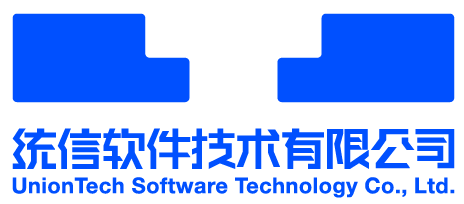
快速启动方案调研



|  |  |  |  |
| --- | --- | --- | --- |
| 配置项编号 |  | | |
| 密级 | 秘密 | 版本 | V0.1.0 |
| 拟制 |  | 日期 |  |
| 审核 |  | 日期 |  |
| 批准 |  | 日期 |  |

**版本变更记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **修订说明** | **修订人** | **修订时间** |
| 0.1.0 | 创建 | 聂诚 | 2024.07.02 |
| 0.1.1 | 新增2.2 4.5.2 4.4.1 | 聂诚 | 2024.11.01 |
|  |  |  |  |
|  |  |  |  |

**目录**

[1. 背景 4](#_Toc1132284520)

[1.1. 快速启动的工作原理 4](#_Toc1818493279)

[1.2. fast startup的问题 4](#_Toc1248152852)

[1.3. 调研目标 5](#_Toc1033273167)

[2. linux相关流程 5](#_Toc1917499262)

[2.1. 开机 5](#_Toc1450918322)

[2.1.1. BIOS和UEFI阶段 5](#_Toc2035989921)

[2.1.2. bootloader/grub 5](#_Toc1663676479)

[2.1.2.1. 阶段1 5](#_Toc1963871993)

[2.1.2.2. 阶段1.5 6](#_Toc1676366524)

[2.1.2.3. 阶段2 6](#_Toc175345158)

[2.1.3. kernel启动 6](#_Toc953503144)

[2.1.3.1. 内核自解压 6](#_Toc783196977)

[2.1.3.2. 架构特定基础硬件初始化 6](#_Toc24186414)

[2.1.3.3. start\_kernel 6](#_Toc1838375086)

[2.1.4. initrd/initramfs 7](#_Toc1912300106)

[2.1.5. systemd 7](#_Toc1539966541)

[2.1.6. lightdm 8](#_Toc1768021725)

[2.2. 系统睡眠状态 8](#_Toc211155060)

[2.2.1. 休眠唤醒 9](#_Toc300454598)

[2.2.1.1. 休眠 9](#_Toc772980074)

[2.2.1.2. 唤醒 9](#_Toc1977089144)

[2.2.1.3. 设备休眠唤醒 10](#_Toc1265988402)

[2.2.2. ACPI 11](#_Toc1303867526)

[3. linux快速开机方案 11](#_Toc963208055)

[3.1. 休眠唤醒 11](#_Toc495197883)

[3.1.1. 测试 11](#_Toc1791317962)

[3.1.2. 分析 12](#_Toc1621310452)

[3.1.3. snapshot boot 12](#_Toc320769229)

[3.1.4. 总结 13](#_Toc388276146)

[3.2. EFIstub 13](#_Toc136528197)

[3.2.1. 测试 14](#_Toc1453053749)

[3.2.2. 总结 14](#_Toc59285777)

[3.3. 一些零散的优化点 14](#_Toc1384681049)

[3.3.1. initrd的异步unpacking 14](#_Toc338843268)

[3.3.2. initramfs-tools等待耗时 14](#_Toc1976785040)

[3.3.3. 模块buildin 15](#_Toc688115723)

[3.3.4. xorg加载优化 16](#_Toc227349541)

[3.3.5. hibernation image加载优化 16](#_Toc1492977871)

[4. 启动耗时分析工具与方法 16](#_Toc504504068)

[4.1. 上游一些优化手段 16](#_Toc1903716066)

[4.2. pm-graph 16](#_Toc1668323029)

[4.2.1. 工具获取 17](#_Toc1458007212)

[4.2.2. 分析启动过程 17](#_Toc539429395)

[4.2.3. 分析休眠唤醒流程 17](#_Toc1692509443)

[4.2.3.1. 休眠 19](#_Toc1148898651)

[4.2.3.2. 唤醒 20](#_Toc304245854)

[4.2.3.3. 如何分析整个唤醒耗时 20](#_Toc1084992336)

[4.2.3.4. 总结 21](#_Toc769436728)

[4.3. systemd-bootchart 21](#_Toc515400914)

[4.4. systemd-analyze 23](#_Toc1385446934)

[4.5. initramfs-tools调试 23](#_Toc1542416802)

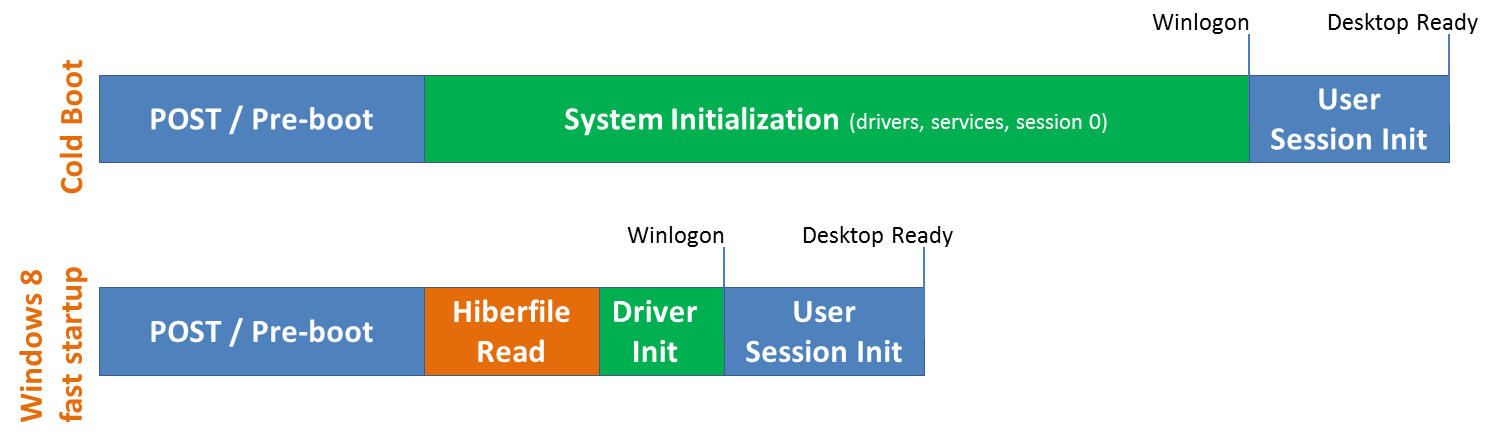
[5. 总结 24](#_Toc345006410)

[6. 缩写 24](#_Toc503951689)

[7. 参考资料 24](#_Toc698800680)

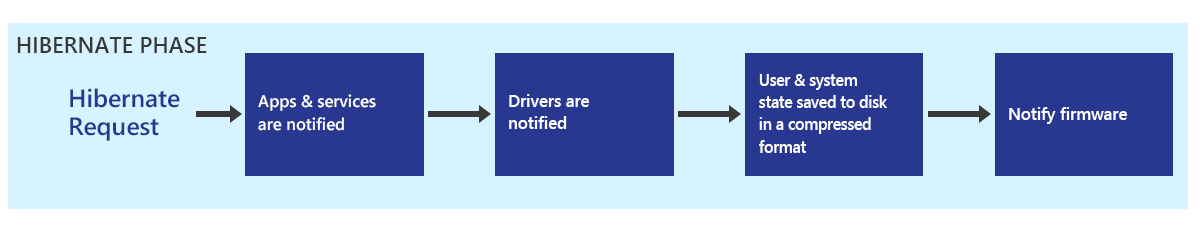
1. 背景

Windows 的快速启动（Fast Startup）是一项在 Windows 8 及其后续版本中引入的功能，旨在缩短系统的启动时间。它结合了传统的关机和休眠模式的优点，通过保存系统内核和驱动程序的状态到一个休眠文件来实现更快的启动过程。

