OSS Europe 2018

Fine-grained Distributed Application Monitoring Using LTTng

EfficiOS

jeremie.galarneau@efficios.com

jgalar

Presenter

Jérémie Galarneau

EfficiOS Inc.

- Vice President
- http://www.efficios.com

Maintainer of

- LTTng-tools
- Babeltrace

Outline

A few words about tracing and LTTng

The (long) road to session rotation mode

How session rotation mode makes distributed trace analyses possible (and how to implement them)

Isn’t tracing just another name for logging?

Tracers are not *completely* unlike loggers

Both save information used to understand the state of an application

- Trying to cater to developers, admins, and end-users

In both cases, a careful balance must be achieved between verbosity, performance impact, and usefulness

- Logging levels
- Enabling only certain events

Different goals, different tradeoffs

Tracers tend to **focus on low-level events**

- Capture syscalls, scheduling, filesystem events, etc.
- More events to capture means we must lower the space and run-time cost per event
- Binary format
- Different tracers use different strategies to minimize cost

Different goals, different tradeoffs

Traces are harder to work with than text log files

- File size
  - Idle 4-core laptop: 54k events/sec @ 2.2 MB/sec
  - Busy 8-core server: 2.7M events/sec @ 95 MB/sec
- Exploration can be difficult
- Must know the application or kernel to make sense of what was captured
- Purpose-built tools are needed

LTTng: Linux Trace Toolkit Next Generation

Open source tracing framework for Linux first released in 2005

Regroups a number of projects

- LTTng-UST `// userspace tracer`
- LTTng-modules `// kernel tracer`
- LTTng-tools `// system daemons`
- LTTng-analyses `// analysis scripts`
- LTTng Scope `// graphical viewer`

LTTng – What it does differently

Unify many information sources

- kernel, C/C++, python `logging`, java (jul and log4j)

Fast

- Kernel: same as ftrace, with syscall payloads
- Userspace: ~130ns / event (32-bit payload, Xeon E5-2630)

Standard trace format (Common Trace Format)

- Vast ecosystem of analysis/post-processing tools

Many ways to deploy LTTng

Allow multiple trace extraction methods

- Locally to disk
- Stream through the network to a trace server (`lttng-relayd`)
- Snapshot
- Live

Local tracing and trace streaming

Great when working on an easily reproducible problem

Used by developers to gather very detailed logs

- Breakdown of where time is spent
- Understand how the kernel behaves under a certain load

These traces are then used with existing tools

Local tracing and trace streaming

Not suited for continuous monitoring

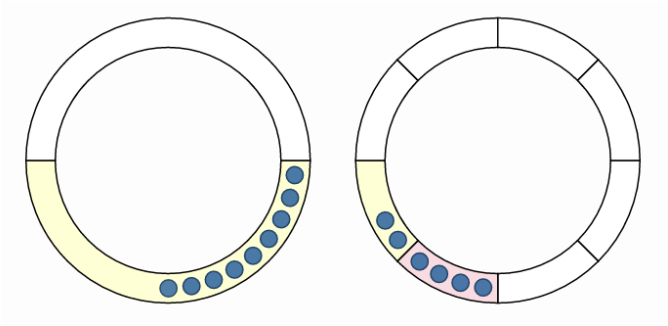
Some users have used these modes to collect “samples”

- Choose a machine
- Trace for a moment
- Run scripts on the trace

Snapshot mode

Also known as “flight-recorder” mode

- Trace to in-memory buffers



Introduced to make it possible to leave tracing active on production machines

- Load a session profile on start-up
- Leave it running
- Capture the buffers when a condition occurs

Snapshot mode

A great improvement, but with some drawbacks

Must react quickly enough when an error occurs

- Not easy to do when the problem is detected on another machine

Capturing a short trace can make it hard to reason about what was happening

- Both for humans and existing tools

Live mode

Introduced at the same time as the snapshot mode

CTF is not a format meant to be consumed while it is being produced

- Self-described layout requires careful synchronization between tracers and viewers
- Ensure that all data, from all domains, produced by all CPUs, is available up to a given point to merge traces

