Boot time optimization

Latest update: August 18, 2020.

Document updates and sources:
https://bootlin.com/doc/training/boot-time

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

Rights to copy

Document sources: https://github.com/bootlin/training-materials/

Hyperlinks in the document

There are many hyperlinks in the document

- Regular hyperlinks:
  `https://kernel.org/`
- Kernel documentation links:
  `dev-tools/kasan`
- Links to kernel source files and directories:
  `drivers/input/`
  `include/linux/fb.h`
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  `platform_get_irq()`
  `GFP_KERNEL`
  `struct file_operations`

bootlin

- Engineering company created in 2004, named "Free Electrons" until Feb. 2018.
- Locations: Orange, Toulouse, Lyon (France)
- Serving customers all around the world
- Head count: 12 - Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel, build systems and low level Free and Open Source Software for embedded and real-time systems.
- Bootlin is often in the top 20 companies contributing to the Linux kernel.
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

Bootlin on-line resources

- All our training materials and technical presentations: https://bootlin.com/docs/
- Technical blog: https://bootlin.com/
- Quick news (Mastodon): https://fosstodon.org/@bootlin
- Quick news (Twitter): https://twitter.com/bootlincom
- Quick news (LinkedIn): https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line: https://elixir.bootlin.com

Mastodon is a free and decentralized social network created in the best interests of its users.

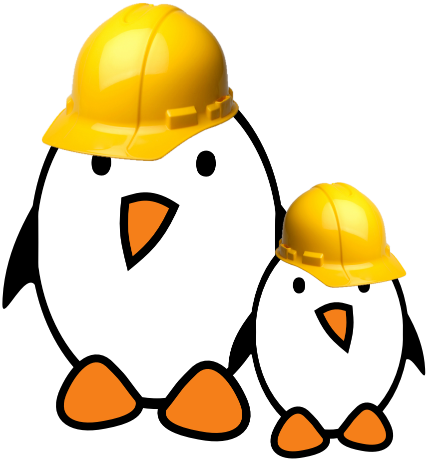
Image credits: Jin Nguyen - https://frama.link/bQwcWHTP

Thanks

Special thanks to

- Zuehlke Engineering (Serbia)
  - For funding a major update to these materials and further development (2 days now)
- Microchip (formerly Atmel Corporation)
  - For funding the development of the first version of these materials (1 day course)

Practical lab - Lab setup and downloading sources



Prepare your lab environment

- Download and extract the lab archive

Start cloning source trees right away

- U-Boot
- Linux kernel
- Buildroot

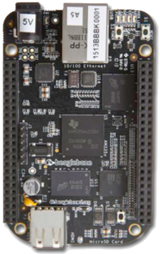
Generic course information

Supported hardware

BeagleBone Black or BeagleBone Black Wireless, from BeagleBoard.org

- Texas Instruments AM335x (ARM Cortex-A8 CPU)
- SoC with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- 512 MB of RAM
- 4 GB of on-board eMMC storage
- USB host and USB device, microSD, micro HDMI
- WiFi and Bluetooth (wireless version), otherwise Ethernet
- 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- A huge number of expansion boards, called *capes*. See `https://elinux.org/Beagleboard:BeagleBone_Capes`.

beagleboard.org



open source hardware

The full system

- Beagle Bone Black board (of course). The Wireless variant should work fine too.
- Beagle Bone LCD cape `https://elinux.org/Beagleboard:BeagleBone_LCD4` Unfortunately, no longer manufactured. You could adapt the course to use an HDMI display instead.
- Standard USB webcam (supported through the `uvcvideo` driver.

Course outline - Day 1

Morning

- Lecture: principles
- Lab: compiling the bootloader, kernel and root filesystem
- Lab: setting up the system, customizing the build system
- Lecture: measuring time, software and hardware methods

Afternoon

- Lab: measuring time, software and hardware methods
- Lecture and lab: finding the optimum toolchain
- Lecture and lab: optimizing the application

Course outline - Day 2

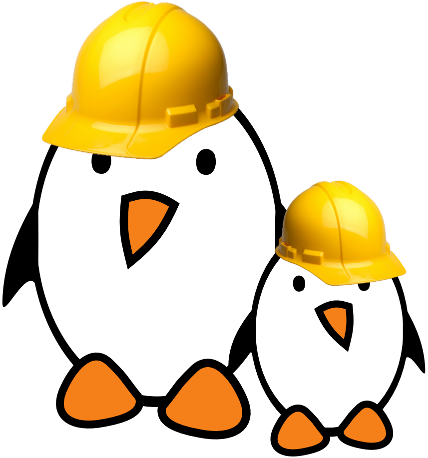
Morning

- Lecture and lab optimizing system initialization
- Lecture and lab: filesystem optimizations

Afternoon

- Lecture and lab: kernel optimizations
- Lecture and lab: bootloader optimizations

Practical lab - Board setup



Prepare your board

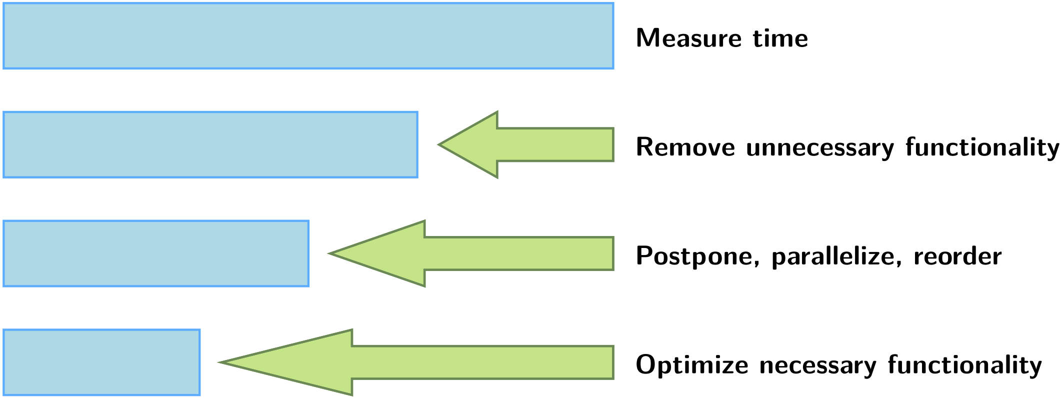
- Access the board through its serial line
- Check the stock bootloader
- Attach the LCD4 cape

Principles

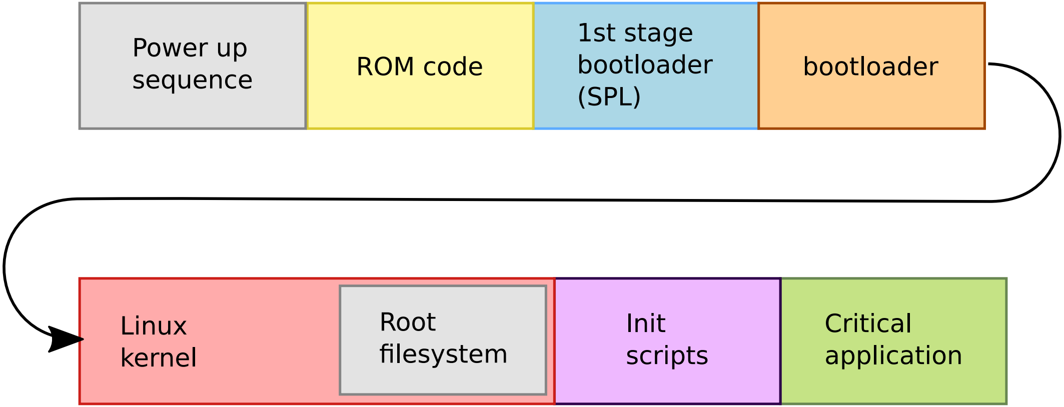
Set your goals

- Reducing boot time implies measuring boot time!
- You will have to choose reference events at which you start and stop counting time.
- What you choose will depend on the goal you want to achieve. Here are typical cases:
  - Showing a splash screen or an animation, playing a sound to indicate the board is booting
  - Starting a listening service to handle a particular message
  - Being fully functional as fast as possible

Boot time reduction methodology

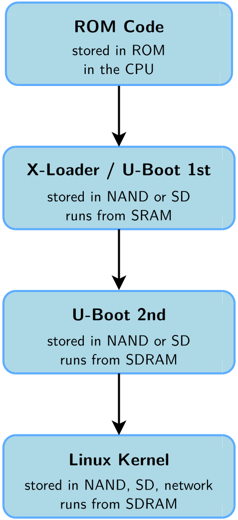


Boot time components



We are focusing on reducing *cold* boot time, from power on to the critical application.

