Kernel dynamic memory allocation tracking and reduction

Consumer Electronics Work Group Project
2012

Ezequiel García

Memory reduction and tracking

- Why do we care?
  - Tiny embedded devices (but really tiny)
  - Virtualization: might be interesting to have really small kernels

- How will we track?
  - ftrace

Kernel memory

Static memory

- Static footprint == kernel code (text) and data

- Simple accounting: **size** command

```
$ size fs/ramfs/inode.o
text    data     bss      dec     hex   filename
1588     492       0     2080     820   fs/ramfs/inode.o
```

Static memory

- The **readelf** command

```
$ readelf fs/ramfs/inode.o -s | egrep "FUNC|OBJECT"
[extract]
Num:    Value  Size Type    Bind  Vis      Ndx Name
22:  000002a8   168 FUNC    LOCAL DEFAULT    1 ramfs_mknod
26:  00000350    44 FUNC    LOCAL DEFAULT    1 ramfs_mkdir
28:  00000388   224 FUNC    LOCAL DEFAULT    1 ramfs_symlink
44:  000001c8    28 OBJECT  LOCAL DEFAULT    3 rootfs_fs_type
```

Dynamic memory

How do we allocate memory?

- Almost every architecture handles memory in terms of **pages**. On x86: 4 KiB.
- alloc_page(), alloc_pages(), free_pages()
- Multiple pages are acquired in sets of 2^N number of pages

Dynamic memory

How do we allocate memory?

- SLAB allocator allows to obtain smaller chunks
- Comes in three flavors: SLAB, SLOB, SLUB
- Object cache API: kmem caches

Dynamic memory

How do we allocate memory?

- SLAB allocator allows to obtain smaller chunks
- Comes in three flavors: SLAB, SLOB, SLUB
- Object cache API: kmem caches
- Generic allocation API: kmalloc()

Dynamic memory

How do we allocate memory?

- SLAB allocator allows to obtain smaller chunks
- Comes in three flavors: SLAB, SLOB, SLUB
- Object cache API: kmem caches
- Generic allocation API: kmalloc()

Wastes memory

Dynamic memory

How do we allocate memory? **vmalloc()**

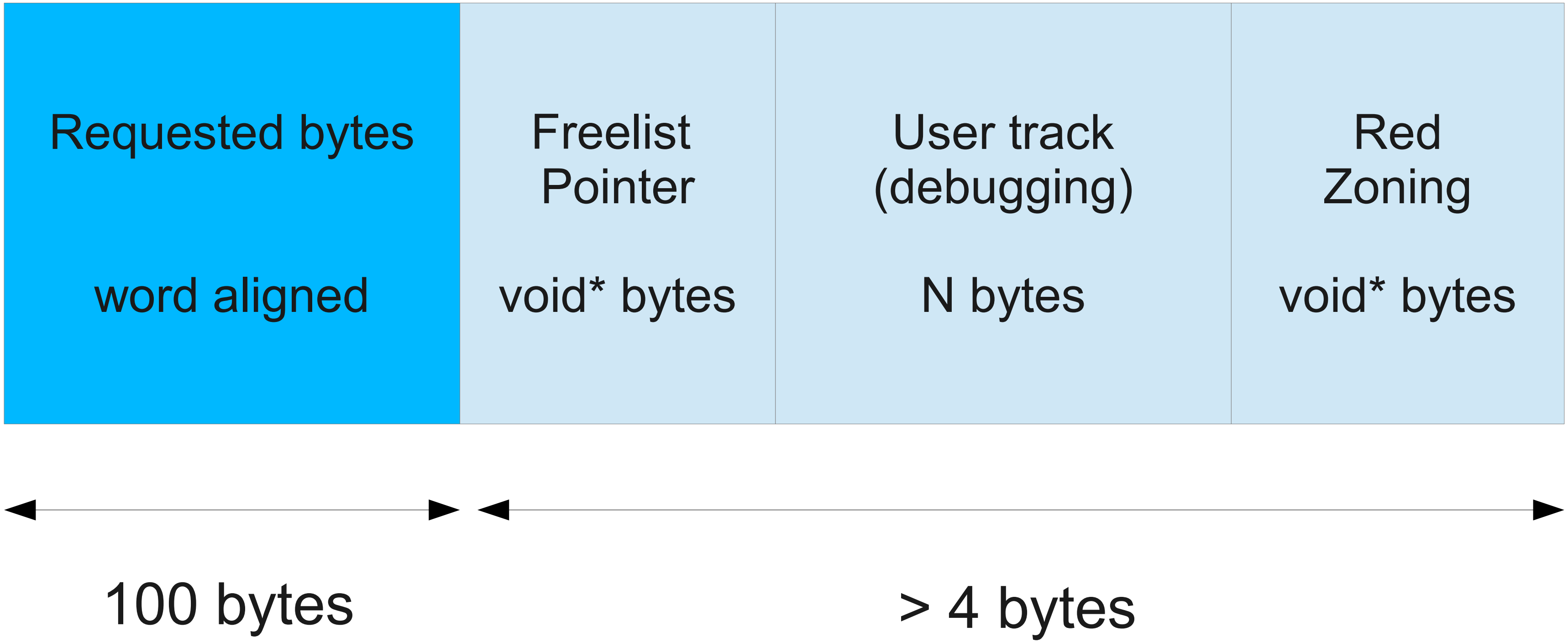
- Obtains a physically discontiguous block
- Unsuitable for DMA on some platforms
- **Rule of thumb:**

chunk < 128 KiB → kmalloc()

chunk > 128 KiB → vmalloc()

Memory wastage: where does it come from?