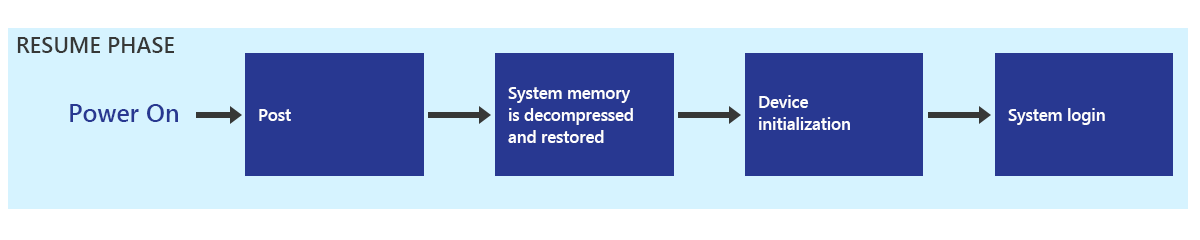


* 1. 快速启动的工作原理

关机时：当你关机时，Windows 会关闭所有正在运行的应用程序和用户会话。但不同于完全关闭系统，快速启动会将系统内核和加载的驱动程序保存到一个称为 hiberfil.sys 的文件中（类似于休眠模式）。



启动时：在下一次启动时，Windows 会从这个休眠文件中恢复内核和驱动程序的状态，而不是重新初始化它们，从而节省了时间，使启动过程更快。



* + 1. resume boot

winresume

* 1. fast startup的问题
* 无法进行系统更新
  1. 调研目标

1. linux中也存在对应的休眠模式，是否可以借鉴windows快速开机方案。
2. 分析开机优化点
3. linux相关流程
   1. 开机

首先我们简单梳理一下linux下的开机流程，开机主要涉及到的流程如下：

BIOS->grub->kernel->initrd->systemd->lightdm->xorg->greeter

* + 1. BIOS和UEFI阶段

BIOS和UEFI是计算机按下电源后最先被执行的程序。

BIOS（代表“基本输入/输出系统”“Basic Input/Output System”）初始化硬件，并通过开机自检（POST）确保所有硬件都正常运行。

BIOS上电自检确认硬件的基本功能正常，然后产生一个BIOS中断和INT 13H，该中断指向某个接入的可引导设备的引导扇区，它所找到的包含有效的引导记录的第一个引导扇区装在到内存中，并且内核的控制权也从引导扇区转移到此段代码。BIOS 的主要工作是加载引导加载程序。

* + 1. bootloader/grub

引导加载程序将内核加载到内存中，然后使用一组内核参数启动内核。最常见的引导加载程序之一是 GRUB，它是一种通用的 Linux 标准。可分为三个阶段:

* + - 1. 阶段1

在 POST 阶段结束时，BIOS 将查找在接入的磁盘中查找引导记录，其通常位于 MBR（主引导记录Master Boot Record）或UEFI引导分区中。它加载它找到的第一个引导记录中到内存中，并开始执行此代码。

引导代码（及阶段 1 代码）必须非常小，因为它必须连同分区表放到硬盘的第一个 512 字节的扇区中。

由于引导记录必须非常的小，它不可能非常智能，且不能理解文件系统结构。因此阶段 1 的唯一功能就是定位并加载阶段 1.5 的代码。为了完成此任务，阶段 1.5 的代码必须位于引导记录与设备第一个分区之间的位置。在加载阶段 1.5 代码进入内存后，控制权将由阶段 1 转移到阶段 1.5。

* + - 1. 阶段1.5

阶段 1.5 的代码必须位于引导记录与设备第一个分区之间的位置。该空间由于历史上的技术原因而空闲。第一个分区的开始位置在扇区 63 和 MBR（扇区 0）之间遗留下 62 bootloader grub个 512 字节的扇区（共 31744 字节），该区域用于存储阶段 1.5 的代码镜像 core.img 文件。该文件大小为 25389 字节，故此区域有足够大小的空间用来存储 core.img。

阶段 1.5 的功能是开始执行存放阶段 2 文件的 /boot 文件系统的驱动程序，并加载相关的驱动程序。

* + - 1. 阶段2

​ GRUB 阶段 2 的主要功能是定位和加载 Linux 内核到内存中，并转移控制权到内核。加载内核映像、加载initrd/initramfs。

* + 1. kernel启动
       1. 内核自解压

在内核映像的开头，有一个小的解压缩程序，它负责解压内核的主体部分。内核映像被加载到内存后，解压缩程序会运行并将压缩的内核映像解压到适当的内存位置。跳转到解压后的内核：一旦解压完成，控制权会被移交给解压后的内核代码的入口点。

* + - 1. 架构特定基础硬件初始化

解压后的内核代码会从一个固定的入口点开始执行，这个入口点是平台和架构相关的。对于x86架构，通常是startup\_32或startup\_64函数。.

根据具体的硬件架构，内核会执行一些必要的初始化步骤，比如设置CPU的运行模式，初始化分页机制，建立基本的内存映射等。

内核设置好自己的堆栈，以便后续的函数调用和操作。然后调用start\_kernel。

* + - 1. start\_kernel

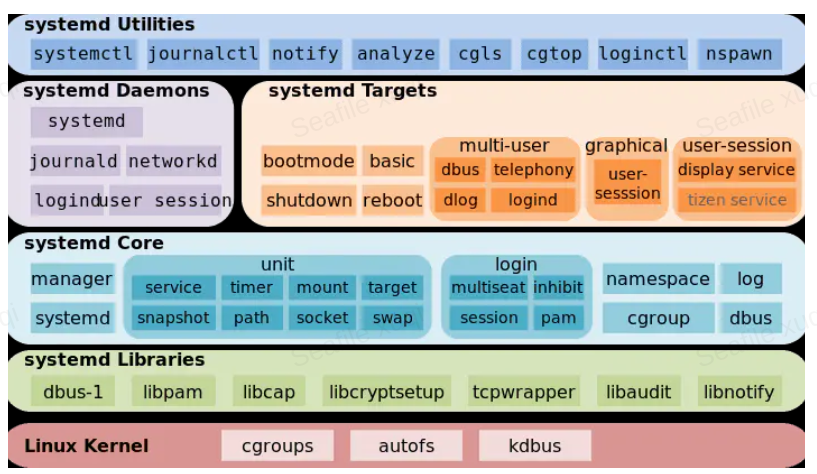
start\_kernel函数位于init/main.c文件中，负责完成大部分内核的初始化工作。

* 初始化控制台：设置内核的打印机制，以便后续的输出可以显示出来。
* 初始化内存管理子系统：建立初始的内存管理结构，准备好内存分配机制。
* 检测和初始化硬件设备：内核会检测并初始化系统中的各种硬件设备和驱动程序。
* 启动中断处理机制：设置和启动中断处理机制，使得内核可以响应硬件中断。
* 初始化内核调度器：初始化内核调度器，以便管理进程调度。
* 加载root文件系统：解压initrd.img，加载root文件系统
* 加载初始进程：内核创建并启动第一个用户空间进程，/init
  + 1. initrd/initramfs