Booting on ARM TI OMAP2+ / AM33xx



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to $<$64 KB. No user interaction possible.
- **X-Loader** or **U-Boot SPL**: runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called `MLO`.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.bin` or `u-boot.img`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

What to optimize first

Start by optimizing the **last steps** of the boot process!

- Don't start by optimizing things that will reduce your ability to make measurements and implement other optimizations.
- Start by optimizing your applications and startup scripts first.
- You can then simplify BusyBox, reducing the number of available commands.
- The next thing to do is simplify and optimize the kernel. This will make you lose debugging and development capabilities, but this is fine as user space has already been simplified.
- The last thing to do is implement bootloader optimizations, when kernel optimizations are over and when the kernel command line is frozen.

We will follow this order during the practical labs.

Worst things first!

Premature optimization is the root of all evil.
Donald Knuth

- Taking the time to measure time carefully is important.
- Find the worst consumers of time and address them first.
- You can waste a lot of time if you start optimizing minor spots first.

Build automation

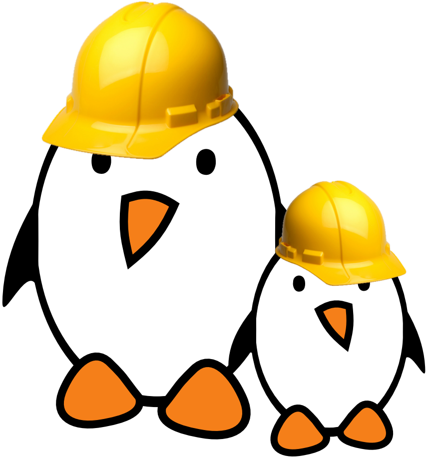
- Very important to automate the way the root filesystem is built, if not done yet. That's always the first thing we do in boot time reduction projects, and it's worth investing 1 or 2 days doing this.
- Otherwise, you may lose existing optimizations or introduce new bugs when making further optimizations. Without a build system, you will waste a lot of time too.
- Can be done through build systems such as Buildroot or Yocto, or using the original build automation of the project.
- Can also be done for kernel and bootloader optimizations, though the need is less critical.

Generic ideas

Some ideas to keep in mind while trying to reduce the boot time:

- The fastest code is code that is not executed
- A big part of booting is actually loading code and data from the storage to RAM. Reading less means booting faster. I/O are expensive!
- The root filesystem may take longer to mount if it is bigger.
- So, even code that is not executed can make your boot time longer.
- Also, try to benchmark different types of storage. It has happened that booting from SD card was actually faster than booting from NAND.

Practical lab - Build and boot the system



Compile your system components and get your system up and running

- Compile the root filesystem with Buildroot
- Compile, install and run the bootloader (U-Boot)
- Compile and install the Linux kernel
- Get the full system up and running

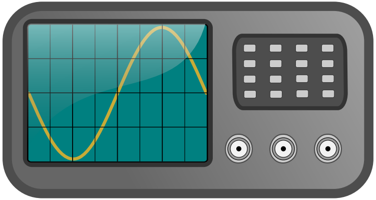
Measuring

bootlin

embedded Linux and kernel engineering

Time measurement equipment: hardware

- The best equipment is an oscilloscope, if you can afford one
- Allows to time the "Power on" event (connected to a power rail), or any event (connected to a GPIO pin, for example), all this in a very accurate way.
- Easy to write to a GPIO at all the stages of system booting (we will explain how to do it)
- Some oscilloscopes are getting affordable. Example: Bitscope Pocket Analyzer (245 AUD, supported on Linux, `https://www.bitscope.com/product/BS10/`)



Measuring with hardware: using an Arduino

https://arduino.cc

- If you don't have an oscilloscope, an Arduino (or any general purpose MCU or MPU board) is a great solution too.
- The main strength of Arduino is its great ease of use and programming, plus all the hardware support libraries that are available.
- You can easily connect board pins to the Arduino analog pins, and watch their voltage.
- In our labs, we will use Arduino Nano boards for measuring the whole boot time.
- Arduino boards are Open Source Hardware. This project is definitely worth supporting!

Arduino Nano

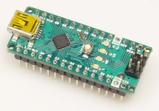
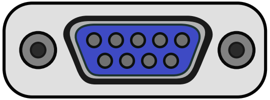


Image credits: https://frama.link/wdhQENrp (Wikimedia Commons)

Time measurement equipment: serial port

- Useful when you don't have monitoring hardware, or don't want to make take any risk connecting wires to the hardware.
- Usually relies on software which times messages received from the board's serial port (serial port absolutely required). Such software runs on a PC connected to the serial port.
- On the board, requires a real serial port (directly connected to the CPU), immediately usable from the earliest parts of the boot process.
- Limitation: won't be able to time the "Power on" event in an accurate way. But acceptable as you can assume that the time to run the first stage bootloader is constant.



Time measurement equipment: USB to serial

- Serial ports over USB **device** are fine if there's an on-board serial-to-USB chip directly connected to the CPU serial port (very frequent).
- Attaching a USB-to-serial dongle to a USB **host** port on the device won't do: USB is available much later and messages go through more complex software stacks (loss of time accuracy).
- All development boards have a standard or USB serial port.

grabserial

https://elinux.org/Grabserial (by Tim Bird)

- A Python script to add timestamps to messages coming from a serial console.
- Key advantage: starts counting very early (bootstrap and bootloader)
- Another advantage: no overhead on the target, because run on the host machine.
- Drawbacks: may not be precise enough. Can't measure power up time.
- Ubuntu package: `grabserial`
  Otherwise available on https://github.com/tbird20d/grabserial/

Using grabserial

Serial device

Reset the time to 0 when this string is found

Print time for each received line

End recording time in seconds

```
$ ~/bin/grabserial -d /dev/ttyACM0 -m RomBOOT  -t -e 30
[0.000002 0.000002] RomBOOT
[0.054440 0.054440]
[0.054594 0.000154]
[0.054719 0.000125] AT91Bootstrap 3.5.2 (Wed Jan 30 18:42:28 CET 2013)
[0.059009 0.004290]
[0.059185 0.000176] 1-Wire: Loading 1-Wire information ...
[0.062645 0.003460] 1-Wire: ROM Searching ... Done, 0x3 1-Wire chips found
[0.110066 0.047421]
[0.110175 0.000109] 1-Wire: BoardName | [Revid] | VendorName
[0.162743 0.052568]   #0x0  SAMA5D34-CM [C9]      EMBEST
[0.214729 0.051986]   #0x1  SAMA5D3x-MB [B9]      FLEX
[0.266321 0.051592]   #0x2  SAMA5D3x-DM [A9]      FLEX
```

Arrival time of the first character

Elapsed time since the previous time stamp

Caution: `grabserial` shows the arrival time of the **first character** of a line. This doesn't mean that the entire line was received at that time.

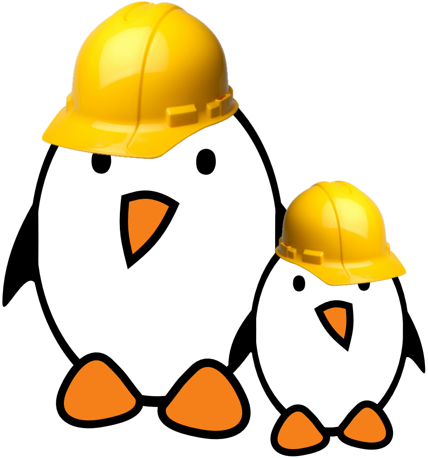
grabserial tips

- You can interrupt `grabserial` manually (with `[Ctrl][c]`) when you have gone far enough.
- The `-m` and `-q` options actually expect a regular expression. A normal string may not match in the middle of a line.
- Example: you may have to replace `-m "Starting kernel"` by `-m ".*Starting Kernel.*"`.

Dedicated measuring resources

Later, we will see specific resources for measuring time

- time for measuring application time
- strace for application tracing
- bootchartd for tracing the execution of system services
- CONFIG_PRINTK_TIME and initcall_debug for tracing kernel code and functions.



Measuring time with software

- Setting up `grabserial`
- Modify the video player to log a notification after the first frame is processed.
- Time the various components of boot time through messages written to the serial console.

Measuring time with hardware

- Setting up an Arduino system
- Timing reset on Beagle Bone Black
- Modifying the video player to toggle a GPIO after the first frame is processed.
- Display the live total time through a 7-segment display

Toolchain optimizations

Best toolchain for your project

Optimizing the cross-compiling toolchain is typically one of the first things to do:

- The benefits of a toolchain change will be more significant and easier to measure if other optimizations haven't been done yet.
- Here's what you can change in a toolchain, with a potential impact on boot time, performance and size:
  - Components: versions of *gcc* and *binutils*
    More recent versions can feature better optimization capabilities.
  - C library: *glibc*, *uClibc*, *musl*
    uClibc and *musl* libraries make a smaller root filesystem
  - Instruction set variant: *ARM* or *Thumb2*, *Hard Float* support or not.
    Can have an impact on code performance and code size (*Thumb2* encodes the same instructions as *ARM* but in a more compact way, at least significantly reducing size).

