Operating Systems

15. File System Implementation

Paul Krzyzanowski

Rutgers University

Spring 2015

Log Structured File Systems

NAND flash memory

- Memory arranged in “pages” – similar to disk blocks
  - Unit of allocation and programming
  - Individual bytes cannot be written
- You cannot just *write* to a block of flash memory
  - It has to be erased first
  - Read-erase-write may be 50…100x slower than writing to an already-erased block!
- Limited erase-write cycles
  - ~100,000 to 1,000,000 cycles
  - Employ wear leveling to distribute writes among all (most) blocks
  - Bad block “retirement”

Problems with conventional file systems

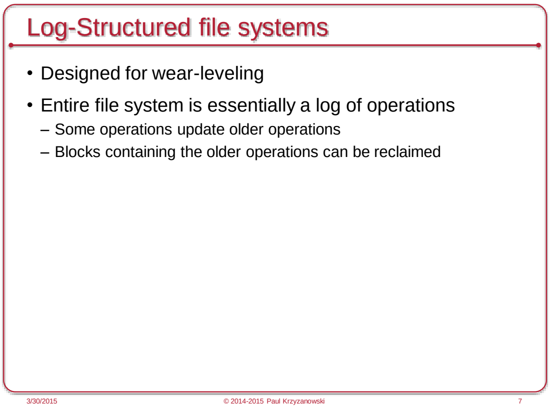
- Modify the same blocks over and over
  - At odds with NAND flash performance
  - Have to rely on FTL or smart controller
- Optimizations to minimize seek time
  - Spatial locality is meaningless with flash memory

Wear leveling

- Dynamic wear leveling
  - Monitors erase count of blocks
  - Map logical block addresses to flash memory block addresses
  - When a block of data is written to flash memory,
    - Write to a free block with the lowest erase count
    - Update *logical* → *physical* block map
  - Blocks that are never modified will not get reused
- Static wear leveling
  - Copy static data with low erase counts to another block so the original block can be reused
  - Usually triggered when the (maximum-minimum) erase cycles reaches a threshold

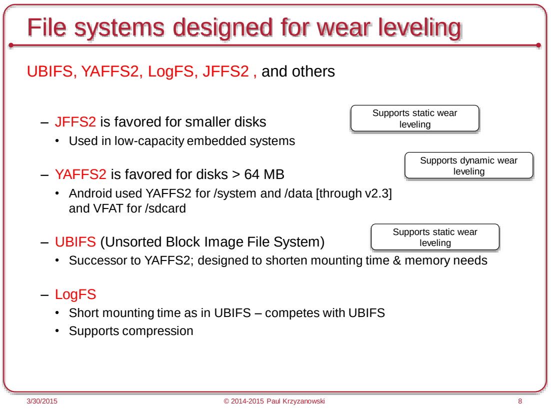
Our options with NAND flash memory

1. NAND flash with a flash memory controller
   - Handles block mapping (logical → physical)
     - Block Lookup Table
   - Employs wear leveling: usually static *and* dynamic
   - Asynchronous garbage collection and erasing
   - Can use conventional file systems – transparent to software
2. Flash Translation Layer (FTL)
   - Software layer between flash hardware & a block device
   - Microsoft’s term: Flash Abstraction Layer (FAL) – sits on top of Flash Media Driver
   - Rarely used now – moved to firmware (1)
3. OS file system software optimized for raw flash storage
   - Write new blocks instead of erasing & overwriting an old one
   - Erase the old blocks later



Log-Structured file systems

- Designed for wear-leveling
- Entire file system is essentially a log of operations
  - Some operations update older operations
  - Blocks containing the older operations can be reclaimed



File systems designed for wear leveling

UBIFS, YAFFS2, LogFS, JFFS2 , and others

- JFFS2 is favored for smaller disks
  - Used in low-capacity embedded systems

Supports static wear leveling

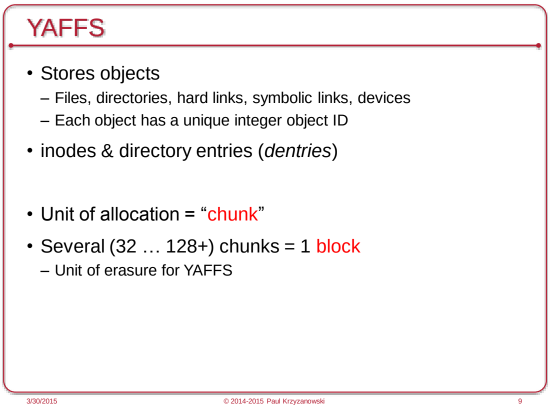
- YAFFS2 is favored for disks > 64 MB
  - Android used YAFFS2 for /system and /data [through v2.3] and VFAT for /sdcard