Makes it possible to consume correlated traces (multiple sources) from a TCP socket

Live mode

Continuous monitoring was not the primary use-case

Meant to provide an experience closer to `strace`

- Time-correlated kernel and user-space traces
- Lower performance impact on traced applications

The protocol is not trivial to implement

- Only supported by `babeltrace` (text viewer)

Live mode

Limitations making it hard to deploy for continuous monitoring

- Only one client may consume the trace at a time
- No way to consume traces in the past without stopping tracing
- Protocol is not designed to handle a high throughput

Back to the drawing board!

Gathered feedback from users

- There is no “typical” deployment, everyone uses different infrastructure components
- Trace processing is slow
- Traces are huge

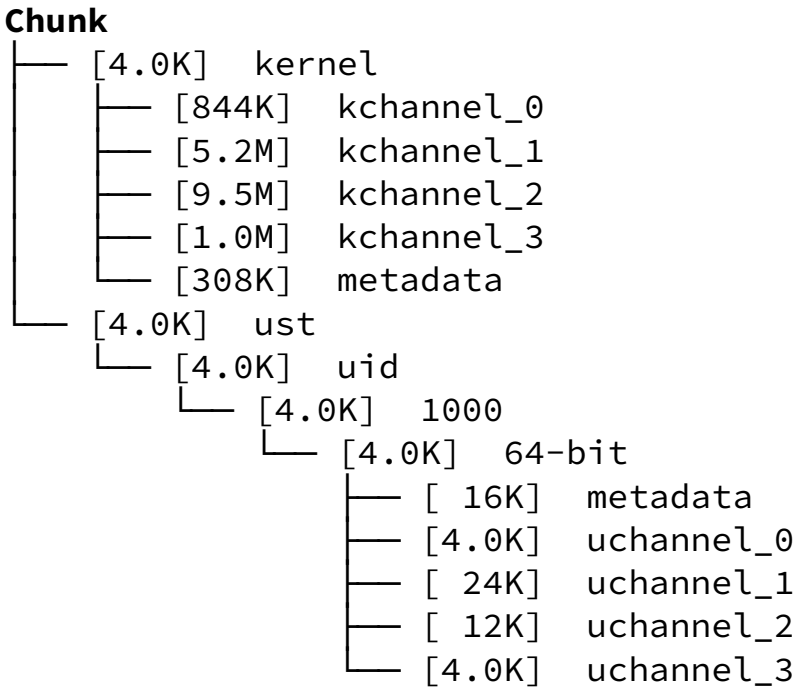
Handle traces like we handle logs

- *Just give us some plain old files and we’ll manage them*

Session rotation

Build and provide independent trace “chunks”

- Can process those traces **when** and **where** we want
- Tracing can keep running, without disturbing the target
- Compatible with existing viewers



Session rotation

Listen for notifications that a trace archive is ready

- Run a script in-place on the target
- Keep a time/size-based backlog of traces
- Compress or encrypt traces
- `rsync`
- Notify workers over a MQ (Kafka, ZeroMQ, Rabbit MQ, etc.)

Using session rotation

Available from LTTng 2.11+

- Currently in *release candidate*, try it out!

Immediate rotation

```
$ lttng rotate --session my_session
```

Scheduled rotation

```
$ lttng enable-rotation --session my_session --timer 30s
$ lttng enable-rotation --session my_session --size 500M
```

What are our users looking for?

Stateless analysis

- Count occurrences of events
- Breakdown errors by categories
- Statistical analysis of event payloads

```
babeltrace my_chunk | grep "MyApp::my_error" | wc -l
```

What are our users looking for?