Choosing the C library

- The C library is hardcoded at toolchain creation time
- Available C libraries:
  - *glibc*: most standard and featureful
  - *uClibc*: smaller and configurable. Has been around for about 20 years.
  - *musl*: an alternative to *uClibc*, developed more recently but mature too.

glibc

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- By default, quite big for small embedded systems. On armv7hf, version 2.23: `libc`: 1.5 MB, `libm`: 492 KB, source: `https://toolchains.bootlin.com`
- But some features not needed in embedded systems can be configured out (merged from the old *eglibc* project).
- `https://www.gnu.org/software/libc/`

Image: `https://bit.ly/2EzHl6m`

uClibc-ng

- https://uclibc-ng.org/
- A continuation of the old uClibc project, license: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface.
  - Supports most embedded architectures, including MMU-less ones (ARM Cortex-M, Blackfin, etc.). The only library supporting ARM noMMU.
  - No guaranteed binary compatibility. May need to recompile applications when the library configuration changes.
  - Some glibc features may not be implemented yet (real-time, floating-point operations...)
  - Focus on size rather than performance
  - Size on armv7hf, version 1.0.24: `libc`: 652 KB, source: https://toolchains.bootlin.com
- Actively supported, but Yocto Project stopped supporting it.

musl C library

https://www.musl-libc.org/

- A lightweight, fast and simple library for embedded systems
- Created while uClibc's development was stalled
- In particular, great at making small static executables
- More permissive license (MIT), making it easier to release static executables. We will talk about the requirements of the LGPL license (glibc, uClibc) later.
- Supported by build systems such as Buildroot and Yocto Project.
- Used by the Alpine Linux distribution (https://www.alpinelinux.org/), fitting in about 130 MB of storage.

glibc vs uclibc-ng vs musl - small static executables

Let's compile and strip a `hello.c` program **statically** and compare the size

- With gcc 6.3, armel, musl 1.1.16:
  7300 bytes
- With gcc 6.3, armel, uclibc-ng 1.0.22 :
  67204 bytes.
- With gcc 6.2, armel, glibc:
  492792 bytes

glibc vs uclibc vs musl (static)

Let's compile and strip BusyBox 1.26.2 **statically** and compare the size

- With gcc 6.3, armel, musl 1.1.16:
  183348 bytes
- With gcc 6.3, armel, uclibc-ng 1.0.22 :
  210620 bytes.
- With gcc 6.2, armel, glibc:
  755088 bytes

Notes:

- BusyBox is automatically compiled with `-Os` and stripped.
- Compiling with shared libraries will mostly eliminate size differences

Other smaller C libraries

- Several other smaller C libraries have been developed, but none of them have the goal of allowing the compilation of large existing applications
- They can run only relatively simple programs, typically to make very small static executables and run in very small root filesystems.
- Choices:
  - Newlib, `https://sourceware.org/newlib/`
  - Klibc, `https://kernel.org/pub/linux/libs/klibc/`, designed for use in an *initramfs* or *initrd* at boot time.

Advise for choosing the C library

- Advice to start developing and debugging your applications with *glibc*, which is the most standard solution.
- Then, when everything works, if you have size constraints, try to compile your app and then the entire filesystem with *uClibc* or *musl*.
- If you run into trouble, it could be because of missing features in the C library.
- In case you wish to make static executables, *musl* will be an easier choice. Note that static executables built with a given C library can be used in a system with a different C library.

Time your commands using the time command

First run

```
> time ffmpeg ...
real 0m 2.06s
user 0m 0.17s
sys 0m 0.26s
```

Total observed time

Time in userspace (running the program and shared libs)

Time in kernel space (accessing files, accessing device data)

Second run (program and libraries already in file cache)

```
> time ffmpeg...
real 0m 0.66s
user 0m 0.17s
sys 0m 0.25s
```

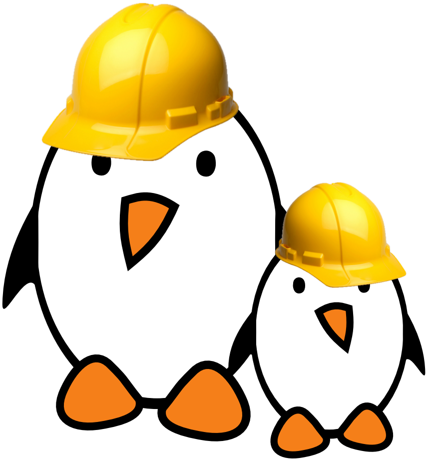
Less waiting time!

real = user + sys + waiting time

Your program cannot run faster than user + sys (unless you optimize the code)

This gives you the best time that can possibly be achieved (with the fastest storage).

Practical lab - Trying a Thumb2 toolchain



- Measure filesystem and `ffmpeg` binary size. Time the execution of the application.
- Re-compile the root filesystem using a Thumb2 toolchain
- Measure size and time again.

Lessons from labs: ARM vs Thumb2

- Testcase: root filesystem with `ffmpeg` and associated libraries (from our training labs)
- Compiled with gcc 7.4, generating *ARM* code:
  Total filesystem size: 3.79 MB
  `ffmpeg` size: 227 KB
- Compiled with gcc 7.4, generating *Thumb2* code:
  Total filesystem size: 3.10 MB (-18 %)
  `ffmpeg` size: 183 KB (-19 %)
- Performance aspect: performance apparently slightly improved with Thumb2 (approximately less than 5 %, but there are slight variations in measured execution time, for one run to another).

Lessons from labs: musl vs uClibc

Tested while preparing the labs (too long to do this during the class)

Tried to replace *uClibc* by *musl* in our video player lab:

- musl library size: 680 KB (size of the `tar` archive for `lib/`)
- uClibc library size: 570 KB
- uClibc saves 110 KB (useful), but otherwise no other significant change in filesystem and code size. Not a surprise when the system is mostly filled with binaries relying on shared libraries.

We stuck to *uClibc*!

Optimizing applications

bootlin

embedded Linux and kernel engineering

Measuring: strace

- Allows to trace all the system calls made by an application and its children.
- Useful to:
  - Understand how time is spent in user space
  - For example, easy to find file open attempts (`open()`), file access (`read()`, `write()`), and memory allocations (`mmap2()`). Can be done without any access to source code!
  - Find the biggest time consumers (low hanging fruit)
  - Find unnecessary work done in applications and scripts. Example: opening the same file(s) multiple times, or trying to open files that do not exist.
- Limitation: you can't trace the `init` process!

strace

System call tracer - `https://strace.io`

- Available on all GNU/Linux systems
  Can be built by your cross-compiling toolchain generator.
- Even easier: drop a ready-made static binary for your architecture, just when you need it. See `https://frama.link/q5WcWayh`
- Allows to see what any of your processes is doing: accessing files, allocating memory... Often sufficient to find simple bugs.
- Usage:
  `strace <command>` (starting a new process)
  `strace -p <pid>` (tracing an existing process)
  `strace -c <command>` (statistics of system calls taking most time)

See the strace manual for details.

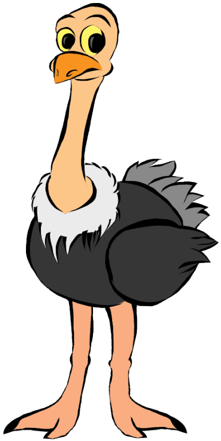


Image credits: `https://strace.io/`

strace example output

```
> strace cat Makefile
execve("/bin/cat", ["cat", "Makefile"], [/* 38 vars */]) = 0
brk(0) = 0x98b4000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f85000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=111585, ...}) = 0
mmap2(NULL, 111585, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f69000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320h\1\0004\0\0\0\344"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1442180, ...}) = 0
mmap2(NULL, 1451632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e06000
mprotect(0xb7f62000, 4096, PROT_NONE) = 0
mmap2(0xb7f63000, 12288, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x15c) = 0xb7f63000
mmap2(0xb7f66000, 9840, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f66000
close(3) = 0
```

Hint: follow the open file descriptors returned by open().

