offwaketime - rev2

`offwaketime` è un profiler per thread su Linux che sfrutta eBPF per catturare informazioni nei momenti in cui un thread entra in attesa di un evento esterno alla CPU e in quelli in cui esso viene risvegliato dall'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
```

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

```
offwaketime -f
```

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]
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

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
```

Si vuole effettuare la profilazione 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 syscall effettuate a tale scopo.

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)
}
```

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
```

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.

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

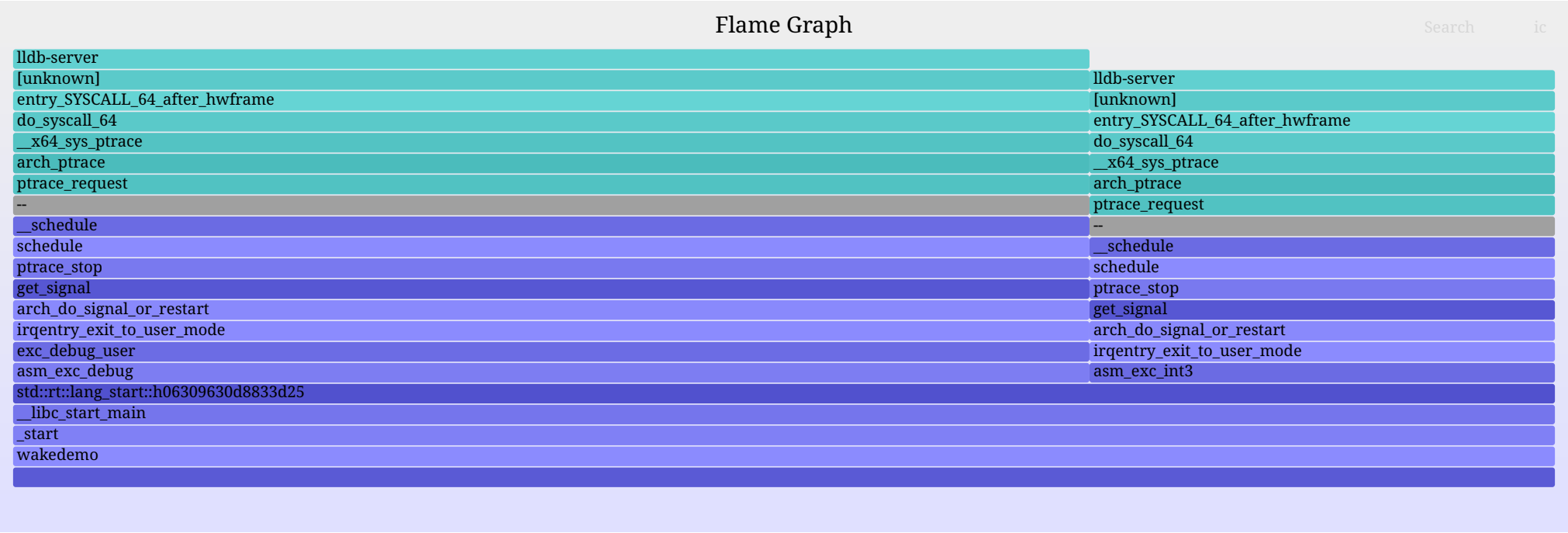
```
offwaketime 30 -p 22353 -f | flamegraph --color='chain' --countname='microsecs' > /root/wakedemo.svg
```

