IoT: Client Devices

Daemons in C

OS Services

Kernel Services, OS Services

- Daemons have access to Kernel and OS services
- Networking, system logging, processes, filesystem

Things daemons don't have

- A console!
- A user!
- A home directory!
- User interaction!

Console Mangement

DAEMONS DON'T HAVE A CONSOLE

- So what do stderr, stdout, and stdin mean?
- Nothing - you need to manage

CLOSE STANDARD FILE DESCRIPTORS

```
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
```

Signal Management

Controlling interactive programs

- Unixes give ctrl-c, ctrl-z, etc.
- Not daemons!
- Remember, these just submit *signals to processes* (see: **man kill**)

Signal Management

```
signal(SIGKILL, _signal_handler);
signal(SIGTERM, _signal_handler);
signal(SIGHUP,  _signal_handler);
```

Logging

No console -> no standard output

- Many programs will log to STDERR, or STDOUT
- But we closed them!

Syslog

- Syslog is a systems-wide logger
- /var/log/messages or /var/log/syslog

Open a log and log to it

- openlog(.), syslog(.), closelog()
- see: **man syslog**

Working Directory

NO USER -> NO DEFAULT WORKING DIRECTORY

- We need a working directory
- We do handle files

MOVING AND SETTING A WORKING DIRECTORY

```
chdir(WORKING_DIR);
```

File Creation

We do create files

- Need to set default permissions on created files
- Usually files only read by privileged users
- In our case, better to leave open

Setting default permissions

```
umask(S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)
```

Sessions

A *Session* has one or more *process groups*

- The first process in the session is default session leader

A *process group* has one or more *Processes*

- The group leader and child processes

Think of sessions as *Terminal Sessions*

`setsid()`

Forking

DON'T LOCK UP SPAWNING PROCESS

- Daemons need their own dedicated processes
- forking a process creates a copy of the program in another process
- Parent process gets PID; child gets 0; error is negative

USING FORK()

```
PID_T PID = FORK();
IF (PID > 0) EXIT(0);
IF (PID < 0) EXIT(1);
```

IoT: Client Devices

A Sample Daemon

```c

int main(void) {
  // Open syslog. We want to write directly to the log with
  // no wait, and we want to include the PID of the daemon process.
  // We are running a daemon, so set the LOG_DAEOMON flag too.
  //
  // Why open here? well, if we have an error forking, we want to
  // register the error to syslog. To do that, we need an open log.
  // The child process will inherit this log, as we don't close it
  // in the parent when it exits, this opening here is A+.
  openlog(DAEMON_NAME, LOG_PID | LOG_NDELAY | LOG_NOWAIT, LOG_DAEMON);
  syslog(LOG_INFO, "staring sampled");

  // We really don't want to take over syslogd or initd, so
  // fork.
  pid_t pid = fork();

  // Well something went wrong. fork() uses standard unix
  // errno functionality, so let's log the problem and exit.
  if (pid < 0) {
NORMAL  master  ./sample_daemon.c        unix  utf-8  c  58%  70:1
```

Getting Started

```c
// We really don't want to take over syslogd or initd, so
// fork.
pid_t pid = fork();

// Well something went wrong. fork() uses standard unix
// errno functionality, so let's log the problem and exit.
if (pid < 0) {
  syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
  return ERR_FORK;
}

// We receive a PID if we're the parent process. If we're then
// parent, let's just exit. We only care about the forked child.
if (pid > 0) {
  return OK;
}

// I'd like to the the leader of the session. If I can't,
// I'm out.
if(setsid() < -1) {
```

NORMAL master ./sample_daemon.c unix utf-8 c 69% 83:1

Forking

```c

// I'd like to the the leader of the session. If I can't,
// I'm out.
if(setsid() < -1) {
  syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
  return ERR_SETSID;
}

// We have console to write to anymore, so these file pointers
// are just silly. Close them.
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

// Are we creating files? Well, not in this example, but let's
// go ahead and set a sane UMASK anyway. This will give us
// read/write, and everybody else gets read permissions.
umask(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

// What's our working directory going to be? Let's make it the root
```

NORMAL master ./sample_daemon.c unix utf-8 c 66% 80:1

Sessions and Output

```c
umask(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

// What's our working directory going to be? Let's make it the root
// directory just to make navigation easier if we needed it.
if (chdir("/") < 0) {
  syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
  return ERR_CHDIR;
}

// Now, we can only be controlled by signals as we have no interactive
// user session. So let's handle those signals, yeah?
signal(SIGTERM, _signal_handler);
signal(SIGHUP, _signal_handler);

// This is the daemon proces runloop. This could be a while statement,
// or some other construct, but daemons have specific responsibilities
// they need to handle, usually in a repeated way, and all that log
// would go here.
_do_work();

```

NORMAL master ./sample_daemon.c unix utf-8 c 89% 108:1

Working Directories and Signals

```c
} error_t;

// This is a signal handler. Signal is the signal passed to
// us to handle.
static void _signal_handler(const int signal) {
  switch(signal) {
    case SIGHUP:
      break;
    case SIGTERM:
      syslog(LOG_INFO, "received SIGTERM, exiting.");
      closelog();
      exit(OK);
      break;
    default:
      syslog(LOG_INFO, "received unhandled signal");
  }
}

// This is where we do the work of the daemon. This example
// just counts and sleeps. Impressive!
```

NORMAL master ./sample_daemon.c unix utf-8 c 21% 25:1

Signal Handling

```c
    syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
    return ERR_CHDIR;
  }

  // Now, we can only be controlled by signals as we have no interactive
  // user session. So let's handle those signals, yeah?
  signal(SIGTERM, _signal_handler);
  signal(SIGHUP, _signal_handler);

  // This is the daemon proces runloop. This could be a while statement,
  // or some other construct, but daemons have specific responsibilities
  // they need to handle, usually in a repeated way, and all that log
  // would go here.
  _do_work();

  // This should actually never be reached. We should be looping in
  // _do_work() until the daemon is killed by a signal, at which
  // point we exit the process.
  return ERR_WTF;
}
```

