Can we boot Linux from just a floppy?

An exercise in minimalism.

Kamila Szewczyk, *Association for Computational Heresy*

19 Sesja Linuksowa, Wroclaw, April 2025

Introduction

- Some common storage media.

(a) CD-ROM

(b) USB drive

(c) SD Card

- Their capacities are usually overwhelming and measured in gigabytes (10^9 bytes).
- All of them constitute popular boot and installation media for Linux.
- Do we really need this much storage just to boot an operating system?

Introduction

- Prior art:
  - Damn Small Linux (DSL) - 50MB - A live CD distribution for 486+ CPUs, 8MB of RAM.
  - Tiny Core Linux - 17MB - Perhaps the most popular minimalist distribution.
  - NanoLinux - 14MB - Discontinued project.
- All of those contenders are quite small and still provide many interesting features like web browsers, text editors or even games.
- They also lag behind in kernel versions.
- What if we didn't want *any* of the nice features, and simply wanted a minimal TTY environment that still formally runs Linux?

Introduction

- As a goalpost of our experiment, we will use this magical device:

Figure: A 3.5" floppy disk.

- Each of these holds about 1.44MB of data, about 0.2% of a standard 700MB CD-ROM.
- The best widespread contender, Tiny Core Linux, needs at least 10 of these to fit.

Toolchain

- It is clear that we will have to compile the kernel and the userland ourselves.
- Because 32-bit x86 code tends to be smaller than its 64-bit counterpart, we will build a i486+ kernel.
  - This has some more advantages: we turn off fragments of the kernel code responsible for complicated hardware instructions that the 486 simply doesn't have.
- But we don't want just *any* 486 ELF toolchain: `glibc` is much too bulky for our needs.
- Instead, we will use `musl`, a lightweight C library, which tends to be much smaller and compatible with `glibc` on many counts.
- Equipped with `i486-linux-musl-gcc`, we may continue.
- In case you want to follow along, you can get this from `https://musl.cc/`.

Toolchain: experiment

```
% cat hello.c
#include <stdio.h>
int main(void)  puts("Hello, world!"); for (;;) ;
% i486-linux-musl-gcc hello.c -o hello -Os -static -s
% wc -c hello && ./hello
4948 hello
Hello, world!
% ldd hello
  not a dynamic executable
```

- The resulting 32-bit program only uses system calls and does not depend on any shared libraries.
- 5KB for a simple program is not fantastic, but it is a good start.
- Let's not get off track. We will get back to that later.