Supports dynamic wear leveling

- UBIFS (Unsorted Block Image File System)
  - Successor to YAFFS2; designed to shorten mounting time & memory needs

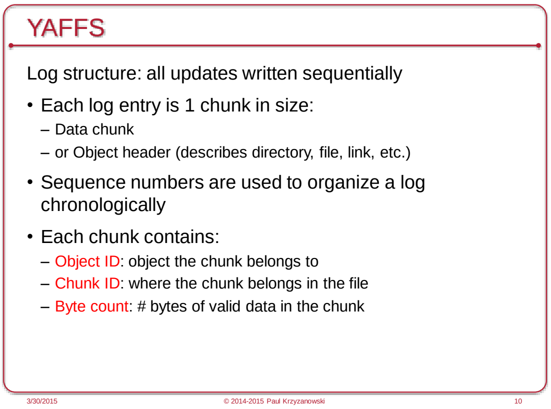
Supports static wear leveling

- LogFS
  - Short mounting time as in UBIFS – competes with UBIFS
  - Supports compression



YAFFS

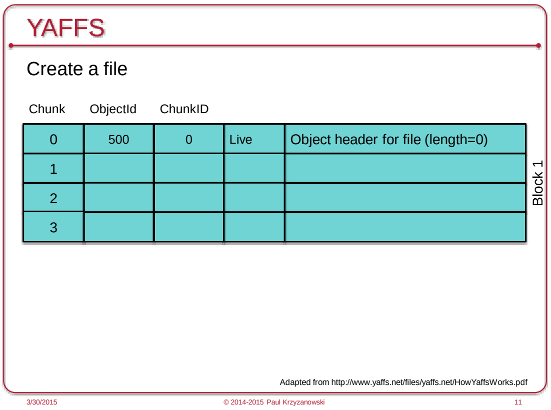
- Stores objects
  - Files, directories, hard links, symbolic links, devices
  - Each object has a unique integer object ID
- inodes & directory entries (*dentries*)
- Unit of allocation = "chunk"
- Several (32 … 128+) chunks = 1 block
  - Unit of erasure for YAFFS



YAFFS

Log structure: all updates written sequentially

- Each log entry is 1 chunk in size:
  - Data chunk
  - or Object header (describes directory, file, link, etc.)
- Sequence numbers are used to organize a log chronologically
- Each chunk contains:
  - Object ID: object the chunk belongs to
  - Chunk ID: where the chunk belongs in the file
  - Byte count: # bytes of valid data in the chunk



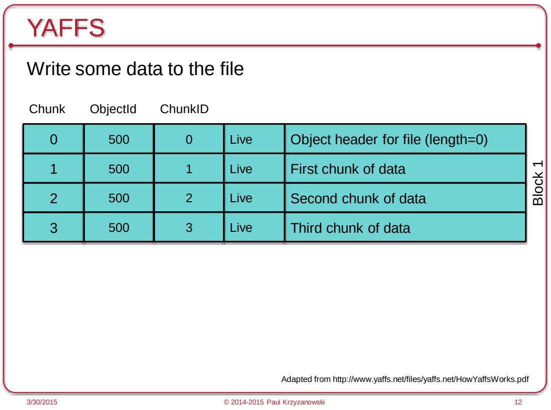
YAFFS

Create a file

Chunk	ObjectId	ChunkID		
0	500	0	Live	Object header for file (length=0)
1				
2				
3				

Block 1

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf



YAFFS

Write some data to the file

Chunk	ObjectId	ChunkID		
0	500	0	Live	Object header for file (length=0)
1	500	1	Live	First chunk of data
2	500	2	Live	Second chunk of data
3	500	3	Live	Third chunk of data

Block 1

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

YAFFS

Close the file: write new header

Chunk	ObjectId	ChunkID			
0	500	0	Deleted	Object header for file (length=0)	Block 1
1	500	1	Live	First chunk of data	
2	500	2	Live	Second chunk of data	
3	500	3	Live	Third chunk of data	
0	500	0	Live	Object header for file (length=*n*)	Block 2