NORMAL master ./sample_daemon.c unix utf-8 c 100% 121:1

Doing Actual Things!

```c
    default:
      syslog(LOG_INFO, "received unhandled signal");
  }
}

// This is where we do the work of the daemon. This example
// just counts and sleeps. Impressive!
static void _do_work(void) {
  for(int i = 0; true; i++) {
    syslog(LOG_INFO, "iteration: %d", i);
    sleep(1);
  }
}

int main(void) {
  // Open syslog. We want to write directly to the log with
  // no wait, and we want to include the PID of the daemon process.
  // We are running a daemon, so set the LOG_DAEOMON flag too.
  //
  // Why open here? well, if we have an error forking, we want to
```

NORMAL master ./sample_daemon.c unix utf-8 c 47% 57:1

Run Loop

IoT: Client Devices

Configuring OS for Daemons

Starting Daemons

INIT, UPSTART, SYSTEMD

- Init was first, it's oldest - initially in Unix System V
- Upstart was next, used in Debian distros and Redhat
- Systemd is the most current service manager

SYSTEMD CRITICISM

- Complex, violates Unix design philosophy
- But still widely used

Our IoT system uses init.

```
/etc
# ls
fstab        hosts        mtab         passwd-      random-seed  shadow-
group        init.d       network      profile      resolv.conf  ssh
group-       inittab      os-release   profile.d    services     ssl
hostname     issue        passwd       protocols    shadow
# ls init.d
S01logging  S20urandom  S40network  S50sshd     rcK         rcS
# 
```

File Structure

init uses filenames to determine startup ordering

```
# Startup the system
::sysinit:/bin/mount -t proc proc /proc
::sysinit:/bin/mount -o remount,rw /
::sysinit:/bin/mkdir -p /dev/pts
::sysinit:/bin/mkdir -p /dev/shm
::sysinit:/bin/mount -a
::sysinit:/bin/hostname -F /etc/hostname
# now run any rc scripts
::sysinit:/etc/init.d/rcS

# Put a getty on the serial port
ttyAMA0::respawn:/sbin/getty -L  ttyAMA0 0 vt100 # GENERIC_SERIAL

# Stuff to do for the 3-finger salute
#::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
::shutdown:/etc/init.d/rcK
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
```

inittab

Things we do when the system starts up

```
# Start all init scripts in /etc/init.d
# executing them in numerical order.
#
for i in /etc/init.d/S??* ;do

     # Ignore dangling symlinks (if any).
     [ ! -f "$i" ] && continue

     case "$i" in
        *.sh)
            # Source shell script for speed.
            (
                trap - INT QUIT TSTP
                set start
                . $i
            )
            ;;
        *)
            # No sh extension, so fork subprocess.
```

rcS

Starting all services on system start

```
# Stop all init scripts in /etc/init.d
# executing them in reversed numerical order.
#
for i in $(ls -r /etc/init.d/S??*) ;do

     # Ignore dangling symlinks (if any).
     [ ! -f "$i" ] && continue

     case "$i" in
        *.sh)
            # Source shell script for speed.
            (
                trap - INT QUIT TSTP
                set stop
                . $i
            )
            ;;
        *)
            # No sh extension, so fork subprocess.
```

rcK

Stopping all services on shutdown

IoT: Client Devices

An Example Startup Script

Particular Script Format

#!/bin/sh
<stuff that always happens>
<start function>
<stop function>
<case handling args>
exit $?

```sh
#!/bin/sh

DAEMON_NAME="DAEMON_NAME"

start() {
  printf "Starting $DAEMON_NAME: "
  echo "OK"
}

stop() {
  printf "Stopping $DAEMON_NAME: "
  echo "OK"
}

restart() {
  stop
  start
}

case "$1" in
  start)
  start
  ;;
  stop)
  stop
  ;;
  restart|reload)
  restart
  ;;
  *)
  echo "Usage: $0 {start|stop|restart}"
  exit 1
esac

exit $?
```

```
Usage: ./sample_framed {start|stop|restart}
cclamb@ubuntu:~ $ ./sample_framed start
Starting DAEMON NAME: OK
cclamb@ubuntu:~ $ ./sample_framed stop
Stopping DAEMON NAME: OK
cclamb@ubuntu:~ $ ./sample_framed restart
Stopping DAEMON NAME: OK
Starting DAEMON NAME: OK
cclamb@ubuntu:~ $
```

Running the Frame

Filling Out the Frame

Let’s use our sample daemon

```
#!/bin/sh

DAEMON_NAME="sampled"

start() {
  printf "Starting $DAEMON_NAME: "
  /usr/sbin/$DAEMON_NAME
  touch /var/lock/$DAEMON_NAME
  echo "OK"
}

stop() {
  printf "Stopping $DAEMON_NAME: "
  killall $DAEMON_NAME
  rm -f /var/lock/$DAEMON_NAME
  echo "OK"
}

restart() {
  stop
  start
}

case "$1" in
  start)
  start
  ;;
  stop)
  stop
  ;;
  restart|reload)
  restart
  ;;
  *)
  echo "Usage: $0 {start|stop|restart}"
  exit 1
esac

exit $?
```

Move sampled to image

MOVE SAMPLED TO DEVICE

- scp -P 2222 sampled <username>@localhost:~/

MOVE IT TO /USR/SBIN

- …this is the location pointed to in start script

MOVE START SCRIPT TO DEVICE

- place in /etc/init.d
- I called S80sampled
- Reboot, login, and check out /var/log/messages

IoT: Client Devices

Daemon Processes

Daemon Processes

Every OS has something like this

- Called a service (windows-land) or a daemon (unix)
- I’ve seen both terms used with MacOS

Startup & shutdown

- Starts with the system, stops with the system

You’ll use these in your project!

