Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Improving Linux Startup Time Using Software Resume (and other techniques)

Hiroki Kaminaga
Sony Corporation
kaminaga@sm.sony.co.jp

Abstract

This paper presents a new resume operation as well as other startup time improvement techniques which are aimed at achieving fast startup time for embedded Linux systems. A new fast boot method called snapshot boot is introduced. Snapshot boot is essentially a resume-from-disk operation, which is a system resume from a semi-permanent snapshot image stored on disk or flash memory, that restores the machine to a known running state. As opposed to a standard resume operation, a snapshot image is made only once, stored on disk or flash memory, and same image is used repeatedly, every time the system is powered on. Other fast boot techniques that are discussed are: use of prelinking, a scheme to reduce the startup cost of symbol relocation overhead for links to dynamic libraries, execute in place (XIP) to reduce or avoid OS and application loading delays, toolchain modifications to collect global constructors in one place to accomplish a locality benefit, and making the program `.data` section demand-paged from flash to avoid fully loading its pages on startup.

Unless otherwise stated, the startup time referred to in this paper is the time from the system power on to the time user can manipulate the device. This includes userland application startup as well as kernel startup time.

1 Software suspend

Snapshot boot is based on the current software suspend technology in the Linux kernel. Software suspend is independent of APM or ACPI, which makes it more applicable to embedded systems, where APM or ACPI is not present in many cases. Before describing snapshot boot, the standard procedure of software suspend is described to assist in understanding the procedure of snapshot boot.

1.1 Suspend states in Linux kernel

There are three suspend states in the Linux kernel [1]. They are:

- Standby state
- Suspend-to-RAM state
- Suspend-to-disk state

Unless otherwise stated, the term *suspend* is referred to as Suspend-to-disk. This is also known as hibernation.

1.2 Suspend procedure

The suspend procedure is shown below:

1. *Trigger*
   Suspend procedure is triggered from writing `disk` to `/sys/power/state` operation. The call stack to the entrance of the suspend procedure is shown in Figure 1.

```
sys_write()
+-vfs_write()
  +-sysfs_write_file()
    +-flush_write_buffer()
      +-subsys_attr_store()
        +-state_store()
          +-enter_state()
            +-pm_suspend_disk()
```

Figure 1: Call graph to entrance of software suspend

2. *Freeze processes*
   This is done by calling the `freeze_processes()` procedure. It freezes user processes, and then freezes kernel tasks.

3. *Free unnecessary memory*
   This is done by calling the `free_some_memory()` procedure. It calls `shrink_all_memory()` inside.

4. *Suspend devices*
   This is done by calling the `device_suspend()` procedure. It calls `suspend_device()` and then the `suspend()` method for all listed active devices.

5. *Power down devices*
   This is done by calling the `device_power_down()` procedure. It calls `suspend_device()` to for all listed power off devices.

6. *Save processor state*
   This is done by calling the `save_processor_state()` procedure. It will save registers other than generic ones, such as segment registers, co-processor registers, and so on.

7. *Save processor registers*
   This is done by calling the `swsusp_arch_suspend()` procedure. It will save general registers. This is written in assembly language, since the stack may not be used.

8. *Allocate memory for snapshot image*
   This is done by calling the `swsusp_alloc()` procedure. Page directories get allocated by calling `__get_free_pages()`, and pages for the image itself gets allocated by `get_zeroes_page()` for each page directory entry.

9. *Copy memory contents to allocated area*
   This is done by calling the `copy_data_pages()` procedure. It calls `memcpy()` for each page to copy.

10. *Restore processor state*
    This is done by calling the `restore_processor_state()` procedure in `swsusp_suspend()`. This is where the software suspend resume procedure comes back. It restores the previously saved processor state.

11. *Power up devices*
    This is done by calling the `device_power_up()` procedure. It resumes system devices and all listed power off device.

12. *Resume devices*
    This is done by calling the `device_resume()` procedure. It resumes devices in power off device list.

13. *Write page, pagedir, header image to swap*
    Now that the devices are active, write to swap could be performed. This is done by calling the `write_suspend_image()` procedure. It writes image data, page directories, and then image header into swap.

14. *Power down devices*
    This is done by calling the `device_shutdown()` procedure. It calls the shutdown() method for each device. Then, the system device is shutdown.

15. *Halt machine*
    This is done by calling the `machine_power_off()` procedure. It calls `pm_power_off()` and the machine halts.

1.3 Resume procedure

The resume procedure is shown below:

1. *Start resume*
   Resume starts by calling the `software_resume()` procedure in `do_initcalls()`, at `late_initcall` timing.