Stateful analysis

- Being able to query a model when a user space event is read
  - Which file is ‘`fd = 42`’?
  - What was my application doing when **`ENOSPC`** occurred?
  - What was happening on the rest of the system

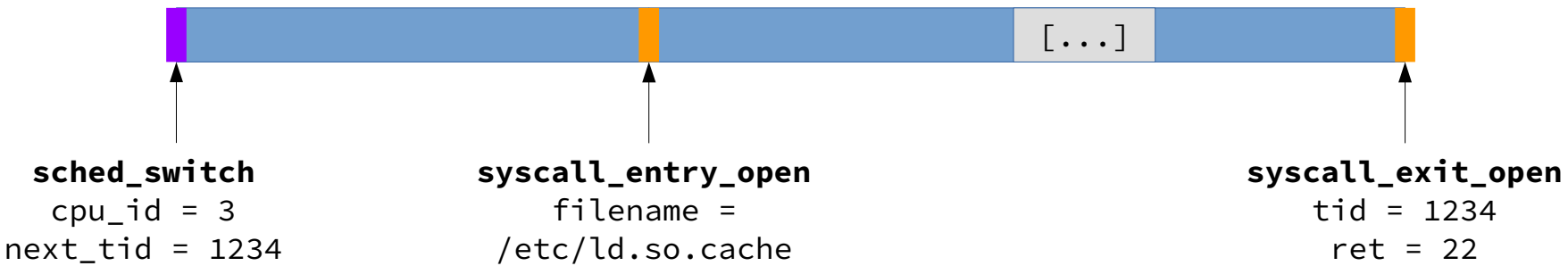
- Present data in a familiar way

```
[03:51:02.799664908] syscall_entry_read: { cpu_id = 1 }, { fd = 3, count = 64 }
[03:51:02.799664908] [thumbnailer] read(“assets/big_buck_bunny.avi”, count = 64);
```

How current analysis tools work

Track the state changes of resources

- Operate by feeding an internal model of the kernel
- Populate a state history database
- Rely on all information being available at all times



How current analysis tools work

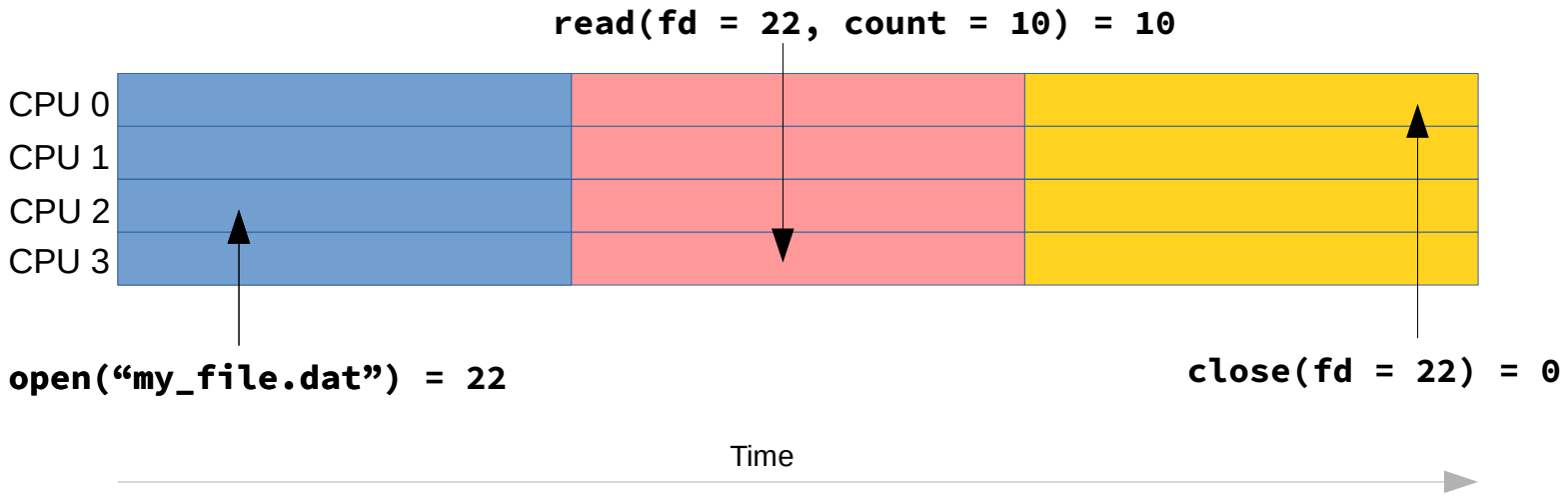
Query the kernel model to augment user space traces

- Application reports an error reading "`fd 42`"
  - Map the file descriptor to a file name
- A `read()` takes a long time to complete
  - What was the I/O activity on that device during this time?
- A request took a long time to complete
  - Was my task preempted during that time?