SLUB object layout wastage



Requested bytes

word aligned

Freelist Pointer

void* bytes

User track (debugging)

N bytes

Red Zoning

void* bytes

100 bytes

> 4 bytes

Memory wastage: where does it come from?

kmalloc() inherent wastage

- kmalloc works on top of fixed sized kmem caches:

8 bytes

16 bytes

32 bytes

Memory wastage: where does it come from?

Big allocations wastage

- kmalloc(6000) → alloc_pages(1) → 8192 bytes
- Pages are provided in sets of 2^N:

  1, 2, 4, 8, …
- kmalloc(9000) → alloc_pages(2) → 16 KiB

Tracking memory

Tracking memory: ftrace
How does it work?

- Ftrace kmem events
- Each event produces an entry in ftrace buffer
  - kmalloc
  - kmalloc_node
  - kfree
  - kmem_cache_alloc
  - kmem_cache_alloc_node
  - kmem_cache_free

Tracking memory: ftrace

- *Advantages*
  - Mainlined, well-known and robust code

- *Disadvantages*
  - Can lose events due to late initialization (`core_initcall`)
  - Can lose events due to event buffer overcommit

Ftrace: enabling

- Compile options

```
CONFIG_FUNCTION_TRACER=y
CONFIG_DYNAMIC_FTRACE=y
```

- How to access

```
/sys/kernel/debug/tracing/...
```

Ftrace: usage
Getting events from boot up

- Kernel parameter

```
trace_event=kmem:kmalloc,
kmem:kmalloc_node,
kmem:kfree
```

- Avoiding event buffer over commit

```
trace_buf_size=1000000
```

Ftrace: usage
Getting events on the run

- Enable events

```
cd /sys/kernel/debug/tracing
echo "kmem:kmalloc" > set_events
echo "kmem:kmalloc_node" >> set_events
echo "kmem:kfree" >> set_events
```

- Start tracing, do something, stop tracing

```
echo "1" > tracing_on;
do_something_interesting;
echo "0" > tracing_on;
```

Ftrace events
What do they look like?

```
# entries-in-buffer/entries-written: 43798/43798   #P:1
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
         linuxrc-1     [000] ....     0.310577: kmalloc: call_site=c00a1198 ptr=de239600
bytes_req=29 bytes_alloc=64 gfp_flags=GFP_KERNEL
         linuxrc-1     [000] ....     0.310577: kmalloc: call_site=c00a122c ptr=de2395c0
bytes_req=24 bytes_alloc=64 gfp_flags=GFP_KERNEL
         linuxrc-1     [000] ....     0.310730: kmalloc: call_site=c00a1198 ptr=de239580
bytes_req=22 bytes_alloc=64 gfp_flags=GFP_KERNEL
         linuxrc-1     [000] ....     0.310730: kmalloc: call_site=c00a122c ptr=de239540
bytes_req=24 bytes_alloc=64 gfp_flags=GFP_KERNEL
         linuxrc-1     [000] ....     0.310883: kmalloc: call_site=c00a1198 ptr=de222940
bytes_req=33 bytes_alloc=64 gfp_flags=GFP_KERNEL
         linuxrc-1     [000] ....     0.310883: kmalloc: call_site=c00a122c ptr=de239500
bytes_req=24 bytes_alloc=64 gfp_flags=GFP_KERNEL
```

Ftrace events
What do they look like?

```
# TASK-PID    CPU#    TIMESTAMP  FUNCTION
#     | |       |          |         |
linuxrc-1    [000]  0.310577: kmalloc: \
# caller address
call_site=c00a1198 \
# obtained pointer
ptr=de239600
# requested and obtained bytes
bytes_req=29 bytes_alloc=64
# allocation flags
gfp_flags=GFP_KERNEL
```

Ftrace events
What do they look like?

```
# TASK-PID   CPU#   TIMESTAMP  FUNCTION

#    | |        |         |         |

linuxrc-1    [000]  0.310577: kmalloc: \

# caller address

call_site=c00a1198 \

# obtained pointer

ptr=de239600

# requested and obtained bytes

bytes_req=29 bytes_alloc=64

# allocation flags

gfp_flags=GFP_KERNEL
```

35 wasted bytes

Obtaining the event call site symbol

- $ cat System.map

```
[...]
c02a1d78 T mtd_point
c02a1e2c T mtd_get_unmapped_area
c02a1eb0 T mtd_write
c02a1f68 T mtd_panic_write
c02a2030 T mtd_get_fact_prot_info
c02a2070 T mtd_read_fact_prot_reg
c02a20cc T mtd_get_user_prot_info
```

Putting it all together
trace_analyze

trace_analyze

- Getting the script

```
$ git clone \
  git://github.com/ezequielgarcia/
  trace_analyze.git
```