2. *Check kernel parameter*
   In `software_resume()`, it checks for the kernel command line for the resume swap device.

3. *Check signature and header of snapshot image*
   This is done by calling the `check_sig()` and `check_header()` procedures. It checks swap image signature for snapshot image, and that the header for the kernel used for suspend and resume is the same.

4. *Allocate memory space for snapshot image*

5. *Read page directory into allocated memory*
   This is done by calling the `read_pagedir()` procedure. It allocates page directory memory space by using `__get_free_pages()` and reads page directory information with `bio_read_page()`.

6. *Relocate page directory (if necessary)*

7. *Read swap image into allocated memory*
   This is done by calling the `data_read()` procedure. The page directory area gets relocated if it collides with the snapshot image. Then the snapshot image is read from swap with `bio_read_page()`.

8. *Prepare resume*

9. *Freeze process*

10. *Free unnecessary memory*

11. *Suspend devices*
    These steps are taken to accomplish consistency between suspend and resume, and in case resume fails. These steps are same as Steps 2 to 4 of the suspend procedure.

12. *Power down devices*
    This step is taken to accomplish consistency between suspend and resume. This step is same as Step 5 of the suspend procedure.

13. *Save processor state*
    These steps are taken in case resume fails. These steps are same as Step 6 of the suspend procedure.

14. *Copy snapshot image in allocated memory to its original address*

15. *Restore processor registers*
    This is done by calling the `swsusp_arch_resume()` procedure. It copies all image pages from the allocated memory address to its original memory address.

It also restores general purpose registers. Since registers are restored, the return address used by this function would be the same as the one in effect for the `swsusp_arch_suspend()` procedure call at suspend time.

16. *Restore processor state*

17. *Power up devices*
    This is exactly the same as Steps 10 to 11 of the suspend procedure.

18. *Free memory allocated for image*
    This is done by calling the `free_image_pages()` procedure. It does `free_page()` for all pages in image. Page directories are also freed.

19. *Resume devices*
    This is done by calling the `device_resume()` procedure. The process is the same as Step 12 of the suspend procedure.

20. *Thaw processes*
    This is done by calling the `thaw_processes()` procedure. This wakes up every thread by calling the `wake_up_process()` procedure.

1.4 Software suspend support for ARM architecture

Software suspend does not support the ARM architecture in a vanilla kernel. To port software suspend for other architectures, a porting note [2] was followed which shows how to port for the ARM architecture. The patch for software suspend ARM support is posted to a public mailing list [3].

1.5 Execution of software suspend

To evaluate software suspend for an embedded system, an ARM-based OMAP 5912 Starter Kit (OSK5912) reference board was used [4]. Since this board does not have a disk, NOR flash is used to store the snapshot image.

To enable software suspend, the `CONFIG_PM` and `CONFIG_SOFTWARE_SUSPEND` configuration options must be set when building the linux kernel. After the target is booted with the new kernel, the following commands were issued to enter suspend.

```
# mkswap /dev/mtdblock3
# swapon /dev/mtdblock3
# mount -t sysfs none /sys
# echo disk >/sys/power/state
```

A kernel message is printed to console, and the system would halt. At the next system power on, passing the argument: `resume=/dev/mtdblock3` (in the above case) will trigger software resume.

2 Snapshot boot

2.1 Preserving snapshot image

For normal use of software suspend and resume, a snapshot image is created and destroyed on every suspend/resume cycle. Since the aim of using software suspend in an embedded product in this paper is to improve startup time, a snapshot image is created only once, stored on disk or flash memory, and the same image is used repeatedly, every time the system is powered on. This is accomplished by not invalidating the snapshot image at resume time.

2.2 Principle of software resume for improving startup time

The time to a certain point in running system state could be roughly expressed as follows:

$$startup\ time = \sum Storage\ to\ RAM + \sum setup\ I/O\ state + \sum setup\ RAM\ state$$

where $\sum Storage\ to\ RAM$ is the time taken to load files in secondary storage to RAM, including kernel, application, and library files, $\sum setup\ I/O\ state$ is the time taken to setup I/O state, $\sum setup\ RAM\ state$ is the time taken to calculate or process data until a certain point in time, including dynamic symbol resolution, global constructor execution, and application-specific initialization and setups.

Software Resume could be thought of replacing the last $\sum setup\ RAM\ state$ calculation and processing by just copying snapshot image back to RAM. On a complex system, it is estimated that this setup RAM state would be the dominant startup time, and startup time could be reduced if this setup RAM state is replaced by just copying the snapshot image to RAM.