Challenges

Processing trace “chunks” independently

- Can’t rely on a *complete* model of the kernel being available
- We still have all the information, it’s just split into a lot of tiny traces



Challenges

This is where the current analyses need the most adjustments

Intuitively, analyses must happen in two phases

- Read the trace and identify spans
  - Keep partial spans *aside*
- Merge results to *stitch* partial spans

This is starting to sound *a lot* like a MapReduce pipeline

Challenges

This is where the current analyses need the most adjustments

Intuitively, analyses must happen in two phases

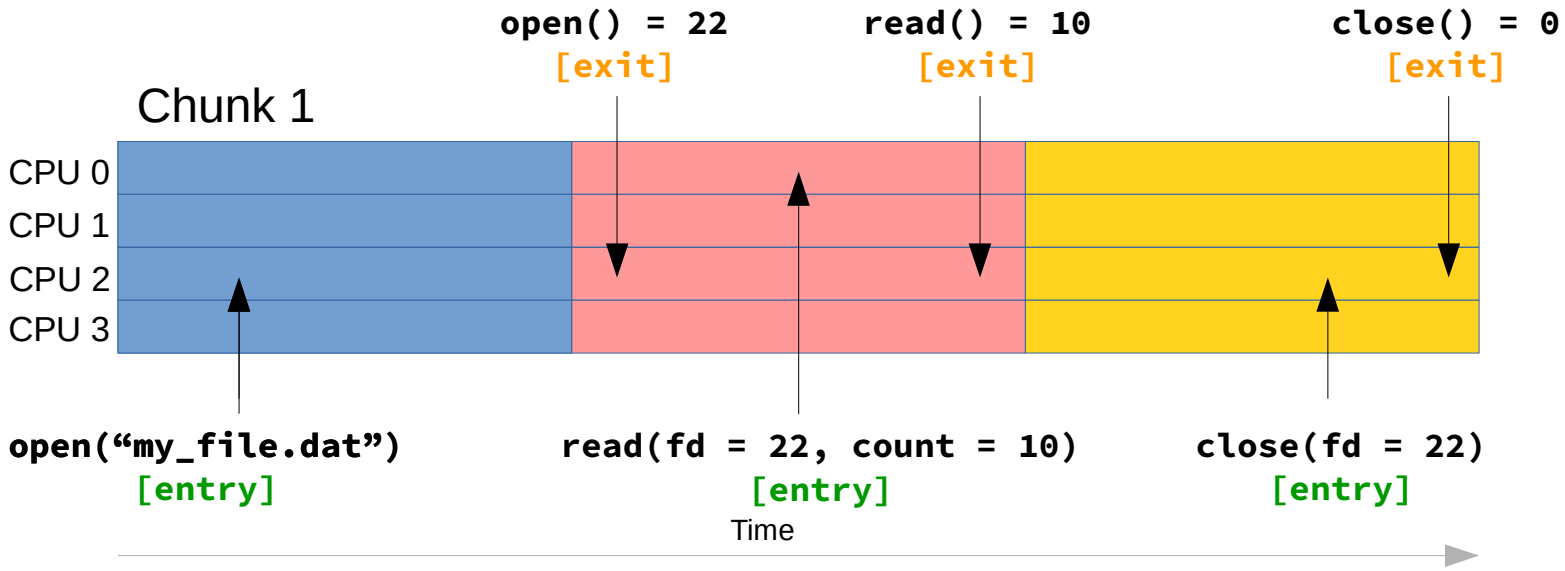
- Read the trace and identify spans `// map()`
  - Keep partial spans *aside*
- Merge results to *stitch* partial spans `// reduce()`

