mode
asm_exc_int3
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:           wakedemo 36360
960

waker:            lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
ptrace_request
--                --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
```

```
irqentry_exit_to_user_mode
asm_exc_int3
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
1190

waker:           lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
ptrace_request
--               --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
irqentry_exit_to_user_mode
asm_exc_int3
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
1007745
```

```
waker:           lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
ptrace_request
--               --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
irqentry_exit_to_user_mode
exc_debug_user
asm_exc_debug
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
1035518

waker:           lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
```

```
ptrace_request
--                --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
arch_do_signal_or_restart
irqentry_exit_to_user_mode
exc_debug_user
asm_exc_debug
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:           wakedemo 36360
1069663

waker:            lldb-server 36341
[unknown]
entry_SYSCALL_64_after_hwframe
do_syscall_64
__x64_sys_ptrace
arch_ptrace
ptrace_request
--                --
finish_task_switch.isra.0
__schedule
schedule
ptrace_stop
get_signal
```

```
arch_do_signal_or_restart
irqentry_exit_to_user_mode
exc_debug_user
asm_exc_debug
wakedemo::main::h52fdaccdcfcbd854
std::rt::lang_start::h06309630d8833d25
__libc_start_main
_start
target:          wakedemo 36360
1231790
```

Se si specifica l'opzione `-f`, `offwaketime` emette output in un formato machine-friendly, separando le tracce con `;`:

```
offwaketime -f
```

Un esempio di esso:

```
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_debug;exc_debug_user;irqe
18
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_int3;irqentry_exit_to_use
[unknown];lldb-server 380
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_debug;exc_debug_user;irqe
[unknown];lldb-server 506
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_int3;irqentry_exit_to_use
[unknown];lldb-server 547
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_int3;irqentry_exit_to_use
[unknown];lldb-server 637
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_debug;exc_debug_user;irqe
[unknown];lldb-server 681214
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_debug;exc_debug_user;irqe
[unknown];lldb-server 690579
```

```
wakedemo;_start;__libc_start_main;std::rt::lang_start::h06309630d8833d25;asm_exc_debug;exc_debug_user;irqe
[unknown];lldb-server 730620
```

Filtraggio delle tracce per durata delle attese

Il profiling di `offwaketime` può essere regolato per includere solo le tracce relative ad attese che rientrano in un dato intervallo di durata.

Se non viene specificato nulla, la profilazione di `offwaketime` include tutte le tracce di attese dalla durata superiore a 1 microsecondo (µs):

```
offwaketime
```

Qualora l'impostazione predefinita restituisse troppe tracce, è possibile incrementare la durata minima delle attese specificando l'opzione `-m` seguita dal numero di microsecondi (µ) desiderato:

```
offwaketime -m 10
```

È possibile anche impostare un tetto superiore alla durata delle attese specificando l'opzione `-M` seguita dal numero di microsecondi (µ) desiderato:

```
offwaketime -M 1000000
```

Le due possono essere combinate per filtrare un intervallo:

```
offwaketime -m 10 -M 1000000
```

Filtraggio delle tracce per stato del thread

Il profiling di `offwaketime` può essere regolato per includere solo le tracce relative ad attese relative a thread con determinate flag di stato.

Se non viene specificato nulla, la profilazione di `offwaketime` include tracce di tutti i thread:

```
offwaketime
```

È possibile filtrare thread relativamente al loro stato specificando l'opzione `--state`, seguita dalla bitmask di i flag di stato che **si richiede siano tutti presenti** sui thread da profilare:

```
offwaketime --state $(( 1 | 2 ))
```

Controintuitivamente, esiste un'eccezione speciale per il valore di `--state 0`, che se specificato seleziona i task senza flag di stato, ovvero quelli in esecuzione:

```
offwaketime --state 0
```

I valori utilizzabili per filtrare i thread in base al loro stato sono:

Valore	Costante	Significato
0	TASK_RUNNING	Il thread è in esecuzione
1	TASK_INTERRUPTIBLE	Il thread è in attesa e in grado di ricevere segnali[3]
2	TASK_UNINTERRUPTIBLE	Il thread è in attesa, ma senza essere in grado di ricevere segnali[4]
256	TASK_WAKEKILL	Il thread è in attesa, ma in grado di ricevere solo segnali che lo ucciderebbero[5]
4096	TASK_RTLOCK_WAIT	Solo in kernel con PREEMPT_RT; il thread è in attesa di un real-time lock[6]
4	__TASK_STOPPED	Il thread è stato messo in pausa da un segnale SIGSTOP[7]; implica TASK_WAKEKILL[8]
8	__TASK_TRACED	Il thread sta venendo debuggato, ed è stato messo in pausa dal debugger[9]
32768	TASK_FROZEN	Il sistema è stato ibernato, ed il thread è in attesa di essere esplicitamente scongelato [10]
8192	TASK_FREEZABLE	Quando il sistema sarà ibernato, questo thread potrà essere congelato e in seguito scongelato[11]

Valore	Costante	Significato
512	TASK_WAKING	Il thread sta venendo risvegliato da un'attesa[12]
1024	TASK_NOLOAD	Il thread è escluso dal conteggio della load average[13]

Filtraggio delle tracce per spazio

Talvolta si può essere interessati a solo le tracce relative all'userspace o al kernelspace di un determinato processo.

Se non viene specificato nulla, `offwaketime` include tutto:

```
offwaketime
```

Specificando l'opzione `-U`, verranno mostrate solo le tracce relative a frame in userspace:

```
offwaketime -U
```

Al contrario, specificando l'opzione `-K`, verranno mostrate solo le tracce relative a frame in kernelspace:

```
offwaketime -K
```

Infine, specificando l'opzione `-d`, verranno inclusi entrambi i tipi di traccia, ma saranno separate da una traccia speciale dal nome di `--`:

```
offwaketime -d
```

Limite del numero di tracce

La struttura dati utilizzata internamente da `offwaketime` ha un limite superiore al numero di tracce contenute negli eventi profilati[14].

Se non viene specificato nulla, ciascun evento potrà contenere fino a 16384 tracce:

```
offwaketime
```

È possibile aumentare o diminuire il numero di tracce specificando l'opzione `--stack-storage-size` seguita dal numero di tracce desiderato:

```
offwaketime --stack-storage-size 3
```

Creazione di un FlameGraph

Per generare un flame graph vettoriale da una chiamata ad `offwaketime`, è necessario:

1. selezionare l'output machine-friendly su `offwaketime` usando l'opzione `-f`
2. *pipe*-are l'output allo script Perl `flamegraph.pl` precendentemente clonato da GitHub, specificando l'opzione `--color='chain'` per colorare appropriatamente le tracce[15], e l'opzione `--countname='microsecs'` per specificare l'unità usata dalle tracce ricevute in input[16], che altrimenti sarebbe genericamente "samples"[17]
3. *pipe*-are l'output ad un file Scalable Vector Graphics (.svg)