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

YAFFS

Open file; modify first chunk; close file

Chunk	ObjectId	ChunkID			
0	500	0	Deleted	Object header for file (length=0)	Block 1
1	500	1	Deleted	First chunk of data	
2	500	2	Live	Second chunk of data	
3	500	3	Live	Third chunk of data	
0	500	0	Deleted	Object header for file (length=*n*)	Block 2
1	500	1	Live	New first chunk of data	
2	500	0	Live	New object header for file (length=*n*)	

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

YAFFS Garbage Collection

- If all chunks in a block are deleted
  - The block can be erased & reused
- If blocks have some free chunks
  - We need to do garbage collection
  - Copy active chunks onto other blocks so we can free a block
    - Passive collection: pick blocks with few used chunks
    - Aggressive collection: try harder to consolidate chunks

YAFFS in-memory structures

Construct file system state in memory

- Map of in-use chunks
- In-memory object state for each object
- File tree/directory structure to locate objects
- Scan the log backwards chronologically
  highest→*lowest* sequence numbers
- Checkpointing: save the state of these structures at unmount time to speed up the next mount

YAFFS error detection/correction

- ECC used for error recovery
  - Correct 1 bad bit per 256 bytes
  - Detect 2 bad bits per 256 bytes
  - Bad blocks:
    if read or write fails, ask driver to mark the block as bad

UBIFS vs YAFFS

- Entire file system state does not have to be stored in memory
- Challenge
  - Index has to be updated out-of-place
  - Parts that refer to updated areas have to also be updated
- UBIFS wandering tree (B+ Tree)
  - Only leaves contain file information
  - Internal nodes = index nodes
- Update to FS
  - Create leaf; add/replace into wandering tree
  - Update parent index nodes up to the root

Special file systems

Pseudo devices

- Device drivers can also provide custom functions
  - Even if there is no underlying device

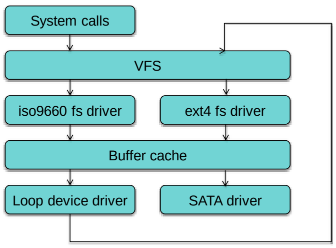
Simple special-function device files

- /dev/null *Null device*
  - Throw away anything written to it; return EOF on reads
- /dev/zero Zero device
  - Return zeros for each read operation
- /dev/random, /dev/urandom Random numbers
  - urandom is non-blocking
  - \Device\KsecDD on Windows NT

Loop pseudo device

- Provides a block device interface to a file
  - Register file as a block device
  - Let the buffer cache know:
    - *request (strategy)* procedure for read/write
    - block size
- The file can then be formatted with a file system and mounted
  - See the *`losetup`* command in Linux
- Common uses
  - installation software
  - CD/DVD images
  - Encrypted file systems

Loop device



System calls

VFS

iso9660 fs driver

ext4 fs driver

Buffer cache

Loop device driver

SATA driver

Example: (1) Create a loop device

Create a 10MB file named file.img

```
# dd if=/dev/zero of=file.img bs=1k count=10000
10000+0 records in
10000+0 records out
```

Associate loop device /dev/loop0 with the file file.img

```
# losetup /dev/loop0 file.img
```

This makes /dev/loop0 a block device whose contents are file.img

```
# ls -l /dev/loop0
brw-rw---- 1 root disk 7, 0 Mar 30 10:55 /dev/loop0
```

Example: (2) Put a file system on the file

Create a file system on /dev/loop0

```
# mke2fs -c /dev/loop0 10000
mke2fs 1.42.9 (4-Feb-2014)
Discarding device blocks: done
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
2512 inodes, 10000 blocks

...
```

Example: (3) Mount it

Create a directory that will be the mount point

```
# mkdir /mnt/here
```

Mount the file system

```
# mount -t ext2 /dev/loop0 /mnt/here
```

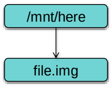
Test it out!

```
# ls -l /mnt/here
total 12
drwx------ 2 root root 12288 Mar 30 10:56 lost+found

# echo hello >/mnt/here/hello.txt

# ls -l /mnt/here
total 13
-rw-r--r-- 1 root root     6 Mar 30 14:31 hello.txt
drwx------ 2 root root 12288 Mar 30 10:56 lost+found

# cat /mnt/here/hello.txt
hello
```



Example: Do it recursively!