There are drawbacks in the usage of software resume, and one of them is remount of file system, to keep consistency between the actual fs tree and kernel fs tree data. The example of this drawback is USB mass storage, to handle startup state for both plugged and unplugged cases.

2.3 Software resume and startup time

Since the focus of this paper is to improve startup time, the time taken in the software suspend phase, when the snapshot image is created, is not significant. The significant part is the time taken during the software resume phase. There are a couple of redundancies in software resume, which can be worked on to improve startup time. They are:

- The trigger of software resume is in a late phase of kernel startup.
- The snapshot image is copied twice.
- There is redundancy in device state transitions during booting.

2.3.1 Software resume starting time

As mentioned in Step 1 of the resume procedure in Section 1.3, the `software_resume()` procedure is called from `do_initcalls()`. At this point, the kernel is almost ready to start, the architecture setup is done, scheduling is initialized, trap, rcu, irqs, timer, and memory are initialized, the init thread is forked, and basic setup, such as populating rootfs, driver initialization, and network initialization, are done. However, if the system is going to resume from software suspend, some of these steps could be skipped, or handled during software resume.

2.3.2 Copying redundancy of snapshot image

As mentioned in Steps 7 and 14 of the resume procedure in Section 1.3, a snapshot image is copied from swap to allocated memory, and then from allocated memory area to its original memory position. Copying the snapshot image twice is needed to keep consistency before suspend and after resume. Consistency is kept by assuring that each device is in the same state before suspend and after resume, and that the PM state is the suspended state. Since the device is suspended, a snapshot image can only be placed in memory. After the device gets resumed, that snapshot image can then be copied to swap.

2.3.3 Device state transition redundancy

As mentioned in the resume procedure steps, each device in the system gets activated, ini-

tialized, and ready to be used, then transitioned to the suspend state. Device initialization and setup is needed for getting the snapshot image from swap, while transitioning to the suspend state is needed for consistency between the system suspend state and the system resume state.

2.4 Improving software resume startup

The most time-consuming part of software resume is copying of the snapshot image, since the image size in 10MB even with almost no processes running, and more than double that size when an application such as mplayer is running.

In order to copy the snapshot image directly from swap to its original memory address, the procedure below was followed, with the involvement of the boot loader.

1. Copy the snapshot image to its original memory address, by boot loader.
2. Setup devices not handled by kernel resume, by boot loader.
3. Devices set to suspend state, by boot loader.
4. Jump to kernel-resume-point, not normal kernel entry point.
5. Devices gets powered on and resumed by kernel.
6. Processes gets thawed by kernel.

The kernel resume point is at Step 15 of the resume procedure in Section 1.3, just after the snapshot image is copied to its original address and the kernel is about to restore registers. This method is named snapshot boot.

2.4.1 Tasks done by the boot loader

Additional tasks done by the boot loader for snapshot boot are to copy the snapshot image from swap to its original memory address, set up devices not handled by the kernel resume procedure, and jump to the kernel-resume-point. For the referenced target, copying the snapshot image to memory was done by simple word-to-word copying. However, if the target supports burst transfer, that should be used to shrink the time taken for copying the image. Kernel areas that needed to be modified for the referenced target were: enhancing the clock speed, timer setup, and enabling MMU. The kernel-resume-point address was obtained from `System.map`.

2.5 Implementation of snapshot boot

A new command for snapshot boot was implemented in `u-boot` [5]. The syntax of the command is shown in Figure 2.

The procedure done by this new command is described in Section 2.4.1.

A new resume entry point function is added on top of the software suspend ARM support. It sets a flag to indicate snapshot boot has been done, and then jumps to Step 15 of the resume procedure in Section 1.3, which is in the middle of the `swsusp_arch_resume()` procedure.

2.6 Evaluation of snapshot boot

Startup time is measured using `printk` time functionality, by setting `CONFIG_PRINTK_TIME`. This configuration emits time at every `printk` output. Two system situations are evaluated, one is until execution of init shell script, with almost no work load, and other

```
PROMPT> bootss <snapshot image address> <kernel resume point>
```

Figure 2: Command for Snapshot Boot in Boot Loader

is while playing an MPEG video file with the mplayer application. Reading time data would cause additional startup time, but is neglected for this evaluation. No optimization regarding application startup is applied to mplayer.

