

Re: [linux-pm] Re: Hibernation considerations

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-07/msg08208.html>

- *From:* "Rafael J. Wysocki" <rjw@xxxxxxx>
 - *Date:* Thu, 19 Jul 2007 22:28:04 +0200
-

On Thursday, 19 July 2007 17:46, Milton Miller wrote:

Hi. I've found this thread from the kjump thread on the kexec mailing list. I'll respond to that one later, but I wanted to respond to several messages in this thread. [Actually, there is a brief outline of a response near the bottom of this note]. I downloaded the archive to get message-ids and references, hopefully I don't break the threading badly.

First, my background is hardware and low level software. I wrote the initial ppc64 support for kexec. I try not to learn too much about x86 details, but have read this thread and the tutorial article mentioned in it. I've studied the suspend code at various times in the past but not in the last year. Several years I mused that kexec could be used restore the suspend image, but I was told that would not be swsuspend.

Next: lets define what we are trying to solve with the kexec approach.

We can solve the case of getting the drivers to quiesce the hardware with requests from userspace suspended in queues. This is what the powermac suspend has been doing for years, and I think its agreed that it will be something similar when we remove the freezer. In the powermac code, there are two notifications to drivers; in the first stage they can allocate memory and interrupts are enabled, in the second its copy enough device state to memory to restart the device.

The problem with all the suspend methods is how do we select what needs to be re-enabled to write the image to stable storage (be it disk, network, or other medium).

The kjump kexec proposal says after we have the io quiesced, we can jump to a totally new kernel, let it initialize the io it needs, and write the image. This has the advantage that there is no confusion as far as which requests should be service or not serviced by the driver and subsystem stacks. It also reuses all the drivers, which means we don't get untested code paths. It also has the advantages that we can use any complicated user stack to access a file system and run any

Re: [linux-pm] Re: Hibernation considerations

desired access methods (eg encryption, raid, etc).

The currently identified problems under discussion include:

- (1) how to interact with acpi to enter into S4.
- (2) how to identify which memory needs to be saved
- (3) how to communicate where to save the memory
- (4) what state should devices be in when switching kernels
- (5) the complicated setup required with the current patch
- (6) what code restores the image

(7) how to avoid corrupting filesystems mounted by the hibernated kernel

I'll now start with quotes from several articles in this thread and my responses.

Message-ID: <200707172217.01890.rjw@xxxxxxx>

On Tue Jul 17 13:10:00 2007, Rafael J. Wysocki wrote:

(1) Upon entering the sleep state, which IMO can be done after the image

has been saved:

- * figure out which devices can wake up
- * put devices into low power states (wake-up devices are placed in the Dx states compatible with the wake capability, the others are powered off)
- * execute the _PTS global control method
- * switch off the nonlocal CPUs (eg. nonboot CPUs on x86)
- * execute the _GTS global control method
- * set the GPE enable registers corresponding to the wake-up devices)
- * make the platform enter S4 (there's a well defined procedure for that)

I think that this should be done by the image-saving kernel.

Message-ID: <87odiag45q.fsf@xxxxxxxxxxxx>

On Tue Jul 17 13:35:52 2007, Jeremy Maitin-Shepard expressed his agreement with this block but also confusion on the other blocks.

I strongly disagree.

(1) as has been pointed out, this requires the new kernel to understand all io devices in the first kernel.

(2) it requires both kernels to talk to ACPI. This is doomed to failure. How can the second kernel initialize ACPI? The platform thinks it has already been initialized. Do we plan to always undo all

acpi initialization?

Good question. I don't know.

(2) Upon start-up (by which I mean what happens after the user has pressed the power button or something like that):

- * check if the image is present (and valid) `_without_` enabling ACPI (we don't do that now, but I see no reason for not doing it in the new framework)
- * if the image is present (and valid), load it
- * turn on ACPI (unless already turned on by the BIOS, that is)
- * execute the `_BFS` global control method
- * execute the `_WAK` global control method
- * continue

Here, the first two things should be done by the image-loading kernel, but the remaining operations have to be carried out by the restored kernel.

Here I agree.

Here is my proposal. Instead of trying to both write the image and suspend, I think this all becomes much simpler if we limit the scope the work of the second kernel. Its purpose is to write the image. After that its done. The platform can be powered off if we are going to S5. However, to support suspend to ram and suspend to disk, we return to the first kernel.