内核在解压完initrd.img之后，会将其加载到内核中，作为一个临时的rootfs，并且其中包含了很多核外驱动镜像。然后内核会执行/init，/init是属于intramfs中的一个脚本程序。脚本完成必要的硬件初始化和配置后，会挂载实际的根文件系统。完成挂载后切换到实际根文件系统，然后移除initrd/initramfs文件。

最后/init脚本会调用/sbin/init，一般来说/sbin/init会链接到systemd上。

* + 1. systemd



* + 1. lightdm

lightdm.service - Light Display Manager 是一个systemd unit，会在开机阶段启动。

LightDM 是一个轻量级的显示管理器，用于启动显示服务，用户会话和用户登录界面。

lightdm会在初始化时拉起Xorg服务，并等待xorg初始化完成。

等待Xorg初始化完成后会拉起greeter。

* 1. 开机时间统计方法

systemd-analyze:

Startup finished in 7.650s (firmware) + 1.352s (loader) + 4.795s (kernel) + 23.250s (userspace) = 37.048s

其中：

- firmware 是 bios 的耗时

- loader 是 grub 的耗时 —— AAA0

- kernel 记录的是从kernel启动到systemd的启动时间(包含整个init阶段) AAA2

- userspace是从systemd启动到systemd任务都启动完成的时间。开机需要管的时间是从systemd启动到greeter的时间，所以，这里的userspace时间没有参考价值

- 内核启动时间的查看方法

sudo dmesg | grep “/init” # initrd启动时间——AAA1

systemd-analyze统计的systemd启动的时间——AAA2

- AAA1-AAA2 就是 initrd 的耗时

systemd-analyze plot | grep lightdm

<text class="left" x="342.906" y="5274.000">lightdm.service (145ms)</text>

342.906ms就是从AAA2到lightdm启动经过的时间。 ———AAA3

查看/var/log/lightdm/lightdm.log日志

**Launching process 1452: /usr/bin/X -background启动x**

**Got signal 10 from process 1452: x启动完成**

**Running command /usr/bin/deepin-greeter greeter启动**

**Session pid=1945: Started with service 'lightdm', username 'uos' greeter启动完成。 ——AAA4**

开机总耗时： AAA0+AAA2+AAA3+AAA4

* 1. 系统睡眠状态

在 Linux 系统中，睡眠状态（Sleep States）是电源管理的一部分，主要是通过swsusp模块来管理系统级别的睡眠状态，依赖于 PM 子系统中的一些关键组件来实现挂起和恢复功能。能减少电力消耗，同时能够快速恢复系统状态。主要的睡眠状态包括：

* Suspend to RAM（S3）：

描述：将系统状态保存在内存中，并关闭大部分硬件设备的电源。内存依然通电以保持数据。

特点：恢复速度快（通常几秒钟内），但仍有少量电力消耗。

命令：echo mem > /sys/power/state 或 systemctl suspend。

* Suspend to Disk（S4）/hibernation休眠：

描述：将系统状态保存在硬盘上，然后完全关闭电源。当重新启动时，从硬盘恢复系统状态。

特点：电力消耗最小（几乎为零），但恢复速度较慢（通常几秒到几分钟）。

命令：echo disk > /sys/power/state 或 systemctl hibernate。

* Hybrid Sleep：

描述：结合 Suspend to RAM 和 Suspend to Disk 的优点。系统状态同时保存在内存和硬盘中。系统首先进入 Suspend to RAM 状态，如果电池耗尽，可以从硬盘恢复状态。

特点：提供快速恢复和数据安全保障。

命令：具体命令依赖于系统配置和支持（一般需要配置 /sys/power/disk 和 /sys/power/state）。

* Freeze（或 Standby）：

描述：将系统进程冻结，进入低功耗状态。

特点：恢复速度非常快，但电力节省效果较弱。

命令：echo freeze > /sys/power/state。

* + 1. 休眠唤醒
       1. 休眠

休眠的流程如下：

1. 冻结进程
2. 释放一些内存
3. 设备suspend(PMSG\_FREEZE)
4. 设备下电
5. 保存CPU状态
6. 制作hibernation镜像
7. 设备上电
8. 设备resume(PMSG\_RESTORE)
9. 将hibernation镜像写入磁盘
10. 整机下电

从上面的流程休眠之前会进行一次suspend 然后进行一次resume，进行suspend的原因是为了让系统达到一个稳定的状态点，而后面又进行resume的原因是需要将image写入磁盘，所以需要相关磁盘设备能正常工作。

* + - 1. 唤醒

唤醒的流程如下：

1. 内核正常启动
2. initrd中检查是否有休眠镜像 or 在late\_initcall中判断
3. 读取休眠镜像到内存中
4. 冻结进程
5. 设备suspend(PMSG\_FREEZE)
6. 设备下电
7. 切换到目标内核中，也就是休眠镜像保存的内核
8. 恢复CPU状态
9. 设备上电
10. 设备resume(PMSG\_RESTORE)
11. 进程解冻

实际上唤醒时，需要正常启动内核一次，这个时候的内核叫boot kernel，然后会检查休眠镜像，如果存在休眠镜像就会进行唤醒动作。先将当前的设备进行supend，主要是为了让设备的状态达到休眠时的状态。

我们可以看到唤醒步骤中7）-11）与休眠步骤中1）-5）是对称的，这样能保证状态一致性。但是实际上

* + - 1. 设备休眠唤醒

每个支持设备挂起和/或恢复的设备驱动程序都需要定义 .suspend() 和 .resume() 等回调函数，并将它们注册到驱动程序模型中。接口如下：

struct dev\_pm\_ops {

...

/\* Suspend callbacks \*/

int (\*suspend)(struct device \*dev);

int (\*resume)(struct device \*dev);

/\* Hibernate callbacks \*/

int (\*freeze)(struct device \*dev);

int (\*thaw)(struct device \*dev);

int (\*poweroff)(struct device \*dev);

int (\*restore)(struct device \*dev);

...

}

有些设备不区分S3与S4，他们在注册回调时.suspend与.freeze会注册相同的函数。

#define SET\_SYSTEM\_SLEEP\_PM\_OPS(suspend\_fn, resume\_fn) \

.suspend = suspend\_fn, \

.resume = resume\_fn, \

.freeze = suspend\_fn, \

.thaw = resume\_fn, \

.poweroff = suspend\_fn, \

.restore = resume\_fn,

* + 1. ACPI

1. linux快速开机方案
   1. 基于linux现有唤醒方案

最简单的方案就是直接用休眠唤醒代替开机，也就是windows快速启动方案。用休眠代替开机的好处是可以减少systemd->lightdm->xorg->greeter阶段的流程。

具体步骤如下：

1. 在系统关闭前，将当前的桌面状态恢复到初始登录阶段，并且释放所有可释放的内存cache，这样做的目的是为了减少hibernation image的尺寸。
2. 执行休眠动作
3. 开机启动

整个流程与休眠基本一样，只不过是在休眠前尽量让使用内存最小。

note： 当前我们的唤醒是在initrd脚本中触发，当前的逻辑会等待amdgpu初始化完成之后再进行唤醒流程，这样会多进行一次设备初始化动作。但是实际上可以直接在initrd脚本之前就触发唤醒流程，通过在cmdline中增加resume=xx 来触发该流程，这样就可以 跳过initrd脚本，能有效优化唤醒流程时间。

* + 1. 测试

针对上面的方案我们在设备（E500，intel+amdgpu）进行了摸测，实际测试休眠唤醒方案从grub到桌面花费时间（20s）比正常开机启动时间（18s）要长。