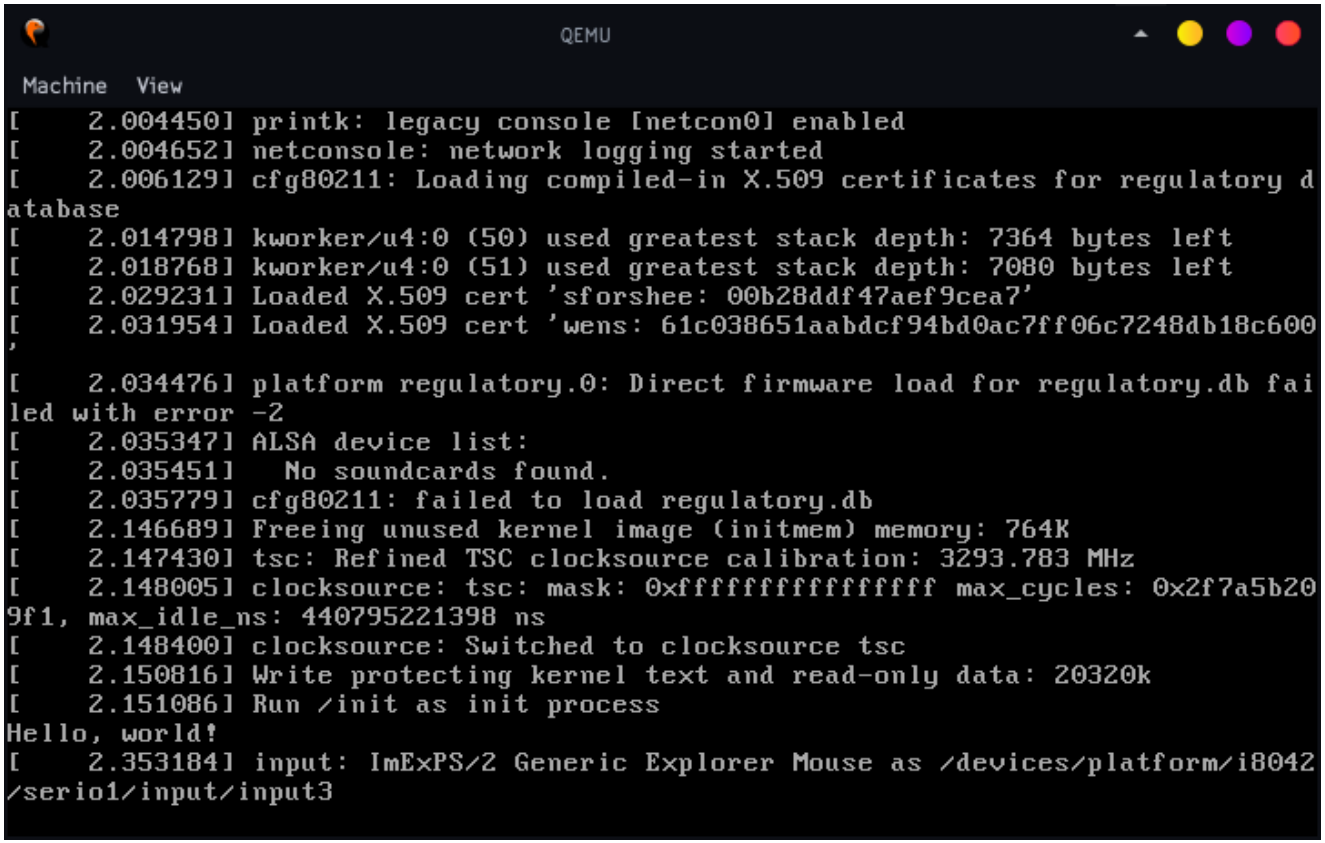
Kernel & initrd

- Let's try to get something booting first.
- The specific kernel that we will be working with is `linux-6.13.8` - most recent version at the time of writing.
- Compiling the kernel by itself is pretty basic nitty-gritty.
  - Use `arch/x86/configs/i386_defconfig` as a base.
  - Enable TTY and `printk` support (for the latter you need expert settings enabled).
  - Then:
    `make ARCH=x86 CROSS_COMPILE=i486-linux-musl- bzImage`.
- Before we can compile and boot the kernel, we need to prepare the initrd.
- The *Hello, world!* program is a good initial candidate: together with a small bootloader, it will leave us with a lot of space for the kernel.
- We put the init program in a CPIO archive, which is a standard format for initrd images.

Kernel & initrd

- Boot with the following command:

```
qemu-system-i386 -kernel arch/x86/boot/bzImage \
                 -initrd ./rootfs.cpio.gz
```



