Kexec Evolutions for LinuxBoot

A series of improvements to userspace kexec in LinuxBoot

Self link: bit.ly/3By04Aa

SPEAKER

David Hu — SWE, Google

Menu

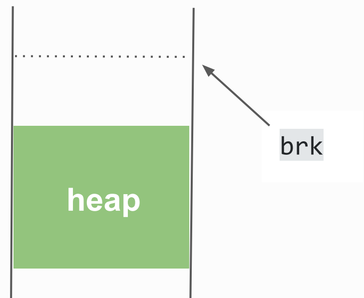
- Introduction
- Getting started
- Classic kexec load Arm64 Image
- Kexec workstream
- Q & A

Introduction

- Recap
  - LinuxBoot relies on kexec sys call to load into next kernel to function as a bootloader
  - 2 kexec syscalls exist today *link*
    - file load, `kernel_fd, initrd_fd, *cmdline, flags` ✅
    - classic kexec load, `entry, nr_segments, segments, flags`
- Problems with file load
  - File load can spike memory usage, though only transitory
    - e.g. need > 3 * N ram, given target image size as N.
  - Can't edit DTB

Getting started

- Problem: netbooting 1.1G image on machine w/ 4G ram would OOM
- Pre-kexec culprits
  - *CachedReader* caches image as it reads, leading to an additional copy lingering around 😠
  - io.ReadAll triggers exponential slice re-allocations 😠
    - Golang runtime also reserves additional memory from OS to enable heap growth
  - Make a read-only copy of kernel+initrd before kexec 😠



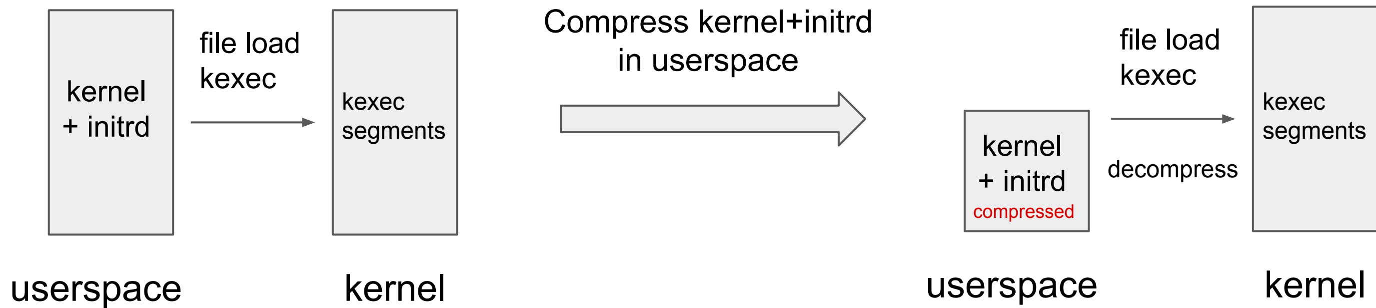
Getting started: Compress kernel and initrd

- Now we only have one copy of kernel and initrd in **userspace** tmpfs before kexec

- But...there is a catch: Kernel (*file load kexec syscall*) would make another copy to begin with further processing 💔

Getting started: Compress kernel and initrd

- In kernel code, file load kexec syscall reads kernel and initrd as a whole, leading to second copy. (Used to prepare for `kexec_segment` for further execution)
- One possible optimization is to compress initrd and kernel before kexec