Not a Regular Program

Operating system services

- Not all services are available (e.g. no console for output)

Follow certain conventions

- Forking, default file permissions, group affiliation

Operating system Configuration

- Linux needs to be configured to start your daemon

Languages

Multiple languages support daemons

- Python and python-daemon
- Ruby with Daemons
- Perl, bash, C

We're embedded though

- No python, perl, ruby, etc.
- We do have bash and C

Pros & Cons of C/Bash

Configuration in Bash

- (+) Well supported standard way to configure Linux startup
- (-) Really, the only supported way to configure Linux startup

Code development in C

- (+) Native to Linux/Unix
- (+) Abundance of services and libraries
- (+) Well supported via build root
- (+) Small footprint
- (-) Wordy
- (-) Complex
- (-) Powerful

When you have a hammer…

IoT: Client Devices

AXFS

Background

Developed by Jared Hulbert

- Originally at Intel, now at Numonyx, released in 2008

Designed for XIP

- Execution-in-place, very handy for mobile/embedded
- Not integrated into Linux kernel well
- Clumsy patch for XIP in common use with CRAMFS

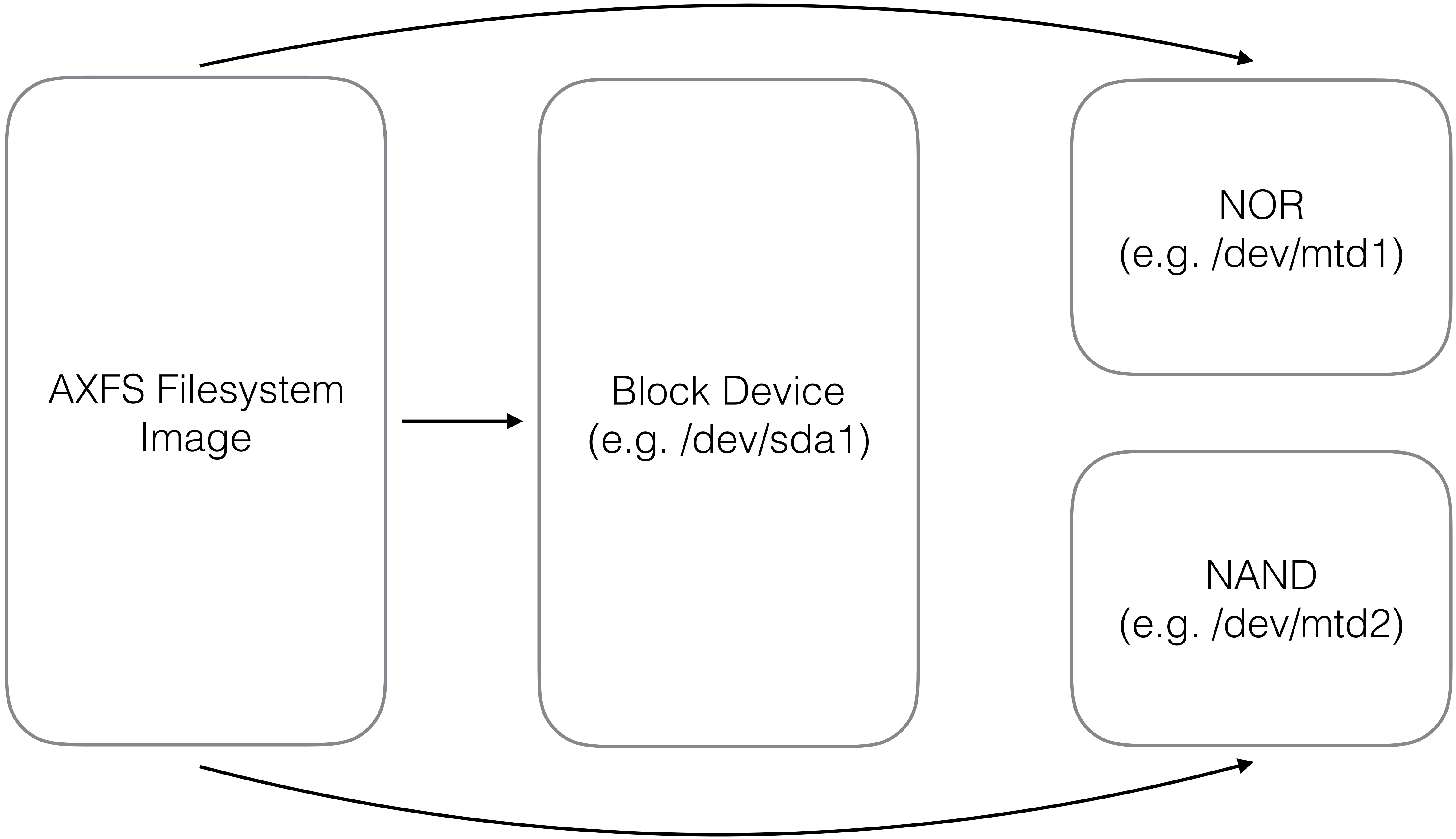
Flash and XIP

So why XIP?

- Mobile and embedded systems are not desktop
- Flash memory is fast, disk is slow
- Running programs directly from Flash is possible
- Saves space, speeds up loading