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
```



Infine, analizziamo l'immagine.

Guardando la parte blu dal basso verso l'alto, possiamo osservare lo stack del nostro programma nei momenti in cui si è verificata 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

Poi, c'è una divisione, che indica che nei momenti di attesa per attività off-CPU lo stack è stato diverso:

1. `asm_exc_debug`, seguito da `exc_debug_user`, era presente 69.81% del tempo; è l'handler della `DEBUG` exception del processore, innescata dal raggiungimento di un breakpoint

2. `asm_exc_int3` era presente 30.19% del tempo; è l'handler della `INT3` exception del processore, innescata anch'essa dal raggiungimento di un breakpoint

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]

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.

Guardiamo ora cos'ha causato il risveglio del nostro programma; lo stack del processo risvegliante è parte in azzurro in cima al flame graph, con le tracce dei frame elencate dall'alto verso il basso.

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`[28]

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

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:

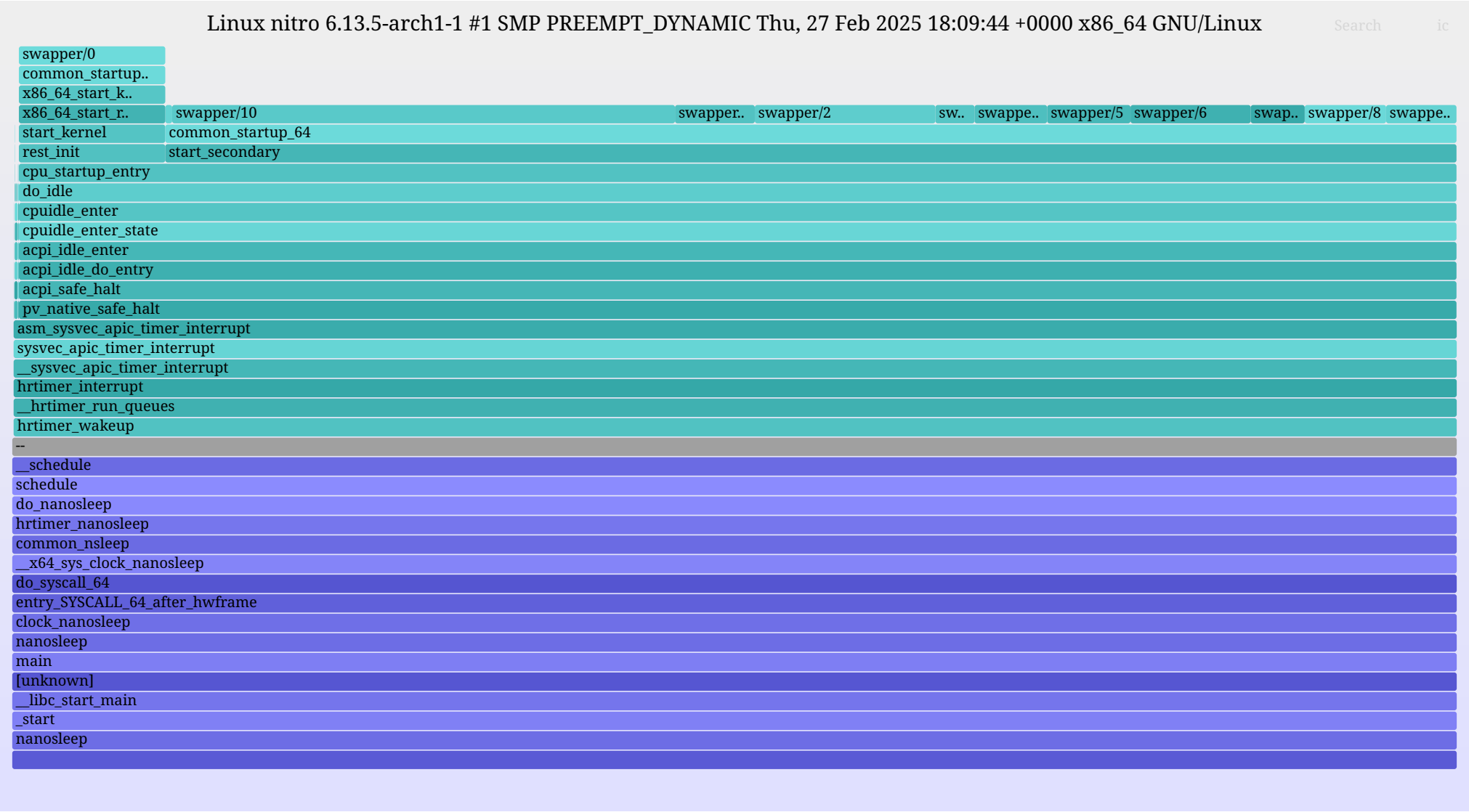
```
#!/usr/bin/env fish

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.