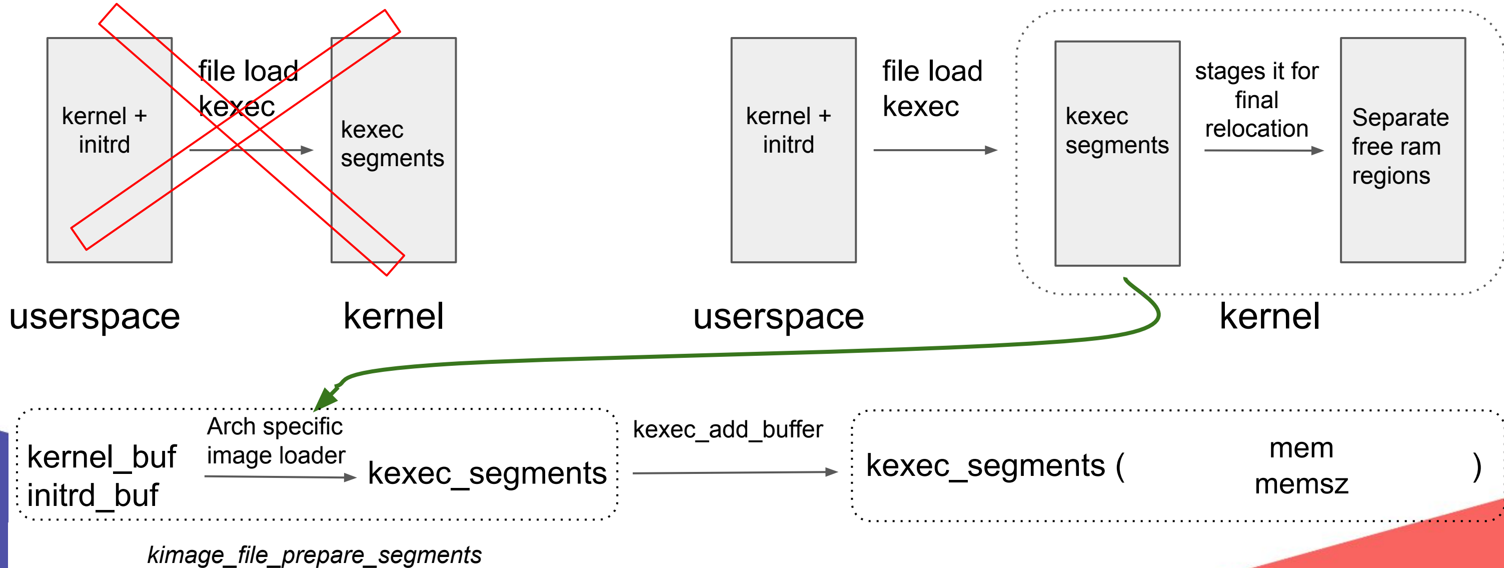


- It adds on X mins in compression to boot time (e.g. gzip 1.1G image can take 5mins). 😡😡😡😡😡

Getting started: Additional copy in kernel space

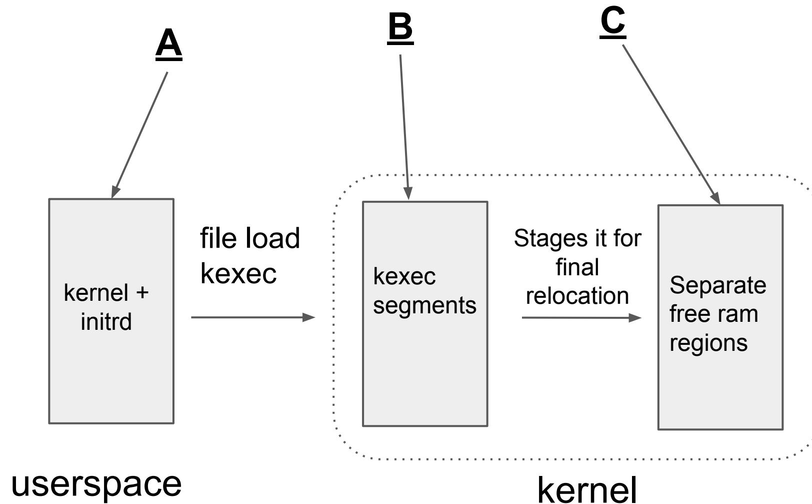
- A deeper look: how file load kexec processes kernel and initrd files.

kimage_file_prepare_segments(*struct kimage *image, int* ***kernel_fd****, int* ***initrd_fd****, …)* *(linux/kernel/kexec_file.c)*



Getting started: Additional copy in kernel space

- Can we eliminate one more copy ? and which one ?



Classic load Arm64: The contract

Boot loader should provide (as a minimum) the following:

- Setup and initialise the RAM
- Setup the device tree
- Decompress the kernel image
- Call the kernel image

Before calling the kernel image

- Primary CPU general-purpose register settings
- ...

more: kernel.org/doc/Documentation/arm64/booting.txt

Classic load Arm64: Implementation

(In LinuxBoot userspace, implement following in **golang**)

- Process Image, Initrd and kernel cmdline into kexec segments
  - Parse memory layout
- Setup device tree
  - Use FDT in from sysfs to begin with (`LoadFDT(dtb io.ReaderAt)`)
  - Purge existing boot param properties from chosen node (`sanitizeFDT(fdt *dt.FDT)`)
  - Add initramfs location