Normal startup is cold startup of system, and time is measured from when the timer is initialized at beginning of kernel startup, to just before the init shell script gets executed. For mplayer, time is measured until the Tux picture, set by kernel, is replaced in LCD panel by MPEG data. Software resume is measured from when the time is initialized at beginning of kernel startup, to the time all processes are thawed. Snapshot boot is measured from timer initialization before copying snapshot image to memory at boot loader, to the time all processes are thawed. For software resume and snapshot boot, image is created after system enters shell and flash as swap is setup and enabled to store image for comparison against normal startup of init. So there is a difference for init shell startup measurement, normal startup is timed till before it gets executed, while software resume and snapshot boot image is created at shell running state. The size of created snapshot image for software resume and snapshot boot were 1424 and 2410 pages for shell and mplayer respectively. Result of the average startup time of 10 trials for each system status is shown in Table 1, and result of each trials are shown in Table 2 and Table 3 respectively.

In both software resume and snapshot boot method, most of the time is taken in copying of snapshot image. For snapshot boot, 80 to 90 percent of the time is occupied by image copying. As mentioned in Section 2.4.1, image is copied on a word basis; however, if the hardware supports burst transfer mode, the snapshot boot startup time would shrink dramatically.

Application	Normal startup	Software resume	Snapshot boot
shell	2.872	6.370	3.580
mplayer	11.1	10.427	5.357

Table 1: Average Time of Each Method [sec]

Trial count	Software resume	Snapshot boot
1	6.831	2.478
2	6.166	4.028
3	6.166	3.132
4	7.538	4.674
5	6.166	3.132
6	6.166	3.133
7	6.166	4.766
8	6.166	4.029
9	6.166	3.949
10	6.166	2.478

Table 2: Shell Resume [sec]

3 Issues met in snapshot boot

3.1 Assumption in snapshot boot

To minimize the effort of the boot loader and to reuse kernel procedures for device manipulation, it is assumed that the device power up and resume procedures in the kernel will handle the devices. However, some devices are initialized and setup at kernel startup only. Some do not have device manipulation at resume. Those devices work with in software resume, since

Trial count	Software resume	Snapshot boot
1	9.793	5.305
2	10.593	5.305
3	9.731	5.305
4	10.593	5.305
5	10.593	6.080
6	10.593	4.460
7	10.593	5.251
8	10.592	5.305
9	10.593	5.953
10	10.593	5.305

Table 3: Mplayer Resume [sec]

kernel startup sequence initializes such devices, before triggering software resume. However, these devices have to be set up somewhere in snapshot boot. Currently, they are handled in the boot loader during the snapshot boot sequence, however, this apparently doubles the effort, and some infrastructure is needed.

The same issues were faced with MMIO. The MMIO registers were initialized and set up during kernel startup, and it worked on software resume, but since snapshot boot doesn't perform the kernel startup sequence, it must be handled somewhere else. The appropriate place to handle these in snapshot boot would be at kernel resume time, where MMIO register related devices do their resume operations.

3.2 Current workaround

Calling the initialization and setup procedures of such devices after snapshot boot has jumped into kernel was considered, but it did not work out, since information in the data related to those devices report that initialization and setups are done, and simply return back. Currently, those initialization and setups are handled on the boot loader side, just before jumping into the kernel resume point. These are implemented one by one, during the implementation and testing cycle of snapshot boot on the target, and appended in the snapshot boot operation of the boot loader. The UART, IRQ configuration, GPIO, DMA, DSP, I2C, TPS65010 chip, UWIRE CS0, UWIRE CS1, OCPI, NOR flash and key pad are handled in this way. Some multiplex setup and pulldown control is also set. Devices processed at kernel resume were the SMC91 network chip, Compact Flash, serial 8250, I2C, TPS65010 chip, and LCD control, in which the resume method existed and taken care of, are serial 8250 and LCD control.

Some interrupt registers and mask registers also had similar problems at snapshot boot, and a similar workaround was used.

3.3 Proper model and infrastructure

To keep the snapshot boot generalized and not system-specific, the boot loader should do minimal work for snapshot boot, and most snapshot boot process should be handled by the kernel. To accomplish this, more devices should be extended to implement the resume method, not handling only resume from RAM, but also resume from disk, taking into account that the device was powered off. Other than that, if the kernel has separate calls for hardware initialization and setup from its related data initialization and setup, that could be used for snapshot boot support. Ideally, the boot loader operation for snapshot boot would be just copying the snapshot image and jumping to the kernel-resume-point.

Regarding the infrastructure issue, the data structure of snapshot image varies at different kernel version. For example, page directories are implemented as array in version 2.6.11, whereas in a recent kernel, it is implemented as a list structure. At implementing snapshot

boot in boot loader side, this would have great impact. Some kind of applicable interface is needed for consistency.