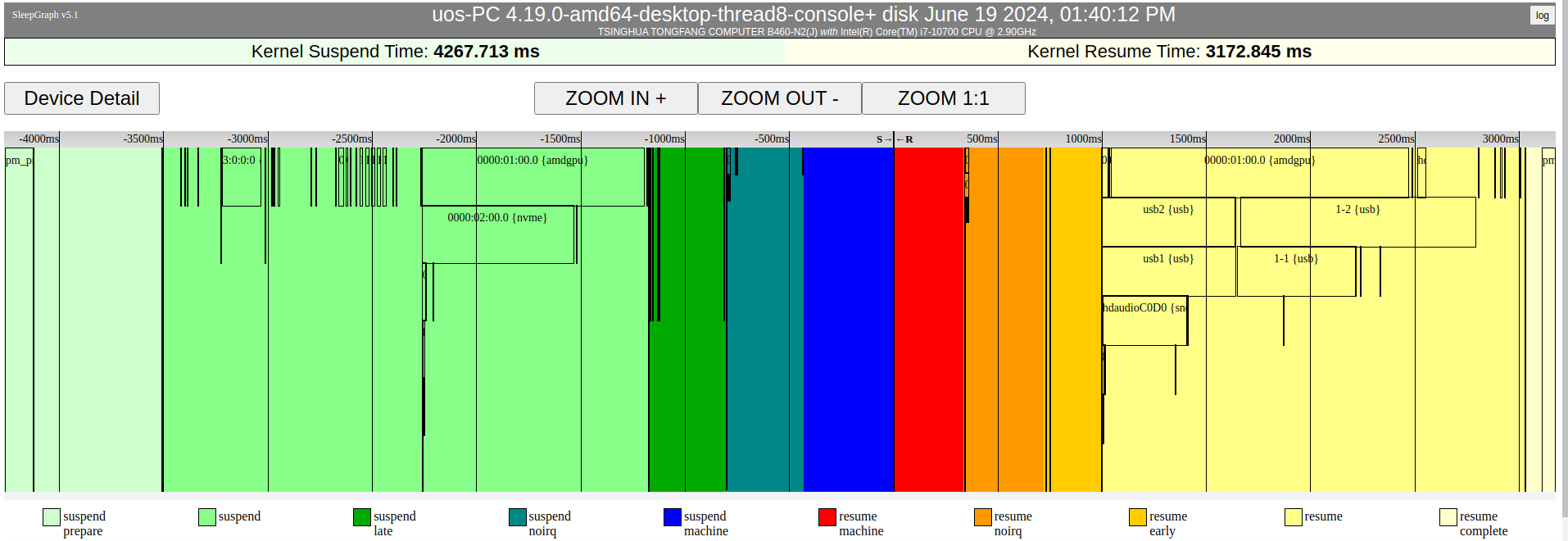
我们前面分析休眠流程可以不经过systemd->lightdm->xorg->greeter阶段，可以减少一些时间，但是Linux在中的唤醒过程中会增加其他步骤导致耗时增加，主要有两点：

1. hibernation image的解压加载耗时。
2. 唤醒流程中5）-10）的suspend与resume步骤。

所以如果休眠方案想要达到优化效果需要保证：load\_hibernation\_image+suspend+resume 的耗时要小于systemd+lightdm+xorg+greeter。

以下是这两时间的对比：

|  |  |
| --- | --- |
| initrd+systemd+lightdm+xorg+greeter | load\_hibernation\_image+suspend+resume |
| 3.6s | 2s+1s+2s |



休眠唤醒的耗时详细，amdgpu占了大部分

* + 1. 分析

从上面的测试结果上看，简单的使用休眠来代替开机并不能优化开机速度，主要原因是当前linux中的休眠流程比开机流程多了很多额外的步骤，这些步骤的耗时抵消了收益。

我们再回过头仔细看windows的快速开机方案，虽然windows方案的基本原理也是借助休眠来代替了开机。但是实际上windows的休眠唤醒流程比Linux的休眠唤醒流程更加简化。

windows的boot manager如果检测到了休眠镜像，识别出需要从休眠恢复，并指示系统恢复，恢复内存和所有体系结构寄存器的内容。

尽管设备驱动程序和服务会收到通知，但它们不会重新启动；它们会恢复到休眠阶段发生时的状态。不清楚windows的做法，按道理它也需要经过supend和resume流程才能保障驱动状态一致性，也有可能是windows的设备驱动在这方面有优化。

从测试结果上看，Linux中的唤醒主要耗时在image的加载，大概需要2s左右。还有就是gpu的suspend（500ms）和resume（1s）耗时比较大，而且优化难度比较大。而且实际上当前我们开机流程中用户态的耗时已经很小了，大概只有3-4s左右。

在某些特性的机型和场景下，如果用户态的初始化时间占比很大，那么当前的休眠方案可能会带来一些收益。实际效果还是要以实际测试为准。具体的评价方式参考4.2.3 分析休眠唤醒流程。

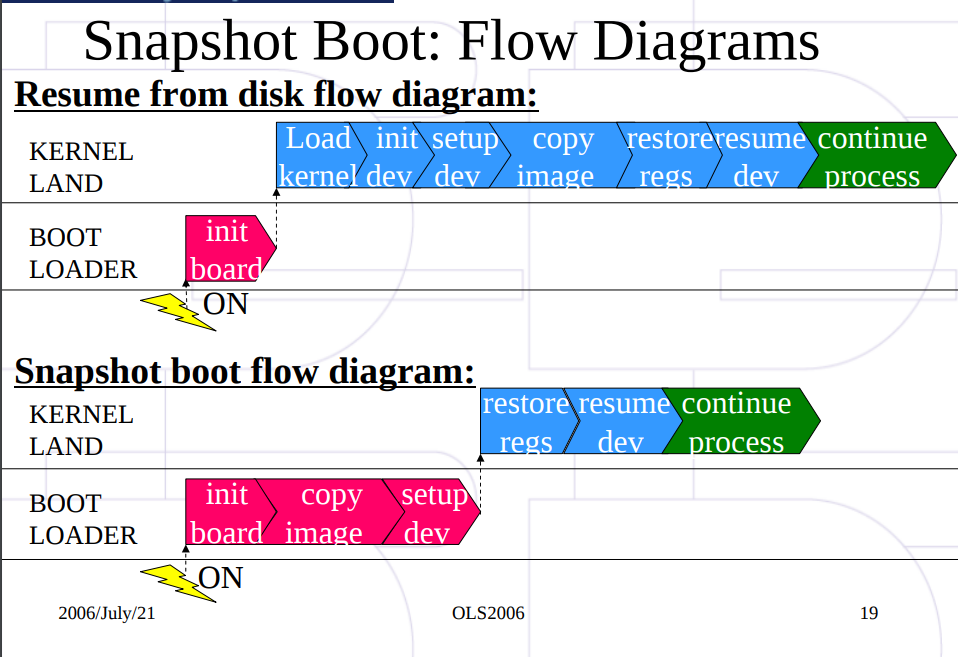
* + 1. snapshot boot

上游对于类似的方案讨论比较少，只有一个在2006年的OLS提出的snapshot boot[方案](https://landley.net/kdocs/ols/2006/ols2006v2-pages-25-34.pdf)，旨在通过休眠来降低启动时间。

snapshot boot主要是借助bootloader完成一部分工作，避免内核进行冗余初始化动作，以及减少不必要的设备初始化。

但是该方案可能需要扩展Bootloader功能，来代替kernel完成部分初始化动作。这个方案是否会导致bootloader功能复杂和一些兼容性问题，目前不得而知。