Classic load Arm64: Implementation

- Set up an executable trampoline with instructions to
  - Save kernel entry to a general purpose register, which we can jump / branch into
  - Save dtb address to x0
  - Zero out x1, x2, and x3
- Then make the syscall
  - ```
    long syscall(SYS_kexec_load, unsigned long entry, unsigned long nr_segments, struct kexec_segment *segments, unsigned long flags);
    ```

    ```

# Classic load Arm64: Trampoline

- Golang assembly, mimicking what kexec-tools does ?
- David Dillow at Google, came up with a simple and minimal *trampoline w/o needing to write any explicit assembly code* 🤔

| Kernel Kexec Userspace ( C ) | LinuxBoot (golang) |
|---|---|
| • SHA256 verifications<br>• Load kernel entry, and dtb address by symbols in assembly<br>• *ldr x17, arm64_kernel_entry*<br>• *ldr x0, arm64_dtb_addr*<br><br>https://github.com/horms/kexec-tools/blob/main/purgatory/arch/arm64/entry.S | • Kernel and dtb addresses are placed at a PC relative memory location (**fixed**), which are then loaded into respective registers by PC relative instructions (LDR)<br><br>*LDR (PC-relative)*<br>*Load register. The address is an offset from the PC*<br><br>https://github.com/u-root/u-root/blob/main/pkg/boot/linux/load_linux_arm64.go#L189 |

# Classic load Arm64: Trampoline

Imaginary 4 bytes block

instructions

data

load kernel entry

load dtb address

zero out x1, 2, 3

branch into kernel entry

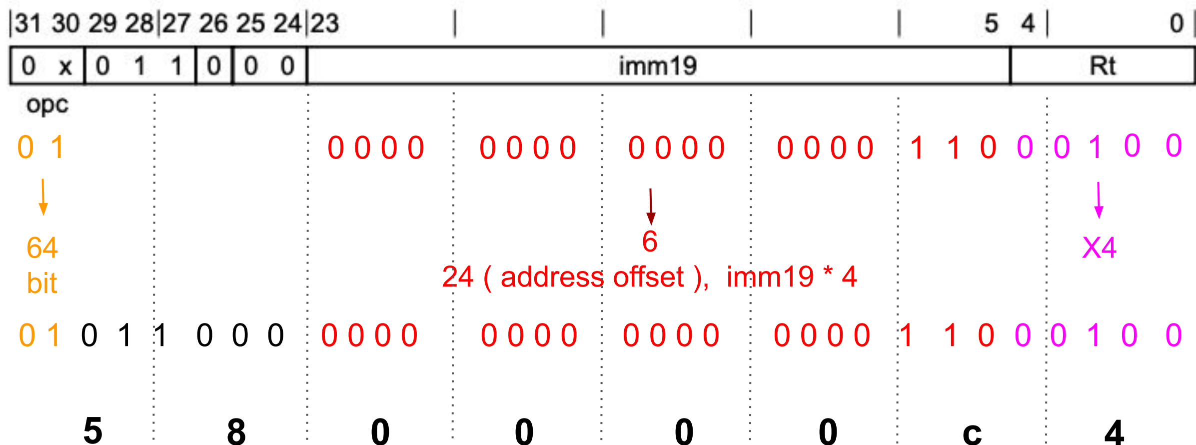
kernel address

dtb address

# Classic load Arm64: load kernel entry

- `0x580000c4 // ldr x4, #0x18 (PC relative: trampoline[6 and 7])`

*(Armv8), C6.2.131, LDR (literal):*



# Classic load Arm64: load dtb address

- `0x580000e0 // ldr x0, #0x1c (PC relative: trampoline[8 and 9])`

*(Armv8), C6.2.131, LDR (literal):*

|31 30 29 28|27 26 25 24|23 | | | | 5 4| 0|

| 0 x | 0 1 1 | 0 | 0 0 | imm19 | Rt |

opc

0 1 | 0000 0000 0000 0000 111 | 00100

64 bit

7

28 ( address offset ), imm19 * 4

X0

0 1 0 1 1 0 0 0 0000 0000 0000 0000 1 1 1 0 0 0 0 0

5 8 0 0 0 0 e 0

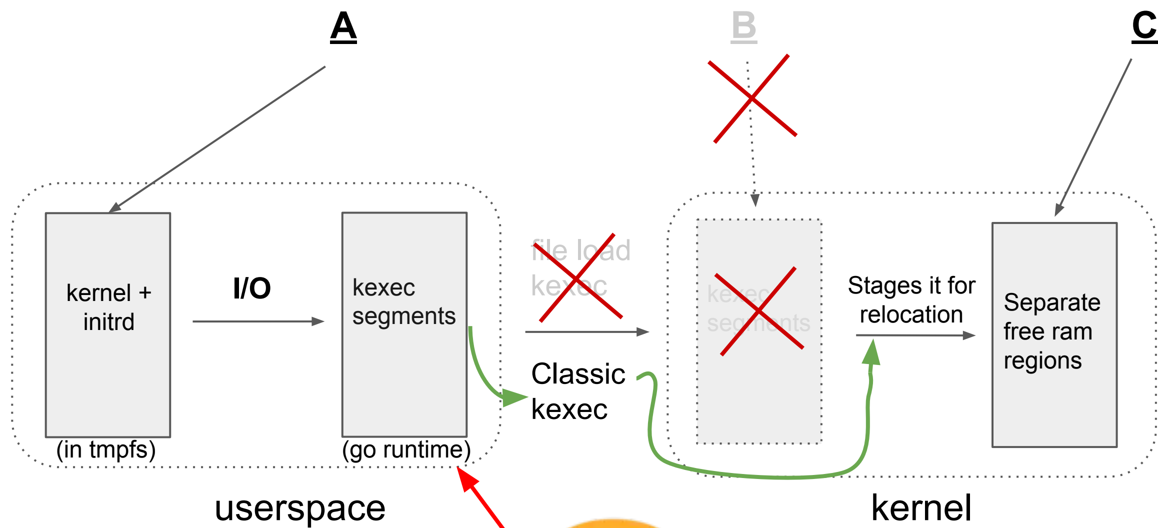
# Classic load Arm64: Trampoline

- "Zero-Assembly" trampoline is position independent (PIC)