- Dependencies

```
$ pip install scipy numpy matplotlib
```

trace_analyze: Usage

- Built kernel parameter: --kernel, -k

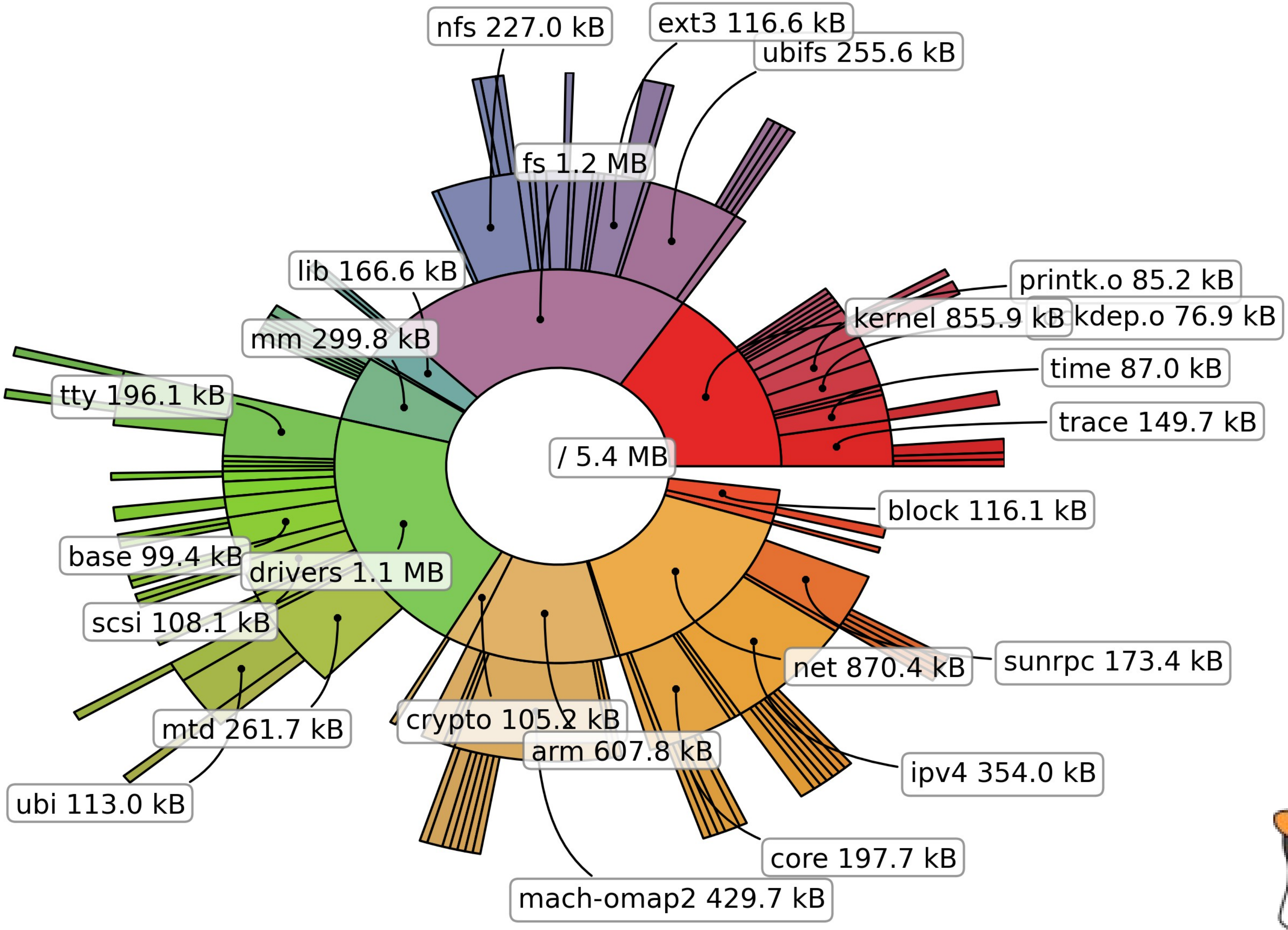
```
$ ./trace_analyze.py \
--kernel=/home/zeta/arm-soc/
```

trace_analyze: Static footprint

```
$ ./trace_analyze.py \
--kernel=/home/zeta/arm-soc/ \
--rings-show

$ ./trace_analyze.py \
--kernel=/home/zeta/arm-soc/ \
--rings-file=static.png
```