This is starting to sound *a lot* like a MapReduce pipeline

Challenges

In fact, it's a *bit* more complicated than that

- Syscalls are not atomic operations; entry and exit may occur in different chunks
- Syscalls may fail



Challenges

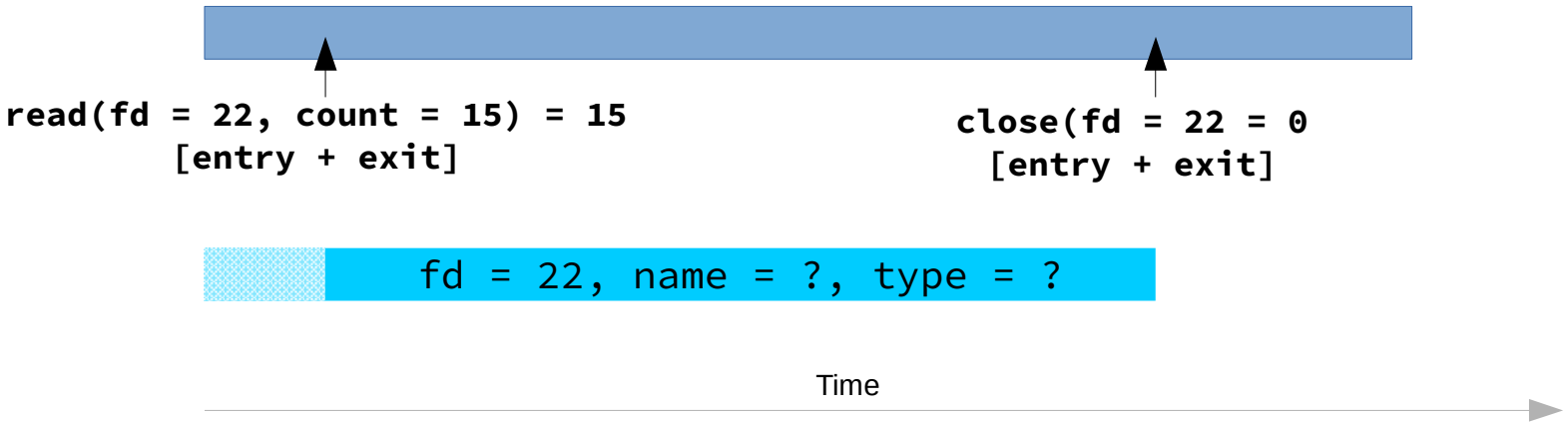
Modelling object lifetime as nested spans makes intuitive sense

- A process lives for a period of time bounded by the kernel
  - Threads live for a period of time bounded by their process / thread group
  - A file descriptor lives for a period of time bounded by its process
  - *Let's not get into `clone()` flags for now...*
- Syscalls can be modelled the same way for **`entry`** and **`exit`**
  - Nests within a Thread span
- The same applies to model the scheduler and track currently running tasks
  - A CPU also has a lifetime (hotplug)
- It's turtles all the way down