We can't do this unless we have frozen tasks (this way, or another) before carrying out the entire operation. In that case, however, the `kexec`-based approach would have only one advantage over the current one. Namely, it would allow us to create bigger images.

This means that the first kernel will need to know why it got resumed. Was the system powered off, and this is the resume from the user? Or was it restarted because the image has been saved, and its now time to actually suspend until woken up? If you look at it, this is the same interface we have with the magic `arch_suspend` hook — did we just suspend and its time to write the image, or did we just resume and its time to wake everything up.

I think this can be easily solved by giving the image saving kernel two resume points: one for the image has been written, and one for we rebooted and have restored the image. I'm not familiar with ACPI.

Re: [linux-pm] Re: Hibernation considerations

Perhaps we need a third to differentiate we read the image from S4 instead of from S5, but that information must be available to the OS because it needs that to know if it should resume from hibernate.

By making the split at image save and restore we have several advantages:

- (1) the kernel always initializes with devices in the init or quiesced but active state.
- (2) the kernel always resumes with devices in the init or quiesced but active state.
- (3) the kjump save and restore kernel does not need to know how to suspend all devices in the platform.
- (4) we have a merged path for suspend to disk, suspend to ram, and suspend to both.
- (5) because of (4), we can implement sleep policys where we save the image to disk but try to stay in ram based on expected remaining battery life.
- (6) we confine all platform (acpi) interaction to the main kernel
- (7) we limit the knowledge needed in the second kernel. It needs to know how to do its job and then put the hardware back how it found it. Nothing more.

This would have been nice if we had been able to do it.

For the suspend to ram and then woken up case, we simply need to invalidate the image before restarting normal kernel operation.

People have worried about how to boot and restore the kernel, and what to do if reading the image fails. They worry about needing memory hotplug or delayed acpi parsing. They are forgetting one thing. This kernel has support for kexec.

This is all easily solved by having the bootloader from the bios always boot the restore kernel.

Well, I think this is not generally acceptable, although I agree that it would be simpler.

It will boot with limited useable memory and no acpi support. If the restore kernel userspace detects that there is

Re: [linux-pm] Re: Hibernation considerations

no restore image, it simply loads the normal main kernel and initrd /
initramfs and calls the normal kexec. The cost is the time to init the
restore kernel, read the kernel with full drivers (vs reading it from
the bootloader). If you want a boot menu, use kboot (on sourceforge).

Well, I'm afraid of adding more and more infrastructure to the mix.

On Jul 17, 2007, at 2:13 PM, Rafael J. Wysocki wrote:

On Tuesday, 17 July 2007 22:27, david@xxxxxxx wrote:

On Tue, 17 Jul 2007, Alan Stern wrote:

But what about the freezer? The original
reason for using kexec was
to
avoid the need for the freezer. With no
freezer, while the original
kernel is busy powering down its devices,
user tasks will be free to
carry out I/O -- which will make the
memory snapshot inconsistent
with
the on-disk data structures.

no, user tasks just don't get scheduled during shutdown.

the big problem with the freezer isn't stopping anything from
happening,
it's selectively stopping things.

Agreed. Or rather, selectively not stopping and resuming things.

I don't quite understand this statement. Can you please elaborate?

It's selectively stopping kernel threads, which is just about right.
If you
that this is a main problem with the freezer, then think again.

with kexec you don't need to let any portion of the original
kernel
or
userspace operate so you don't have a problem.

Re: [linux-pm] Re: Hibernation considerations

In fact, the main problem with the freezer is that it is a coarse-grained solution. Therefore, what I believe we should do is to evolve in the direction of more fine-grained solutions and gradually phase out the freezer.

The kexec-based approach is an attempt to replace one coarse-grained solution (the freezer) with even more coarse-grained solution (stopping the entire kernel with everything), which IMO doesn't address the main problem.

I think this addresses the problem. It's probably a bit harder than powermac because we have to fully quiesce devices; we can't cheat by leaving interrupts off. But once the drivers save the state of their devices and stop their queues, it should be easy to audit the paths to powerdown devices and call the platform suspend and ram wakeup paths.