```
offwaketime -f | flamegraph --color='chain' --countname='microsecs' | tee "offwaketime.svg" | display
```

Struttura interna

`offwaketime` consiste in:

- uno script Python
  - che fa uso del package `bcc`
  - per generare e poi compilare un programma eBPF
    - contenente strutture dati per immagazzinare tracce
    - e contenente funzioni che scrivono sulle strutture dati la traccia del frame attuale
  - che usa kprobe per registrare dei callback alle funzioni del programma generato
  - per poi attendere fino al termine del profiling
  - e infine stampare nel formato desiderato i contenuti delle strutture dati del programma eBPF

Esempio: debug di programma Rust con LLDB

Si vuole effettuare la profilazione durante il debugging di un piccolo programma Rust sincrono che:

1. crea un file
2. vi scrive all'interno
3. chiude il relativo file descriptor
4. attende due secondi
5. apre il file
6. ne rilegge i contenuti in un buffer
7. chiude di nuovo il file descriptor

Verifica della versione del kernel in esecuzione

Innanzitutto, verifichiamo la versione del kernel attualmente in esecuzione:

```
uname -a
```

```
Linux nitro 6.10.2-rt14-arch1-5-rt #1 SMP PREEMPT_RT Sat, 08 Feb 2025 13:50:34 +0000 x86_64 GNU/Linux
```

Realizzazione del programma

Scriviamo un programma Rust che realizzi gli obiettivi proposti, utilizzando un buffer di dimensioni sufficienti da far durare le operazioni di lettura e scrittura più di 1 µs:

```
use std::fs::File;
use std::io::{Read, Write};
use std::thread::sleep;
use std::time::Duration;

// a 512 MiB buffer of ASCII zeroes, b"0"
```

```
const ZEROES: [u8; 536870912] = [48; 536870912];

fn main() {
    {
        let mut file = File::create("example.txt")
            .expect("File creation failed");

        let _bytes_written = file.write(&ZEROES)
            .expect("File write failed");
    }

    sleep(Duration::from_secs(2));

    {
        let mut file = File::open("example.txt")
            .expect("File open failed");

        let mut buffer = Vec::<u8>::new();
        file.read_to_end(&mut buffer)
            .expect("File read failed");

        assert_eq!(buffer, ZEROES);
    }

    // wait indefinitely so that bcc may collect debug symbols
    sleep(Duration::MAX)
}
```

Esecuzione del programma

Eseguiamo il programma con un debugger, inserendo un breakpoint all'inizio della funzione `main`, in modo che non venga effettuato

nulla prima che `offwaketime` sia in ascolto.

Identificazione del pid del programma

Usiamo `ps` per identificare il processo del programma che abbiamo avviato:

```
ps -m -L -u steffo
```

```
PID   LWP   TTY TIME     CMD
...
22353       - ? 00:00:00 wakedemo
-     22353 -   00:00:00 -
...
```

Osserviamo che ha il process id 22353, e utilizza un singolo thread con il thread id 22353.

Esecuzione di offwaketime

Possiamo quindi avviare `offwaketime` in modo che monitori il processo per 30 secondi e generi un flamegraph:

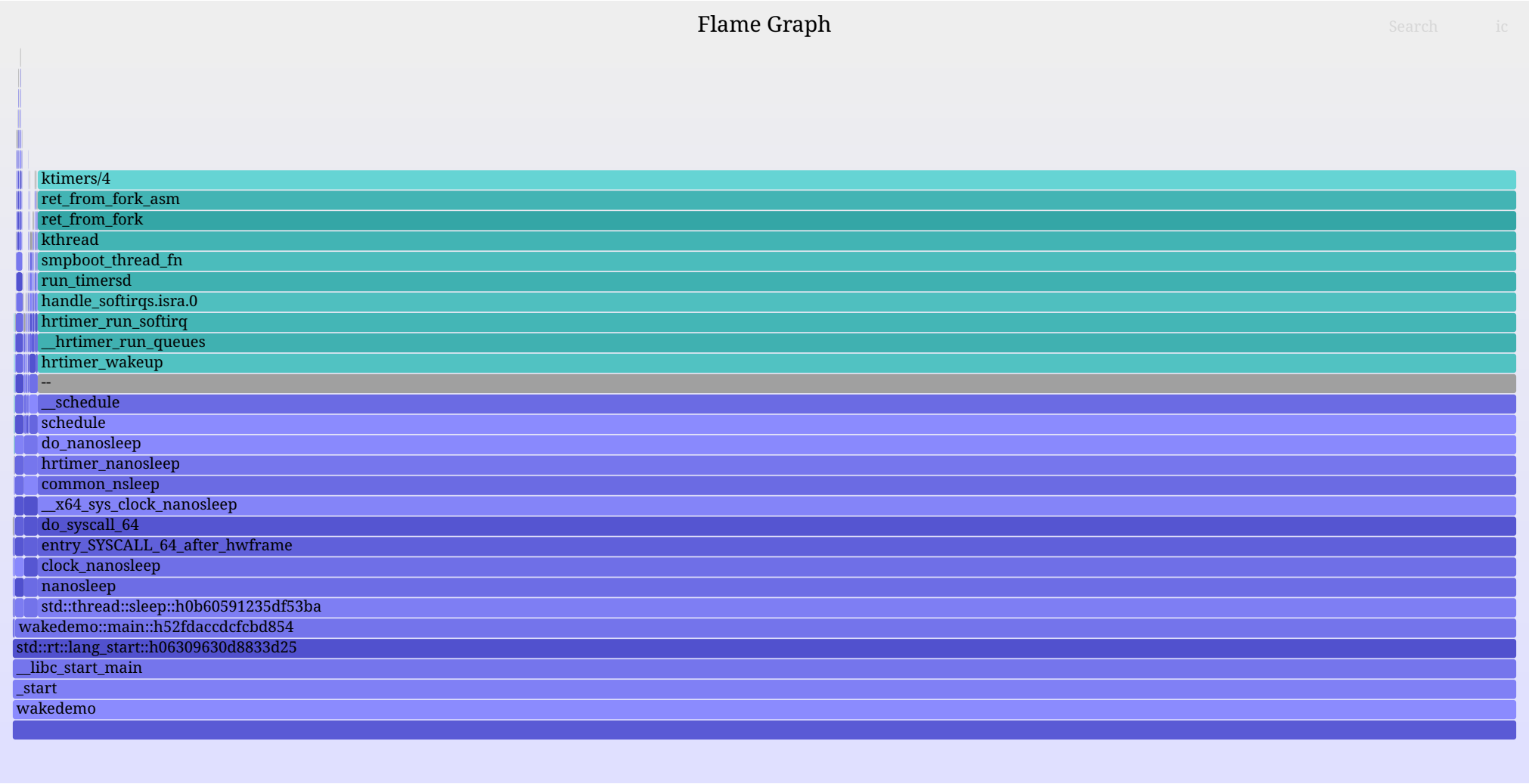
```
sudo offwaketime 30 -p 22353 -f | flamegraph --color='chain' --countname='microsecs' > wakedemo.svg
```

Ora che `offwaketime` è in ascolto, facciamo riprendere al debugger l'esecuzione del nostro programma.

Una volta fatto, senza chiudere il nostro programma[18], aspettiamo che `offwaketime` termini la profilazione.

Immagine prodotta

Alleghiamo l'immagine prodotta da `offwaketime` a questo documento:



Possiamo vedere due gruppi di blocchi colorati, uno blu e uno azzurro, separati da un blocco grigio.

Ciascun blocco corrisponde alla traccia di un frame catturato durante il profiling.

I blocchi blu sono i frame del momento di inizio attesa, i blocchi azzurri sono frame del momento di fine attesa.

Mentre lo stack dei blocchi blu cresce dal basso verso l'alto come è solito nei flame graph, lo stack dei blocchi azzurri è disposto dall'alto verso il basso, commettendo un abuso di notazione che però facilita la lettura del diagramma[19].

Interattività

Ci accorgiamo che l'immagine prodotta contiene anche degli `<script>` che la rendono interattiva, che per motivi di compatibilità funzionano solo se essa viene aperta individualmente come scheda di un browser; pertanto, la apriamo in questo modo.

Mettendo il mouse su ciascuno di questi blocchi, possiamo vedere comparire in basso il testo `Function: NOMEFUNZIONE (DURATA microsecs, PERCENTUALE%)`; ciò ci permette di capire:

- `NOMEFUNZIONE`: il nome completo della funzione nel frame ispezionato, qualora non ci fosse sufficiente spazio nell'immagine per visualizzarlo
- `DURATA`: il totale delle durate in microsecondi di tutte le attese iniziate contenenti quel frame
- `PERCENTUALE`: la percentuale di tempo passato in un'attesa contenente quel frame

La larghezza di ciascun blocco è relativa al blocco "radice" più in basso, sempre denominato `all` e largo tutta l'immagine: è una rappresentazione grafica di `PERCENTUALE`.

Cliccando su un blocco possiamo impostarlo come nuova "radice", rendendolo largo tutta l'immagine, allargando proporzionalmente tutti i blocchi sovrastanti e rimuovendo dall'immagine tutti gli altri blocchi non sovrastanti, facendo inoltre comparire in alto a sinistra un bottone `Reset Zoom` per riportare `all` alla radice.

In alto a destra, in grigio molto molto chiaro, possiamo trovare un bottone `Search`: cliccandolo, possiamo inserire una JavaScript Regular Expression, evidenziando in magenta tutti in cui `NOMEFUNZIONE` la soddisfa fino a quando il bottone non viene nuovamente premuto.

A destra del bottone `Search`, troviamo un bottone `ic`, che aggiunge/toglie il flag `ignoreCase` alla Regular Expression.

Tracce: entrypoint del programma

Guardando la parte blu dal basso verso l'alto, possiamo osservare lo stack del nostro programma nei momenti in cui è iniziata un'attesa.

In ordine:

1. `wakedemo`, la chiamata di avvio al nostro programma

2. `_start`, l'entrypoint del programma
3. `__libc_start_main`, l'inizializzazione di glibc[20]
4. `std::rt::lang_start::*`, l'entrypoint del runtime di Rust[21] e l'ultima funzione dello stack in user space

I quattro blocchi riempiono il 100% dello spazio orizzontale dell'immagine, indicando che quei frame sono stati presenti in tutte le cause di inizio attesa del nostro programma.

Mettendo il mouse sopra ad essi, possiamo scoprire la durata di tutte le attese off-CPU avvenute nel corso dell'esecuzione del programma: `(9.249.467 microsecs, 100.00%)`.

Tracce: cause di attesa

Continuando a ispezionare l'immagine, possiamo vedere una divisione al livello superiore a `std::rt::lang_start::*`, che la causa delle attese off-CPU è stata diversa in momenti diversi.

Da sinistra a destra, possiamo vedere:

- `alloc::vec::partial_eq`, l'implementazione di confronto tra slice di Rust[22], utilizzata dall'asserzione `assert_eq!` alla fine del programma
- `asm_exc_debug`, l'handler di un interrupt configurato dal debugger[23]
- `wakedemo::main`, la funzione main del nostro programma
    - `std::io::Read::read_to_end`, l'implementazione della lettura su un buffer in Rust[24], utilizzata per leggere il file
    - `std::io::Write::write`, l'implementazione della scrittura da un buffer in Rust[25], utilizzata per creare e scrivere sul file
    - `std::thread::sleep`, l'implementazione di un'attesa per un tempo predefinito in Rust[26], utilizzata per attendere due secondi tra scrittura e lettura

Tracce: confronto tra slice

Osserviamo che `alloc::vec::partial_eq` è stata causa di attese per `(1.686 microsecs, 0.08%)`.

Clicchiamo su `alloc::vec::partial_eq` per zoomarla, e continuiamo a salire ai livelli superiori per comprendere lo stack che ha causato l'attesa.

Vediamo una ulteriore divisione in tre:

- `asm_common_interrupt`, l'handler di un generico interrupt hardware, `(797 microsecs, 0.04%)`
- `asm_sysvec_apic_timer_interrupt`, l'handler di un interrupt dovuto a un timer ad alta precisione[27], `(571 microsecs, 0.03%)`
- `asm_sysvec_reschedule_ipi`, l'handler per la ricezione e il rescheduling di un interrupt hardware[28], `(318 microsecs, 0.02%)`

Possiamo capire che durante il confronto tra la slice ZER0ES in-RAM e quella letta dal file si è verificato un interrupt hardware esterno, la cui gestione è stata rimandata a dopo.

Tutti e tre gli stack continuano con:

1. `irqentry_exit_to_user_mode`, che configura il kernel per tornare in usermode in seguito ad un interrupt[29]
2. `schedule`, corrispondente al lavoro dello scheduler[30]
3. `__schedule`, corrispondente a una iterazione dello scheduler[31]

Essendo la gestione stata rimandata a dopo, questo è il kernel che rischedula l'esecuzione del nostro programma al primo momento possibile.

Sopra a `__schedule`, troviamo in grigio il separatore `--`, che separa lo stack che ha causato l'inizio di un'attesa da quello che ne ha causata la fine.

Poichè il thread è stato risvegliato nello stesso momento in cui è stato messo in attesa, non troviamo nulla sopra al separatore.

Tracce: debugging

Osserviamo che `asm_exc_debug` è stata causa di attese per `(915 microsecs, 0.04%)`.

Zoomandola, ne osserviamo lo stack:

1. `exc_debug_user`, funzione che delega la gestione dell'interrupt di un'eccezione di debug a un programma in user space
2. `irqentry_exit_to_user_mode`, che configura il kernel per tornare in usermode in seguito ad un interrupt[32]