4 Other techniques for startup improvement

There are various other existing techniques for improving startup time. Although some of these focus on kernel startup, most are focused on application startup. This is because application startup takes longer than kernel startup in many cases. One example of an improvement focusing on kernel startup time is kernel execute-in-place (XIP). Some examples of improvement focusing application startup time are prelinking, allocate-on-write of `.data` section, and gathering global constructors in one place for locality benefit.

Some of these existing techniques may have a benefit when used along with snapshot boot, whereas others may not. Of the listed existing techniques, ones that may have benefit when used along with snapshot boot are XIP and allocate-on-write of `.data` section. Ones that probably do not have a benefit when used along with snapshot boot are prelinking and gathering global constructors in one place for locality benefit.

4.1 XIP and snapshot boot

A description of XIP is documented at [6] and kernel XIP at [7]. The basic idea is to execute programs directly from Flash or ROM, and save RAM usage (the data section still needed to be placed into RAM) as well as short-cut program loading time and improve startup time. Currently, no tests have been done using both XIP and snapshot boot. The assumption is that XIP could reduce the snapshot image size, and might contribute to faster startup for a snapshot boot.

4.2 Prelinking and snapshot boot

A description of prelinking is documented at [8]. The basic idea is to perform the task of dynamic linking, such as symbol resolution, in advance, and save the information, so that some of the tasks of dynamic linking could be skipped. When used along with snapshot boot, prelinking may or may not have benefit in startup, depending on the timing when the snapshot image is created. In case the program is already loaded before creating the snapshot image, prelinking for that application would not gain any benefit. However, for programs that are not yet loaded, it would gain a benefit, since dynamic linking has not yet been performed.

4.3 Allocate-on-write of `.data` section and snapshot boot

A description of allocate-on-write of the `.data` section is documented at [9]. The basic idea is to modify the dynamic linker, and change the `.data` section mapping attribute by dropping the `PROT_WRITE` bit in `mmap()` and call `mprotect()` immediately after, and then set the `PROT_WRITE` bit. Currently, no tests have been done using both allocate-on-write of the `.data` section and snapshot boot. The assumption is that, like the XIP technique, it could reduce snapshot image size, and thus might contribute to faster startup of snapshot boot.

4.4 Gathering of global constructors and snapshot boot

The basic idea here is to allocate global constructors and destructors in one place, so that

a data locality benefit can shorten startup time. It is done by first collecting global constructors and destructors in one original section, and later merging them to the `.data` section of the program. Currently, no tests have been done using both the gathering of the global constructors technique and snapshot boot. The assumption is that, there would be less, or no benefit toward improving the snapshot boot startup time.

4.5 Deduction

Whether using another startup improvement technique with snapshot boot would gain a benefit or not would depend on the feature. When the technique has only an instant effect at startup, such as prelink, it would be likely that little or no startup improvement would be obtained when used together with snapshot boot. On the other hand, if the technique has a side effect, such as reduced load on RAM, it is likely that the system would gain a startup improvement by using both techniques.

5 Future work

As mentioned in Section 3, an urgent issue regarding snapshot boot is generalization and need of infrastructure for device manipulation. This must be discussed at the community level to draw agreement and diffusion. Further testing and evaluation of snapshot boot is needed, such as effect when used with other existing techniques for improving startup.

6 Conclusion

Snapshot boot is a technique for improving startup time, based on software suspend. The existing techniques for improving startup time, such as XIP, prelink, and others show effect at real startup time, by short-cutting some process, such as load instruction code to RAM, symbol resolution, etc., at various points, from the kernel startup to the application startup. Snapshot boot could be viewed as "short-cutting all startup," by using run-time system snapshot image.

References

[1] *System Power Management States* `Documentation/power/states.txt`

[2] *SwSuspendPortingNotes* `http://tree.celinuxforum.org/CelfPubWiki/SwSuspendPortingNotes`

[3] *swsusp for OSK* `http://lists.osdl.org/pipermail/linux-pm/2005-July/001077.html`

[4] *OMAP 5912 Starter Kit* `http://tree.celinuxforum.org/CelfPubWiki/OSK`

[5] *Das U-Boot - Universal Bootloader* `http://sourceforge.net/projects/u-boot/`

[6] *Execute in Place(XIP)* `http://www.montavista.co.jp/products/tech/saving_ram.html`

[7] *Kernel XIP* `http://tree.celinuxforum.org/CelfPubWiki/KernelXIP`

[8] *Prelink* `http://people.redhat.com/jakub/prelink.pdf`

[9] *Making Mobile Phone with CE Linux* `http://tree.celinuxforum.org/CelfPubWiki/ITJ2005Detail1_2d2`