Challenges

Mapping

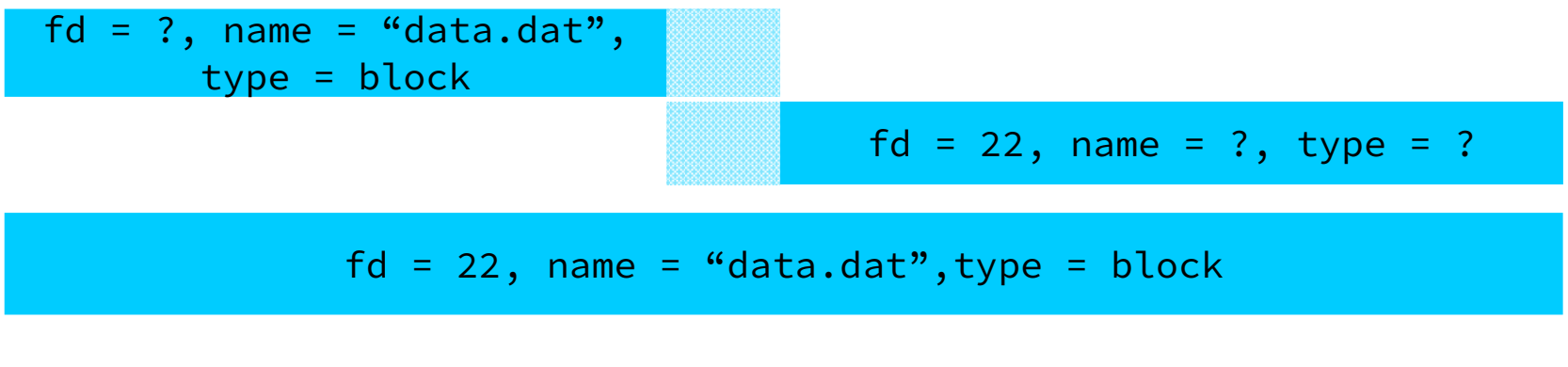
- Identify all spans in a trace
- Attempt to evaluate analysis queries against the modelThe creation of a file may not be visible, but we may see it being used
- We know fd 22 exists in that process (*soft* begin bound)



Challenges

Reducing amounts to stitching spans

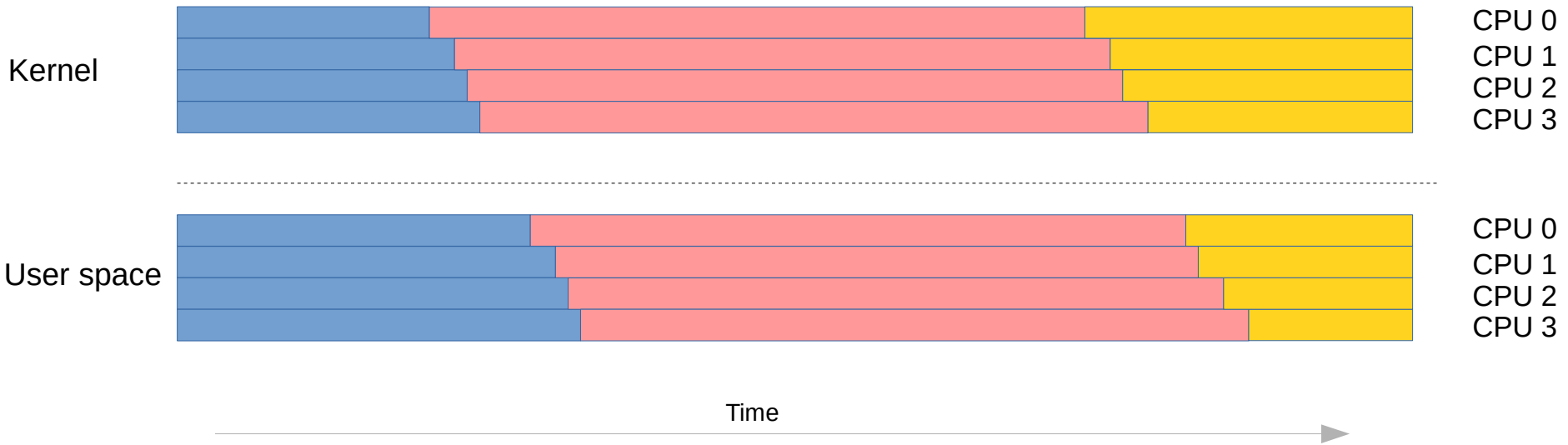
- *Soft* bounds become *hard* bounds
- Attributes are resolved
- Queries for attributes that were unknown are re-evaluated



Challenges

In fact, it's a *bit* more complicated than that, still...