3. `arch_do_signal_or_restart`, che verifica che ci sia un segnale da inviare a un processo prima di provare a consegnarlo[33]

4. `get_signal`, che controlla quale segnale deve essere inviato[34], determinando `SIGTRAP`[35]
5. `ptrace_stop`, che mette in pausa un processo impostandogli lo stato `TASK_TRACED`[36]
6. `schedule`, corrispondente al lavoro dello scheduler[37]
7. `__schedule`, corrispondente a una iterazione dello scheduler[38]

Capiamo che si è tratta di un breakpoint hardware, impostato attraverso uno dei registri `DR*` della CPU.

Risveglio

Possiamo osservare nuovamente il separatore in grigio `--`, ma questa volta sopra ad esso ci sono dei blocchi azzurri, che descrivono cosa ha risvegliato il nostro programma:

1. `lldb-server`, il nostro debugger in user space
2. `[unknown]`, probabilmente una funzione di LLDB che effettua una syscall `ptrace`
3. `entry_SYSCALL_64_after_hwframe`, l'entrypoint in kernel space della syscall
4. `do_syscall_64`, il gestore delle syscall
5. `__x64_sys_ptrace`, il gestore della syscall `ptrace` per l'architettura `x86_64`
6. `arch_ptrace`, il gestore della syscall `ptrace` agnostico all'architettura del processore
7. `ptrace_request`, che elabora la syscall, probabilmente `PTRACE_CONT`[39], rimuovendo lo stato `TASK_TRACED`[40]

Capiamo quindi che si tratta di un breakpoint che `lldb` ha deciso di ignorare silenziosamente.

Tracce: lettura

Osserviamo che `std::io::Read::read_to_end` è stata causa di attese per `(11.996 microsecs, 0.59%)`.

La zoomiamo, e otteniamo una bella visualizzazione di tutte le funzioni kernel coinvolte nella lettura di dati da file in un filesystem btrfs.

Ancora, in ordine gerarchico, alcune che notiamo sono:

1. `std::io::default_read_to_end`, metodo privato della standard library di Rust
2. `read`, syscall di glibc

3. `[unknown]`, probabilmente una funzione interna di glibc che effettua la syscall vera e propria
4. `entry_SYSCALL_64_after_hwframe`, l'entrypoint in kernel space della syscall
5. `do_syscall_64`, il gestore delle syscall
6. `ksys_read`, funzione principale per la lettura da disco
7. ...

Notiamo anche che nessuna di queste stack ha blocchi sopra il separatore `--`, ma anche che tutti gli stack terminano con `preempt_schedule_irq`[41]; questo indica che tutte le attese sono terminate su un altro core prima che il processo potesse essere interrotto e schedulato da quello attuale[42].

Tracce: scrittura

Osserviamo che `std::io::Write::write` è stata causa di attese per `(18.697 microsecs, 0.92%)`.

Zoomiamo anche su questo, e vediamo una stack estremamente simile a quella per la lettura:

1. `write`, syscall di glibc
2. `[unknown]`, probabilmente una funzione interna di glibc che effettua la syscall vera e propria`
3. `entry_SYSCALL_64_after_hwframe`, l'entrypoint in kernel space della syscall
4. `do_syscall_64`, il gestore delle syscall
5. `ksys_write`, funzione principale per la scrittura su disco
6. ...

Anche qui si verifica lo stesso fenomeno svoltosi in lettura: tutti gli stack di attesa terminano con `preempt_schedule_irq`[43], e non hanno uno stack di risveglio.

Tracce: sleep

Osserviamo che `std::thread::sleep` è stata causa di attese per `(2.000.062 microsecs, 98.34%)`.

Il suo stack è visibile senza bisogno di zoom, essendo larga il 98.34% dell'immagine:

1. `nanosleep`, chiamata di sistema che sospende l'esecuzione del programma per una certa durata[44]

2. `clock_nanosleep`, altra chiamata di sistema che sospende l'esecuzione del programma fino a un certo momento[45]
3. `entry_SYSCALL_64_after_hwframe`, l'entrypoint in kernel space della syscall
4. `do_syscall_64`, il gestore delle syscall`
5. `__x64_sys_clock_nanosleep`, il gestore della syscall `clock_nanosleep`
6. `common_nsleep`, che gestisce nanosleep per i real-time clock[46]
7. `hrtimer_nanosleep`, che configura un timer ad alta risoluzione per gestire la nanosleep[47]
8. `do_nanosleep`, che modifica effettivamente lo stato del thread per poi avviare il timer ad alta risoluzione[48]
9. `schedule`, corrispondente al lavoro dello scheduler[49]
10. `__schedule`, corrispondente a una iterazione dello scheduler[50]

Tempo in eccesso

Stranamente, però, la durata misurata dell'attesa è stata più lunga di `2.000.000 microsecs`, nonostante `nanosleep` permetta di creare attese dalla durata ad alta risoluzione.

Per spiegare i restanti 62 microsecondi dobbiamo aprire una parentesi su quando `offwaketime` prende misure del tempo:

- la misura di inizio viene presa prima che la funzione kernel `finish_task_switch` venga eseguita[51], ovvero dopo l'esecuzione dello stack di attesa e dopo aver cambiato stato al thread
- la misura di fine viene presa prima che la funzione kernel `try_to_wake_up` venga eseguita[52], ovvero dopo l'esecuzione dello stack di risveglio e prima di aver cambiato stato al thread

Il tempo di esecuzione del cambio di stato e dello stack di risveglio è quindi incluso nella misura, e corrisponde ai 62 microsecondi in eccesso.

Risveglio

Ancora una volta troviamo il separatore, `--`, ma questa volta è seguito dallo stack di risveglio, dato che il thread è stato effettivamente messo in attesa:

1. `ktimers/*`
2. `ret_from_fork_asm`

3. `ret_from_fork`
4. `kthread`
5. `smpboot_thread_fn`
6. `run_timersd`
7. `handle_softirqs.isra.0`
8. `hrtimer_run_softirq`
9. `__hrtimer_run_queues`
10. `hrtimer_wakeup`

Senza entrare in dettaglio su come funzioni la gestione degli interrupt dovuti ai timer ad alta precisione in un kernel `PREEMPT_RT`, possiamo capire che la causa del risveglio è stato un interrupt software dovuto al completamento del timer.

Altre tracce

Notiamo che nel flamegraph non appaiono nè il debugger che ha messo il processo in pausa inizialmente, nè la chiamata per la creazione del file, nè quelle per la chiusura dei file descriptor, nè la chiamata finale a `sleep`.

Le chiamate di creazione e chiusura non sono presenti in quanto l'attesa relativa ad esse ha avuto una durata inferiore a 1 microsecondo; sarebbe stato lo stesso per `read` e `write` se il buffer scritto all'interno nel file non avesse avuto la dimensione esagerata di 512 MiB.

L'interrupt del debugger e la seconda `sleep` invece sono assenti perchè l'attesa relativa ad esse ha rispettivamente *inizio prima dell'inizio della profilazione* e *fine dopo la fine della stessa*, che porta `offwaketime` a ignorarle[53].

Ricerca: timer ad alta risoluzione su diversi kernel

Vogliamo farci un'idea di come si comportino i timer ad alta risoluzione su diverse build del kernel.

Realizzazione del programma

Creiamo un piccolo programma C che esegua un milione di nanosleep dalla durata di 10 nanosecondi ciascuno, per un totale di 10 secondi:

```
#include <time.h>

int main(void) {
    const struct timespec wait = {
        .tv_sec = 0L,
        .tv_nsec = 10L,
    };

    for(int i = 0; i < 1000000; i++) {
        nanosleep(&wait, NULL);
    }

    return 0;
}
```

Esecuzione del programma

Creiamo poi uno script shell che compili il programma, lo esegua in background, e vi connetta la pipeline `offwaketime-flamegraph` appena possibile:

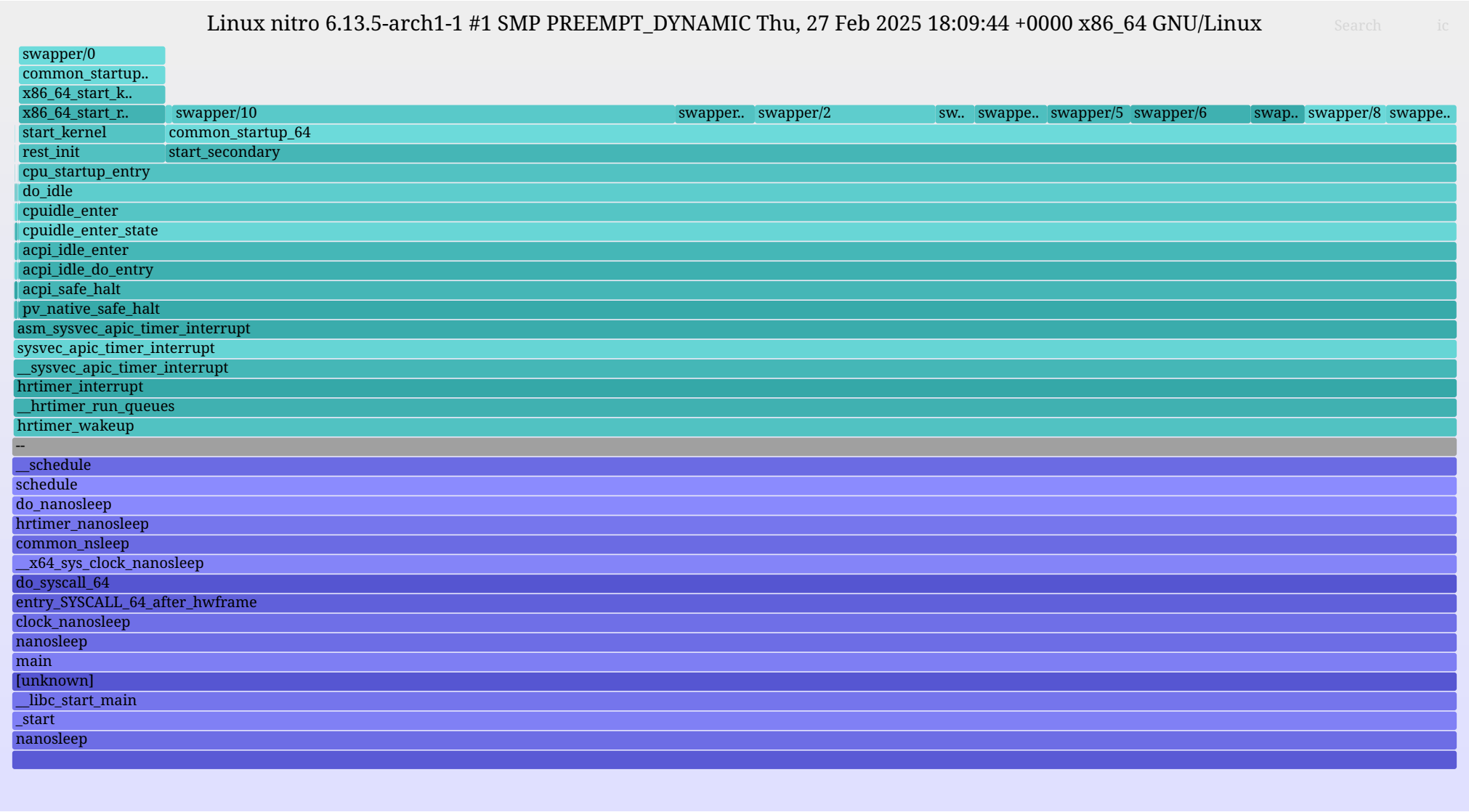
```
#!/usr/bin/env fish

