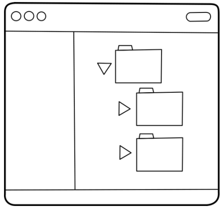
Linux Playground and Storage Stack

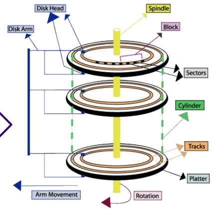
ECE 469, March 31

Aravind Machiry

Preview: File System Abstraction



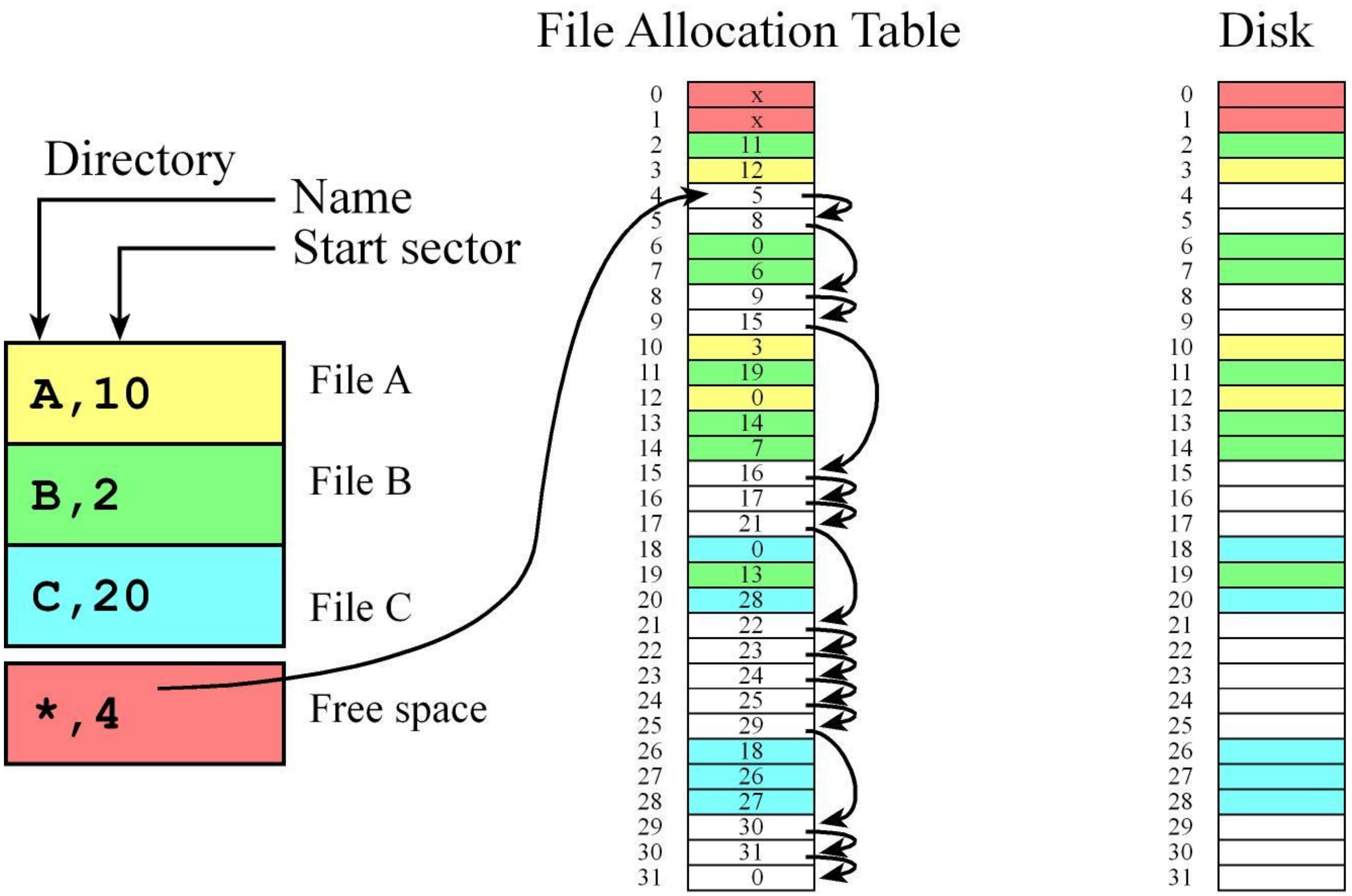
File System Abstraction



Preview: File Allocation Table (FAT)

- Simple.
- Easy to implement.
- Still used in Phones and Thumb drives.