文章还提到，如果驱动能将设备的硬件初始化和相关的驱动数据初始化分开，那么就不需要bootloader来做设备初始化过程，而仅仅只是完成copy snapshot boot 和跳转到内核的resume点。



* + 1. Bootloader based hibernation

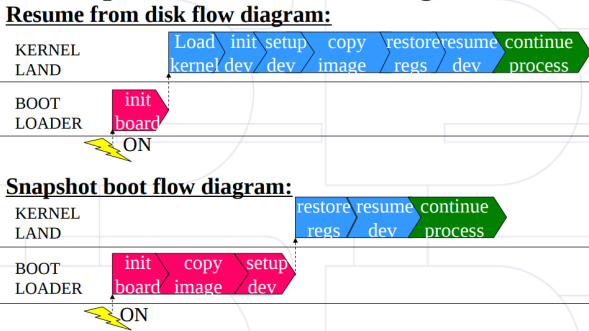
<https://lore.kernel.org/lkml/1652860121-24092-1-git-send-email-quic_vivekuma@quicinc.com/>

* + 1. Intel Rapid Start Technology
    2. 优化点

唤醒流程的可优化点：

1. 加载hibernation image耗时，与image大小相关
2. 唤醒需要走一遍完整的内核初始化，实际上有很多冗余步骤
3. 设备需要多次初始化，suspend，resume，冗余状态转换
4. image需要两次copy过程
   1. 优化方案fastboot

综合windows和上游一些实现，总结一下当前的休眠相关快速开机的可行方案：



fastboot

1. 不进行boot kernel，直接在bootloader中进行必要设备的初始化。然后加载hibernation image到RAM。
2. 在bootloader 设置MMU ，然后跳转到内核resume point中。
3. 为了简化bootloader的流程和兼容性，需要设备的restore实现能处理未初始化状态。为了与restore区分，可能需要增加一个新的fastboot状态。*如果无差异可不区分，直接使用restore状态。*（可能厂商驱动支持）
4. 关键驱动（GPU）的restore耗时优化。（厂商支持）
5. 关闭一些非boot相关程序，减小image的大小。

上述方案的关键点在于，少了唤醒过程中的freeze动作，需要对应的设备驱动支持直接从未初始化状态进行restore。

如果无法支持则需要在bootloader中针对这些设备进行初始化，甚至freeze操作。

* + 1. 方案诉求

1. 关键驱动，例如GPU驱动支持在设备未初始化状态下进行restore,为了与restore区分，可能需要增加一个新的fastboot状态，*如果无差异可不区分，直接使用restore状态。*。fastboot状态：与restore的区别，设备当前可能没有进行初始化或者状态未知。
2. 关键驱动GPU的fastboot或者(restore+freeze)的耗时优化，期望优化到500ms。
   * 1. 待补充

当前方案还需要补充：

1. 快速开机启动失败处理
2. 快速开机开关
3. image的更新机制
   * 1. 可行性分析
   1. EFIstub

EFISTUB是The EFI Boot Stub。原理是跳过传统的bootloader启动引导器加载内核阶段，直接加载内核。这个方案可以跳过GRUB加载流程，能有效的减少开机时间。

目前已经在W515（PanguV）、W585（PanguV战斗版）、L420（kelvinV）、L540（kelvinV战斗版）等机器上落地实现。

EFISTUB当前只支持在X86和ARM平台，内核打开配置CONFIG\_EFI\_STUB，然后通过efibootmng工具配置启动项目，配置完成后即可成功加载。

例子：

efibootmgr -c -d /dev/nvme0n1 -p 2 -L "UOSDIRECT BOOT 4.19" -l '\EFI\UOS\vmlinuz-4.19.0-amd64-desktop' -u "root=UUID=1491b637-4f94-444b-b3e1-8f12248440a3 ro checkreqprot=1 useclinux=1 video=efifb:nobgrt splash quiet ima\_appraise=off add\_efi\_memmap initrd=\EFI\UOS\initrd.img-4.19.0-amd64-desktop pcie\_aspm=off DEEPIN\_GFXMODE= modprobe.blacklist=i915 luks.crypttab=n libahci.ignore\_sss=1"

* + 1. 测试

在E500机器上进行了测试，使用EFISTUB方案比正常GRUB加载方案优化了2-3s左右，基本上就是GRUB的耗时。

* + 1. 总结

EFISTUB能有效减少开机时间，但是要商用落地要考虑很多异常回退的场景。具体可以参考华为的方案。

* 1. 一些零散的优化点
     1. initrd的异步unpacking

4.19内核中initramfs 的unpacking过程属于rootfs\_initcall，内核的初始化会阻塞在这个流程中。但是实际上后续的late\_initcall大多数不依赖于这个流程，所以可以将这个流程异步。

具体的[patch](https://github.com/torvalds/linux/commit/e7cb072eb988e46295512617c39d004f9e1c26f8)。可优化100ms-200ms左右时间，取决于具体的机型。

* + 1. initramfs-tools等待耗时

initrd在挂载真实的根文件系统之前，会检查块设备是否ready，local\_mount\_root->local\_device\_setup。在local\_device\_setup中会存在一个sleep 1s的等待，也就是每1s会检查一下设备状态。这个检查的粒度太长，最大会导致耗时增加1s。而且这个时间明显可以减小。

实际上我们在/usr/share/initramfs-tools/scripts/init-top/udev中已经增加了等待blk设备的流程，rootdev="$(blkid --uuid "${root\_uuid}")"。

但是这个新增的代码实际上没有生效，因为udev脚本中设置了#!/bin/sh -e，这样就会导致在rootdev="$(blkid --uuid "${root\_uuid}")"没有查询到blk设备时脚本直接退出了。

#!/bin/sh -e # 脚本运行时，只要命令返回值是非0，则脚本会自动退出，不需要去加判断然后退出。

PREREQS=""

prereqs() { echo "$PREREQS"; }

case "$1" in

    prereqs)

    prereqs

    exit 0

    ;;

esac

if [ -w /sys/kernel/uevent\_helper ]; then

        echo > /sys/kernel/uevent\_helper

fi

if [ "${quiet:-n}" = "y" ]; then

        log\_level=notice

else

        log\_level=info

fi

root\_uuid=${ROOT#UUID=}

SYSTEMD\_LOG\_LEVEL=$log\_level /lib/systemd/systemd-udevd --daemon --resolve-names=never

udevadm trigger --type=subsystems --action=add

udevadm trigger --type=devices --action=add

for i in $( seq 1 25 )

do

        rootdev="$(blkid --uuid "${root\_uuid}")" # 查询blk设备

        if [ -n "${rootdev}" ];then

                break

        fi

                sleep 0.2

done

*# Leave udev running to process events that come in out-of-band (like USB*

*# connections)*

* + 1. 模块buildin

有一些常用的模块可以直接buildin到内核中，理论上避免加载和模块查找的开销。

例子：initrd在monut 根文件系统的时候，显示mount流程耗时很长。通过增加维测确认monut耗时长是因为在第一次mount时候，会根据fs文件系统的类型加载对应的ext4模块。

do\_mount

└── do\_new\_mount

└── get\_fs\_type

└── request\_module // 查找ext4模块并加载。

我们将ext4修改为buildin，在初始化时直接加载就可以避免一些查找动作，而且由于初始化时尽量进行并行化加载，可以避免启动流程阻塞在mount中。

* + 1. xorg加载优化

xorg初始化时会根据驱动配置来加载一些驱动，也就是会通过dlopen加载一些.so文件。这样就会产生IO读操作，如果这个时候同时存在其他的任务进行IO读操作，可能就会当时dlopen的耗时变长。