make nanosleep

sudo echo

./nanosleep &

sudo offwaketime 10 \
    -p $last_pid \
    -f \
```

```
| flamegraph \
    --color='chain' \
    --countname='microsecs' \
    --title=(uname -a) \
> nanosleep.svg
```

Eseguiamo infine lo script con kernel diversi per vedere come cambia il risultato.

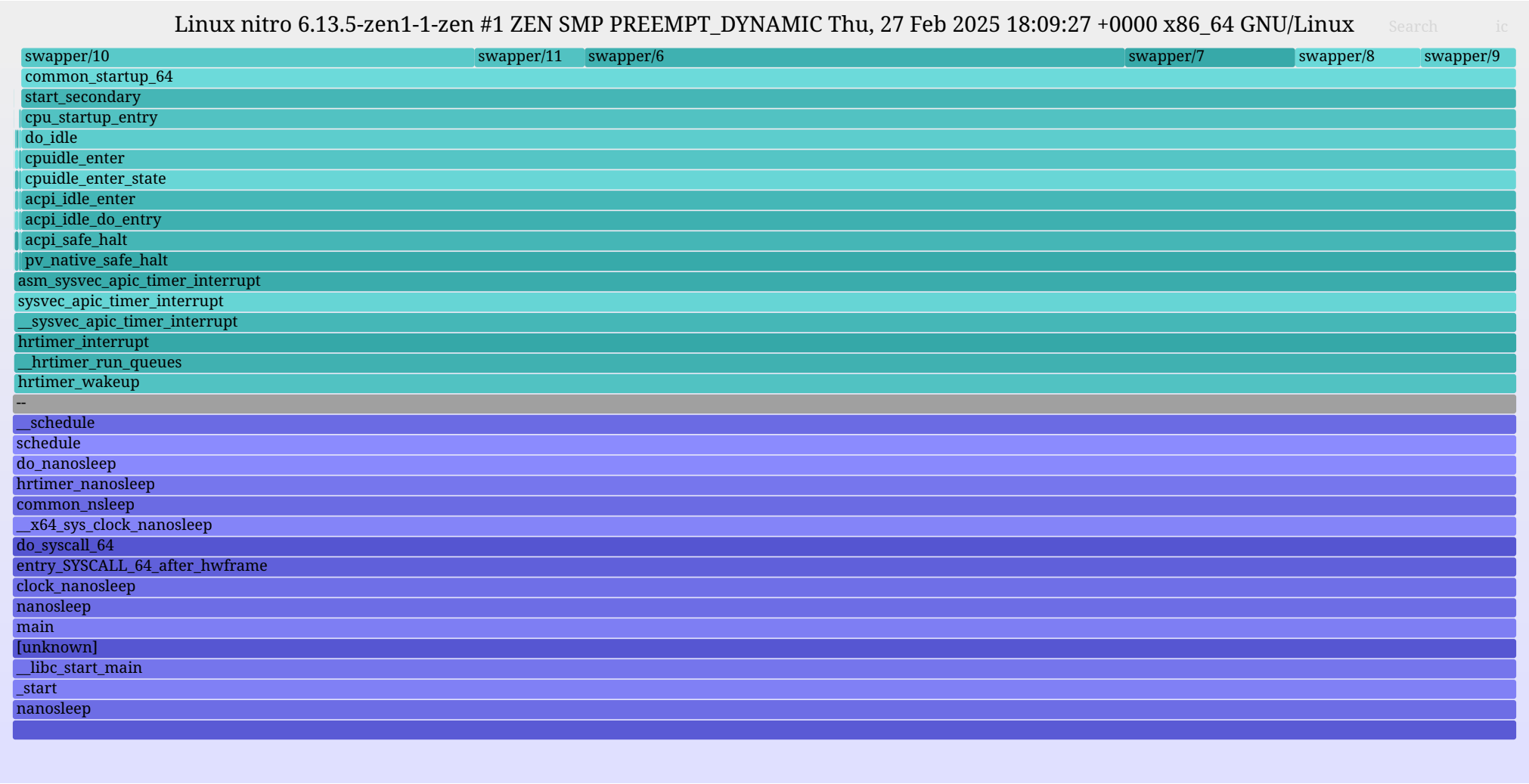
Kernel `linux`

Possiamo vedere che nella gran parte dei casi il programma viene schedulato (`schedule`) e poi risvegliato da un interrupt (`asm_sysvec_apic_timer_interrupt`) mentre la CPU è idle (`acpi_idle_enter` e simili), ma in una frazione molto ridotta dei casi (circa 76 dei 822845 catturati) il task viene direttamente risvegliato dalla stessa chiamata che lo ha messo in attesa (`schedule` sia sopra sia sotto).



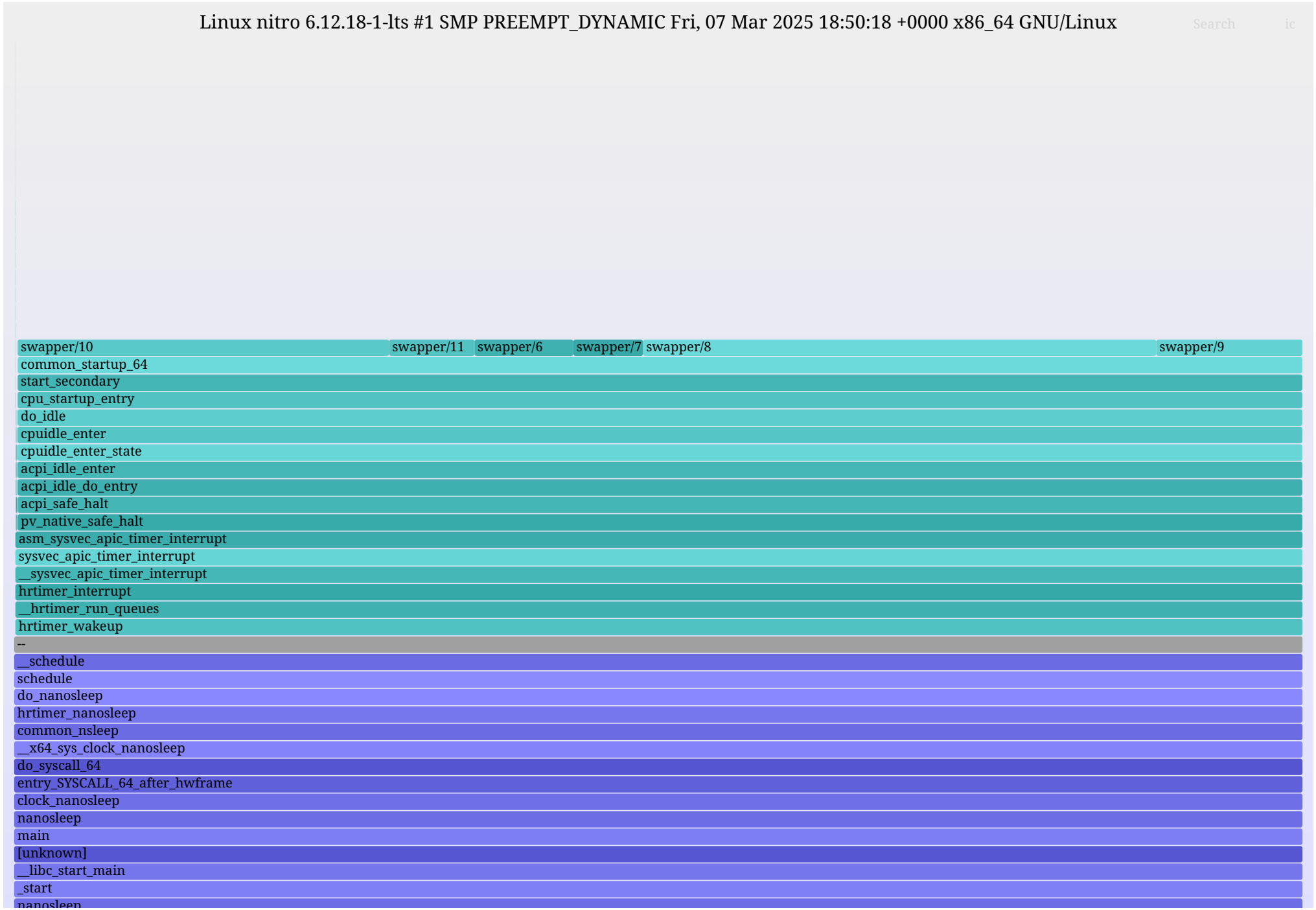
Kernel `linux-zen`

Non si presentano differenze significative negli stack di risveglio dal precedente kernel.



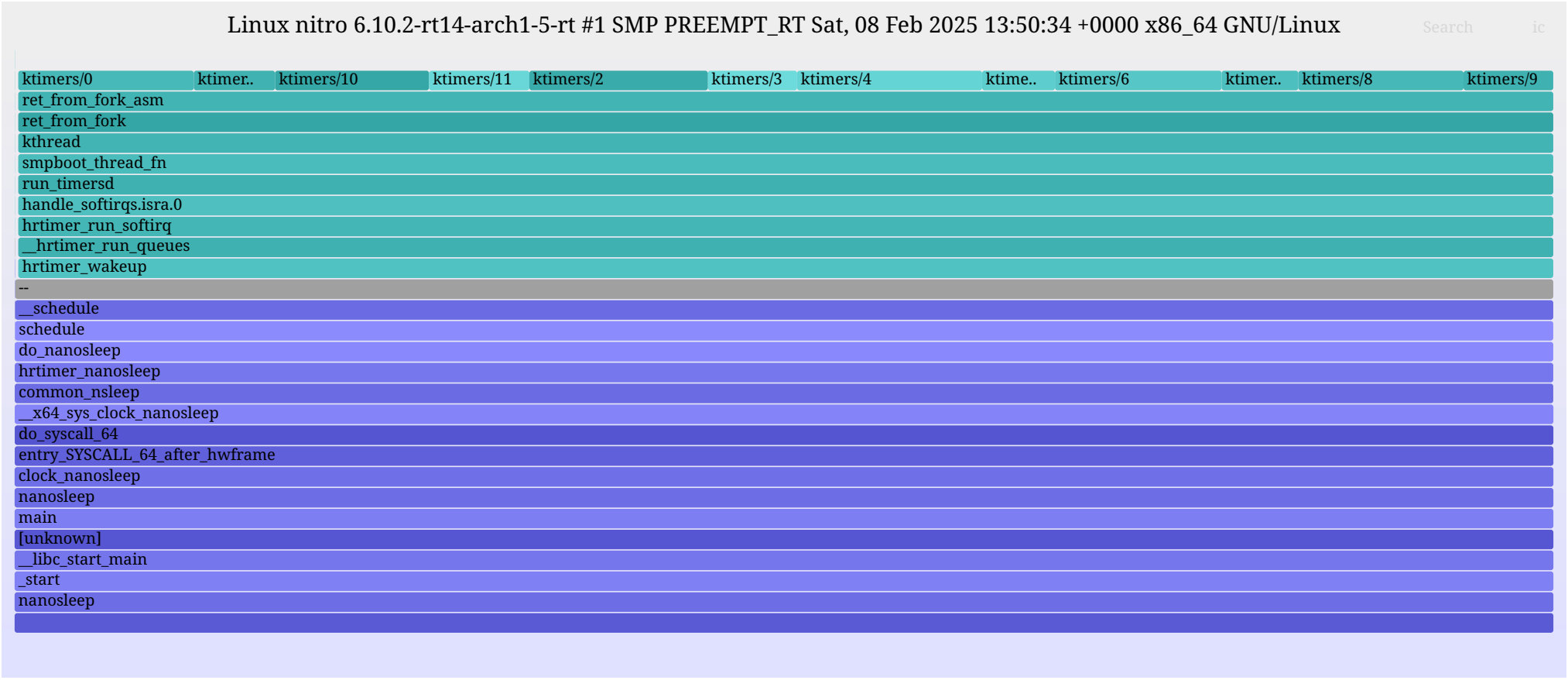
Kernel `linux-lts`

Neanche con questo kernel possiamo osservare differenze significative dagli stack degli altri due, con l'eccezione di circa 150 casi in cui il call stack del risveglio è sconosciuto (`[unknown]`).


Linux nitro 6.12.18-1-lts #1 SMP PREEMPT_DYNAMIC Fri, 07 Mar 2025 18:50:18 +0000 x86_64 GNU/Linux
Search
ic
swapper/10
swapper/11
swapper/6
swapper/7
swapper/8
swapper/9
common_startup_64
start_secondary
cpu_startup_entry
do_idle
cpuidle_enter
cpuidle_enter_state
acpi_idle_enter
acpi_idle_do_entry
acpi_safe_halt
pv_native_safe_halt
asm_sysvec_apic_timer_interrupt
sysvec_apic_timer_interrupt
__sysvec_apic_timer_interrupt
hrtimer_interrupt