- Key data structure: File Allocation Table
  - List of all disk blocks.
  - File: Linked list of blocks.



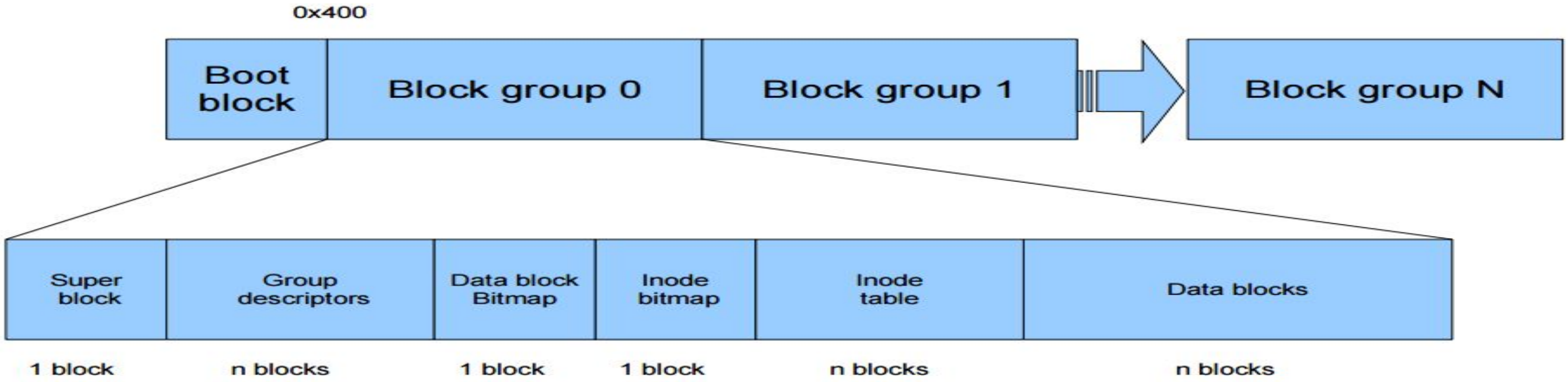
Directory
Name
Start sector
A,10
B,2
C,20
*,4
File A
File B
File C
Free space
File Allocation Table
0 x
1 x
2 11
3 12
4 5
5 8
6 0
7 6
8 9
9 15
10 3
11 19
12 0
13 14
14 7
15 16
16 17
17 21
18 0
19 13
20 28
21 22
22 23
23 24
24 25
25 29
26 18
27 26
28 27
29 30
30 31
31 0
Disk

Preview: FAT

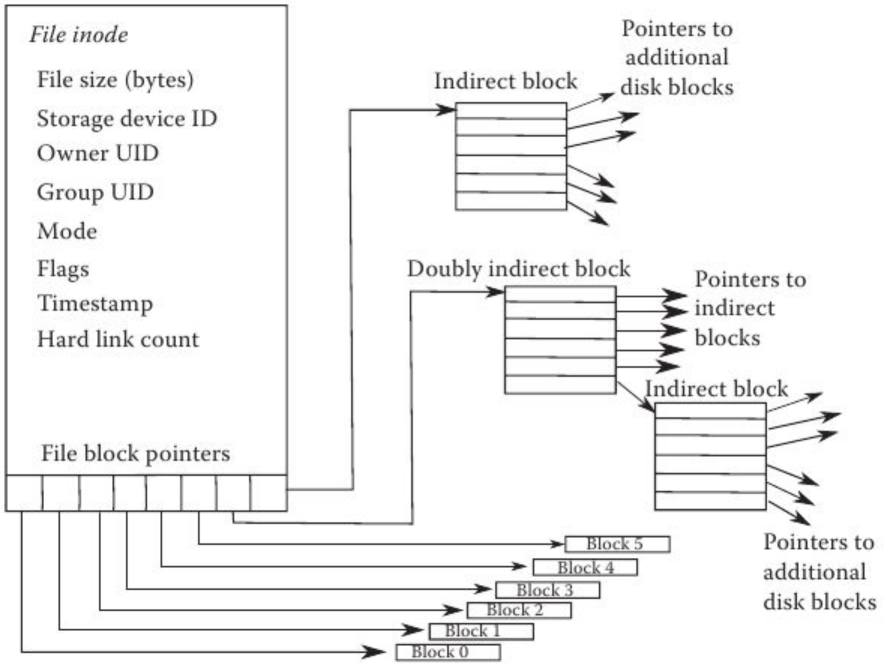
- Pros: Simple
  - Easy to find free block.
  - Easy to append file.
  - Easy to delete a file.