Obstacles

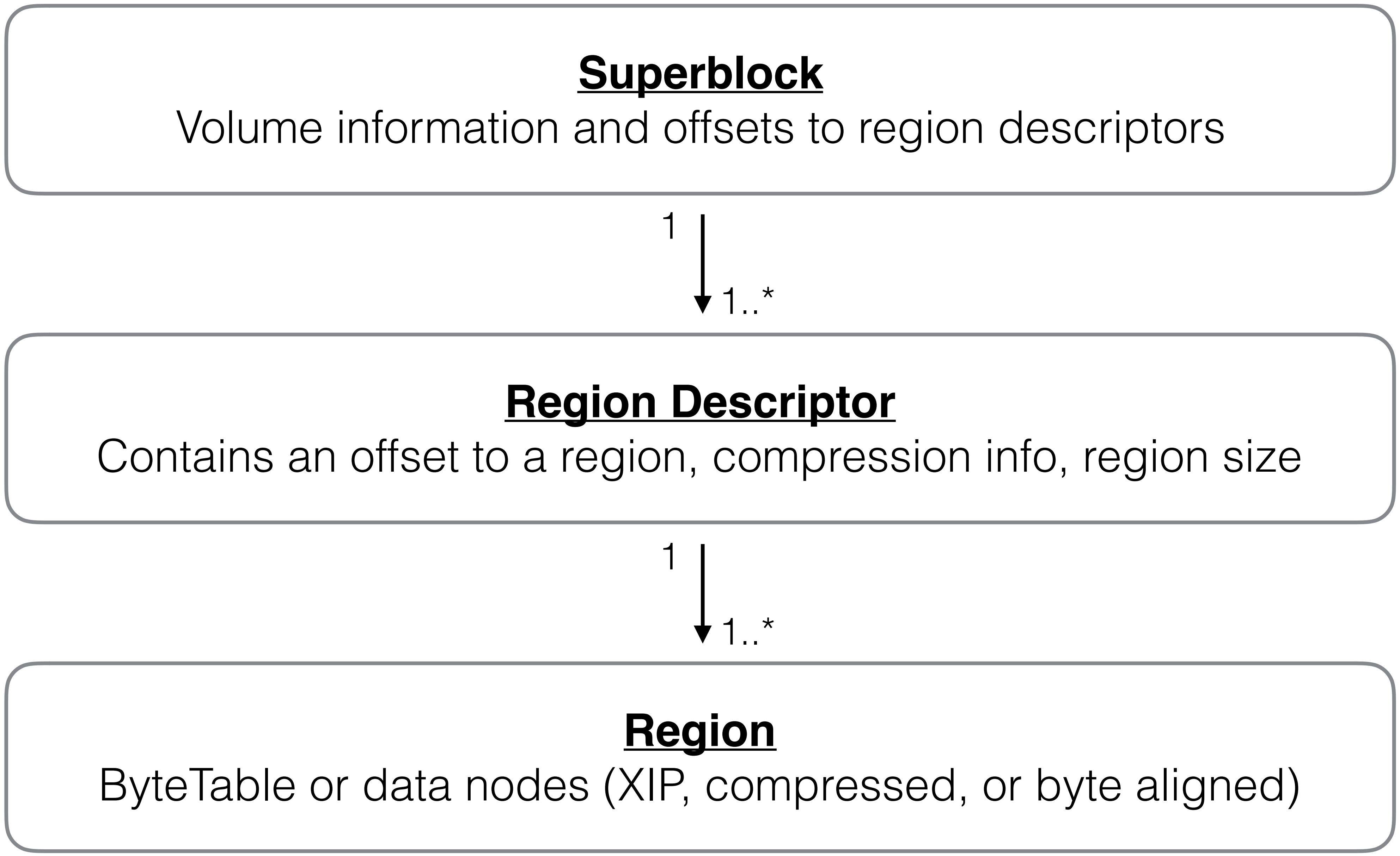
- Program loading is remarkably different
  - Shared libraries, code relocation, other dynamic loading



Design - Mounting

Split Mounting

NOR: XIP Programs **NAND**: Media Storage



Design - Data

(See Hulbert's *Introducing the Advanced XIP File System* for more details)

IoT: Client Devices

ext2

Background

Typical Linux General Purpose Filesystems

- Remy Card, 1993
- Basic design in use today (see: ext4)
- Non-journaling
- Based on earlier Berkeley Fast Filesystem