strace -c example output

```
> strace -c cheese
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 36.24    0.523807          19     27017           poll
 28.63    0.413833           5     75287       115 ioctl
 25.83    0.373267           6     63092     57321 recvmsg
  3.03    0.043807           8      5527           writev
  2.69    0.038865          10      3712           read
  2.14    0.030927           3     10807           getpid
  0.28    0.003977           1      3341        34 futex
  0.21    0.002991           3      1030       269 openat
  0.20    0.002889           2      1619       975 stat
  0.18    0.002534           4       568           mmap
  0.13    0.001851           5       356           mprotect
  0.10    0.001512           2       784           close
  0.08    0.001171           3       461       315 access
  0.07    0.001036           2       538           fstat
...
```

oprofile

A system-wide profiler

- Two ways of working: *legacy* mode and *perf_events* mode
- *legacy* mode:
  - Low accuracy, use a kernel driver to profile
  - `CONFIG_OPROFILE`
  - User space tools: `opcontrol` and `oprofiled`
- *perf_events* mode:
  - Uses hardware performance counters
  - `CONFIG_PERF_EVENTS` and `CONFIG_HW_PERF_EVENTS`
  - User space tools: `operf`

oprofile: usage

- *legacy* mode:

```
opcontrol --vmlinux=/path/to/vmlinux # optional step
opcontrol --start
/my/command
opcontrol --stop
```

- *perf_events* mode:

```
operf --vmlinux=/path/to/vmlinux /my/command
```

- Get the results with:

```
opreport
```

oprofile

```
# opreport
Using /var/lib/oprofile/samples/ for samples directory.
CPU: CPU with timer interrupt, speed 393 MHz (estimated)
Profiling through timer interrupt
          TIMER:0|
  samples|      %|
------------------
     1540 78.3715 no-vmlinux
      105  5.3435 libQtGui.so.4.8.4
       66  3.3588 libc-2.15.so
       64  3.2570 libQtCore.so.4.8.4
       58  2.9517 ld-2.15.so
       45  2.2901 libgobject-2.0.so.0.3000.3
       37  1.8830 libgstreamer-0.10.so.0.30.0
       13  0.6616 libglib-2.0.so.0.3000.3
        9  0.4580 libQtScript.so.4.8.4
        6  0.3053 libgcc_s.so.1
        4  0.2036 libQtDeclarative.so.4.8.4
        4  0.2036 libstdc++.so.6.0.17
        3  0.1527 libpthread-2.15.so
        2  0.1018 busybox
        2  0.1018 libQtSvg.so.4.8.4
        2  0.1018 libQtWebKit.so.4.9.3
        2  0.1018 libgthread-2.0.so.0.3000.3
        1  0.0509 HomeAutomation
        1  0.0509 libQtNetwork.so.4.8.4
        1  0.0509 libphonon_gstreamer.so
```

perf

- Uses hardware performance counters
- `CONFIG_PERF_EVENTS` and `CONFIG_HW_PERF_EVENTS`
- User space tool: `perf`. It is part of the kernel sources so it is always in sync with your kernel.
- Usage:

```
perf record /my/command
```

- Get the results with:

```
perf report
```

perf

```
20.91%  gst-launch-0.10  libavcodec.so.53.35.0         [.] 0x00000000003bdaa1
15.45%  gst-launch-0.10  libgstflump3dec.so            [.] 0x0000000000014b42
 3.16%  gst-launch-0.10  libglib-2.0.so.0.3600.2       [.] 0x00000000000882c9
 2.99%  gst-launch-0.10  libc-2.17.so                  [.] __memcpy_ssse3_back
 2.37%  gst-launch-0.10  liboil-0.3.so.0.3.0           [.] 0x000000000004417e
 2.24%  gst-launch-0.10  libgobject-2.0.so.0.3600.2    [.] g_type_value_table_peek
 1.53%  gst-launch-0.10  libc-2.17.so                  [.] vfprintf
 1.37%  gst-launch-0.10  libgstreamer-0.10.so.0.30.0   [.] 0x0000000000026fd8
 1.29%  gst-launch-0.10  ld-2.17.so                    [.] do_lookup_x
 0.99%  gst-launch-0.10  libpthread-2.17.so            [.] pthread_mutex_lock
 0.98%  gst-launch-0.10  libgobject-2.0.so.0.3600.2    [.] g_type_check_value
 0.93%  gst-launch-0.10  libgstavi.so                  [.] 0x00000000000119f9
 0.88%  gst-launch-0.10  libgstreamer-0.10.so.0.30.0   [.] gst_value_list_get_type
 0.85%  gst-launch-0.10  libc-2.17.so                  [.] __random
 0.66%  gst-launch-0.10  [kernel.kallsyms]             [k] clear_page_c_e
 0.62%  gst-launch-0.10  [kernel.kallsyms]             [k] try_to_wake_up
 0.61%  gst-launch-0.10  [kernel.kallsyms]             [k] page_fault
 0.58%  gst-launch-0.10  libgobject-2.0.so.0.3600.2    [.] g_type_is_a
 0.57%  gst-launch-0.10  libc-2.17.so                  [.] __strcmp_sse42
 0.57%  gst-launch-0.10  [kernel.kallsyms]             [k] radix_tree_lookup_element
 0.57%  gst-launch-0.10  libc-2.17.so                  [.] malloc
 0.57%  gst-launch-0.10  libc-2.17.so                  [.] _int_malloc
 0.55%  gst-launch-0.10  libgobject-2.0.so.0.3600.2    [.] g_type_check_instance_is_a
 0.53%  gst-launch-0.10  [kernel.kallsyms]             [k] __ticket_spin_lock
 0.53%  gst-launch-0.10  libgobject-2.0.so.0.3600.2    [.] g_type_check_value_holds
 0.53%  gst-launch-0.10  libgstffmpeg.so               [.] 0x000000000001e40c
 0.51%  gst-launch-0.10  libgstreamer-0.10.so.0.30.0   [.] gst_structure_id_get_value
 0.50%  gst-launch-0.10  libc-2.17.so                  [.] _IO_default_xsputn
 0.50%  gst-launch-0.10  [kernel.kallsyms]             [k] tg_load_down
```

Linker optimizations (1)

Group application code used at startup

- Find the functions called during startup, for example using the `-finstrument-functions` gcc option.
- Create a custom linker script to reorder these functions in the call order. You can achieve that by putting each function in their own section using the `-ffunction-sections` gcc option.
- Particularly useful for flash storage with rather big MTD read blocks. As the whole read blocks are read, you end up reading unnecessary data.

Details: `http://blogs.linux.ie/caolan/2007/04/24/controlling-symbol-ordering/`

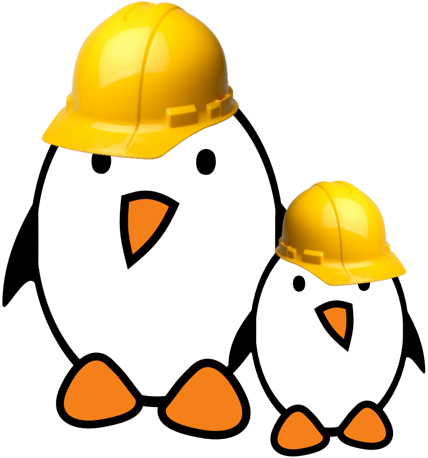
Linker optimizations (2)

- Here's a very simple way to find the maximum savings you can expect from this technique:
  - Start the application once and measure its startup time.
  - Start the application and measure its startup time again. Its code should still be in the Linux file cache, and the code loading time will be zero.
- You now know how much time it took to load the application code (and its libraries) the first time. Linker optimizations will save less than this upper limit.
- You can then decide whether this could be worth the effort. Such optimizations are costly, as the way you compile your applications has to be modified.

Prelink

- Prelinking reduces the time needed to start an executable
- It is extensively used on Android
- It has to be configured to know which libraries needs to be prelinked and will assign a fixed address for each available symbol thus removing the need to relocate symbols when starting an executable.
- Be careful of security implications, as executable code is always loaded at the same address.
- Code and paper at `http://people.redhat.com/jakub/prelink/`
- Supports ARM but no release since 2013. Not supported in Buildroot either. However, easy to try on x86.

Practical lab - Optimizing the application



- Compile the video player with just the features needed at run time.
- Trace and profile the video player with `strace`
- Observe size savings

Optimizing init scripts and system startup

embedded Linux and kernel engineering

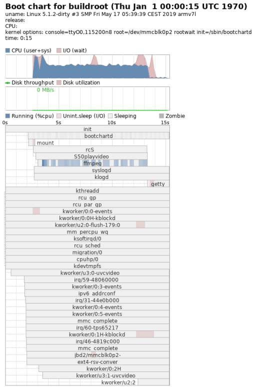
Methodology

There are multiple ways to reduce the time spent in init scripts before starting the application:

- Start the application as soon as possible after only the strictly necessary dependencies.
- Simplify shell scripts
- Even starting the application before `init`

Measuring - bootchart

If you want to have a more detailed look at the userland boot sequence than with `grabserial`, you can use `bootchart`.