我们可以通过systemd-bootchart工具来观察初始化阶段CPU、IO资源的使用情况。通过对比可以初步筛选出占用IO较大的服务。具体方法详见4.2 systemd-bootchart。

找到IO占用较大的服务后，将其停止或者增加合理的依赖关系，避免与xorg同时加载即可。

* + 1. hibernation image加载优化

HIBERNATION当前使用的是LZO方式，并且支持多线程解压。目前默认最大线程数是3，通过LZO\_THREADS变量控制。

我们可以通过修改LZO\_THREADS来提升解压线程数，在E500机器下，将LZO\_THREADS设置为8，可以提升300ms左右的时间。

1. 启动耗时分析工具与方法
   1. 上游一些优化手段

参考：<https://elinux.org/Boot_Time>

https://tinylab.org/elinux-org-boot-time-optimization/

* 1. pm-graph

pm-graph 工具是intel公司针对linux系统的suspend, resume, and boot流程推出的一个可视化分析工具。

- 使用户能够快速识别这些流程中的低效率和瓶颈。

工具的原理通过获取ftrace和dmesg的信息抓取相关流程的日志。

* + 1. 工具获取

$> sudo apt-get install python-configparser python-requests linux-tools-common

$> git clone http://github.com/intel/pm-graph.git

$> cd pm-graph

$> sudo make install

$> man sleepgraph

依赖:

CONFIG\_PM=y

CONFIG\_PM\_DEBUG=y

CONFIG\_PM\_SLEEP\_DEBUG=y

CONFIG\_FTRACE=y

CONFIG\_FUNCTION\_TRACER=y

CONFIG\_FUNCTION\_GRAPH\_TRACER=y

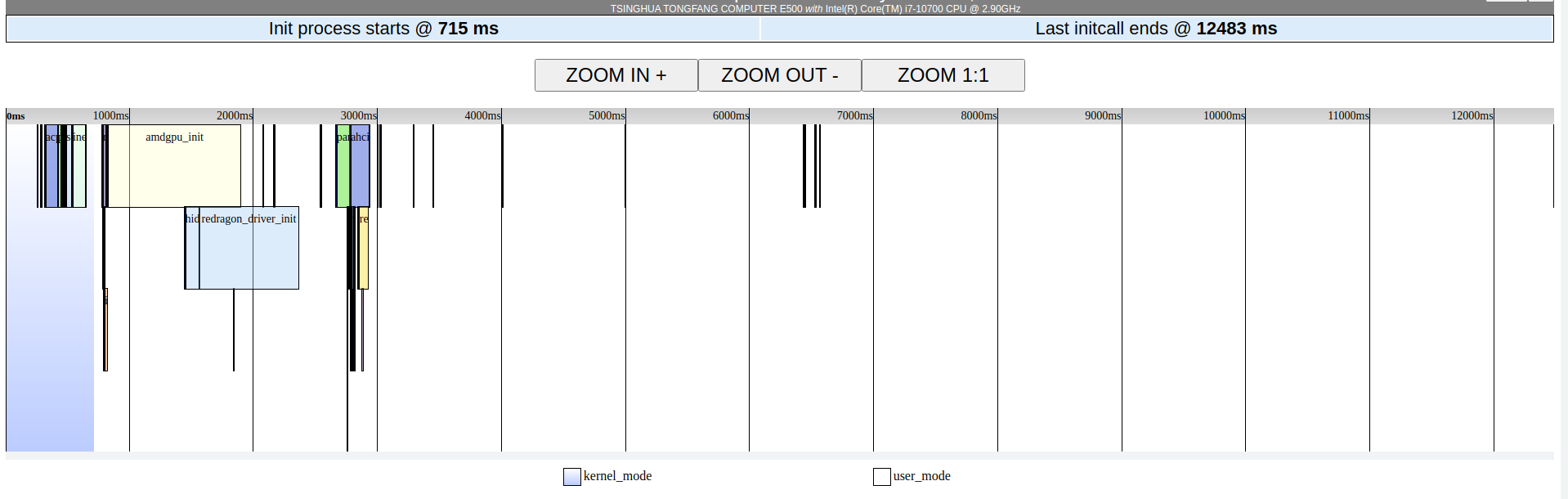
CONFIG\_KPROBES=y

CONFIG\_KPROBES\_ON\_FTRACE=y

CONFIGURING KERNEL COMMAND LINE OPTIONS

* + 1. 分析启动过程

sudo bootgraph -reboot -f



* + 1. 快速验证方法

在一些极限的非商用场景中可以关闭一些不影响开机的启动项目，首先通过上一个步骤中的图获取关键初始化函数，然后在cmdline中增加initcall\_blacklist=xxx，即可跳过这个初始化。

* + 1. 分析休眠唤醒流程

修改suspend-callgraph.cfg文件，mem 为disk。

执行下面命令会自动触发休眠唤醒流程。

sudo ./sleepgraph.py -config config/suspend-callgraph.cfg

Checking this system (uos-PC)...

    have root access: YES

    is sysfs mounted: YES

    is "disk" a valid power mode: YES

    is ftrace supported: YES

    are kprobes supported: YES

    timeline data source: FTRACE (all trace events found)

    is rtcwake supported: YES

baseboard-manufacturer  : TSINGHUA TONGFANG COMPUTER

baseboard-product-name  : B460-N2(J)

baseboard-version       : V1.1

bios-release-date       : 09/19/2020

bios-vendor             : American Megatrends Inc.

bios-version            : J006NJ0

chassis-manufacturer    : TSINGHUA TONGFANG COMPUTER

processor-manufacturer  : Intel(R) Corporation

processor-version       : Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz

system-manufacturer     : TSINGHUA TONGFANG COMPUTER

system-product-name     : E500

system-serial-number    : 2201280002097200586

cpucount                : 16

memtotal                : 16237644 kB

memfree                 : 14908196 kB

INITIALIZING FTRACE...

Setting trace buffers to 3145728 kB (196608 kB per cpu)