Going back to the requirements document that started this thread:

Message-ID: <200707151433.34625.rjw@xxxxxxx>
On Sun Jul 15 05:27:03 2007, Rafael J. Wysocki wrote:

- (1) Filesystems mounted before the hibernation are untouchable

This is because some file systems do a fsck or other activity even when mounted read only. For the kexec case, however, this should be "file systems mounted by the hibernated system must not be written". As has been mentioned in the past, we should be able to use something like dm snapshot to allow fsck and the file system to see the cleaned copy while not actually writing the media.

We can't require users to use the dm snapshot in order for the hibernation to work, sorry.

And by reading from a filesystem you generally update metadata.

The kjump kernel must not have any knowledge retained if we reuse it.

- (2) Swap space in use before the hibernation must be handled with care

Yes. Actually, even though they have been used by the write-in-the

Re: [linux-pm] Re: Hibernation considerations

kernel users, they will be among the most difficult devices to use for snapshots by a userspace second kernel.

(3) There are memory regions that must not be saved or restored

because they may not exist. This means that we must identify the memory to be saved and restored in a format to be passed between the kernel.

(4) The user should be able to limit the size of a hibernation image

This means the suspending kernel must arrange to reduce its active memory. The limited save can be done by providing a limited list in (3).

It seems to me that you don't understand the problem here.

Assume you have 90% of RAM allocated before the hibernation and the user has requested the image to be not greater than 50% of RAM. In that case you have to free some memory *_before_* identifying memory to save and you must not race with applications that attempt to allocate memory while you're doing it.

(5) Hibernation should be transparent from the applications' point of view

People have pointed out they may want userspace to be aware of the suspend. I believed this can be done with `/proc/apm` emulation today or by other means; it seems that should be hooked up to dbus in some fashion.

Not a solution, because there still will be programs not needing to know anything about hibernation. After all, we don't require all applications to know anything about SMP, even if they are executed on an SMP system.

(6) State of devices from before hibernation should be restored, if possible

related to suspend should be transparent ... yes.

Re: [linux-pm] Re: Hibernation considerations

(7) On ACPI systems special platform-related actions have to be carried out at the right points, so that the platform works correctly after the restore

I believe I have explained my suggestion.

(8) Hibernation and restore should not be too slow

We control the added code. We are using full runtime drivers and will run at hardware speeds.

That may not be enough. If you're going to save, say, 80% of RAM on a 2 GB machine, then you'll have to be using image compression.

(9) Hibernation framework should not be too difficult to set up

Ok the current patch is presently too difficult. But I think it will be much simpler with a few small changes.

As noted in the thread

Message-ID: <873azxwqhr.fsf@xxxxxxxxxxxx>
Subject: [linux-pm] Re: hibernation/snapshot design
on Mon Jul 9 08:23:53 2007, Jeremy Maitin-Shepard wrote:

Both would work. One would eat 8-64MB of your RAM, permanently;

As I have stated in other messages, the kdump approach would not waste any RAM permanently.

...

Immediately before jumping to the new kernel, the first X bytes (where X is the amount of memory the new kernel will get, typically 16MB or 64MB) of physical memory are backed up into the arbitrary discontinuous pages that are made available. This will not take very long, because copying even 64MB of memory is extremely fast. Then the new kernel is free to use the first X bytes of contiguous physical memory. Problem solved.

Ok, now let's look at my list again:

(1) how to interact with acpi to enter into S4.

This was discussed.

(2) how to identify which memory needs to be saved

We need to generate a list. We need it to fit in a computable size so that we can free and allocate the pages before suspending IO in the first kernel.

One possibility is to use something like the kexec copy list. If we are imaging a small fraction of ram this is appropriate, but if we are doing dense saves we need something extent based. We should be able to extend the list.

(3) how to communicate where to save the memory

This is an interesting topic. The suspended kernel has most IO and disk space. It also knows how much space is to be occupied by the kernel. So communicating a block map to the second kernel would be the obvious choice. But the second kernel must be able to find the image to restore it, and it must have drivers for the media. Also, this is not feasible for storing to nfs.

I think we will end up with several methods.

One would be supply a list of blocks, and implement a file system that reads the file by reading the scatter list from media. The restore kernel then only needs to read an anchor, and can build upon that until the image is read into memory. Or do this in userspace.

I don't know how this compares to the current restore path. I wasn't able to identify the code that creates the on disk structure in my 10 minute perusal of kernel/power/.

The structure is created at two levels.