Drivers for every major OS

- Though they may be third-party

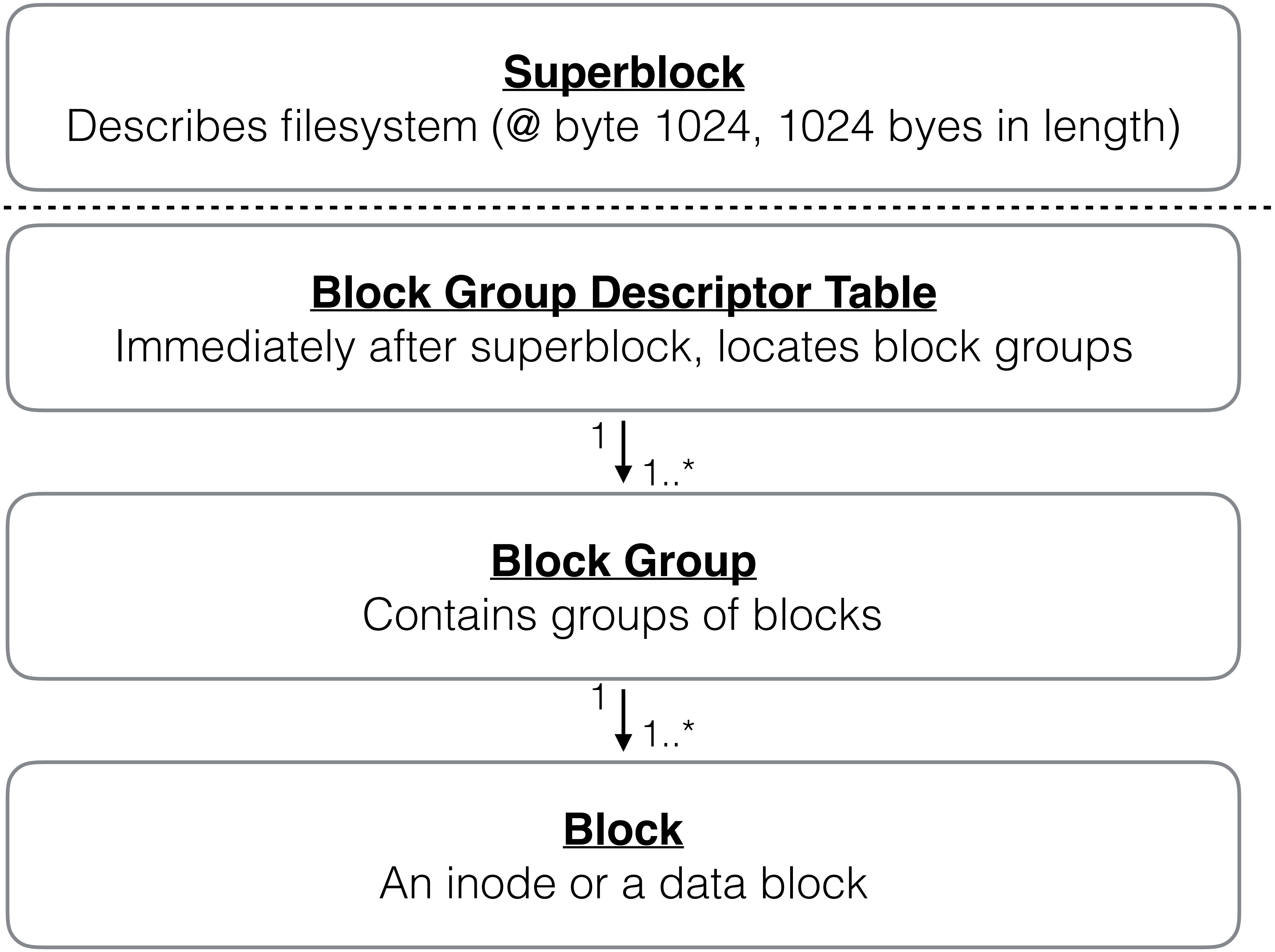
Aside: Journaling FS

Like a database of changes

- Write to changes to *journal* before committing to disk
- *File writes* usually require many separate *disk writes*
- Non-atomicity leads to potential corruption

Crash recovery

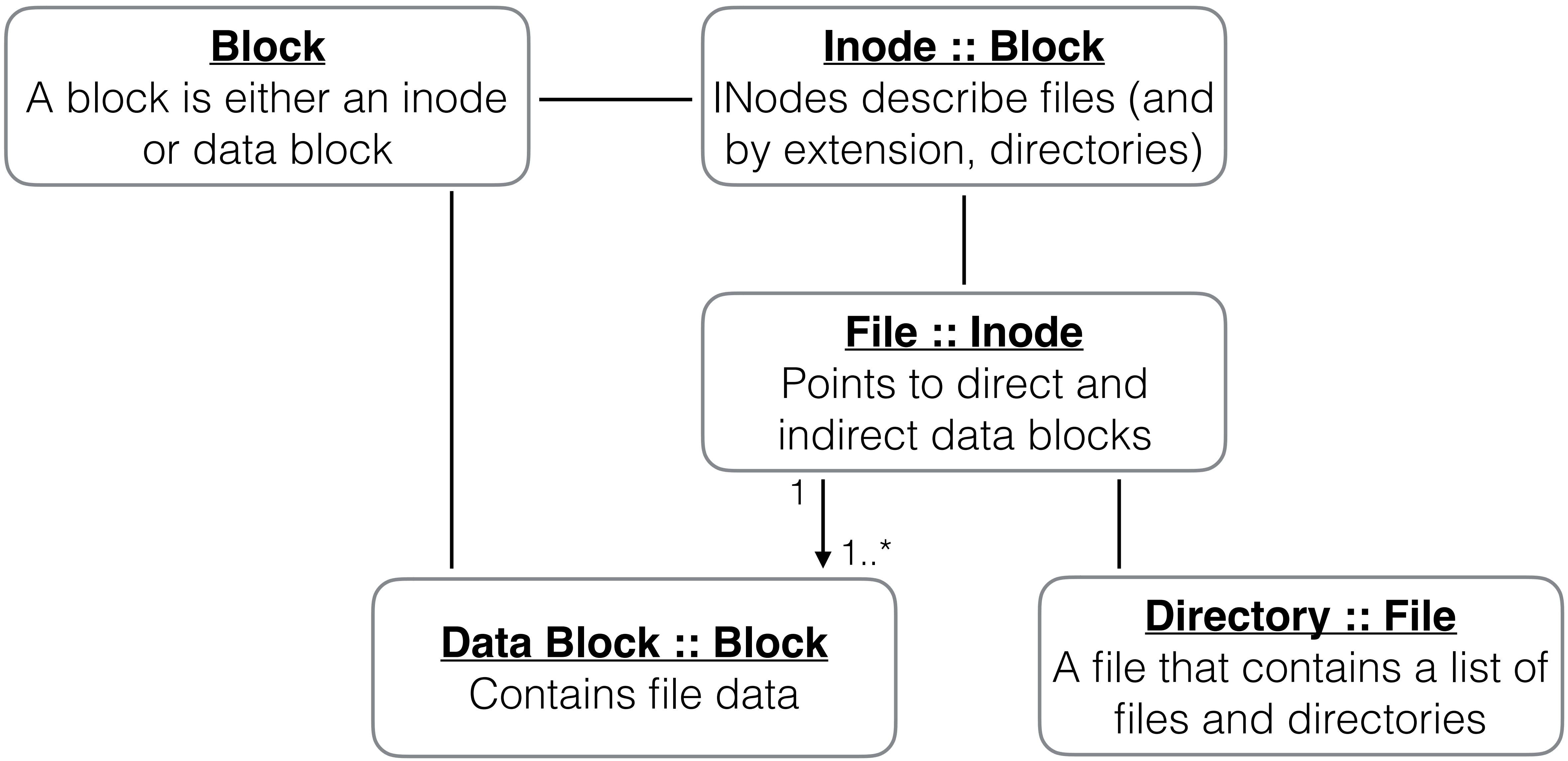
- Requires FS walk if no journal, not always recoverable
- Journal played back to ensure that all changes are applied