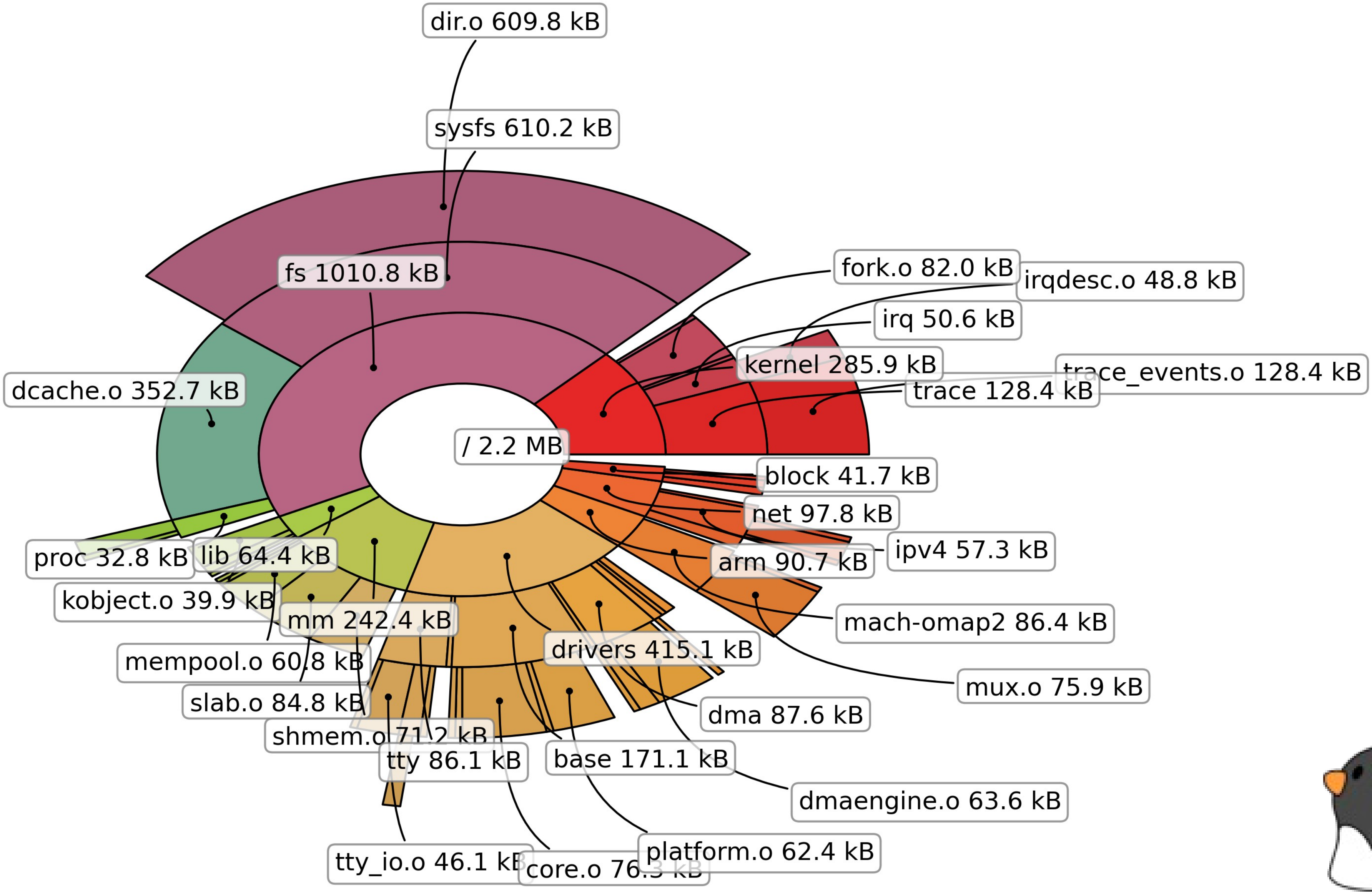
trace_analyze: Static footprint



trace_analyze: Dynamic footprint

```
$ trace_analyze.py \
--file kmem.log \
--rings-file dynamic.png \
--rings-attr current_dynamic
```