Kernel & initrd

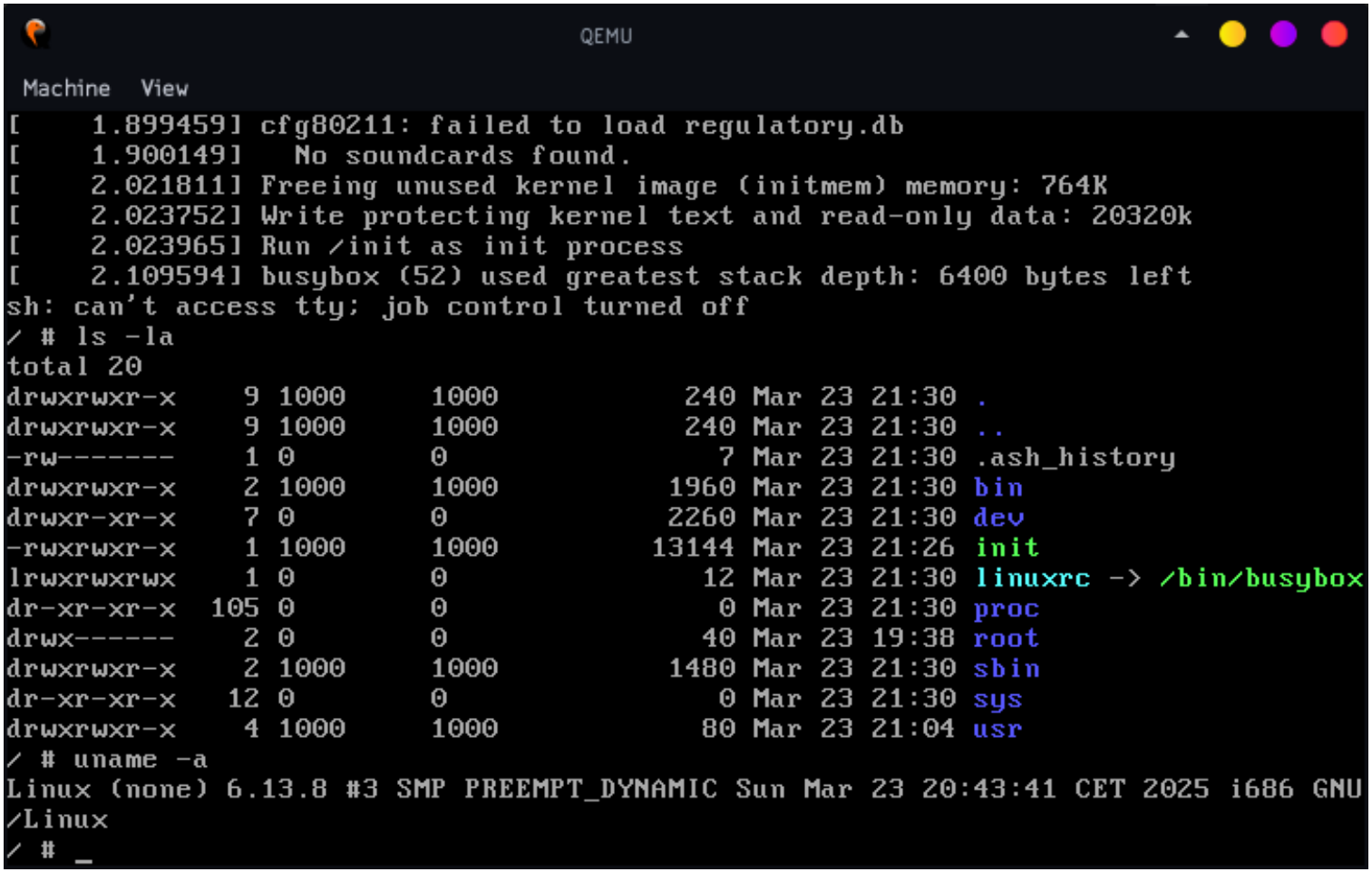
- Excellent! We have a working kernel and initrd.
- Bad news: the kernel image is 10.8 MiB.
- Before we start trying to remove stuff from the kernel, let's try to build a functional initrd using BusyBox. Then it's easier to tell if the stuff we are removing is actually used by something.
- Specifically, we will use BusyBox 1.35.0. The binary is quite large, at around 1.0 MiB, but it is statically linked and contains a lot of useful tools.

Kernel & initrd

- `init` program: install BusyBox, mount `devtmpfs` and run `/bin/init`.

```c
int main(int argc, char *argv[]) {
  pid_t pid = fork();
  if (pid == 0) {
    char * args[] = { "/bin/busybox", "--install", "-s", NULL };
    execv(args[0], args);
    perror("execv failed"), exit(1);
  } else if (pid < 0)
    perror("fork failed"), exit(1);
  int status; waitpid(pid, &status, 0);
  if (!WIFEXITED(status))
    perror("waitpid"), exit(1);
  if (mount("none", "/dev", "devtmpfs", 0, "")) perror("mount"), exit(1);
  if (mount("none", "/proc", "proc", 0, "")) perror("mount"), exit(1);
  if (mount("none", "/sys", "sysfs", 0, "")) perror("mount"), exit(1);
  // /bin/init is dmesg -n 1;exec sh
  char * args[] = { "/bin/busybox", "sh", "/bin/init", NULL };
  execv(args[0], args);  perror("execv"); exit(1);
}
```

Kernel & initrd



Stripping the kernel

- We have a Linux system that fits in about 11 MiB. The CPIO with the `initrd` is about 699 KiB: we will have to look into that, because surely we can improve on it.
- The next two steps are compressing these two components well enough to fit in a total of 1.44 MB. We will start with the kernel, as it's the biggest offender.
  - First step: Use XZ compression, disable the support for all other codecs except `gzip` for `initrd`, compile with `-Os`.
  - These changes alone bring the kernel down to about 6.7 MiB.
- Unfortunately, together with the initrd we are still quite far away from our goal:

```
6763008 arch/x86/boot/bzImage
 669206 rootfs.cpio.gz
7432214 total
```

Stripping the kernel

- We will squeeze the kernel as much as possible by removing device drivers, filesystems, networking features, debugging options, and other features that we don't need.
- Further, the build script supports embedding the initrd in the kernel image. We will use `xz` compression (keeping in mind that we need `--check=crc32` for the kernel to boot).
- The result? `1282560 arch/x86/boot/bzImage` - the kernel + initrd fit in 1.28MB, much below the 1.44MB limit.