Design - Blocks

Filesystem is based on blocks and block management

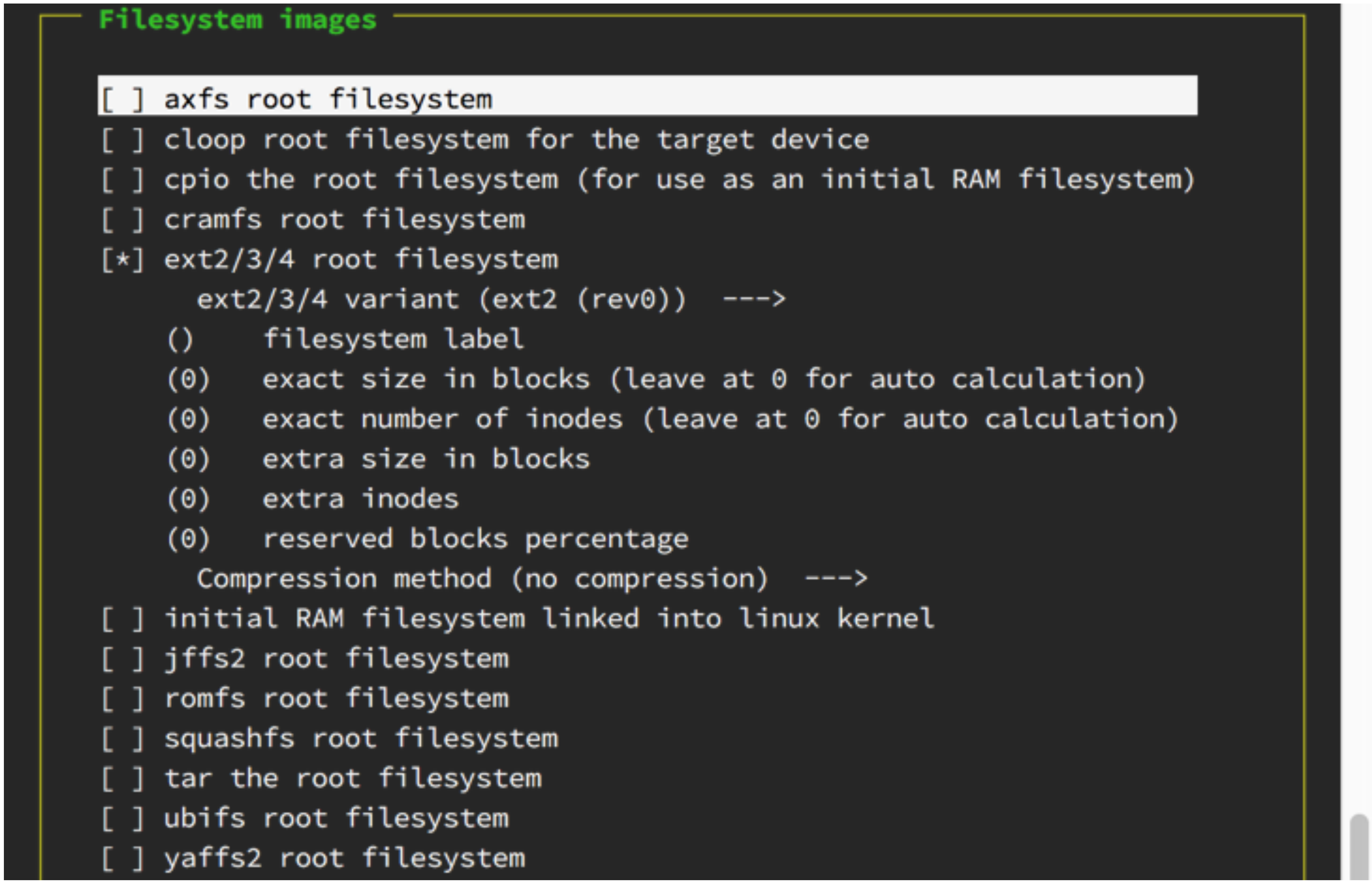
block size not tied to physical blocks



Design - Files and Directories

IoT: Client Devices

Filesystems



Lots of Options

so what do you choose? We'll take it from the top.

Always pay attention to licensing

AXFS

What is it?

- Advanced XIP Filesystem
- Compressed, read-only, execute-in-place

Why would you use it?

- Fast boot and load, small footprint

Why wouldn't you use it?

- You need writeable filesystem
- Hardware support

CLOOP

WHAT IS IT?

- Compressed block driver, similar to Apple DMG
- Read-only block devices, transparent compression

WHY WOULD YOU USE IT?

- Frequently used with Live CD
- Not a filesystem, but block device support (under FS)

WHY WOULDN'T YOU USE IT?

- You want to use a full filesystem solution

CRAMFS

What is it?

- Compressed ROM Filesystem, Read-only, Linux
- Simple, fast, small

Why would you use it?

- Only if you're stuck with it; It's obsoleted by SquashFS

Why wouldn't you use it?

- Don't use it for new systems (unless forced for some reason)

ext2/3/4

WHAT IS IT?

- Very common Linux filesystem(s)
- ext2 non-journaling, used with Flash and SD cards

WHY WOULD YOU USE IT?

- Writeable filesystem (moving programs to image, caching, etc.)
- Limit writes to storage (no journal to maintain)

WHY WOULDN'T YOU USE IT?

- Writeable is more exploitable, speed

RAM filesystem

WHAT IS IT?

- A filesystem configured in RAM (i.e. a RAMDisk)

WHY WOULD YOU USE IT?

- Very fast, gives initial filesystem while other loads

WHY WOULDN'T YOU USE IT?

- Need more space than RAMDisk will provide
- Limit RAM usage

JFFS2

What is it?

- Journaling Flash Filesystem

Why would you use it?

- You want journaling, but you're using Flash memory
- You don't care about write degradation
- Compression

Why wouldn't you use it?

- There's successor filesystems (e.g. YAFFS)
- Slow boot, difficult filesystem analysis

ROMFS

What is it?

