

offwaketime - rev4

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-formatto:

```
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::h52fdaccdcfcdbd854
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 (μ s):

```
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` precedentemente 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 crea un file, vi scrive all'interno, e chiude il relativo file descriptor, aspettando di vedere tracce delle attese avvenute per il debug e potenzialmente di attese avvenute per l'interazione con il file.

Verifica della versione del kernel in esecuzione

Innanzitutto, verifichiamo la versione del kernel attualmente in esecuzione:

```
uname -a
```

```
Linux nitro 6.13.4-arch1-1 #1 SMP PREEMPT_DYNAMIC Sat, 22 Feb 2025 00:37:05 +0000 x86_64 GNU/Linux
```

Realizzazione del programma di esempio

Il programma Rust realizzato è il seguente:

```
use std::fs::File;
use std::io::Write;
use std::thread::sleep;
use std::time::Duration;

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

    let _ = file.write(b"hello world")
        .expect("File write failed");
```

```
// unnecessary; explicitly closes the file descriptor for clarity
drop(file);

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

Esecuzione del programma di esempio

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.

Con il programma in pausa, avviamo un terminale come superutente:

```
sudo -i
```

Identificazione del pid del programma di esempio

Da lì, 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:

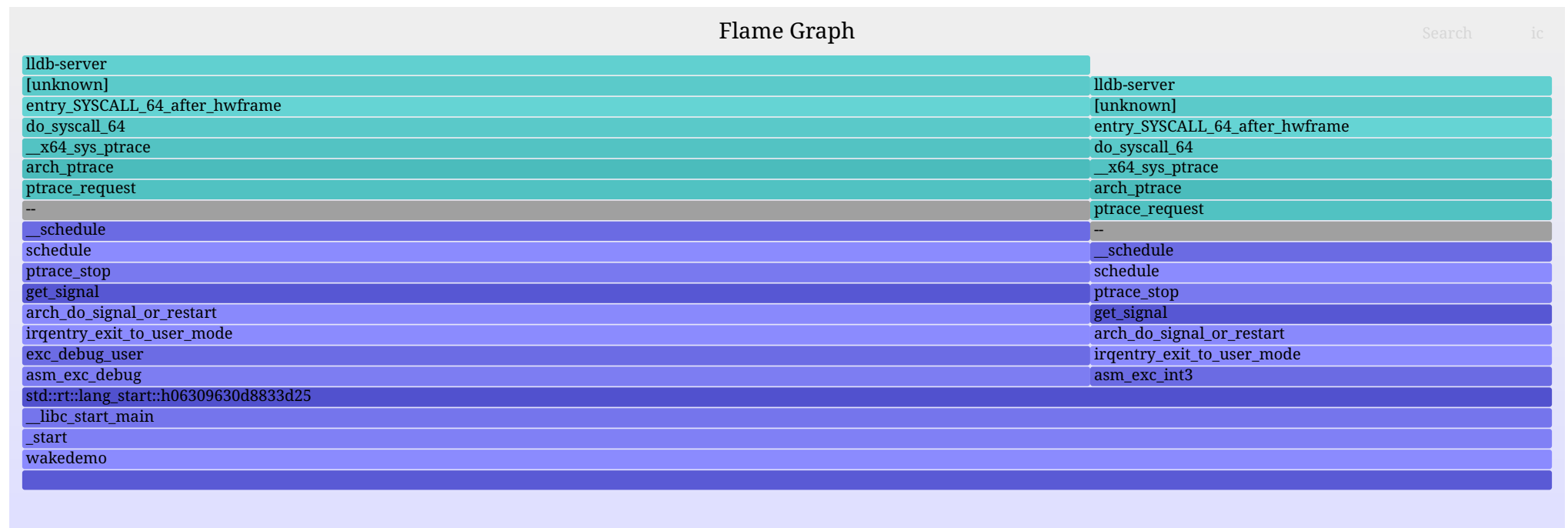
```
offwaketime 30 -p 22353 -f | flamegraph --color='chain' --countname='microsecs' > /root/wakedemo.svg
```

Ora che offwaketime è in ascolto, effettuiamo step di una linea di codice alla volta fino a quando il programma non raggiunge l'ultima riga.

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

Alleghiamo poi l'immagine prodotta a questo documento:

```
mv /root/wakedemo.svg ~/wakedemo.svg
chown steffo: ~/wakedemo.svg
```



Analisi del risultato

Infine, analizziamo l'immagine (aprendola in un browser web grafico con JavaScript abilitato per garantirne l'interattività).

Descrizione dell'immagine

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.

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.

Stack di inizio attesa (blu), parte comune

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^[19]
4. `std::rt::lang_start::_`, l'entrypoint del runtime di Rust^[20] 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 vedere la loro durata, ovvero il tempo totale che il nostro programma è stato in attesa di eventi off-CPU: (14.718.328 microsecs, 100.00%).

Stack di inizio attesa (blu), parte divisa

Continuando a ispezionare l'immagine, possiamo vedere una divisione, che indica che nei momenti di attesa per attività off-CPU lo stack è stato diverso:

- `asm_exc_debug`, seguito da `exc_debug_user`, l'handler della DEBUG exception del processore, innescata dal raggiungimento di un breakpoint
- `asm_exc_int3`, l'handler della INT3 exception del processore, innescata anch'essa dal raggiungimento di un breakpoint

Osserviamo i loro parametri mettendo il mouse sopra ad essi:

- `asm_exc_debug` è stato la causa di attese per una durata totale di (10.205.062 microsecs, 69.81%)
- `asm_exc_int3` è stato la causa di attese per una durata totale di (4.412.246 microsecs, 30.19%)

Entrambi gli stack poi chiamano entrambi:

1. `irqentry_exit_to_user_mode`, che configura il kernel per delegare la gestione di un interrupt^[21]
2. `arch_do_signal_or_restart`, che verifica che ci sia un segnale da inviare a un processo prima di provare a consegnarlo^[22]
3. `get_signal`, che controlla quale segnale deve essere inviato^[23], determinando SIGTRAP^[24]

4. `ptrace_stop`, che mette in pausa un processo impostandogli lo stato `TASK_TRACED`^[25]
5. `schedule`, corrispondente al lavoro dello scheduler^[26]
6. `__schedule`, corrispondente a una iterazione dello scheduler^[27]

Separatore (grigio)

Il separatore in grigio -- denota che lo stack del programma ha termine lì.

Possiamo capire quindi che tutti gli eventi off-CPU che sono avvenuti sono le interruzioni avvenute in corrispondenza al nostro stepping del programma tramite il debugger.

Stack di risveglio (azzurro)

Guardiamo ora cos'ha causato il risveglio del nostro programma.

Lo stack del processo risvegliante è la parte in azzurro in cima al flame graph, in cui viene commesso un abuso di notazione per facilitare la lettura delle tracce elencando i frame dall'alto verso il basso^[28].

Questo significa che, in entrambi i casi, il processo del nostro programma è stato risvegliato dal seguente stack:

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_SINGLESTEP`, rimuovendo lo stato `TASK_TRACED`^[29]

Capiamo quindi che la causa del risveglio è stata quindi una richiesta del nostro debugger di far continuare il nostro processo.

Ulteriori informazioni

Notiamo che nel flamegraph non appaiono nè la chiamata a `sleep`, nè chiamate di sistema per la creazione e scrittura su file.

sleep è assente perchè l'attesa relativa ad essa ha termine *dopo* la chiusura del profiler, e quindi non viene registrata.

Le chiamate di sistema per la creazione e scrittura su file non sono presenti perchè non avendo inserito una chiamata a flush, LLVM ha deciso di rimuovere interamente drop, lasciando che il file buffer venga flushato automaticamente al termine del programma.

Caso di studio: timer ad alta risoluzione su diversi kernel

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;
}
```

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.

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).

```
swapper/0
common_startup..
x86_64_start_k..
x86_64_start_r.. swapper/10 swapper.. swapper/2 sw.. swappe.. swapper/5 swapper/6 swap.. swapper/8 swappe..
start_kernel common_startup_64
rest_init 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
```

linux-zen

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

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					

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]).

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					

XXXXXXXXXX

[illegible]

ktimers/0	ktimer..	ktimers/10	ktimers/11	ktimers/2	ktimers/3	ktimers/4	ktimer..	ktimers/6	ktimer..	ktimers/8	ktimers/9
ret_from_fork_asm											
ret_from_fork											
kthread											
smpboot_thread_fn											
run_timersd											
handle_softirqs.isra.0											
hrtimer_run_softirq											
__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											