Measuring - bootchart

- You can use bootchartd from busybox (CONFIG_BOOTCHARTD=y)
- Boot your board passing init=/sbin/bootchartd on your kernel command line
- Copy /var/log/bootlog.tgz from your target to your host
- Generate the timechart:

```
cd bootchart-<version>
java -jar bootchart.jar bootlog.tgz
```

bootchart is available at http://www.bootchart.org

Measuring - systemd

If you are using `systemd` as your `init` program, you can use `systemd-analyze`. See https://www.freedesktop.org/software/systemd/man/systemd-analyze.html.

```
$ systemd-analyze blame
  6207ms udev-settle.service
   735ms NetworkManager.service
   642ms avahi-daemon.service
   600ms abrtd.service
   517ms rtkit-daemon.service
   396ms dbus.service
   390ms rpcidmapd.service
   346ms systemd-tmpfiles-setup.service
   316ms cups.service
   310ms console-kit-log-system-start.service
   309ms libvirtd.service
   303ms rpcbind.service
   298ms ksmtuned.service
   281ms rpcgssd.service
   277ms sshd.service
...
```

Init

Starting as soon as possible after all the dependencies are started:

- Depends on your `init` program. Here we are assuming sysV `init` scripts.
- `init` scripts run in alphanumeric order and start with a letter (K for stop (**k**ill) and S for **s**tart).
- You want to use the lowest number you can for your application.
- You can even replace `init` with your application!

How fast would we be if we could be the first started application?

Optimizing init scripts

- Start all your services directly from a single startup script (e.g. `/etc/init.d/rcS`). This eliminates multiple calls to `/bin/sh`.
- You could mount your filesystems directly in the C code of your application:

```
#include <stdio.h>
#include <sys/mount.h>

int main (void)
{
        int ret;
        ret = mount("sysfs", "/tmp/test", "sysfs", 0, NULL);
        if(ret < 0)
                perror("Can't mount sysfs\n");
}
```

Reduce forking (1)

- fork/exec system calls are very expensive. Because of this, calls to executables from shells are slow.
- Even an echo in a BusyBox shell results in a fork syscall!
- Select Shells -> Standalone shell in BusyBox configuration to make the shell call applets whenever possible.
- Pipes and back-quotes are also implemented by fork/exec. You can reduce their usage in scripts. Example:

```
cat /proc/cpuinfo | grep model
```

Replace it with:

```
grep model /proc/cpuinfo
```

See https://elinux.org/Optimize_RC_Scripts

Reduce forking (2)

Replaced:

```
if [ $(expr match "$(cat /proc/cmdline)" '.* debug.*')\
       -ne 0 -o -f /root/debug ]; then
DEBUG=1
```

By a much cheaper command running only one process:

```
res=`grep " debug" /proc/cmdline`
if [ "$res" -o -f /root/debug ]; then
DEBUG=1
```

This only optimization allowed to save 87 ms on an ARM AT91SAM9263 system (200 MHz)!

Reduce size (1)

- Strip your executables and libraries, removing ELF sections only needed for development and debugging. The `strip` command is provided by your cross-compiling toolchain. That's done by default in Buildroot.
- `superstrip`: `https://muppetlabs.com/~breadbox/software/elfkickers.html`. Goes beyond `strip` and can strip out a few more bits that are not used by Linux to start an executable. Buildroot stopped supporting it because it can break executables. Try it only if saving a few bytes is critical.

Reduce size (2)

You may try `mklibs`, available at https://packages.debian.org/sid/mklibs.

- `mklibs` produces cut-down shared libraries that contain only the routines required by a particular set of executables. Really useful with big libraries like OpenGL and QT. It even works without having the source code.
- Available in Yocto, but not in Buildroot (2019.02 status).
- Limitation: `mklibs` could remove `dlopen`ed libraries (loaded "manually" by applications) because it doesn't see them.

Quick splashscreen display (1)

Often the first sign of life that you are showing!

- You could use the `fbv` program (`http://freshmeat.sourceforge.net/projects/fbv`) to display your splashscreen.
- On `armel`, you can just use our statically compiled binary: `https://github.com/bootlin/static-binaries/tree/master/fbv/`
- However, this is slow: 878 ms on an Microchip AT91SAM9263 system!

Quick splashscreen display (2)

- To do it faster, you can dump the framebuffer contents:

```
fbv -d 1 /root/logo.bmp
cp /dev/fb0 /root/logo.fb
lzop -9 /root/logo.fb
```

- And then copy it back as early as possible in an initramfs:

```
lzopcat /root/logo.fb.lzo > /dev/fb0
```

Results on an Microchip AT91SAM9263 system:

	fbv	plain copy (dd)	lzopcat
Time	878 ms	54 ms	52.5 ms

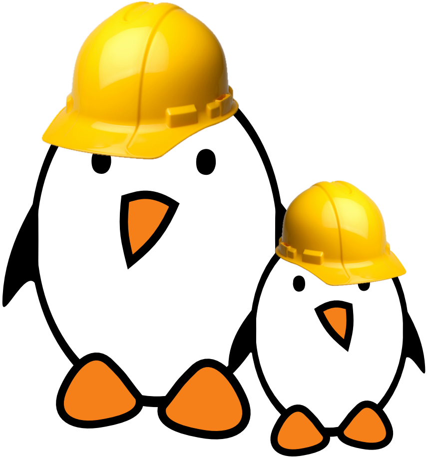
https://bootlin.com/blog/super-fast-linux-splashscreen/

Animated splashscreen

Still slow to read and write entire screens. Just draw useful pixels and even create an animation!

- Create a simple C program that just animates pixels and simple geometric shapes on the framebuffer!
- Example: `https://bootlin.com/pub/code/fb/anim.c`. On a 400 MHz ARM9 system: starts drawing in 10 ms
  Size: 24 KB, compiled statically.

Practical lab - Reducing time in init-scripts



- Regenerate the root filesystem with Buildroot
- Use bootchart to measure boot time

Filesystem optimizations

Filesystem impact on performance

Tuning the filesystem is usually one of the first things we work on in boot time projects.

- Different filesystems can have different initialization and mount times. In particular, the type of filesystem for the root filesystem directly impacts boot time.
- Different filesystems can exhibit different read, write and access time performance, according to the type of filesystem activity and to the type of files in the system.

Different filesystem for different storage types

- Block storage (including memory cards, eMMC)
  - ext2, ext3, ext4
  - xfs, jfs, reiserfs
  - btrfs
  - f2fs
  - SquashFS
- Raw flash storage
  - JFFS2
  - YAFFS2
  - UBIFS
  - ubiblock + SquashFS

See our embedded Linux training materials for full details:
`https://bootlin.com/doc/training/embedded-linux/`
See also our flash filesystem benchmarks:
`https://elinux.org/Flash_Filesystem_Benchmarks`.

Block filesystems

For block storage

- ext4: best for rather big partitions, good read and write performance.
- xfs, jfs, reiserfs: can be good in some read or write scenarii as well.
- btrfs, f2fs: can achieve best read and write performance, taking advantage of the characteristics of flash-based block devices.
- SquashFS: best mount time and read performance, for read-only partitions. Great for root filesystems which can be read-only.

JFFS2

For raw flash storage

- Mount time depending on filesystem size: the kernel has to scan the whole filesystem at mount time, to read which block belongs to each file.
- Need to use the `CONFIG_JFFS2_SUMMARY` kernel option to store such information in flash. This dramatically reduces mount time.
- Benchmark on ARM:
  from 16 s to 0.8 s for a 128 MB partition.
- Rather poor read and write performance,
  compared to YAFFS2 and UBIFS.

YAFFS2

For raw flash storage

- Good mount time
- Good read and write performance
- Drawbacks: no compression, not in the mainline Linux kernel

UBIFS

For raw flash storage

- Advantages:
  - Good read and write performance (similar to YAFFS2)
  - Other advantages: better for wear leveling (can operate on the whole UBI space, not only within a single partition).
- Drawbacks:
  - Not appropriate for small partitions (too much metadata overhead). Use JFFS2 or JAFFS2 instead.
  - Not so good mount time, because of the time needed to initialize UBI (*UBI Attach*: at boot time or running `ubi_attach` in user space).
  - Addressed by *UBI Fastmap*, introduced in Linux 3.7.
    See next slides.