- Cons:
  - Small file access is slow.
  - Random file access is very slow.
  - Fragmentation:
    - Blocks of a small file could be heavily scattered.
    - Problem becomes worse as the usage increases.

Preview: EXT2 File System



Preview: EXT2 File System : inode

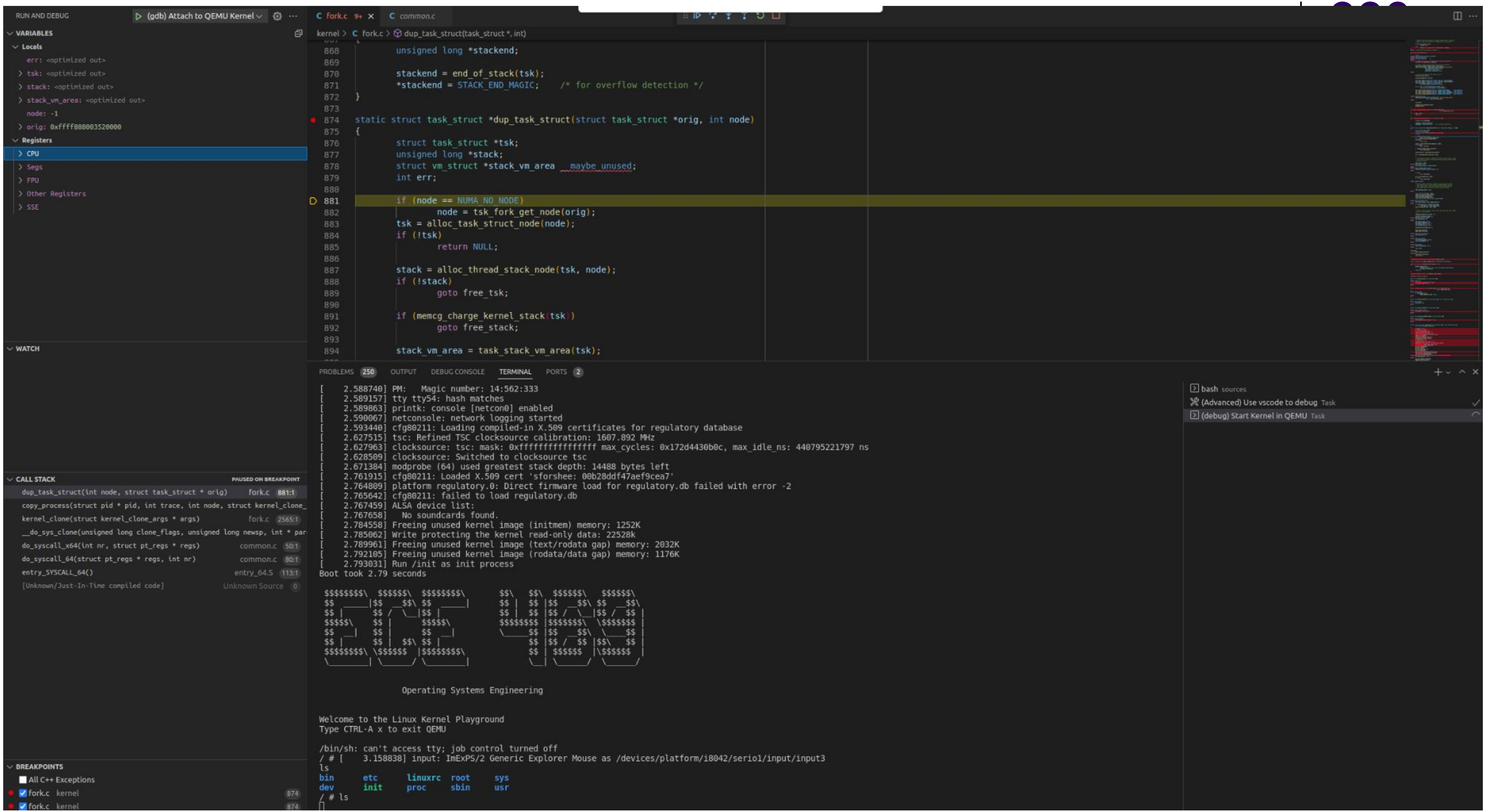


Preview: EXT2 File Size (Block Size: 4K)

- 12 File Block Pointers = 12*4 = 48K