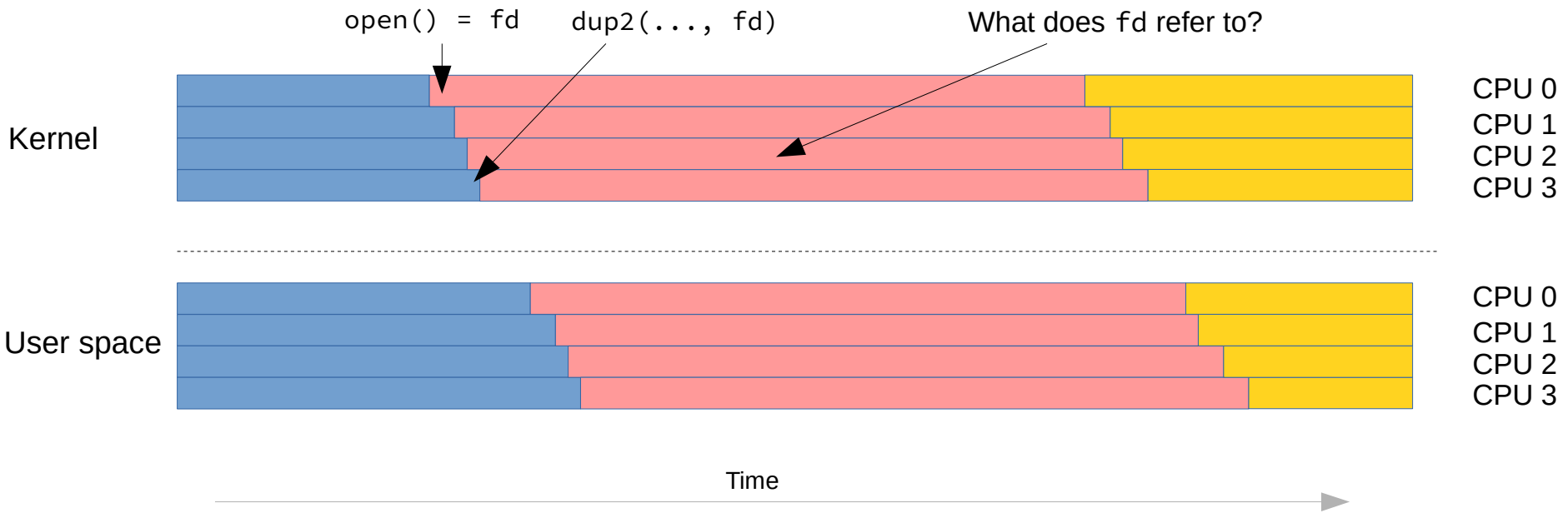
- Rotations are not atomic across CPUs and domains
- ~28 ms to perform the switch across domains (12 threads)



Challenges

Processing “raw” chunks adds more uncertainty to the model

- It becomes hard to reason about any span that can start and end on different CPUs



Challenges

There are a number of solutions to this problem

- The simplest is to use two chunks to “simulate” a clean bound
- Simple to perform since the traces all use the same clock source

Demo

- Started working on a trace analysis pipeline *proof of concept* to prepare this talk
  - Goal is to provide the same kind of data as LTTng-analyses
    - Breakdown of time spent in each processing phase of a user space application
    - Show time spent in syscalls (with added information)
  - Address shortcomings of LTTng-analyses
    - Slow
    - Not distributable (requires a complete trace)

Demo

A server application receives requests to generate a thumbnail from a video at a given time

- Open video file
- Seek in video file
- Decode video
- Encode thumbnail
- Send

Generate a breakdown of time spent in syscalls, per processing phase, and show it in a Grafana dashboard.

Conclusion

The tracing infrastructure needed to easily distribute trace processing is now available in LTTng

Creating distributed analyses is challenging, but this POC shows that it is viable

Open questions

- What form should this POC take in the long term?
- How could it integrate with existing monitoring tools?
- How hard will it be to extend to multiple hosts?
  - e.g. critical path analysis across hosts, determining network latency, etc.
  - I assume the same model works, but does it really hold up?

Questions ?

- lttng.org
- lttng-dev@lists.lttng.org
- @lttng_project
- `#lttng OFTC`