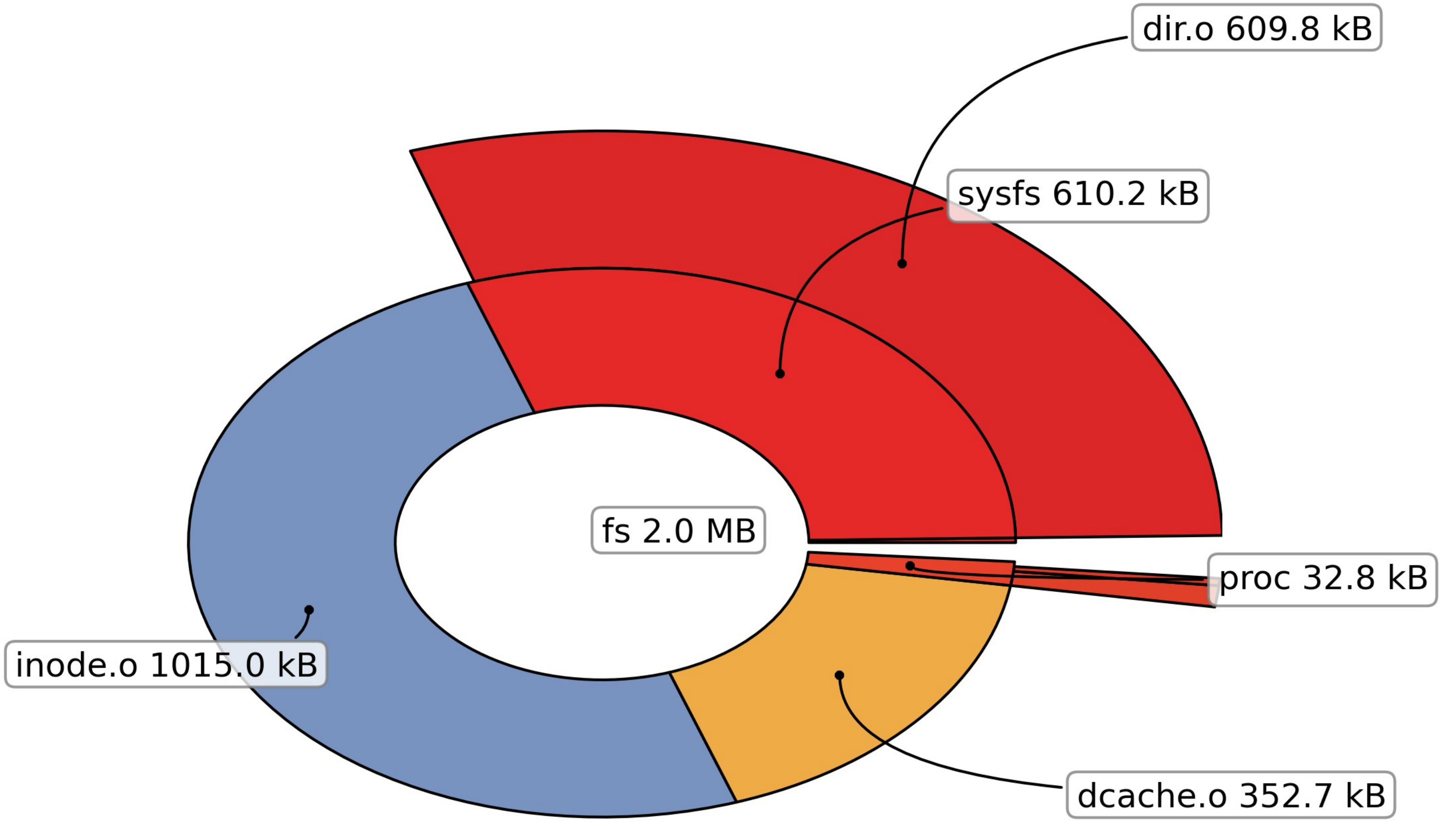
trace_analyze: Dynamic footprint



trace_analyze: Picking a root directory

```
$ trace_analyze.py \
--file kmem.log \
--rings-file fs.png \
--rings-attr current_dynamic \
--start-branch fs
```

trace_analyze: Picking a root directory

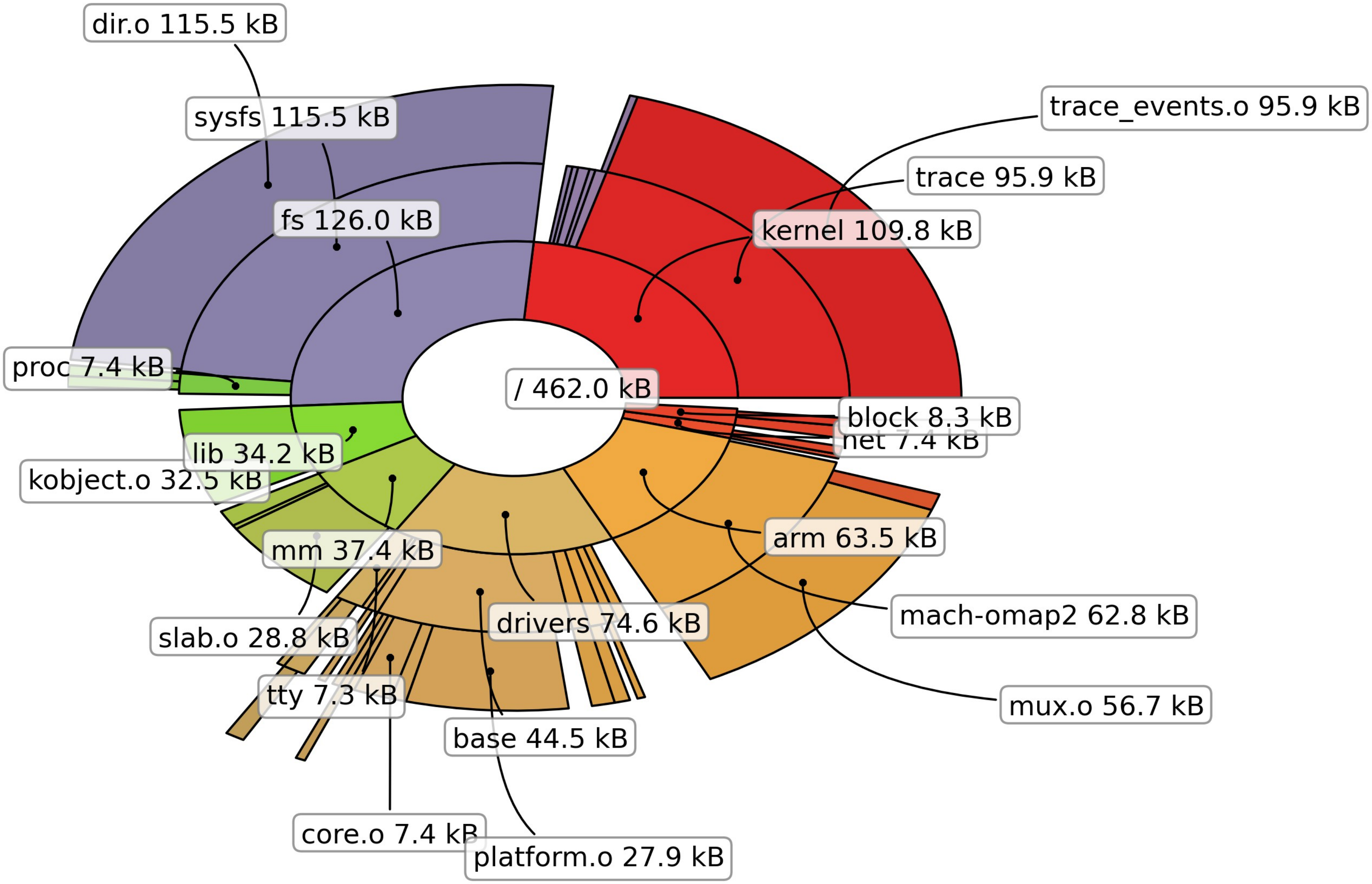


trace_analyze: Waste

- waste == **allocated** minus **requested**

```
$ trace_analyze.py \
--file kmem.log \
--rings-file dynamic.png \
--rings-attr waste
```