- 1 Indirect block pointer (4K) = 1K direct block pointers = 1K*4K = 4MB
- 1 doubly indirect block pointer (4K) = 1K Indirect block pointers = 1K*4MB = 4 GB
- 1 triply indirect block pointer (4K) = 1K doubly Indirect block pointers = 1K*4GB = 4 TB
- Total Size = 48K + 4MB + 4GB + 4TB

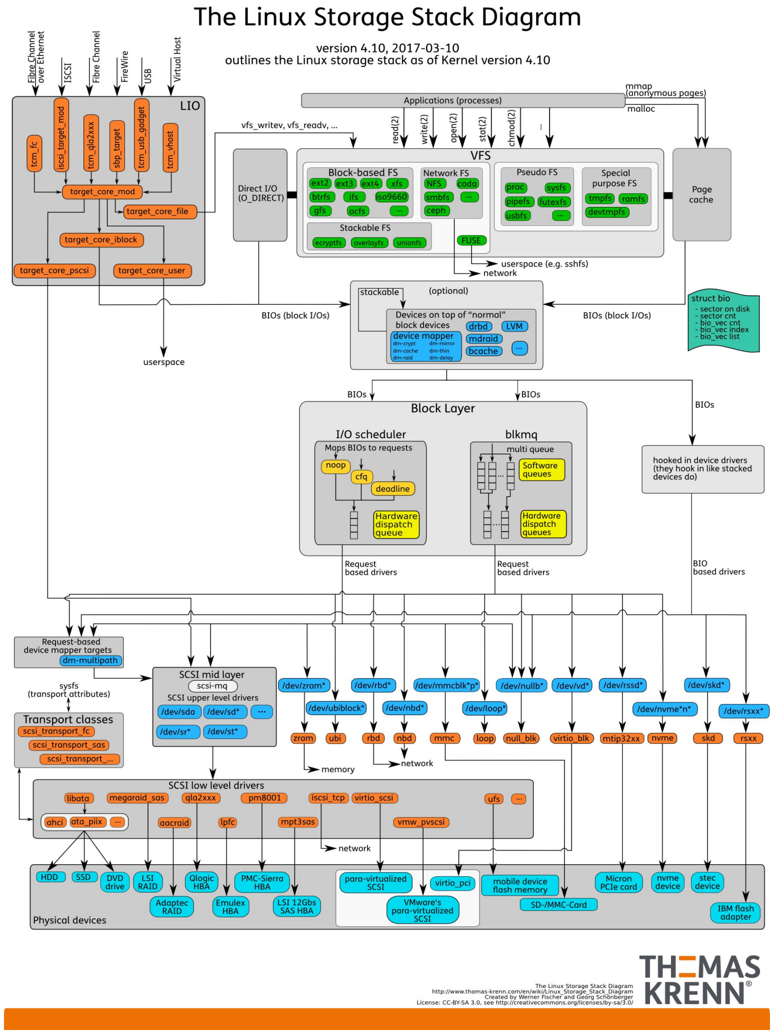
Linux Kernel PlayGround

- https://github.com/purs3lab/linux-playground
- Linux kernel debugging through Docker container and VSCode