How UBI Fastmap works

- *UBI Attach*: needs to read UBI metadata by scanning all erase blocks. Time proportional to the storage size.
- *UBI Fastmap* stores such information in a few flash blocks (typically at UBI detach time during system shutdown) and finds it there at boot time.
- This makes *UBI Attach* time constant.
- If *Fastmap* information is invalid (unclean system shutdown, for example), it falls back to scanning (slower, but correct, and *Fastmap* will work again during the next boot).
- Details: ELCE 2012 presentation from Thomas Gleixner: `https://elinux.org/images/a/ab/UBI_Fastmap.pdf`

Using UBI Fastmap

- Compile your kernel with `CONFIG_UBI_FASTMAP`
- Boot your system at least once with the `ubi.fm_autoconvert=1` kernel parameter.
- Reboot your system in a clean way
- You can now remove `ubi.fm_autoconvert=1`

UBI Fastmap benchmark

- Measured on the Microchip SAMA5D3 Xplained board (ARM), Linux 3.10
- UBI space: 216 MB
- Root filesystem: 80 MB used (Yocto)
- Average results:

	Attach time	Diff	Total time
Without *UBI Fastmap*	968 ms		
With *UBI Fastmap*	238 ms	-731 ms	-665 ms

- Expect to save more with bigger UBI spaces!

Note: total boot time reduction a bit lower probably because of other kernel threads executing during the attach process.

ubiblock + SquashFS

For raw flash storage

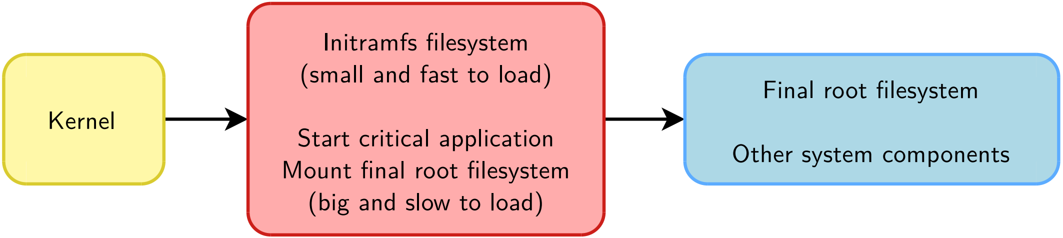
- *ubiblock*: read-only block device on top of UBI (`CONFIG_MTD_UBI_BLOCK`). Available in Linux 3.15 (developed on his spare time by Ezequiel Garcia, a Bootlin contractor).
- Allows to put SquashFS on a UBI volume.
- Expecting great boot time and read performance. Great for read-only root filesystems.
- Benchmarks not available yet.

Finding the best filesystem

- Raw flash storage: UBIFS with `CONFIG_UBI_FASTMAP` is probably the best solution.
- Block storage: SquashFS best solution for root filesystems which can be read-only. Btrfs and f2fs probably the best solutions for read/write filesystems.
- Fortunately, changing filesystem types is quite cheap, and completely transparent for applications. Just try several filesystem options, as see which one works best for you!
- Do not focus only on boot time.
  For systems in which read and write performance matters, we recommend to use separate root filesystem (for quick boot time) and data partitions (for good runtime performance).

Initramfs

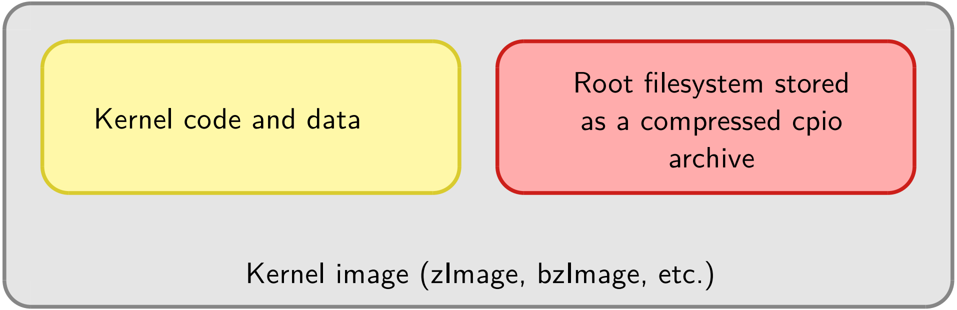
An idea is to use a very small initramfs, just enough to start the critical application and then switch to the final root filesystem.



rootfs in memory: initramfs (1)

- It is also possible to have the root filesystem integrated into the kernel image
- It is therefore loaded into memory together with the kernel
- This mechanism is called **initramfs**
  - It integrates a compressed archive of the filesystem into the kernel image
  - Variant: the compressed archive can also be loaded separately by the bootloader.
- It is useful for two cases
  - Fast booting of very small root filesystems. As the filesystem is completely loaded at boot time, application startup is very fast.
  - As an intermediate step before switching to a real root filesystem, located on devices for which drivers not part of the kernel image are needed (storage drivers, filesystem drivers, network drivers). This is always used on the kernel of desktop/server distributions to keep the kernel image size reasonable.

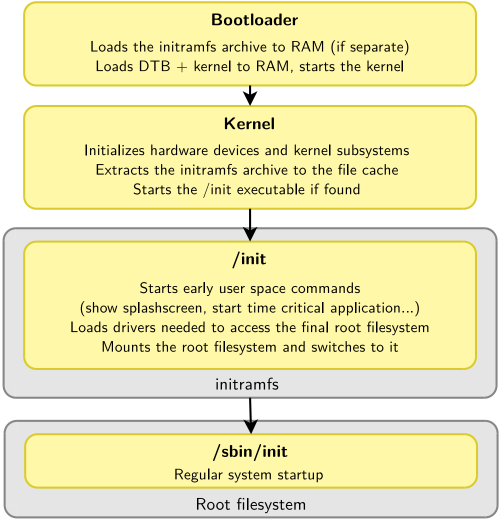
rootfs in memory: initramfs (2)



rootfs in memory: initramfs (3)

- The contents of an initramfs are defined at the kernel configuration level, with the `CONFIG_INITRAMFS_SOURCE` option
  - Can be the path to a directory containing the root filesystem contents
  - Can be the path to a cpio archive
  - Can be a text file describing the contents of the initramfs (see documentation for details)
- The kernel build process will automatically take the contents of the `CONFIG_INITRAMFS_SOURCE` option and integrate the root filesystem into the kernel image
- Details (in kernel sources):
  `Documentation/filesystems/ramfs-rootfs-initramfs.txt`
  `Documentation/early-userspace/README`

Overall booting process with initramfs



Initramfs for boot time reduction

Create the smallest initramfs possible, just enough to start the critical application and then switch to the final root filesystem with `switch_root`:

- Use a light C library reduced to the minimum, *uClibc* if you are not yet using it for your root filesystem
- Reduce BusyBox to the strict minimum. You could even do without it and implement `/init` in C.
- Use statically linked applications (less CPU overhead, less libraries to load, smaller initramfs if no libraries at all). `BR2_STATIC_LIBS` in Buildroot.

Statically linked executables: licensing constraints

- Statically linked executables are very useful to reduce size (especially in small initramfs), and require less work to start.
- However, the LGPL license in the uClibc and glibc libraries require to leave the user the ability to relink the executable with a modified version of the library.
- Solution to keep static binaries:
  - Either provide the executable source code (even proprietary), allowing to recompile it with a modified version of the library. That's what you do when you ship a static BusyBox.
  - Or also provide a dynamically linked version of the executable (in a separate way), allowing to use another library version.
  - Easiest solution: build your static executables with the `musl` library (MIT license: no trouble)
- References:
  https://gnu.org/licenses/gpl-faq.html#LGPLStaticVsDynamic
  https://gnu.org/copyleft/lesser.html#section4

Do not compress your initramfs

- If you ship your initramfs inside a compressed kernel image, don't compress it (enable `CONFIG_INITRAMFS_COMPRESSION_NONE`).
- Otherwise, by default, your initramfs data will be compressed twice, and the kernel will be bigger and will take a little more time to load and uncompress.
- Example on Linux 5.1 with a 1.60 MB initramfs (tar archive size) on Beagle Bone Black: this allowed to reduce the kernel size from 4.94 MB to 4.74 MB (-200 KB) and save about 170 ms of boot time.

Kernel optimizations

Measure - Kernel initialization functions

To find out which kernel initialization functions are the longest to execute, add `initcall_debug` to the kernel command line. Here's what you get on the kernel log:

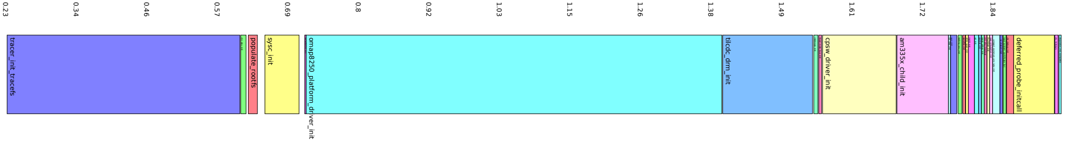
```
...
[    3.750000] calling  ov2640_i2c_driver_init+0x0/0x10 @ 1
[    3.760000] initcall ov2640_i2c_driver_init+0x0/0x10 returned 0 after 544 usecs
[    3.760000] calling  at91sam9x5_video_init+0x0/0x14 @ 1
[    3.760000] at91sam9x5-video f0030340.lcdheo1: video device registered @ 0xe0d3e340, irq = 24
[    3.770000] initcall at91sam9x5_video_init+0x0/0x14 returned 0 after 10388 usecs
[    3.770000] calling  gspca_init+0x0/0x18 @ 1
[    3.770000] gspca_main: v2.14.0 registered
[    3.770000] initcall gspca_init+0x0/0x18 returned 0 after 3966 usecs
...
```

You might need to increase the log buffer size with `CONFIG_LOG_BUF_SHIFT` in your kernel configuration. You will also need `CONFIG_PRINTK_TIME` and `CONFIG_KALLSYMS`.