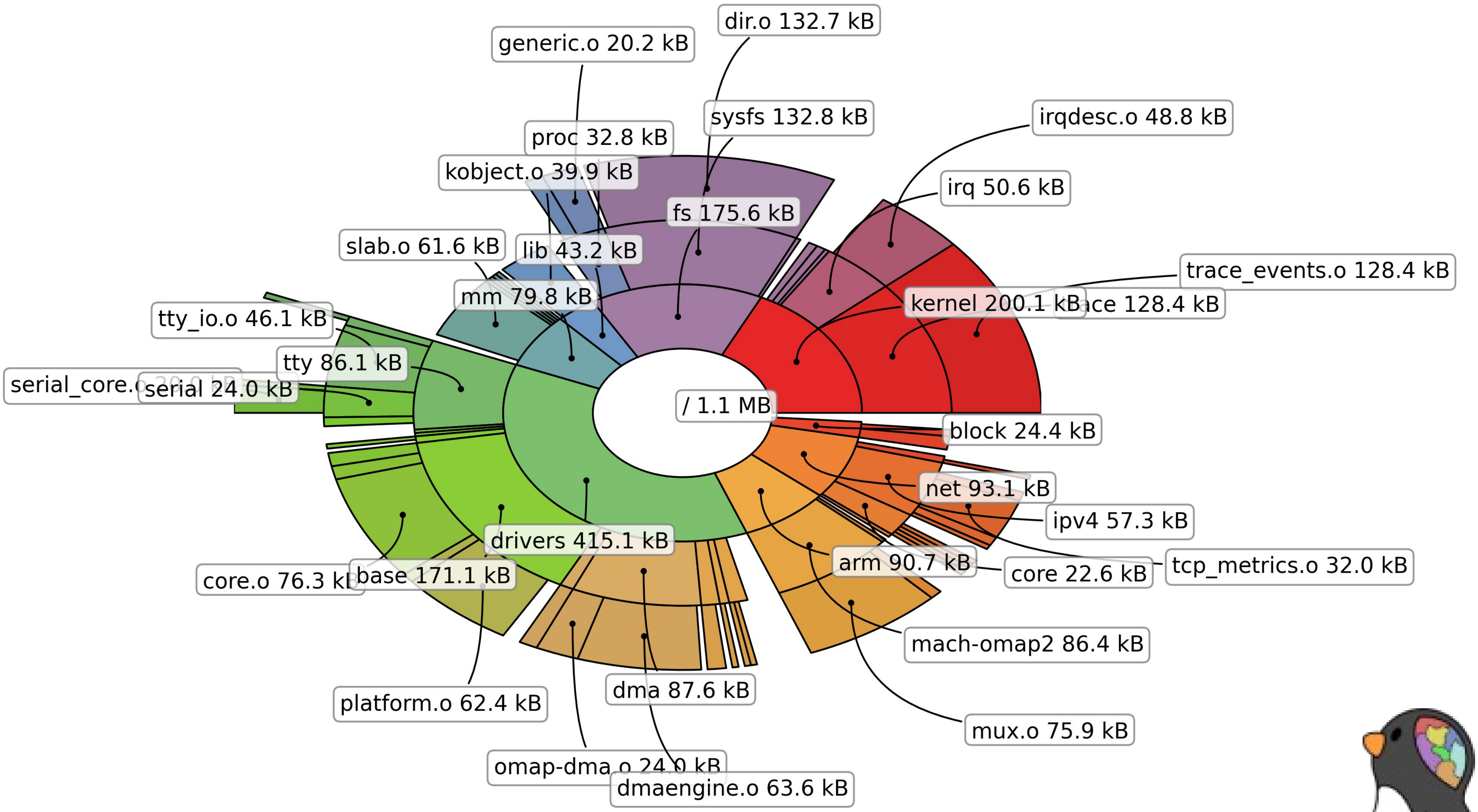
trace_analyze: Waste



trace_analyze: Filtering events

```
$ trace_analyze.py \
--file kmem.log \
--rings-file dynamic.png \
--rings-attr current_dynamic \
--malloc
```