Stripping the kernel

- The result? `1282560 arch/x86/boot/bzImage` - the kernel + initrd fit in 1.28MB, much below the 1.44MB limit.
- That's very nice, but we can't really do anything with our operating system other than idling in the shell.
- I personally don't like `vi` and would prefer a different text editor. Also, I would like to put some cool programs in the initramfs to play with - that would be best achieved with a dynamically linked libc.
- Also, we haven't actually figured out how to make this boot off an actual floppy (yet).

Goals

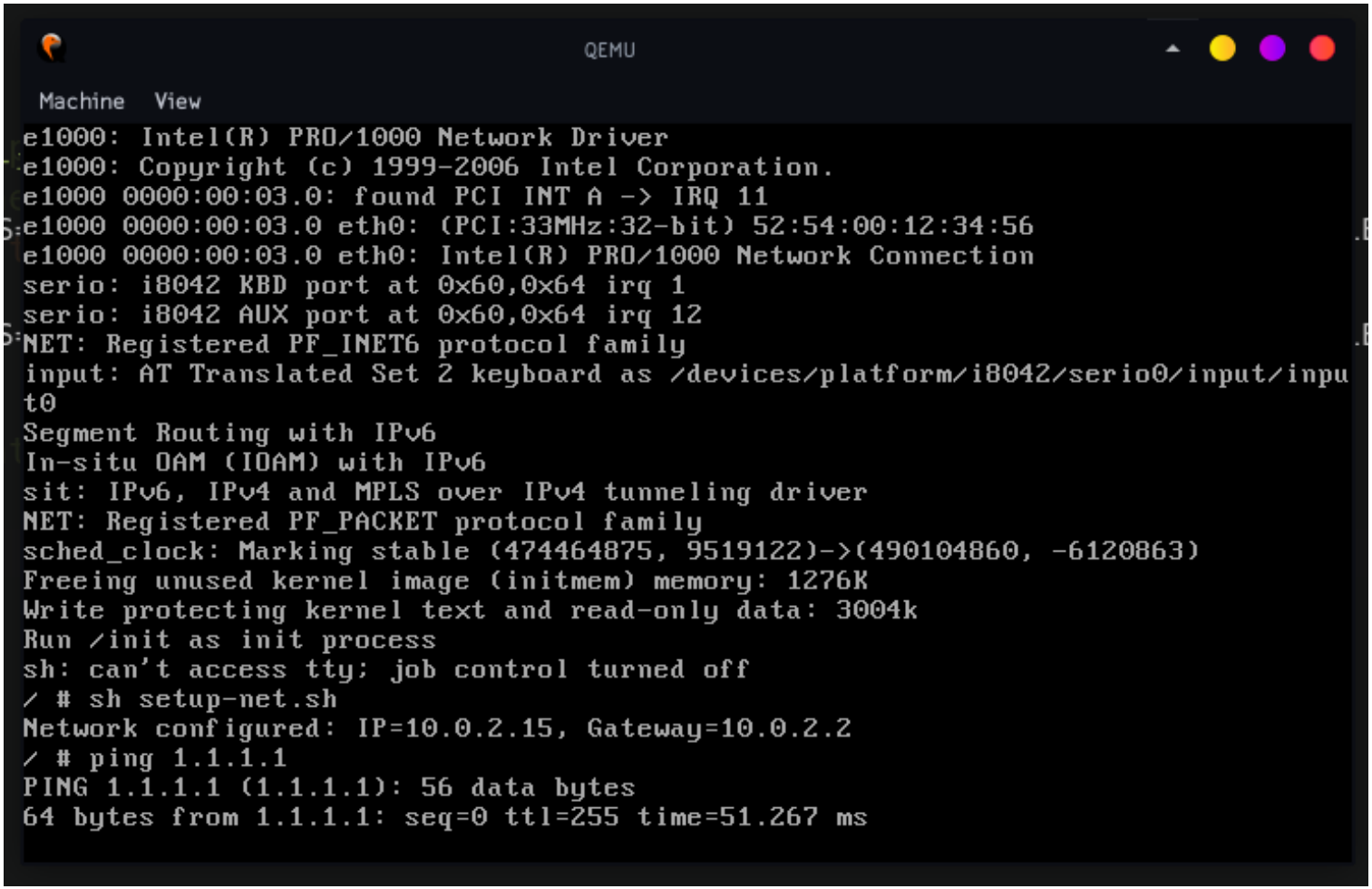
- Now that we have a MVP, we want to accomplish the following:
  1. Build a bootable floppy disk instead of relying on `qemu-system-i386 -kernel arch/x86/boot/bzImage` to boot.
  2. Strip down busybox to remove things that we won't need. For example user management, various mkfs/fsck programs for filesystems that we don't support.
  3. Add more programs to the initramfs and features to the kernel: interpreters, compilers, games, etc.
- Of course, eventually we will run out of space. Then we will look into custom compression.