START TRACING

SUSPEND START

will issue an rtcwake in 15 seconds

RESUME COMPLETE

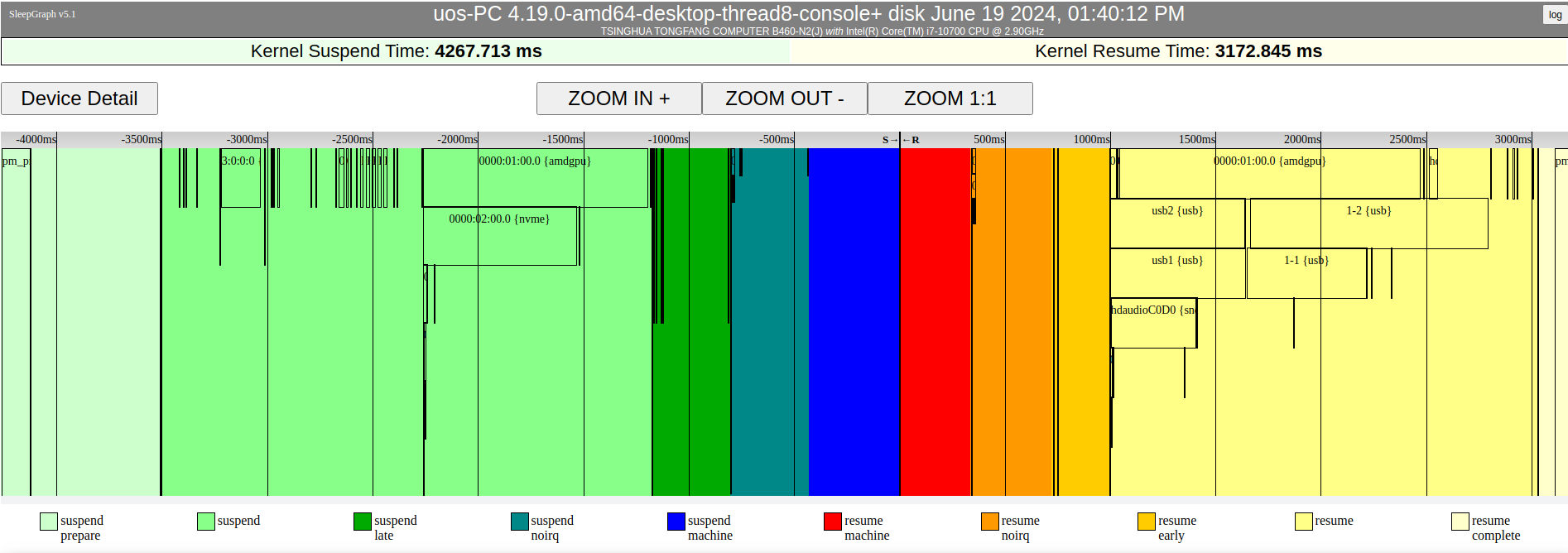
CAPTURING TRACE

CAPTURING DMESG

PROCESSING DATA

DONE

重启完成后查看日志，可以看到suspend和resume的耗时，以及具体哪个阶段的耗时过大。



点击log，我们可以看到如下对于每个阶段耗时的统计。

Analyzing the ftrace data (uos-PC\_disk\_ftrace.txt)...

Callgraph found for task 5294: 133.378ms, pm\_prepare\_console

Callgraph found for task 5294: 0.310ms, \_\_pm\_notifier\_call\_chain

Callgraph found for task 5294: 61.026ms, pm\_restore\_console

Command:

sleepgraph.py -config config/disk\_suspend-callgraph.cfg

Timeline Details:

test start: 7597.033601

kernel suspend start: 7597.033601

suspend\_prepare: 7597.033601 - 7597.787979 (850 devices)

suspend: 7597.787979 - 7600.120675 (98 devices)

suspend\_late: 7600.120675 - 7600.492957 (22 devices)

suspend\_noirq: 7600.492957 - 7600.867738 (22 devices)

suspend\_machine: 7600.867738 - 7601.301314 (0 devices)

resume\_machine: 7601.301314 - 7601.635852 (0 devices)

resume\_noirq: 7601.635852 - 7602.019301 (22 devices)

resume\_early: 7602.019301 - 7602.292042 (3 devices)

resume: 7602.292042 - 7604.323838 (95 devices)

resume\_complete: 7604.323838 - 7604.474159 (826 devices)

kernel resume end: 7604.474159

test end: 7604.474159

Creating the html timeline (suspend-uos-PC-240619-133955-cg/uos-PC\_disk.html)...

上面的图中显示的是一次suspend和一次resume流程，但是实际上 std的休眠唤醒流程涉及到了多次suspend和resume。那上面的日志的时间suspend和resume又是如何对应的呢？

* + - 1. 休眠

我们先看一下休眠流程（下面只是选取了几个比较耗时的流程，不代表全部流程）

1. 申请 hibernation image需要的内存 = 》 suspend\_prepare

2. 冻结任务 = 》 suspend\_prepare

3. Suspend设备 = 》 suspend suspend\_late suspend\_noirq

4. 非BOOT CPU下线 = 》 suspend\_machine

5. 保存image

6. resume 部分设备

7. CPU上线

8. 唤醒设备

9. 将image写入磁盘

10. 关机

从上面的流程休眠之前会进行一次suspend 然后又会进行一次resume。 在suspend之后会保存内存的现场，而后面又进行resume的原因是需要将image写入磁盘，所以需要相关磁盘设备能正常工作。

实际上pm-graph的流程中分析的suspend的耗时就是这次suspend的过程。但是这里的resume并不是。

* + - 1. 唤醒

我们再分析唤醒流程：

1. 系统正常初始化，此时设备都初始化完成，和一次正常的开机一样。

2. 检查是否存在hibernation image，加载hibernation 进行休眠唤醒

3. 设备suspend

4. 非BOOT CPU下线

5. 加载image到内存，跳转到image对应的内核中。

6. CPU上线 =》 resume\_machine

7. 设备resume =》 resume\_noirq resume\_early resume

STD的唤醒流程会经过一个正常的初始化，然后加载hibernation image ，并将设b备suspend。 最后再次resume设备。所以上面pm-graph抓取的就是这次resume流程。

* + - 1. 如何分析整个唤醒耗时

我们从pm-graph只能获取resume的耗时，无法获取初始化耗时、suspend等其他过程耗时。

实际上休眠时候的suspend与唤醒时的suspend很多步骤都是相同的，我们可以通过分析休眠时的suspend时间来评估唤醒时的suspend时间。

suspend + suspend\_late + suspend\_noirq + suspend\_machine

我们假设唤醒的几个过程分别如下：

内核初始化： kernel\_init

加载image： load\_hibernation\_image

设备suspend：

suspend + suspend\_late + suspend\_noirq + suspend\_machine （这次suspend\_prepare的流程与休眠时的不一样，并且时间很短可以忽略）

设备resume：

resume\_machine + resume\_noirq + resume\_early + resume

那么总时间就是：

kernel\_init + load\_hibernation\_image + suspend + suspend\_late + suspend\_noirq + suspend\_machine + resume\_machine + resume\_noirq + resume\_early + resume。

上面这些只有kernel\_init + load\_hibernation\_image 的耗时还不知道。kernel\_init的耗时我们可以使用sudo bootgraph -reboot -f获取。唤醒时的kernel\_init要一直到amdgpu初始化完成后再加载hibernation image，所以kernel\_init的时间要统计到amdgpu完成的时间。

而load\_hibernation\_image 的耗时需要在串口日志中查看，如下：

   12.705598] PM: Basic memory bitmaps created

[   12.891516] PM: Using 8 thread(s) for decompression

[   12.896417] PM: Loading and decompressing image data (1234173 pages)...

[   12.927164] PM: Image loading progress:   0%

[   13.971981] PM: Image loading progress:  10%

[   14.458278] PM: Image loading progress:  20%

[   15.200351] PM: Image loading progress:  30%

[   15.665689] PM: Image loading progress:  40%

[   16.295272] PM: Image loading progress:  50%

[   16.783812] PM: Image loading progress:  60%

[   17.244957] PM: Image loading progress:  70%

[   17.716996] PM: Image loading progress:  80%

[   18.197732] PM: Image loading progress:  90%

[   18.651664] PM: Image loading progress: 100%

[   18.656228] PM: Image loading done

[   18.659774] PM: Read 4936692 kbytes in 5.75 seconds (858.55 MB/s)

[   18.667239] PM: Image successfully loaded

* + - 1. 总结

总的来说想要测量唤醒的耗时需要综合好几个日志，并且结果也并不是一个非常准确的值。但是通过上面的流程可以获取一个基本的耗时分布，我们可以通过这个分布来确认瓶颈进行有针对性的优化。