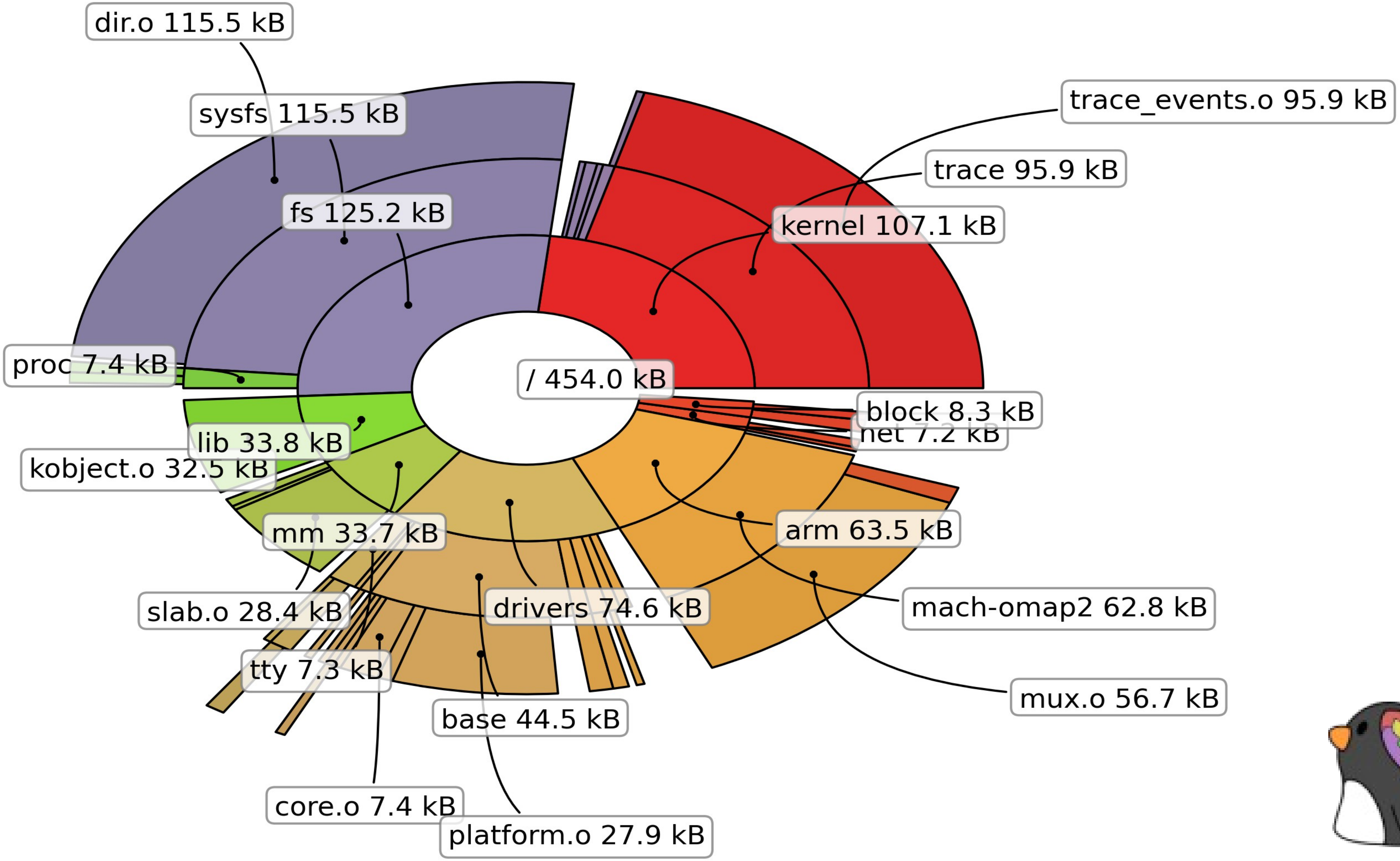
trace_analyze: Filtering events



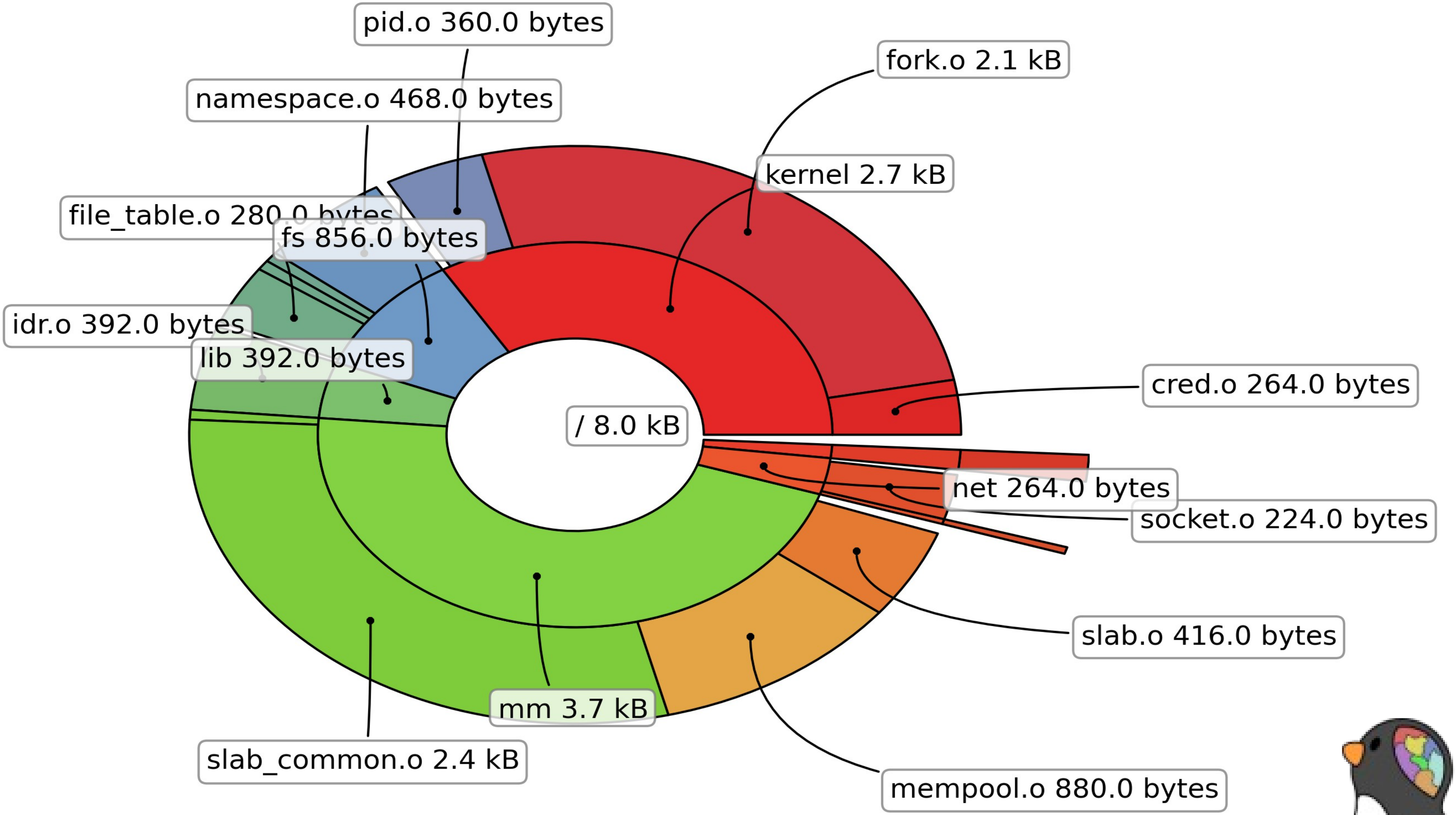
trace_analyze: Use case kmalloc vs. kmem_cache wastage

```
$ trace_analyze.py \
--file kmem.log \
--rings-file dynamic.png \
--rings-attr waste \
--malloc \
...
--cache \
```