Adding networking

- We will enable networking, E1000 support, the TCP/IP stack and the basic security features.
- This unfortunately makes the kernel swell up quite a bit.
- Configuring networking with busybox is tricky. I wrote the following script:

```
#!/bin/sh
ip link set eth0 up
LEASE=$(udhcpc -i eth0 -n 2>&1)
IP=$(echo "$LEASE" | awk '/lease of/ {print $4}')
GATEWAY=$(echo "$LEASE" | awk '/obtained from/ {print $7}' | tr -d ',')
if [ -n "$IP" ] && [ -n "$GATEWAY" ]; then
  ip addr flush dev eth0
  ip addr add "$IP/24" dev eth0
  ip route add default via "$GATEWAY"
  echo "Network configured: IP=$IP, Gateway=$GATEWAY"
else
  echo "Failed to obtain an IP address via DHCP." && exit 1
fi
```

Networking


QEMU
Machine View
e1000: Intel(R) PRO/1000 Network Driver
e1000: Copyright (c) 1999-2006 Intel Corporation.
e1000 0000:00:03.0: found PCI INT A -> IRQ 11
e1000 0000:00:03.0 eth0: (PCI:33MHz:32-bit) 52:54:00:12:34:56
e1000 0000:00:03.0 eth0: Intel(R) PRO/1000 Network Connection
serio: i8042 KBD port at 0x60,0x64 irq 1
serio: i8042 AUX port at 0x60,0x64 irq 12
NET: Registered PF_INET6 protocol family
input: AT Translated Set 2 keyboard as /devices/platform/i8042/serio0/input/input0
Segment Routing with IPv6
In-situ OAM (IOAM) with IPv6
sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
NET: Registered PF_PACKET protocol family
sched_clock: Marking stable (474464875, 9519122)->(490104860, -6120863)
Freeing unused kernel image (initmem) memory: 1276K
Write protecting kernel text and read-only data: 3004k
Run /init as init process
sh: can't access tty; job control turned off
/ # sh setup-net.sh
Network configured: IP=10.0.2.15, Gateway=10.0.2.2
/ # ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1): 56 data bytes
64 bytes from 1.1.1.1: seq=0 ttl=255 time=51.267 ms

Checkpoint: list of features

- `printk` for debugging, ISA bus support, R/W block layer, compacting memory manager, PCI bus support.
- SATA/ATA, SCSI, E1000 (generic network card interface) support, IPv4 networking, DNS resolution, FAT32 file systems.
- BusyBox with a DHCP network setup script, `dmesg`, `wget`, `vi`, `sh`, `awk`, `dc`, `bc`, `httpd`, and so on.
- Caveat: the XZ image is 1'835'520. We will implement a custom compression algorithm to fit this in a floppy.

Kernel Compression

- Basic idea: we put the initrd and kernel in the same logical container, which is then bootstrapped by `mkpiggy` (a custom program that Linux uses for compressed kernels).
- Nominally, this is simple on paper: we start with some existing compression wrapper (e.g. XZ), blank out the source code, change the makefile to use a different compressor, then embed the decompressor in the kernel.
- Issues:
  - We are looking for a somewhat fast statistical compressor that can beat XZ quite significantly.
  - We would prefer it to not use gobs of memory and to be able to run on a 486.
  - It needs to fit in the piggyback module together with the compressed kernel and bootloader, i.e. it can not be too large.

Kernel Compression

Kernel Compression

- We will reuse some common ideas: PAQ-like context-adaptive binary arithmetic coding, a context model with a rudimentary context mixer between finite order context models and a match model that seldom disables it.
- The mixer output is filtered by a chain of a few APMs. The input is filtered by an E8E9 transform together with a simple disassembly pass that recognises prefixes and opcodes.
- The biggest issue is integrating this with the Linux kernel, which is somewhat unfamiliar to me as a C code base.
- If you want to know what any of this means, it will probably be explained in my compression book that I talked about here about a year ago.