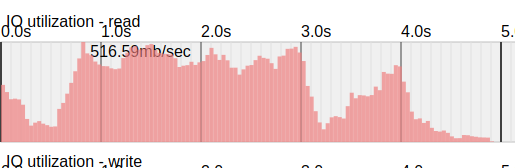
* 1. systemd-bootchart

用于生成系统启动过程的性能图表，帮助分析和优化启动性能，主要是系统资源利用率的角度来分析。

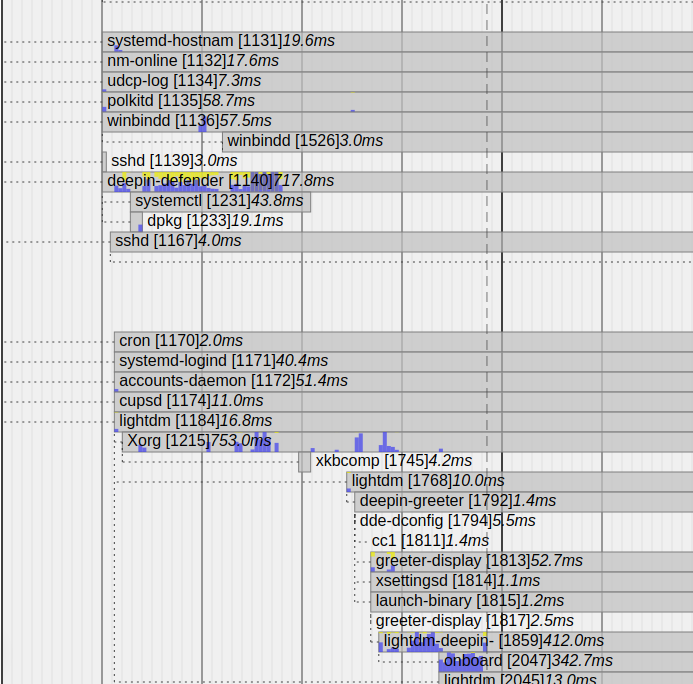
1. sudo apt install systemd-bootchart
2. systemctl status systemd-bootchart.service
3. reboot
4. ls /run/log/bootchart-20240726-0929.svg

例子：

观测资源使用情况，可以看到IO的读取在1-3s左右使用率很高，这个时候正好的xorg的初始化阶段，xorg在初始化时会读取大量的.so，而此时的IO使用率可能会导致xorg加载耗时增加。



我们可以查看在1-3s左右比较活跃的任务，比如下图中，deepin-defender-monitornetflow.service任务比较活跃，并且有一些cpu wait的情况，猜测这个服务可能有大量的io操作，我们可以通过查看/pro/1140/io确认。



找到异常的服务之后就可以评估这个服务的启动时间是否合适，通过关闭和增加依赖关系来将服务延后到开机启动之后。

* 1. systemd-analyze

systemd-analyze用于分析和调试 systemd 系统和服务管理器。它提供了一些命令和选项来帮助用户了解系统启动过程、服务依赖关系以及性能瓶颈。

1. 启动时间分析

显示系统启动过程中各个阶段所花费的时间，包括固件、引导加载器、内核和用户空间的时间。

systemd-analyze

2. 启动时间图示

生成一个图示，显示各个服务启动的时间顺序和耗时。

systemd-analyze plot > bootup.svg

1. 关键路径分析

列出启动过程中的关键路径，显示哪些服务在启动过程中占用了较多时间。

systemd-analyze critical-chain

1. 单个服务的依赖关系图

生成一个服务的依赖关系图，可以用 dot 工具进行可视化

systemd-analyze dot | dot -Tsvg > dependencies.svg

* + 1. 优化方法

systemd-analyze critical-chain lightdm.service

lightdm.service +145ms

└─systemd-user-sessions.service @1.084s +31ms

└─network.target @1.083s

└─NetworkManager.service @724ms +346ms

└─dbus.service @720ms

└─basic.target @706ms

└─sockets.target @706ms

└─virtlockd.socket @706ms

└─sysinit.target @705ms

└─sys-fs-fuse-connections.mount @1.082s +33ms

└─systemd-modules-load.service @133ms +567ms

└─systemd-journald.socket @130ms

└─system.slice @129ms

└─-.slice @129ms

结合bootup.svg图中纵坐标的关系，找一些可疑的服务，尝试关闭。

* 1. initramfs-tool
     1. 调试

打开日志：cmdline中添加 debug

日志位置：/run/initramfs/initramfs.debug

增加时间戳的方法：

vim /usr/share/initramfs-tools/scripts/functions

# 修改

\_log\_msg() {

    if [ "${quiet?}" = "y" ]; then return; fi

*# 获取当前时间戳，精确到毫秒*

    timestamp=$(cat /proc/uptime)

*# 打印带有时间戳的消息*

*# shellcheck disable=SC2059*

    printf "[%s] " "$timestamp"

    printf "$@"

}

# 重新生成inintrd

sudo update-initramfs -k 4.19.0-amd64-desktop -u

* + 1. 裁剪

update-initramfs 主要是将 `/lib/modules/xx` 中的相关驱动文件，源码目录 `/usr/share/initramfs-tools`，配置目录 `/etc/initramfs-tools`，根据一定的要求进行打包，最后生成了对应的 `initrd.img-xxxx` 文件。

裁剪主要就是裁剪不需要的/lib/modules/下的.ko。

1. 查看开机需要哪些必须的modules

`lsmod` 中就是开机自动加载的mod，也就是开机阶段可能需要的。

通过脚本将除了lsmod中的其他所有.ko文件全部删除。（删除前先备份/lib/modules/xxxx ）

#!/bin/bash

*# 指定内核模块目录*

MODULE\_DIR="/lib/modules/4.19.0-loongson-3-desktop"

*# 获取当前加载的模块名并存入数组*

loaded\_modules=($(lsmod | awk '{print $1}' | tail -n +2))

*# 查找所有 .ko 文件，包括子目录*

find "$MODULE\_DIR" -name '\*.ko' | while read -r ko\_file; do

*# 提取模块名*

module\_name="${ko\_file##\*/}"

module\_name="${module\_name%.ko}"

*# 检查模块是否在已加载模块中*

if [[ ! " ${loaded\_modules[@]} " =~ " ${module\_name} " ]]; then

echo "Deleting: $ko\_file"

rm "$ko\_file"

fidone

1. 更新initramfs

sudo update-initramfs -k 4.19.0-loongson-3-desktop -u

1. 查看/boot 目录下initramfs文件大小是否优化。
2. 总结
3. 缩写

BIOS：Basic I/O System

POST: Power On Self Test 自检

1. 参考资料

<https://learn.microsoft.com/en-us/windows-hardware/test/weg/delivering-a-great-startup-and-shutdown-experience>

<https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/distinguishing-fast-startup-from-wake-from-hibernation>

https://learn.microsoft.com/zh-cn/archive/blogs/b8/delivering-fast-boot-times-in-windows-8

<https://tinylab.org/elinux-org-boot-time-optimization/>

1. Joe I , Lee S C .Bootup time improvement for embedded Linux using snapshot images created on boot time[C]//International Conference on Next Generation Information Technology.IEEE, 2011.DOI:doi:http://dx.doi.org/.

<https://elinux.org/Boot_Time>

<https://wiki.deepin.org/zh/04_%E5%B8%B8%E8%A7%81%E9%97%AE%E9%A2%98FAQ/DDE/%E6%B5%85%E8%B0%88DDE%E6%A1%8C%E9%9D%A2%E8%BF%90%E8%A1%8C%E6%9C%BA%E5%88%B6>

https://elinux.org/Boot\_Time