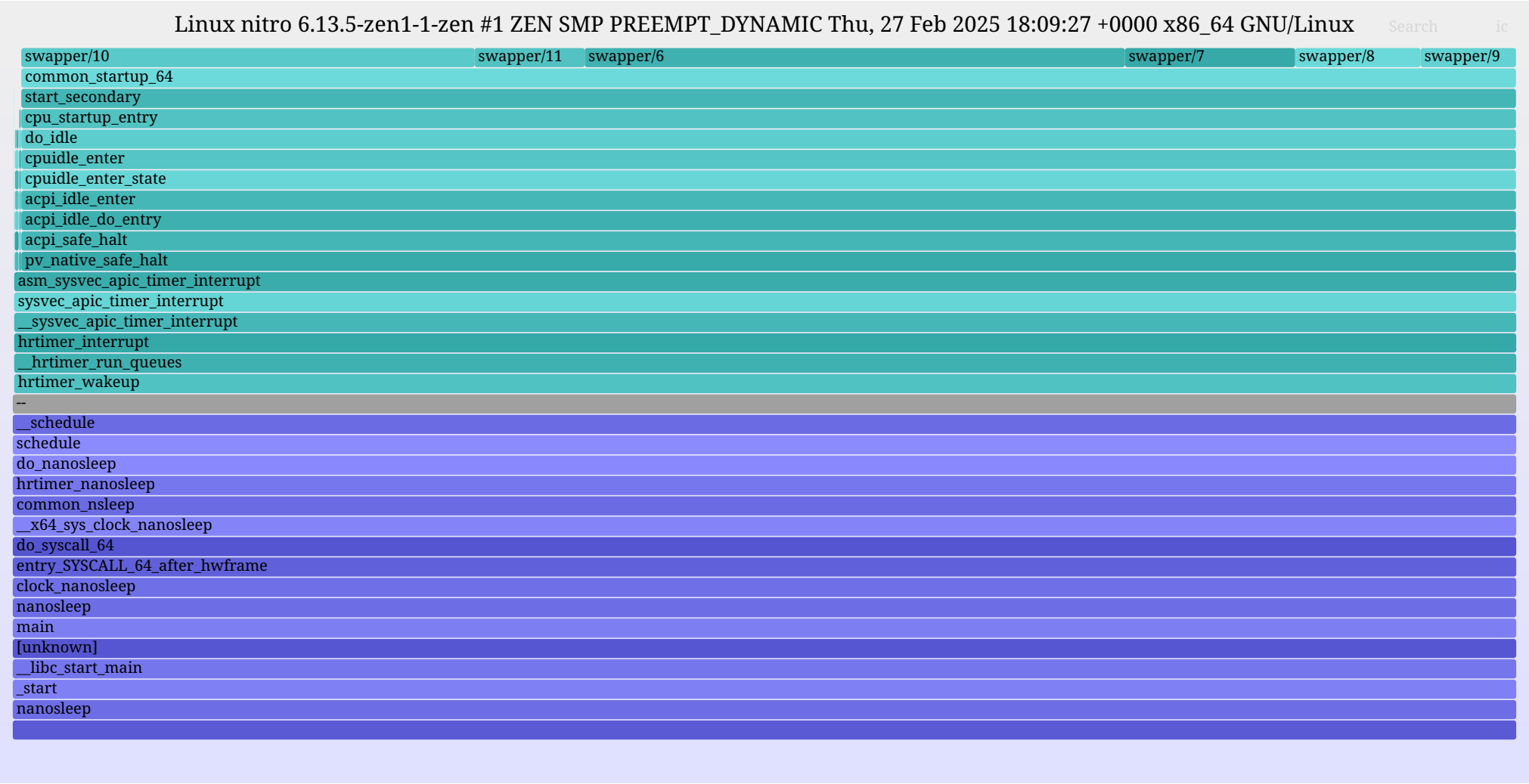
linux

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).