```
L1   var trampoline [10]uint32
L2   trampoline[0] = 0x580000c4 // ldr x4, #0x18 (PC relative: trampoline[6 and 7])
L3   trampoline[1] = 0x580000e0 // ldr x0, #0x1c (PC relative: trampoline[8 and 9])
L4   // Zero out x1, x2, x3
L5   trampoline[2] = 0xaa1f03e1 // mov x1, xzr
L6   trampoline[3] = 0xaa1f03e2 // mov x2, xzr
L7   trampoline[4] = 0xaa1f03e3 // mov x3, xzr
L8   // Branch register / Jump to instruction from x4.
L9   trampoline[5] = 0xd61f0080 // br x4
L10 trampoline[6] = uint32(uint64(kernelEntry) & 0xffffffff)
L11 trampoline[7] = uint32(uint64(kernelEntry) >> 32)
L12 trampoline[8] = uint32(uint64(dtbBase) & 0xffffffff)
L13 trampoline[9] = uint32(uint64(dtbBase) >> 32)
```

- https://github.com/u-root/u-root/blob/main/pkg/boot/linux/load_linux_arm64.go#L189

Classic load Arm64: Outcome



Classic load Arm64: One more thing

A

kernel +
initrd

(files in tmpfs)

I/O

kexec
segments

(go runtime)

userspace

MMU

…

…

Physical page frames

mmap instead of
I/O

A

mmap

kernel +
initrd

kexec
segments

(files in tmpfs)

(go runtime)

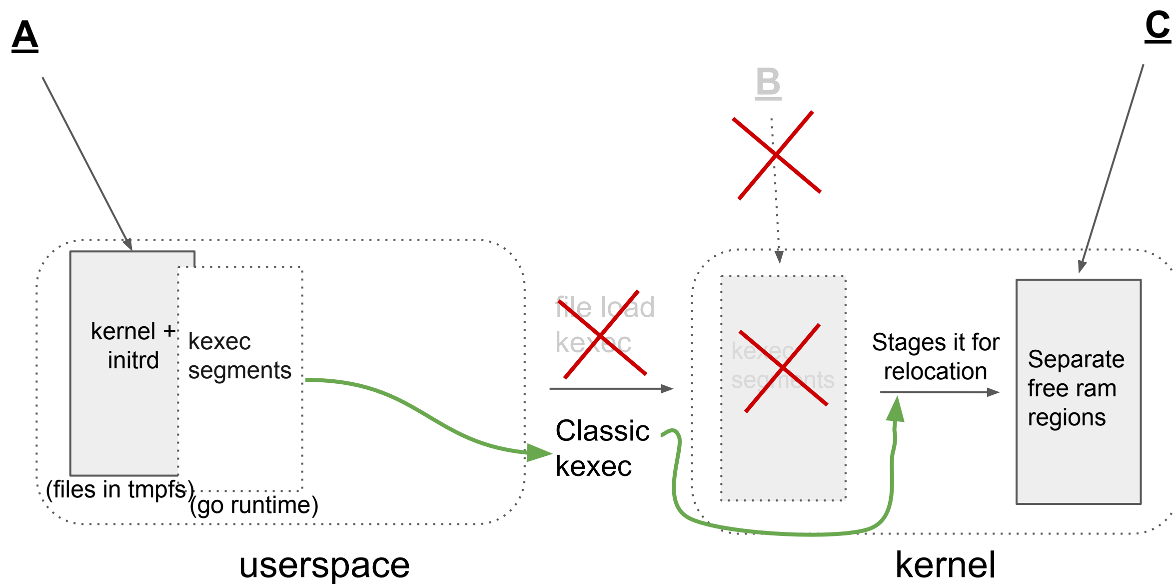
userspace

MMU

…

Physical page frames

Classic load Arm64: Final outcome



Question to the audience: can we do even better ?

Call for action: Kexec workstream

- Open Source Firmware Foundation kexec workstream
- Get involved
  - Share your problems
  - Try out fixes by others
  - Contribute

Questions?