Kernel boot graph

With initcall_debug, you can generate a boot graph making it easy to see which kernel initialization functions take most time to execute.

- Copy and paste the output of the dmesg command to a file (let's call it boot.log)
- On your workstation, run the scripts/bootgraph.pl script in the kernel sources:
  scripts/bootgraph.pl boot.log > boot.svg
- You can now open the boot graph with a vector graphics editor such as inkscape:



Using the kernel boot graph (1)

Start working on the functions consuming most time first. For each function:

- Look for its definition in the kernel source code. You can use Elixir (for example `https://elixir.bootlin.com`).
- Be careful: some function names don't exist, the names correspond to *modulename*_`init`. Then, look for initialization code in the corresponding module.
- Remove unnecessary functionality:
  - Find which kernel configuration parameter compiles the code, by looking at the Makefile in the corresponding source directory.

Using the kernel boot graph (2)

- Postpone:
  - Find which module (if any) the function belongs to. Load this module later if possible.
- Optimize necessary functionality:
  - Look for parameters which could be used to reduce probe time, looking for the `module_param` macro.
  - Look for delay loops and calls to functions containing `delay` in their name, which could take more time than needed. You could reduce such delays, and see whether the code still works or not.

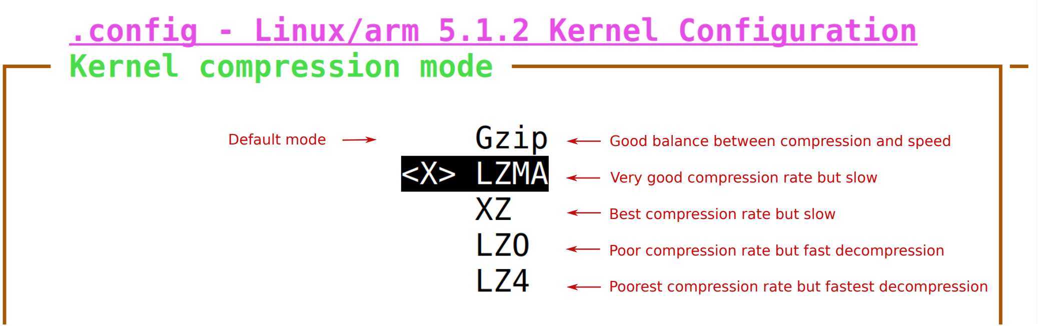
Reduce kernel size

First, we focus on reducing the size without removing features

- The main mechanism is to use kernel modules
- Compile everything that is not needed at boot time as a module
- Two benefits: the kernel will be smaller and load faster, and less initialization code will get executed
- Remove features that are not used by userland: `CONFIG_KALLSYMS`, `CONFIG_DEBUG_FS`, `CONFIG_BUG`
- Use features designed for embedded systems: `CONFIG_SLOB`, `CONFIG_EMBEDDED`

Kernel Compression

Depending on the balance between your storage reading speed and your CPU power to decompress the kernel, you will need to benchmark different compression algorithms. Also recommended to experiment with compression options at the end of the kernel optimization process, as the results may vary according to the kernel size.



Kernel compression options

Results on TI AM335x (ARM), 1 GHz, Linux 5.1

Timestamp	gzip	lzma	xz	lzo	lz4
Size	2350336	1777000	**1720120**	2533872	2716752
Copy	0.208 s	0.158 s	**0.154 s**	0.224 s	0.241 s
Time to userspace	1.451 s	2.167 s	1.999s	**1.416 s**	1.462 s

Gzip is close. It's time to try with faster storage (SanDisk Extreme Class A1)

Timestamp	gzip	lzma	xz	lzo	lz4
Size	2350336	1777000	**1720120**	2533872	2716752
Copy	0.150 s	0.114 s	**0.111 s**	0.161 s	0.173 s
Time to userspace	1.403 s	2.132 s	1.965 s	**1.363 s**	1.404 s

Lzo and Gzip seem the best solutions. Always benchmark as the results depend on storage and CPU performance.

Optimize kernel for size (1)

- CONFIG_CC_OPTIMIZE_FOR_SIZE: possibility to compile the kernel with gcc -Os instead of gcc -O2.
- Such optimizations give priority to code size at the expense of code speed.
- Results: the initial boot time is better (smaller size), but the slower kernel code can offset the benefits. Your system will run a bit slower!

Optimize kernel for size (2)

Results on BeagleBone Black, Linux 5.1, lzo compression

	O2	Os	Diff
Size	2533872	2390608	-5.7 %
Copy time	0.161 s	0.153 s	-8 ms
Starting kernel	0.912 s	0.904 s	-8 ms
Starting userspace	1.363 s	1.359 s	-4 ms
Total boot time	2.961 s	2.957s	-4 ms

Results on Microchip SAMA5D3 Xplained, Linux 3.10, gzip compression:

Timestamp	O2	Os	Diff
Starting kernel	4.307 s	4.213 s	-94 ms
Starting init	5.593 s	5.549 s	-44 ms
Login prompt	21.085 s	22.900 s	+ 1.815 s

Deferring drivers and initcalls

- If you can't compile a feature as a module (e.g. networking or block subsystem), you can try to defer its execution.
- Your kernel will not shrink but some initializations will be postponed.
- Typically, you would modify `probe()` functions to return `-EPROBE_DEFER` until they are ready to be run.
- See `https://lwn.net/Articles/485194/` for details about the infrastructure supporting this.

Turning off console output

- Console output is actually taking a lot of time (very slow device). Probably not needed in production. Disable it by passing the `quiet` argument on the kernel command line.
- You will still be able to use `dmesg` to get the kernel messages.
- Time between starting the kernel and starting the `init` program, on Microchip SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
Without `quiet`	2.352 s	
With `quiet`	1.285 s	-1.067 s

- Less time will be saved on a reduced kernel, of course.
- Don't do it too early if you're using `grabserial`

Preset loops per jiffy

- At each boot, the Linux kernel calibrates a delay loop (for the `udelay()` function). This measures a number of loops per jiffy (*lpj*) value. You just need to measure this once! Find the `lpj` value in the kernel boot messages:

```
Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

- Now, you can add `lpj=<value>` to the kernel command line:

```
Calibrating delay loop (skipped) preset value.. 996.14 BogoMIPS (lpj=4980736)
```

- Tests on BeagleBone Bloack (ARM), Linux 5.1: -82 ms
  Time measured at the first kernel messages... the calibration loop is run before the message is issued.

Multiprocessing support (CONFIG_SMP)

- SMP is quite slow to initialize
- It is usually enabled in default configurations, even if you have a single core CPU (default configurations should support multiple systems).
- So make sure you disable it if you only have one CPU core.
- Results on BeagleBone Black:
  Compressed kernel size: -188 KB

Kernel: last milliseconds (1)

To shave off the last milliseconds, you will probably want to remove unnecessary features:

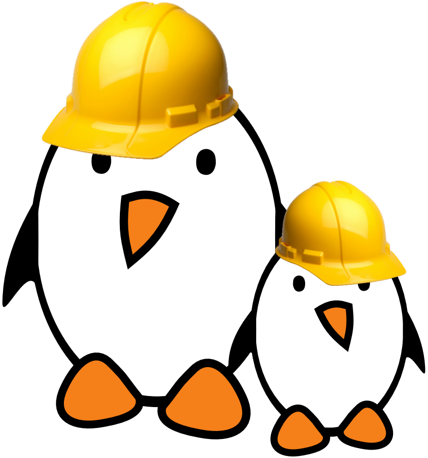
- `CONFIG_PRINTK=n` will have the same effect as the `quiet` command line argument but you won't have any access to kernel messages. You will have a significantly smaller kernel though.
- Compile your kernel in *Thumb2* mode: `CONFIG_THUMB2_KERNEL` (any ARM toolchain can do that).

Kernel last milliseconds (2)

More features you could remove:

- Module loading/unloading
- Block layer
- Network stack
- USB stack
- Power management features
- `CONFIG_SYSFS_DEPRECATED`
- Input: keyboards / mice / touchscreens
- Reduce the `CONFIG_LEGACY_PTY_COUNT` value or set the `pty.legacy_count` kernel parameter

Practical lab - Reduce kernel boot time



- Recompile the kernel, switching to an initramfs
- Use `initcall_debug` to find the biggest time consumers
- Remove unused features and drivers
- Tune kernel command line parameters

Bootloader optimizations

bootlin

embedded Linux and kernel engineering

Bootloader

- Remove unnecessary functionality.
  Usually, bootloaders include many features needed only for development. Compile your bootloader with less features.
- Optimize required functionality.
  Tune your bootloader for fastest performance.
  Skip the bootloader and load the kernel right away.

U-Boot - Remove unnecessary functionality

Recompile U-Boot to remove features not needed in production

- Disable as many features as possible in `include/configs/<soc>-<board>.h`
- Examples: MMC, USB, Ethernet, dhcp, ping, command line edition, command completion...
- A smaller and simpler U-Boot is faster to load and faster to initialize.

U-Boot - Remove the boot delay

- Remove the boot delay:
  `setenv bootdelay 0`
- This usually saves several seconds!
- Before you do that, recompile U-Boot with `CONFIG_ZERO_BOOTDELAY_CHECK`, documented in `doc/README.autoboot`. It allows to stop the autoboot process by hitting a key even if the boot delay is set to `0`.

U-Boot - Simplify scripts

Some boards have over-complicated scripts:

Running nested scripts