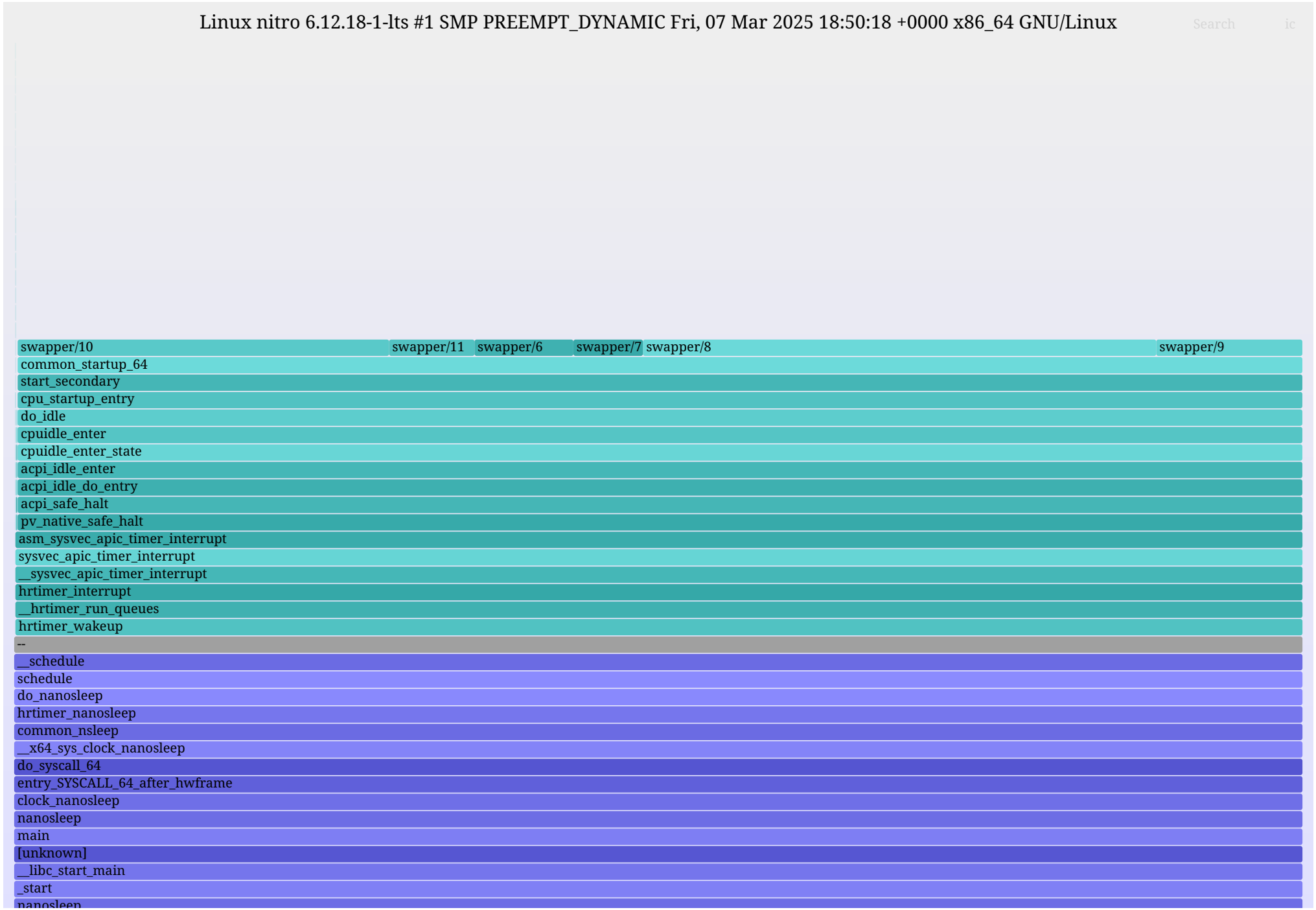
linux-zen

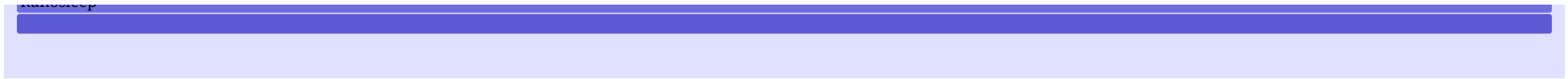
Non si presentano differenze significative negli stack di risveglio dal precedente kernel.



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



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`)[29].