__hrtimer_run_queues
hrtimer_wakeup
--
__schedule
schedule
do_nanosleep
hrtimer_nanosleep
common_nsleep
__x64_sys_clock_nanosleep
do_syscall_64
entry_SYSCALL_64_after_hwframe
clock_nanosleep
nanosleep
main
[unknown]
__libc_start_main
_start
nanosleep

Kernel `linux-rt`

Questo kernel, che ha `CONFIG_PREEMPT_RT` abilitato, si comporta in maniera significativamente diversa dagli altri, che invece usano `CONFIG_PREEMPT_DYNAMIC`: non sono interrupt a risvegliare il nostro programma, bensì thread dedicati alla gestione dei timer ad alta risoluzione (`run_timersd`), che poi effettuano un "interrupt" a livello di software (`hrtimer_run_softirq`)[54].



Ricerca: stima del tempo di risveglio necessario in seguito a nanosleep

È possibile sfruttare il modo in cui `offwaketime` prende le misure di tempo per determinare quanto tempo viene impiegato per risvegliare il task dalle attese.

Quando usiamo `nanosleep`, siamo a conoscenza a priori della durata dell'attesa, quindi possiamo sottrarre quel tempo alla misura presa per ottenere il tempo di risveglio:

$$nanosleep + wakeup = measure$$

Ovvero, ipotizzando di chiamare `nanosleep` con una attesa di 1 secondo:

$$wakeup = measure - 1\,000\,000\ \mu s$$

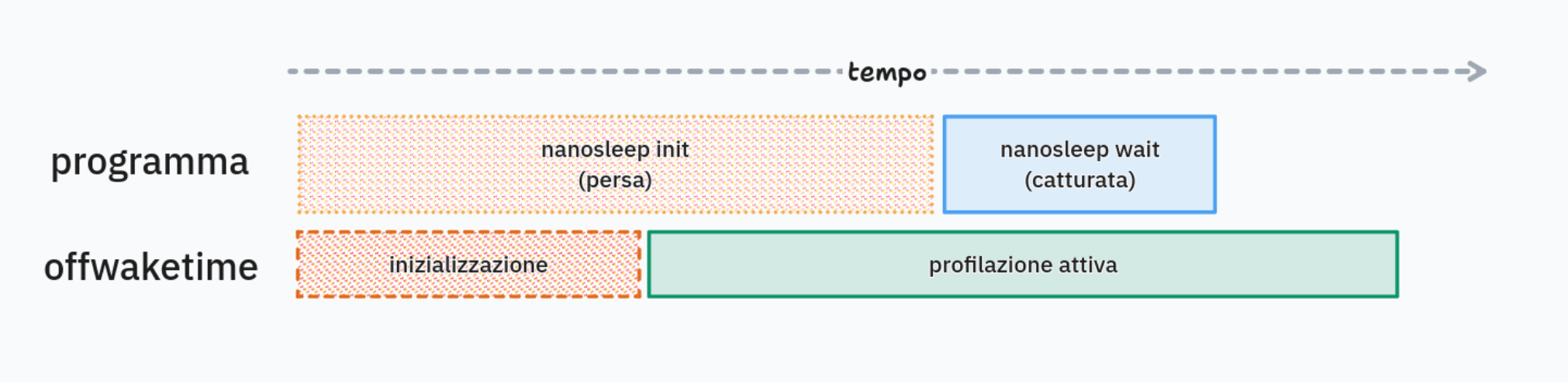
Realizzazione del programma

Ricordando che `offwaketime` cattura tracce solamente delle attese che iniziano e finiscono mentre è attivo, e che impiega tra i 2 e i 4 secondi per attivarsi, realizziamo un programma C che gli dia sufficiente tempo di avviarsi e connettersi, poi chiami una `nanosleep` dalla durata fissa di 1 secondo:

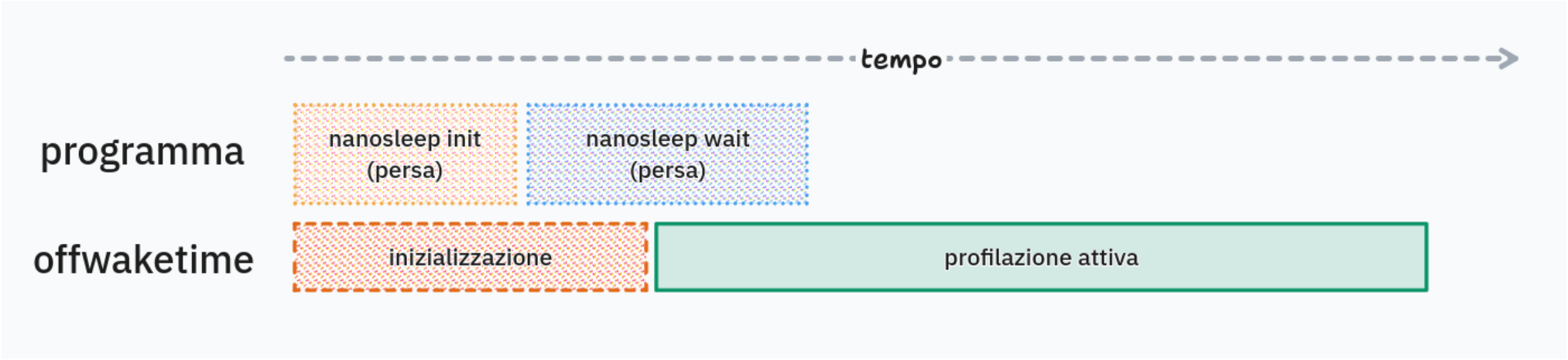
```
#include <time.h>