```
bootcmd=run bootf0
bootf0=run ${args0}; setenv bootargs ${bootargs} \
maximasp.kernel=maximasp_nand.0:kernel0; nboot 0x70007fc0 kernel0
```

Let's replace this by:

```
setenv bootargs 'mem=128M console=tty0 consoleblank=0
console=ttyS0,57600 \
mtdparts=maximasp_nand.0:2M(u-boot)ro,512k(env0)ro,512k(env1)ro,\
4M(kernel0),4M(kernel1),5M(kernel2),100M(root0),100M(root1),-(other)\
rw ubi.mtd=root0 root=ubi0:rootfs rootfstype=ubifs earlyprintk debug \
user_debug=28 maximasp.board=EEKv1.3.x \
maximasp.kernel=maximasp_nand.0:kernel0'
setenv bootcmd 'nboot 0x70007fc0 kernel0'
```

This saved 56 ms on this ARM9 system (400 MHz)!

Bootloader: copy the exact kernel size

- When copying the kernel from flash to RAM, we still see many systems that copy too many bytes, not taking the exact kernel size into account.
- In U-Boot, use the `nboot` command:
  `nboot ramaddr 0 nandoffset`
- U-Boot using the kernel size information stored in the `uImage` header to know how many bytes to copy.

U-Boot - Optimize kernel loading

- After copying the kernel `uImage` to RAM, U-Boot always moves it to the load address specified in the `uImage` header.
- A CRC check is also performed.

```
[16.590578 0.003404] ## Booting kernel from Legacy Image at 21000000 ...
[16.595204 0.004626]    Image Name:   Linux-3.10.0+
[16.597986 0.002782]    Image Type:   ARM Linux Kernel Image (uncompressed)
[16.602881 0.004895]    Data Size:    3464112 Bytes = 3.3 MiB
[16.606542 0.003661]    Load Address: 20008000
[16.608903 0.002361]    Entry Point:  20008000
[16.611256 0.002353]    Verifying Checksum ... OK
[17.134317 0.523061] ## Flattened Device Tree blob at 22000000
[17.137695 0.003378]    Booting using the fdt blob at 0x22000000
[17.141707 0.004012]    Loading Kernel Image ... OK
[18.005814 0.864107]    Loading Device Tree to 2bb12000, end 2bb1a0b6 ... OK
```

Kernel CRC check time

Kernel memmove time

U-Boot - Remove unnecessary memmove (1)

- You can make U-Boot skip the `memmove` operation by directly loading the `uImage` at the right address.
- Compute this address:

```
Addr = Load Address - uImage header size
Addr = Load Address - (size(uImage) - size(zImage))
Addr = 0x20008000 - 0x40 = 0x20007fc0
```

```
[16.590927 0.003407] ## Booting kernel from Legacy Image at 20007fc0 ...
[16.595547 0.004620]    Image Name:   Linux-3.10.0+
[16.598351 0.002804]    Image Type:   ARM Linux Kernel Image (uncompressed)
[16.603228 0.004877]    Data Size:    3464112 Bytes = 3.3 MiB
[16.606907 0.003679]    Load Address: 20008000
[16.609256 0.002349]    Entry Point:  20008000
[16.611619 0.002363]    Verifying Checksum ... OK
[17.135046 0.523427] ## Flattened Device Tree blob at 22000000
[17.138589 0.003543]    Booting using the fdt blob at 0x22000000
[17.142575 0.003986]    XIP Kernel Image ... OK
[17.156358 0.013783]    Loading Device Tree to 2bb12000, end 2bb1a0b6 ... OK
```

Kernel CRC check time
Kernel `memmove` time (skipped)

U-Boot - Remove unnecessary memmove (2)

Results on Microchip SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
Default	1.433 s	
Optimum load address	0.583 s	-0.85 s

Measured between `Booting kernel` and `Starting kernel ...`

U-Boot - Remove kernel CRC check

- Fine in production when you never have data corruption copying the kernel to RAM.
- Disable CRC checking with a U-boot environment variable:
  `setenv verify no`

Results on Microchip SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
With CRC check	583 ms	
Without CRC check	60 ms	-523 ms

Measured between `Booting kernel` and `Starting kernel ...`

Further U-Boot optimizations

- Silence U-Boot console output. You will need to compile U-Boot with `CONFIG_SILENT_CONSOLE` and `setenv silent yes`.
  See `doc/README.silent` for details.

- Principle: instead of loading the bootloader and then the kernel, load the kernel right away!
- For example, on Microchip AT91, is is easy to implement with `at91bootstrap v3`. You just need to configure it with one of the `linux` or `linux_dt` configurations:

```
make at91sama5d3xeknf_linux_dt_defconfig
make
```

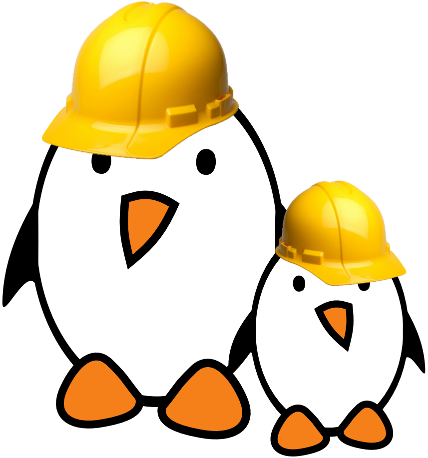
Full details on https://bootlin.com/blog/starting-linux-directly-from-at91bootstrap3/

Skipping the bootloader - U-Boot Falcon mode

A generic solution!

- U-Boot is split in two parts: the SPL (Secondary Program Loader) and the U-Boot image. U-Boot can then configure the SPL to load the Linux kernel directly, instead of the U-Boot image.
  See `doc/README.falcon` for details and `http://schedule2012.rmll.info/IMG/pdf/LSM2012_UbootFalconMode_Babic.pdf` for the original presentation.
- This is supported in the same way on all the boards with U-Boot support for SPL.

Practical lab - Reduce bootloader time



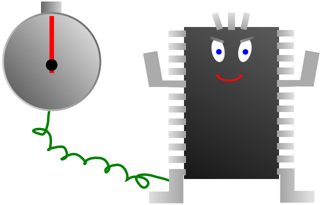
- Reduce boot time by compiling U-Boot with less features
- Optimizing the way U-Boot is used

Hardware initialization

Hardware initialization

The hardware needs time to initialize

- Voltage regulation, crystal stabilization
- Can be up to 200 ms
- As a software developer, you can't do anything about this part.
- All you can do is measure this time with an oscilloscope and ask the hardware board designers whether the can do anything about this. However, there are delays in the CPU which may not be possible to reduce (see the CPU datasheet).



References

Conference presentations

- Andrew Murray - The Right Approach to Minimal Boot Time (2010)
  Video: https://frama.link/nrf696Hy - Slides: https://frama.link/uCBH9jQM
  Great talk about the methodology.
- Chris Simmonds - A Pragmatic Guide to Boot-Time Optimization (2017)
  Video: https://frama.link/Vnmj5t1m - Slides: https://frama.link/TC0YKM9N
- Jan Altenberg - How to Boot Linux in One Second (2015)
  Video: https://frama.link/BztbLy9T - Slides: https://frama.link/bFkvgLFR
- Michael Opdenacker - Embedded Linux size reduction techniques (2017) Video: https://youtu.be/ynNLlzOElOU - Slides: https://frama.link/fS5gQZZq