Linux “Time Travel” mode and network simulation

Johannes Berg
johannes.berg@intel.com
johannes@sipsolutions.net

netdev 0x14

Introduction

- “Time Travel”
  - Term coined in mailing list discussions
  - Skip time forward when system is idle
  - Cannot go backwards!
- User Mode Linux (UML)
  - A kind of virtual machine
  - Port of the Linux kernel to its own userspace
  - Userspace inside running as ptrace’d processes

Motivation

Testing!

- Speed
  - Delays/timeouts collapse due to time forwarding
- Ability
  - E.g. when physical device doesn’t exist yet
  - Device simulation might be slower than real time
  - Network topologies, …
- Debug checks
  - Without affecting timing
  - E.g. kernel debug options (slub debugging, object debugging, …)
- Manual debugging
  - Time stops when in gdb

Implementation

“Time Travel” modes

- **`time-travel`**
  skip time forward if possible, but never slower than real time
  (will not cover the details here)

- **`time-travel=inf-cpu`**
  Skip time forward, and simulate infinite CPU speed, i.e. time doesn’t move until the system is idle or in a delay. Note that there’s no preemption in any way here and even user space infinite loops will hang the system/simulation.

- **`time-travel=ext:/path/to/controller-socket`**
  Like `=inf-cpu` but integrate with multiple UML instances

Implementation - underlying mechanisms

Four central points (that we need to modify), everything else (timing related) derives from this:

- **“What time is it?”**
  Clock source (`struct clocksource`)
- **“Please wake me in …”**
  Clock event source (`struct clock_event_device`)
- **“Wait just a little *without scheduling*.”**
  Delays (`ndelay`, `mdelay`, `cpu_relax`)
- **“There’s nothing to do.”**
  Idle loop (`arch_cpu_idle`)

“What time is it?”

- In UML, normally just asks the host OS using `clock_gettime(CLOCK_MONOTONIC, …)`
- In time-travel mode, just read internal “current time” (`time_travel_time`)
- Caveat: sometimes user space has loops so make this cost a little bit of time (otherwise get infinite timeout loops e.g. in python socket servers)

“Please wake me in …”

- In UML, normally just arm a host OS timer with `timer_settime(...)`
- In time-travel mode, just remember when the next wakeup should happen.

“Wait just a little *without scheduling*.”

- In UML, there’s normally no special implementation. Just delay per the normal loops per jiffy, or do a “nop” for cpu_relax().

- In time-travel mode this must “take time”, so move time forward by an appropriate number of nanoseconds.

“There’s nothing to do.”

- In UML, normally just sleep for a second - will be interrupted by timer
- In time-travel mode, “sleep” for up to a second, i.e. move clock forward to the next wakeup time and trigger the timer interrupt

Implementation - so far

- Can speed up delays in tests in a single virtual machine now

- Already useful: e.g. wpa_supplicant tests (this is upstream)
  - **>6x speedup**
    (for example, DFS channel tests that require 120s CAC no longer take nearly that long)
  - Kernel debug options used to be problematic, causing due to userspace timeouts, not now
  - Disconnected from real time, so can oversubscribe CPUs without simulation noticing in form of timeouts

But we always want more!

Multiple Machines

Multiple Machines - Modifications

- Cooperative scheduling between the different instances
- Simple protocol (`include/uapi/linux/um_timetravel.h`)
  - REQUEST runtime
  - WAIT for my turn
  - GET current time
  - UPDATE current time
  - RUN now
  - FREE_UNTIL (for optimisation)
- Delay/Idle changes to not just move time/skip to the next event, but
  - REQUEST from controller
  - WAIT until it’s my turn
  - RUN when told
  - repeat

Multiple Machines - Controller application

- Contains the overall “calendar” that keeps track of each participant’s next event
- Notifies which one is allowed to run
- Distributes time updates

Working to release this as open source, including a framework for device simulation.

Devices

Devices

Conceptually simple? Need to communicate with

- the time controller (just like a virtual machine), and
- the device driver.

Devices - virtio/vhost-user

We already have:

- VirtIO
  - Standard model
  - Existing infrastructure and drivers
- vhost-user
  - pulls device implementation out of the hypervisor

Implemented vhost-user support in UML. Done?

Devices - virtio/vhost-user

Let’s transmit a network frame:

Normal vhost-user model

- **Host** puts frame on the virtqueue
- **Host** notifies **device** using *eventfd*
- **Device** handles frame

Simulation model

- **Host** puts frame on the virtqueue
- **Host** notifies **device** using *in-band signal*
- **Device** asks **Controller** for time to run**
- **Device** sends ACK back to **host**
- **Host** continues running until idle/delay
- **Host** returns to **Controller**
- **Controller** tells **device** to run
- **Device** handles frame

All handled by `arch/um/drivers/virtio_uml.c` and device-side vhost-user library code.

**: this may cause more messages, including communication with the Host and Device

Devices - Wireless

Within a single machine, **mac80211_hwsim** and **wmediumd** can simulate wireless networks. Extend:

- Transport netlink protocol over virtio
- Teach wmediumd to be a vhost-user device implementation that has a device for every socket connection

Demo

Summary

Summary

- Time-travel mode can disconnect simulated time from real time
  - CPU bound - faster or slower than real time depending on simulation complexity
- Already used for testing in hostapd/wpa_supplicant
- Multiple machines & devices can be in a common simulation using the `um_timetravel.h` protocol and the controller application
- VirtIO devices are supported with vhost-user, using the “in-band signalling” and “reply-ack” protocol extensions
- Already used for testing wireless with real firmware & driver at Intel