- A very small, simple filesystem for EEPROMs

Why would you use it?

- Kernel module storage

Why wouldn't you use it?

- If you don't know you need it, don't use it

SquashFS

WHAT IS IT?

- Successor to cramfs
- Compressed, read-only,large block support, low-overhead

WHY WOULD YOU USE IT?

- You want a modern, read-only, compressed filesystem
- You don't care about XIP

WHY WOULDN'T YOU USE IT?

- You want a writeable filesystem
- You care about XIP

UBIFS

WHAT IS IT?

- Unsorted block image filesystem
- Successor to JFFS2

WHY WOULD YOU USE IT?

- Better than JFFS2 for large NAND Flash
- Better failure tolerance, compression support

WHY WOULDN'T YOU USE IT?

- Locked into JFFS2, Hardware limitations

YAFFS2

What is it?

- Yet Another Flash Filesystem
- Log structured, high data-integrity goals

Why would you use it?

- Portable, Fast, supports modern hardware

Why wouldn't you use it?

- Hardware restrictions or licensing
- No compression

IoT: Client Devices

Embedded Kernels

Embedded Kernels

I’m not going to discuss the Linux kernel in depth.

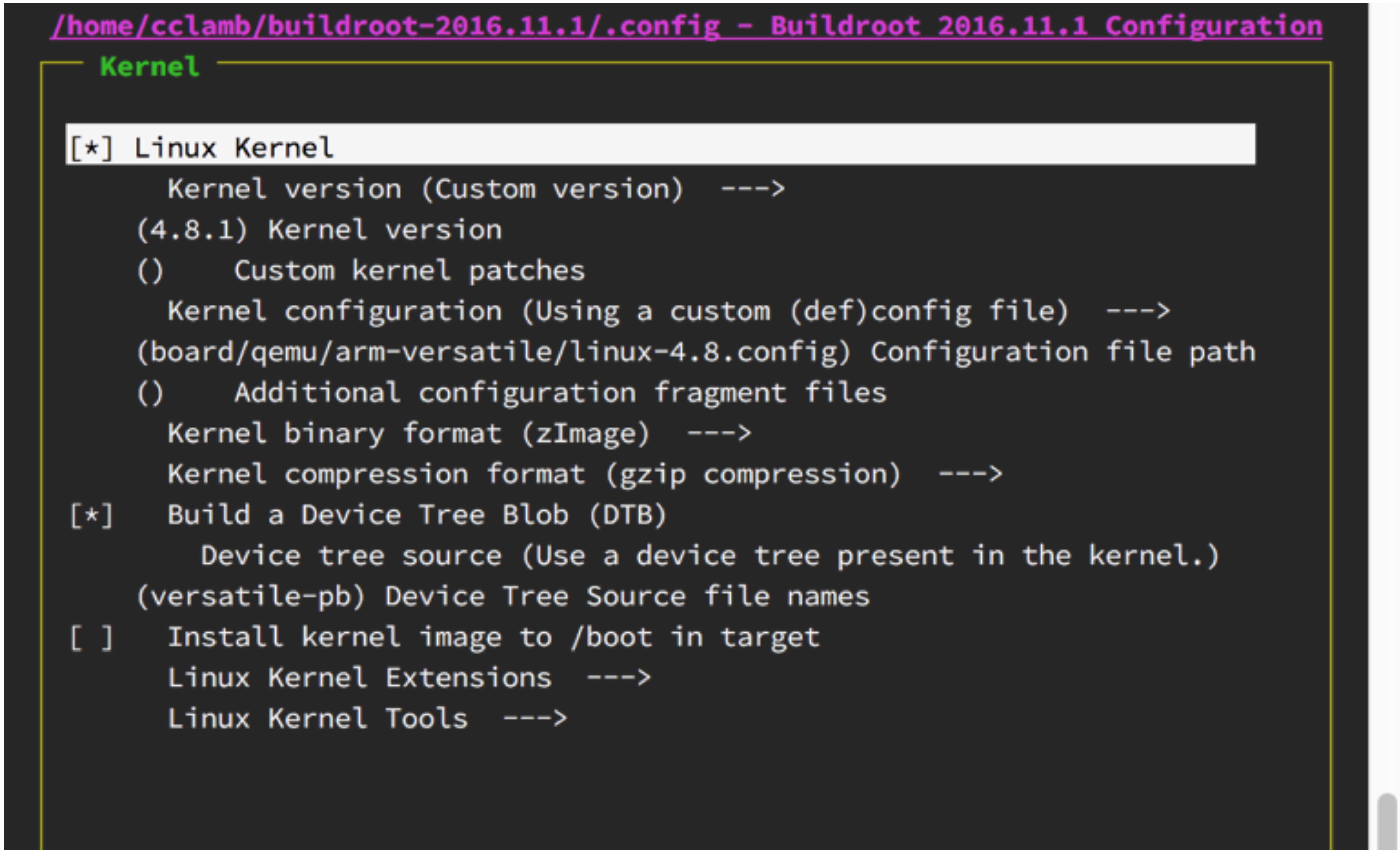
Guidelines for Kernel

You may not have much choice

- The kernel modules you need are dictated by hardware
- You'll likely have a fixed config file

Buildroot gives you some options

- For the most part, use the defaults



Changing Defaults

If you do change defaults…

…back up your config(s) first!

```
cclamb@ubuntu:~/buildroot-2016.11.1 $ head -20 board/qemu/arm-versatile/linux-4.
8.config
CONFIG_SYSVIPC=y
CONFIG_MODULES=y
CONFIG_MODULE_UNLOAD=y
# CONFIG_ARCH_MULTI_V7 is not set
CONFIG_ARCH_VERSATILE=y
CONFIG_PCI=y
CONFIG_PCI_VERSATILE=y
CONFIG_AEABI=y
CONFIG_NET=y
CONFIG_PACKET=y
CONFIG_UNIX=y
CONFIG_INET=y
CONFIG_SCSI=y
CONFIG_BLK_DEV_SD=y
CONFIG_SCSI_SYM53C8XX_2=y
CONFIG_NETDEVICES=y
CONFIG_8139CP=y
CONFIG_PHYLIB=y
CONFIG_INPUT_EVDEV=y
CONFIG_SERIO_AMBAKMI=y
cclamb@ubuntu:~/buildroot-2016.11.1 $
```

Defconfig Files

IoT: Client Devices

Bootloaders

We’re Not Using One!