Create a 1000KB file called another.img within the file.img file system

```
# dd if=/dev/zero of=/mnt/here/another.img bs=1k count=1000
1000+0 records in
1000+0 records out
```

Make /dev/loop1 be a loop device that points to another.img

```
# losetup /dev/loop1 /mnt/here/another.img
```

Create a file system

```
# mke2fs -c /dev/loop1 1000
mke2fs 1.42.9 (4-Feb-2014)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
128 inodes, 1000 blocks
...

#
```

another.img is a file containing a file system

It exists within file.img, which is also a file containing a file system

Example: Do it recursively!

Create a directory (`/mnt/there`) that will be the mount point

```
# mkdir /mnt/there
```

mount the file system

```
# mount -t ext2 /dev/loop1 /mnt/there
```

Test it!

```
# echo hey! >/mnt/there/test
# ls -l /mnt/there
total 13
drwx------ 2 root root 12288 Mar 30 14:35 lost+found
-rw-r--r-- 1 root root     5 Mar 30 14:36 test
```

/mnt/there/text:
File in a file system (`/mnt/there`)
that is a file (`another.img`)
within a file system (`/mnt/here`)
that is a file (`file.img`)
within a file system (top-level)

It works! another.img is a file system within file.img which is a file system on the disk

```
# ls -l /mnt/here
total 1018
-rw-r--r-- 1 root root 1024000 Mar 30 14:35 another.img
-rw-r--r-- 1 root root       6 Mar 30 14:31 hello.txt
drwx------ 2 root root   12288 Mar 30 10:56 lost+found
```

Generic Interfaces via VFS

VFS gives us a generic interface to file operations
- We don't need to have persistent storage underneath
  … or even storage!

procfs: process file system

- `/proc`
  - View & control processes & kernel structures as files
- Origins: Plan 9 from Bell Labs
  - Look into and control processes
- procfs is a file system driver
  - Registers itself with VFS
  - When VFS calls to request inodes as files & directories are accessed, /proc creates them from info within kernel structures.

procfs: process file system

- Remove the need for system calls to get info, read config parameters, and inspect processes
- Simplify scripting
- Just a few items:

– /proc/cpuinfo	info about the cpu
– /proc/devices	list of all character & block devices
– /proc/diskstats	info on logical disks
– /proc/meminfo	info on system memory
– /proc/net	directory containing info on the network stack
– /proc/swaps	list of swap partitions
– /proc/uptime	time the system has been up
– /proc/version	kernel version

- Plan 9 allowed remote access to /proc

procfs: process info

```
$ ls /proc/27325
attr             cwd       loginuid     oom_adj       smaps
auxv             environ   maps         oom_score     stack
cgroup           exe       mem          pagemap       stat
clear_refs       fd        mountinfo    personality   statm
cmdline          fdinfo    mounts       root          status
comm             io        mountstats   sched         syscall
coredump_filter  latency   net          schedstat     task
cpuset           limits    numa_maps    sessionid     wchan
```

Naming Devices

Device Names in Windows

- Windows Object Manager
  - Owns the system namespace
  - Manages Windows resources: devices, files, registry entries, processes, memory, …
  - Programs can look up, share, protect, and access resources
  - Resource access is dedicated to the appropriate subsystem
    - I/O Manager gets requests to parse & access file names
- When a device driver is loaded by the kernel
  - Driver init routine registers a device name with the Object Manager
    - \Device\CDRom0, \Device\Serial0
  - Win32 API requires MS-DOS device names
    - Names also live in the Object Manager
    - Created as symbolic links in the \?? directory

Devices in Linux & OS X

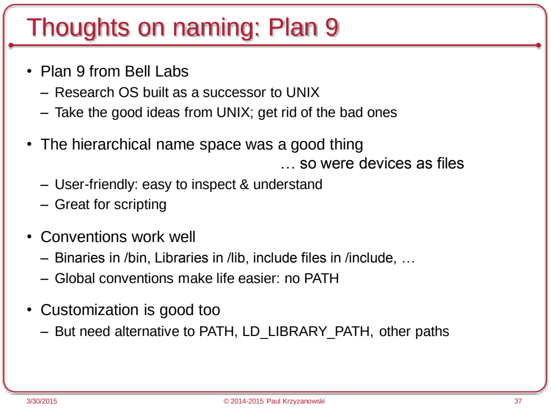
- In the past: Devices were static; explicitly created via *mknod*
- Now: Devices come & go