Kernel Compression

- To not go into too much detail, the decompression stub was easily produced in a few hours.
- It's somewhat fast (20s to decode the kernel) on my machine.
- Most importantly, the kernel now fits into 1.44MB!

```
1413632 arch/x86/boot/bzImage
```

- Two more goal posts to score: a standalone boot loader ($\Rightarrow$ a working bootable floppy image) and some cool stuff in the user land if we can squeeze it in.

The Bootloader

- Written from scratch in x86 assembly, about 300 lines of code.
- Core difficulties are:
  - Enabling the A20 gate (requires some trickery because of how many disjoint methods there are - via BIOS, via keyboard, etc).
  - Operating in Unreal Mode so we can use BIOS interrupts to move stuff from conventional memory containing sectors read to a high memory address where the kernel expects to be loaded.
  - Loading the GDT and properly invoking the kernel, but that's easy.
- It ends up taking less than two sectors in total even with my sloppy and rushed programming.

End Result

- Direct download (1.44M floppy image):
  - `https://palaiologos.rocks/doc/projects/floppy-linux.img`
- This also version features `fasm` (Flat Assembler by Tomek Grysztar) in the initrd.
- Adding an alternative program that is about 120KB in size was also possible, but it quite like `fasm` and don't see the point in doing that.

End Result: Full list of features (I)

- *All* of BusyBox; abbreviated list below:
  - Shell utilities: the `ash` shell, the `hush` shell, `base32`, `base64`, `chattr`, `chmod`, `cp`, `date`, `dd`, `df`, `egrep`, `fgrep`, `getopt`, `iostat`, `kill`, `ln`, `mknod`, `mktemp`, `mount`, `more`, `less`, `netstat`, `nice`, `pipe-progress`, `ps`, `rm`, `sync`, `bc`, `cal`, `crc32`, `dc`, `diff`, `du`, `expand`, `factor`, `find`, `fold`, `hexdump`, `install`, `md5sum`, `nproc`, `openvt`, `pgrep`, `printf`, `pscan`, `pstree`, `script`, `seq`, `sha*sum`, `shuf`, `sort`, `split`, `strings`, `tail`, `tee`, `uniq`, `man`.
  - Compression/archiving: `cpio`, `gzip`, `gunzip`, `lzop`, `rpm`, `tar`, `vi`, `ar`, `bunzip2`, `bzip2`, `dpkg`, `lzcat`, `lzma`, `unzip`, `xz`.
  - System management: `dmesg`, `setfont`, `powertop`, `crond`, `crontab`, `lsof`, `lspci`.
  - Networking: `ping`, `udhcpc`, `tftpd`, `telnet`, `sendmail`, `ntpd`, `inetd`, `httpd`, `ftpd`, `arping`, `ftpget`, `ftpput`, `nc`, `traceroute`, `wget`.
  - Editors: `ed`, `sed`, `awk`, `vi`, `hexedit`, `patch`.

End Result: Full list of features (II)

- Kernel: ATA, SATA, PCI, IDE support; MS-DOS filesystems (read/write), mitigations.
- SCSI, E1000 (generic network card interface) support, IPv4 networking, DNS resolution.
- Notably no IPv6 or EXT4 support (they're too bulky).
- Needs at least 70MB of RAM for the decompressor. Not that uncommon among i486/i586 machines.
- Games: 2048, `chess`; Demos: `donut`.

Live demo!

Hacking

- Future ideas:
  - Strip down the BusyBox a bit more to make some space in the initrd for cooler stuff.
  - Add a `.profile` and a cool rice to the floppy disk Linux distribution.
- Programs to bundle:
  - Terminal games.
  - Interpreters and compilers.
  - System rescue tools?
- Package manager:
  - Download a platform toolchain (`https://skarnet.org/toolchains/native/i486-linux-musl_i486-14.2.0.tar.xz`), compile source packages.

Thanks!

- `https://palaiologos.rocks`
- `https://github.com/kspalaiologos`
- `mailto:kszewczyk@acm.org`
- I hope that you liked this talk. Unfortunately due to real life circumstances and my university obligations - I am currently working on a high-profile thesis about genome compression - I started working on these slides and the code, from scratch, only on Monday 24/03 and was finished by Wednesday 26/03.
- This explains why the decompression performance is not top notch and the boot sector is holey.