First, the code in snapshot.c makes the image available to the code in swap.c as a stream of pages. The first page is the header, followed by some pages containing the PFNs of the page frames to which the image data pages are to be

Re: [linux-pm] Re: Hibernation considerations

restored, followed by the image data pages themselves (the ordering of the PFNs must be the same as the ordering of data pages that correspond to them). Still, the low-level image format only needs to be known by the restore code in snapshot.c .

Second, the code in swap.c writes the image pages to a storage adding some metadata making it possible to reproduce their original ordering during the restore.

The fact that we use swap spaces as the storage is related to implementation simplicity rather than anything else.

A second method will be to supply a device and file that will be mounted by the save kernel, then unmounted and restored. This would require a partition that is not mounted or open by the suspended kernel (or use nfs or a similar protocol that is designed for multiple client concurrent access).

A third method would be to allocate a file with the first kernel, and make sure the blocks are flushed to disk. The save and restore kernels map the file system using a snapshot device. Writing would map the blocks and use the block offset to write to the real device using the method from the first option; reading could be done directly from the snapshot device.

The first and third option are dead on log based file systems (where the data is stored in the log).

All in all, we have three different and working implementation of the image-writing and image-reading code at our disposal. Why would you want to break the open doors?

(4) what state should devices be in when switching kernels

My proposal is either initialized and untouched or quiesced.

This is reasonable, but in general we also need to save some information about the pre-hibernation state of devices, so that we can put them into the same state, if reasonably possible, during the restore.

(5) the complicated setup required with the current patch

I think a few simple changes to kjump will make this much simpler. See below.

(6) what code restores the image

The save kernel, loaded at boot. People have suggested booting the first kernel, and using current restore code. However, I think that ignores that (1) we saved from a different kernel, so the backed up region will be restored to its backed up random pages,

This problem has already been solved.

(2) the code was written to restore the same kernel,

Not exactly. In fact, the current implementation only relies on the tiny portion of the restore code being in the same place in both kernels, but we can change the code not to make this assumption (it'll be more complicated, but that's perfectly doable).

so the text and data will be replaced by identical text. Its much simpler conceptually to use the same kernel to save and restore the image.

Here I agree. :-)

Simplifying kjump: the proposal for v3.

The current code is trying to use crash dump area as a safe, reserved area to run the second kernel. However, that means that the kernel has to be linked specially to run in the reserved area. I think we need to finish separating kexec_jump from the other code paths.

(1) add a new command line argument that specifies the kexec_jump target area.

(2) add a kjump flag to the flags parameter, used by kexec_load. When loading a jump kernel, it is loaded like a normal kernel, however, additional control pages are allocated to (a) save the kexec_jump target area (b) save the backed up region that is used by all kernels like crash dump, and (c) space for invoking relocate_new_kernel that will get its args from the execution entry point and will restore the kernel then call resume and suspend.

(3) replace jump_huf_pfn with two command line addresses that specify the (a) return point for after resume, and (b) the return point for after image save. Actually these can be done in userspace; the second

Re: [linux-pm] Re: Hibernation considerations

restore kernel can just specify the null copy list and the entry points supplied by the suspended kernel. To do resume we also need (c) where to store resume address for the save kernel.

As a first stage of suspend and resume, we can save to dedicated partitions all memory (as supplied to `crash_dump`) that is not marked `nosave` and not part of the save kernel's image.

A little problem here: there are "nosave" areas that are not marked as nosave.

The fancy block lists and memory lists can be added later.

On the majority of systems that will work. On some of them it won't.

mmaking these changes will allow us to use a normal kernel invoked with `acpi=off apm=off mem=xxk` as the save and restore kernel.

If we want to keep the second kernel booted, then we need to add a save area for the booted jump target. Note that the save and restore lists to `relocate_new_kernel` can be computed once and saved. Longer term we could implement `sys_kexec_load(UNLOAD)` that would retrieve the saved list back to application space to save to disk in a file. This means you could save the booted save kernel, it just couldn't have any shared storage open.

I'll try to expand on this in the jump v2 thread, but it may be 36+ hours before I do so.

Well, I have no experience with kexec, so I really can't comment your kexec-related suggestions.

Greetings,
Rafael

"Premature optimization is the root of all evil." – Donald Knuth

–

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to `majordomo@xxxxxxxxxxxxxxxxx`
More majordomo info at <http://vger.kernel.org/majordomo-info.html>
Please read the FAQ at <http://www.tux.org/lkml/>