- devfs: special file system mounted on /dev
  - Presents device files
  - Device driver registers with *devfs* upon initialization via *devfs_register*
  - Avoids having to create device special files in /dev
  - *Obsolete since Linux 2.6; still used in OS X and others*
- udev device manager
  - User level process; listens to *uevents* from the kernel via a *netlink* socket
    - Detect new device initialization or removal
  - Persistent device naming – guided by files in /etc/udev/rules.d

FUSE: Filesystem in Userspace

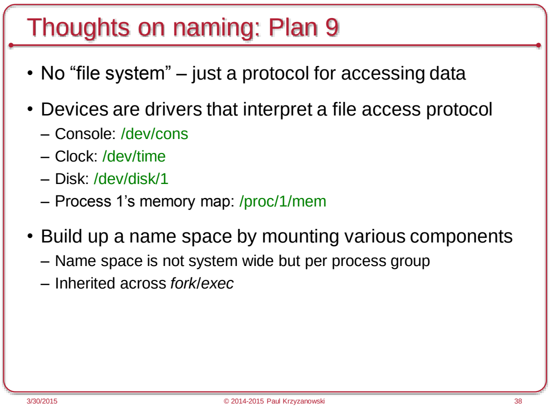
- File system can run as a normal user process

- FUSE module
  - Conduit to pass data between VFS and user process
  - Communication via a special file descriptor obtained by opening /dev/fuse



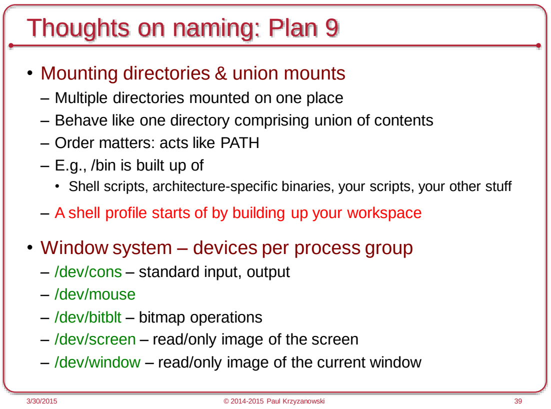
Thoughts on naming: Plan 9

- Plan 9 from Bell Labs
  - Research OS built as a successor to UNIX
  - Take the good ideas from UNIX; get rid of the bad ones
- The hierarchical name space was a good thing
  … so were devices as files
  - User-friendly: easy to inspect & understand
  - Great for scripting
- Conventions work well
  - Binaries in /bin, Libraries in /lib, include files in /include, …
  - Global conventions make life easier: no PATH
- Customization is good too
  - But need alternative to PATH, LD_LIBRARY_PATH, other paths



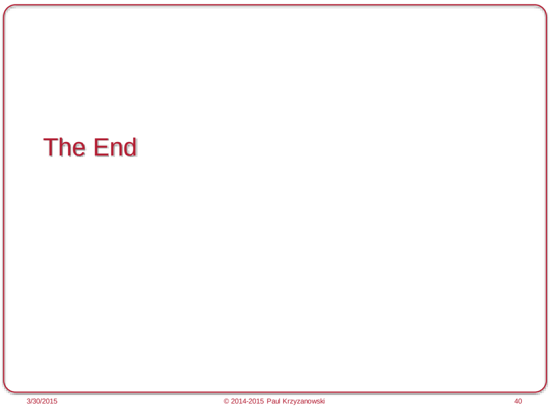
Thoughts on naming: Plan 9

- No “file system” – just a protocol for accessing data
- Devices are drivers that interpret a file access protocol
  - Console: /dev/cons
  - Clock: /dev/time
  - Disk: /dev/disk/1
  - Process 1’s memory map: /proc/1/mem
- Build up a name space by mounting various components
  - Name space is not system wide but per process group
  - Inherited across *fork*/*exec*



Thoughts on naming: Plan 9

- Mounting directories & union mounts
  - Multiple directories mounted on one place
  - Behave like one directory comprising union of contents
  - Order matters: acts like PATH
  - E.g., /bin is built up of
    - Shell scripts, architecture-specific binaries, your scripts, your other stuff
  - A shell profile starts of by building up your workspace
- Window system – devices per process group
  - /dev/cons – standard input, output
  - /dev/mouse
  - /dev/bitblt – bitmap operations
  - /dev/screen – read/only image of the screen
  - /dev/window – read/only image of the current window



The End