We’re running a simulator

- So we’re not using one
- In real life you’ll need one

Boot loaders

- DAS U-Boot, gummiboot, AT91 Bootstrap, etc
- Take a look at the Bootloader menu

Bootloaders

Yeah, these are complex too.

Bootloader does What?

Loads the Operating System

- Transitions control to OS

Handles initial configuration

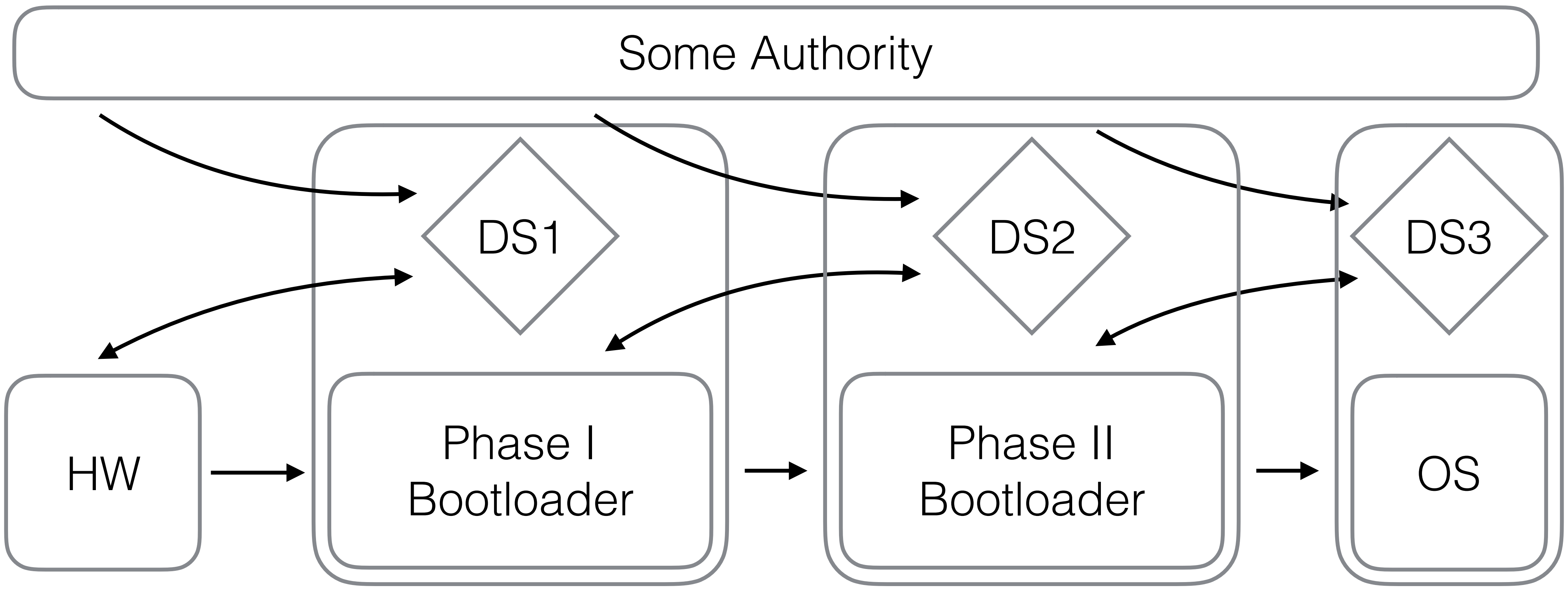
- Power on self tests first…
- …then loads bootloader…
- …then loads OS.

Commonly use multiple stages

- Different loaders of different complexity

IoT: Client Devices

Bootloaders



Secure Boot

How do we do this?

IoT: Client Devices

Securing the Kernel and OS

Securing the Kernel

OKAY, YOU'RE NOT A KERNEL/OS DEVELOPER

- You're not going too be writing secure kernel code

YOU ARE RESPONSIBLE FOR A SECURE KERNEL

- …on your device

So what does this mean?

Use a Secure Kernel

Don't use buggy code

- This includes kernels, libraries, etc.

Old? Known issues?

- Don't use it
- Use the newest most secure code you can

Ship Secure Tooling

OS IMAGES COME WITH LOTS OF STUFF

- Only ship what you need
- Don’t ship things you don’t

ONGOING MAINTENANCE AND ANALYSIS

- This is a real need too
- Make sure anything that you ship on your device is as secure as possible!

DON’T WRITE YOUR OWN

- Don’t create your own protocols or encryption

Build Updatable Devices

VULNERABILITIES **WILL** POP UP

- They always do.

MAKE SURE THAT WHEN THEY DO YOU CAN FIX THEM

- You'll need to do this at scale
- You'll need to secure this too

IoT: Client Devices

Securing your Application

Yes, you.

SECURING KERNELS, OS

- You're responsible for this
- You don't have much control

You have total control over **YOUR** software.

Know Typical Flaws

KNOW TYPICAL SOFTWARE FLAWS

- C: buffer overflows, integer underflows, bad coding, etc.
- Web: encryption, authentication, authorization, etc.

KNOW RESOURCES

- C secure programming standard
- SANS top 20
- OWASP secure programming practices

Red Teams & Pentest

Code reviews

- Review code with development colleagues

Red teams

- Have adversarial design and code reviews

Attack!

- Penetration test the final product

Don’t DYI

OTHER EYES == BETTER

- Find others to review your code
- Other organizations to red team and pentest

STUDY SECURE CODING

- It’s not hard, you just need to do it!