Optimizing suspend/resume

Saravana Kannan (Google)

LINUX
PLUMBERS
CONFERENCE Vienna, Austria / Sept. 18-20, 2024

Key phases of suspend/resume

Suspend:

- suspend_enter
  - sync_filesystems
  - freeze_processes
- dpm_prepare
- dpm_suspend
- dpm_suspend_late
- dpm_suspend_noirq
- Power off CPUs

Resume:

- Power on CPUs
- dpm_resume_noirq
- dpm_resume_early
- dpm_resume
- dpm_complete
- thaw_processes

Optimizing suspend_enter

sync_filesystems() - flushes all dirty pages to “disk”

- Wasteful on systems with frequent suspend/resume.
  - On average 30ms to 66ms on phones and watches.
  - Can be about 10-35% of the duration to suspend.
- Userspace can’t abort suspend during a long sync_filesystem()
  - Even 10% dirty pages on a 8GB device is 800 MB.
  - Most of these writes will be random I/O – so about 30 MB/s on flash.
  - Can take more than 25 seconds!

Questions:

- Can we add a way to abort a suspend request from userspace?
- Can we extend /sys/power/sync_on_suspend to mitigate data loss risk?
  - Sync for every Nth suspend?
  - Sync if X minutes have passed since last sync on suspend?

Optimizing suspend_enter

freeze_processes()

What's going on with the 2nd half? Is this normal?

It can't be all just setting the flags for all the kernel threads, can it?

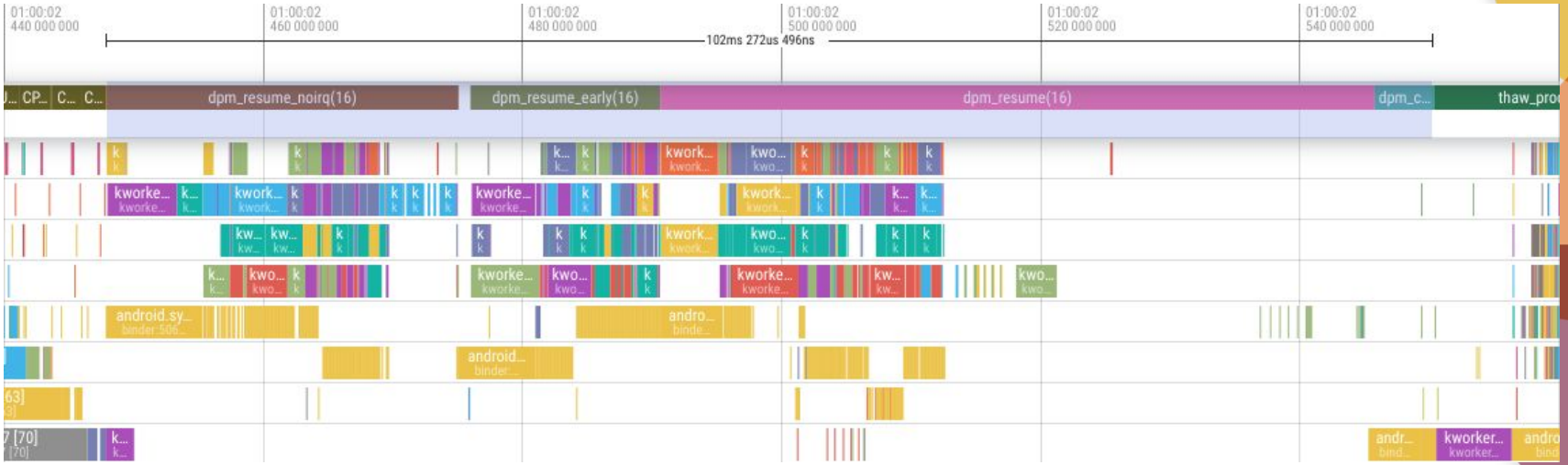
Optimizing dpm_resume*() stages

What does resume look like without any optimization?

It takes 82ms but the CPUs are mostly idle.

Optimizing dpm_resume*() stages

What does resume look like if we enable async suspend/resume for all devices?



It actually works and is stable because fw_devlink is tracking all the dependencies.

The CPUs are very busy, but it takes 102ms. That's 20ms worse!

Optimizing dpm_suspend*() stages

What does suspend look like without any optimization?

It takes 126ms but the CPUs are mostly idle.

Optimizing dpm_suspend*() stages

What does suspend look like if we enable async suspend/resume for all devices?

It takes 94ms. That's 32ms better, but there might be room to improve.

Ironically, async suspends are triggered in a less async manner than async resumes.

Why is async hurting/not helping much?

Too much overhead

- Lots of devices have no ops or quick (microseconds) ops.
- The overhead is more than the actual work:
  - Work queuing
  - kworker wakeups
  - context switches
  - wait_for_completion()s (on consumers, children, suppliers and parent)

Still synchronous:

- dpm_prepare() and dpm_complete() are fully serialized even for devices with async flag set.
- Is there a reason for this? Can we async this too?

Hard cut off between suspend/resume stages:

- All devices should finish a stage before the kernel moves on to the next one.
- Can we kick off the next stage for a device if all its dependencies have finished the current stage?
  - Except noirq of course.

Avoiding async overhead (Proposal 1): Auto async slow devices

1. Have userspace set a “sync threshold”.
2. Kernel tracks worst case time to suspend/resume each device.
3. Any time a device’s worst case time exceeds sync threshold:
    a. Set the async flag for it.
    b. Set the async flag for all it’s consumers, consumers of consumers, etc.
    c. Set the async flag for all it’s suppliers, suppliers of suppliers, etc.

3b & 3c are needed to avoid an async device waiting on a sync consumer/supplier that is deep in the sync devices list.

Concerns:

- Doing all this ends up with many “quick” devices using async and ends up adding too much overhead.
- Still doesn’t parallelize as much as possible.

Avoiding async overhead (Proposal 2): Breadth-first suspend/resume

Suspend

1. Suspend leaf node devices in parallel.
2. This creates more leaf nodes in the graph.
3. Goto 1 until all nodes are suspended.

Resume is just the reverse.

Avoids the overhead:

- No kworker wakeups or context switch overhead.
- Removes repeated wait_for_completion() by picking up only ready to go devices.
- Might need refcounting of consumer/children and parent/suppliers to avoid looping through lists.

Am I missing something?

Any known issues which approach?

Still need to somehow make this work for systems without proper dependency tracking.

Optimizing device suspend/resumes

For a mostly idle system with periodic and short wakeups:

- Total work done to suspend/resume is 3-5% of the total work.
- That is approximately 43 extra minutes of battery life per day.

Resume latency also affects time to first frame rendered:

- Why resume the display if the screen isn't going to get turned on?
- Why serialize resuming storage and rendering a frame on screen?
- Why resume a device if it's not needed right now?

fw_devlink=rpm (default) enforces runtime PM dependency, improving its stability.

Questions:

AFAIK, lazy resume with runtime PM is already supported.

Why is it not more prevalent?

How can we make it easier for driver developers?

S2Idle woes

S2Idle works and is so much faster than S2RAM.

_IF_ the firmware supports it and isn't buggy.

S2Idle vs S2RAM gap increases with ever increasing CPU counts.

Getting firmware updates for existing devices is next to impossible.

Can we get S2idle like behavior by using S2RAM firmware calls?

- Add a fake C-state that just calls S2RAM hotplug API
- Power up CPU using S2RAM firmware calls before sending IPI to wake up the CPUs from the fake C-state

Can we get something like this working?