int main(void) {
    // empirically determined, may require different values on different systems
    const struct timespec init = {
        .tv_sec = 5L,
        .tv_nsec = 0L,
    };

    nanosleep(&init, NULL);

    const struct timespec wait = {
        .tv_sec = 1L,
        .tv_nsec = 0L,
```

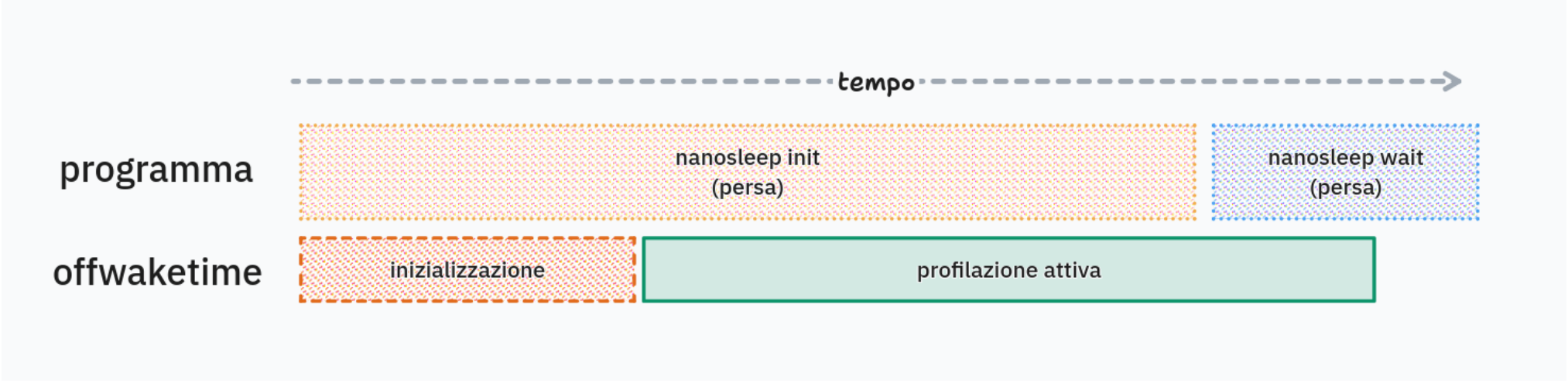
```
    };

    nanosleep(&wait, NULL);

    return 0;
}
```

L'idea è quella di trovarci in casi come il seguente:



E non come i seguenti:





Esecuzione del programma

Innanzitutto, creiamo uno script shell che esegua il programma ed emetta in output il numero di microsecondi misurati da `offwaketime`:

```
#!/usr/bin/env fish

sudo echo
```

```
make nanosleep

./nanosleep &

sudo /usr/share/bcc/tools/offwaketime 7 --pid "$last_pid" --folded | string trim | grep do_nanosleep |
cut --delimiter=" " --fields=2
```

Ora, facciamo in modo che ripeta la misura 1000 volte, saltando i casi in cui essa viene persa, catturata due volte, o nanosleep viene interrotta anticipatamente, e facendogli scrivere i risultati su file:

```
#!/usr/bin/env fish

sudo echo

make nanosleep

set kernel (uname -r)

for i in (seq 1000)
    ./nanosleep &

    set time (sudo /usr/share/bcc/tools/offwaketime 7 --pid "$last_pid" --folded | string trim | grep
do_nanosleep | cut --delimiter=" " --fields=2)

    for t in $time
        set adjusted (math "$t - 1_000_000")

        if [ "$adjusted" -gt 0 ]
            echo "$adjusted" | tee --append "$kernel.txt"
        else
```

```
            echo "[invalid]" > /dev/stderr
        end
    end
end
```

Kernel linux-rt

Dopo un'esecuzione, otteniamo un file con 849 misure, che si riportano qui senza newline per evitare di allungare il report:

```
99 69 68 69 181 90 78 71 86 67 73 208 78 75 70 70 175 70 70 70 75 82 69 73 71 71 70 73 88 87 95 95 73
102 74 75 85 69 66 81 79 63 221 79 92 74 73 82 91 77 67 70 72 95 65 96 84 102 187 86 89 109 81 67 71
233 67 90 96 181 96 521 105 177 93 69 81 137 77 76 74 66 72 89 86 74 190 67 68 81 69 93 68 67 73 107
100 72 105 106 99 81 71 117 104 88 106 98 75 192 101 101 71 102 101 87 104 113 66 226 101 103 82 99 104
167 104 103 90 557 224 103 70 107 216 98 86 89 196 227 86 104 105 104 116 110 303 86 108 81 82 90 89 75
217 69 99 67 121 92 105 74 103 104 208 89 237 92 69 74 83 251 104 212 186 102 88 80 104 70 92 122 331
203 217 71 80 104 104 66 122 105 120 220 100 102 97 684 108 80 82 107 98 70 73 105 105 98 216 82 112
103 223 73 99 241 111 202 107 103 71 199 101 756 206 282 110 148 163 96 112 72 87 85 125 103 72 100 70
121 84 69 107 71 100 205 96 800 110 175 150 100 217 105 100 105 82 114 96 94 94 182 60 107 102 80 89
165 97 96 73 90 87 210 248 107 226 523 90 83 93 105 61 102 116 80 81 104 91 105 199 66 73 222 111 128
88 72 91 706 98 78 76 104 108 88 105 99 89 220 117 119 103 70 125 107 109 87 98 102 226 90 92 80 72 106
184 89 105 87 216 108 88 86 107 88 96 217 175 106 217 222 88 132 106 86 99 100 147 530 224 106 85 86
111 229 107 93 218 86 113 95 238 92 121 87 66 106 233 105 114 77 108 88 509 68 96 71 284 167 118 569
221 84 108 96 216 66 261 82 336 105 105 86 70 100 98 87 111 69 70 77 83 84 124 73 122 63 103 100 70 106
107 87 68 84 218 234 86 116 181 86 87 94 87 84 86 175 68 90 181 1006 88 82 108 101 189 106 73 70 99 88
111 82 69 80 87 75 86 89 90 66 95 74 79 82 84 88 88 75 172 84 69 286 73 83 83 173 116 83 81 67 82 112
71 102 108 69 245 108 105 122 96 70 69 105 335 109 166 310 74 82 225 104 74 101 84 89 73 86 82 86 102
108 111 117 148 113 105 68 111 95 102 104 86 68 282 89 245 83 67 245 100 212 78 102 161 93 69 105 124
677 69 97 69 215 109 103 80 142 152 652 90 86 66 103 80 104 77 75 82 786 74 80 79 114 254 93 83 224 79
105 81 101 91 94 107 107 234 104 166 80 106 502 70 93 215 104 105 199 530 73 106 99 69 72 255 119 107
82 146 81 88 71 73 118 90 91 86 478 102 104 106 67 113 82 92 99 159 97 97 107 73 77 263 430 98 65 216
133 105 225 513 82 74 128 214 103 72 86 164 520 71 85 293 91 111 85 113 524 104 102 107 124 117 114 83
```

```
177 104 209 83 94 206 68 112 97 200 108 105 280 75 88 68 69 87 172 71 106 74 100 61 91 82 78 222 79 166
87 104 86 199 77 239 100 94 89 76 73 75 74 92 260 174 171 91 301 87 94 72 71 57 86 66 122 106 68 70 112
103 298 80 116 95 91 82 71 76 86 72 92 150 85 75 67 75 85 73 70 70 76 73 90 77 82 88 79 113 79 90 99 69
85 76 92 72 65 77 80 168 71 82 86 176 95 78 105 74 79 73 65 73 77 72 78 100 74 85 77 187 91 100 68 70
69 115 73 88 71 70 68 69 67 73 107 69 75 84 68 102 90 75 71 87 239 67 100 170 69 78 87 88 70 86 72 281
83 77 94 69 74 86 387 72 86 108 72 167 241 106 84 76 71 71 88 78 90 60 154 177 75 93 73 69 80 72 65 68
69 90 86 68 74 84 70 74 82 72 78 92
```

Da esse, ricaviamo:

- media: ***120 µs***
- deviazione standard: ***95 µs***
- mediana: ***92 µs***
- 90esima percentile: ***216 µs***
- 99esima percentile: ***563 µs***