trace_analyze: Use case **kmalloc** wastage



trace_analyze: Use case
kmem_cache wastage



trace_analyze: SLAB accounting

- Based in Matt Mackall's patches for SLAB accounting
- An updated patch for v3.6:

  http://elinux.org/File:0001-mm-sl-aou-b-Add-slab-accounting-debugging-feature-v3.6.patch

trace_analyze: SLAB accounting What do they look like?

```
total     waste      net alloc/free caller
---------------------------------------------
507920        0   488560  6349/242  sysfs_new_dirent+0x50
506352        0   361200  3014/864  __d_alloc+0x2c
139520   118223   135872  2180/57   sysfs_new_dirent+0x34
72960         0    72960   120/0    shmem_alloc_inode+0x24
393536      576    12672   559/541  copy_process.part.56+0x694
10240      1840    10240    20/0    __class_register+0x40
8704       2992     8704    68/0    __do_tune_cpucache+0x1c4
8192          0     8192     1/0    inet_init+0x20
```

trace_analyze: SLAB accounting Getting most frequent allocators

```
$ trace_analyze.py \
--file kmem.log
--account-file account.txt
--start-branch drivers
--malloc
--order-by alloc_count
```

trace_analyze: SLAB accounting Getting most frequent allocators

```
total   waste    net alloc/free caller
--------------------------------------------
46848   5856  46848   366/0  device_private_init+0x2c
111136  4176  11136   174/0  scsi_dev_info_list_add_keyed+0x8c
65024      0  65024   127/0  dma_async_device_register+0x1b4
24384      0  24384   127/0  omap_dma_probe+0x128
6272    3528   6272    98/0  kobj_map+0xac
36352   1136  36352    71/0  tty_register_device_attr+0x84
29184    912  29184    57/0  device_create_vargs+0x44
```

trace_analyze: SLAB accounting Getting most frequent allocators

```
total  waste    net alloc/free caller

--------------------------------------------

46848   5856  46848   366/0  device_private_init+0x2c

111136  4176  11136   174/0  scsi_dev_info_list_add_keyed+0x8c

65024      0  65024   127/0  dma_async_device_register+0x1b4

24384      0  24384   127/0  omap_dma_probe+0x128

6272    3528   6272    98/0  kobj_map+0xac

36352   1136  36352    71/0  tty_register_device_attr+0x84

29184    912  29184    57/0  device_create_vargs+0x44
```

These are candidates for kmem_cache_{} usage

trace_analyze: Pitfall GCC function inline

- Automatic GCC inlining can report an allocation on the wrong function

trace_analyze: Pitfall
GCC function inline

- Automatic GCC inlining can report an allocation on the wrong function
- Can be disabled adding GCC options

```
KBUILD_CFLAGS  += -fno-default-inline \
+       -fno-inline \
+       -fno-inline-small-functions \
+       -fno-indirect-inlining \
+       -fno-inline-functions-called-once
```

trace_analyze: Pitfall GCC function inline

- Automatic GCC inlining can report an allocation on the wrong function
- Can be disabled adding GCC options

```
KBUILD_CFLAGS  += -fno-default-inline \
+        -fno-inline \
+        -fno-inline-small-functions \
+        -fno-indirect-inlining \
+        -fno-inline-functions-called-once
```

… but it can break compilation!

trace_analyze: Future?

- Integrate trace_analyze with perf?
  (suggested by Pekka Enberg)
- Extend it to report a page owner?
  (suggested by Minchan Kim)
- Find trace_analyze a better name!

Conclusions

- Care for bloatness:
  - OOM printk

```
dev = kmalloc(sizeof(*dev), GFP_KERNEL);
if (!dev) {
        pr_err("memory alloc failure\n");
}
```

Conclusions

- Care for bloatness:
  - OOM printk

```
$ git grep  "alloc fail" drivers/ | wc -l
  305
```

Conclusions

- Care for bloatness:
  - OOM printk

```
$ git grep  "alloc fail" drivers/ | wc -l
  305
```

- Don't roll your own tracing, use **ftrace**!
  - **Powerful**
  - **Flexible**
  - See **pytimechart**

Questions?