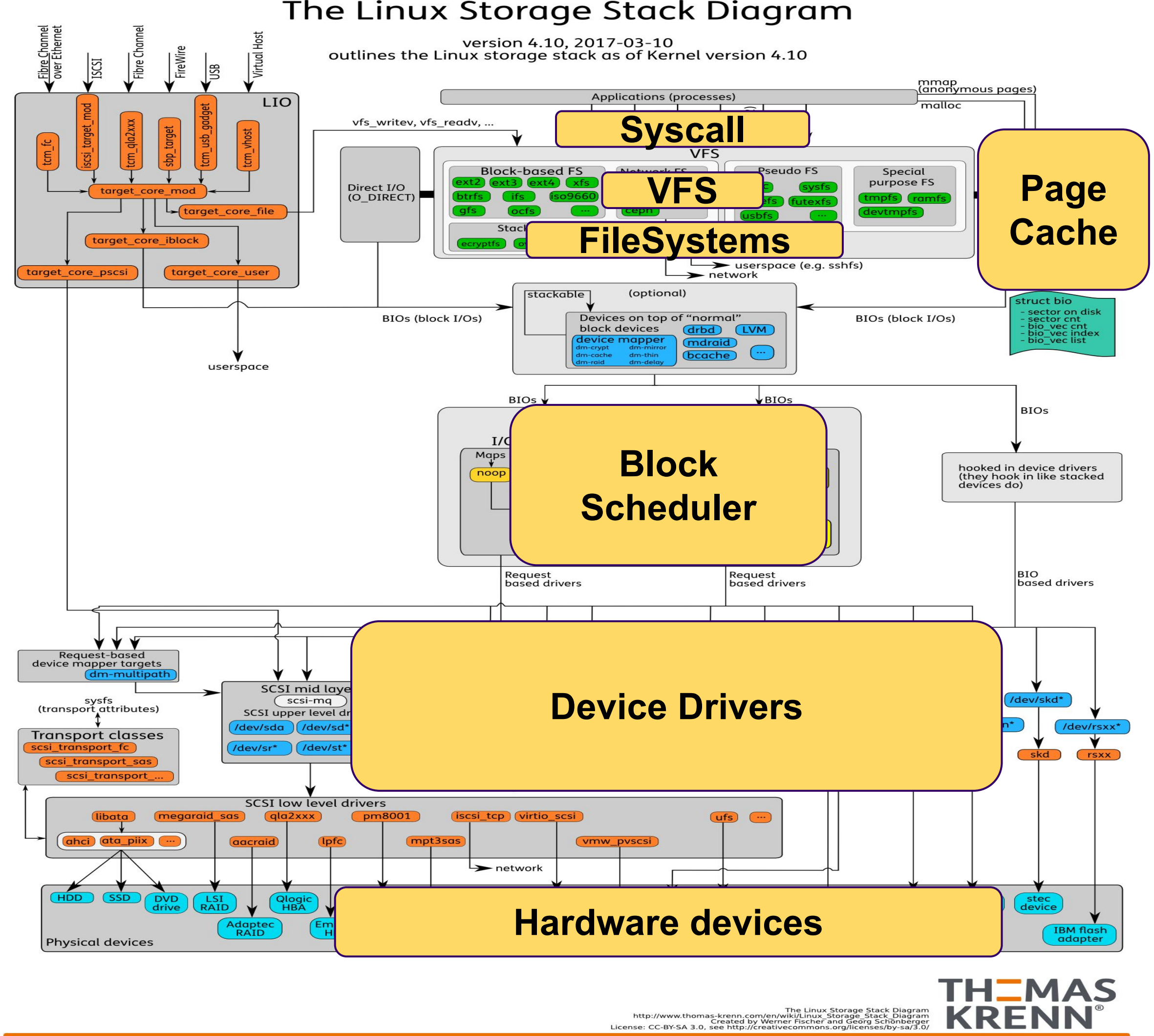

RUN AND DEBUG
(gdb) Attach to QEMU Kernel
VARIABLES
Locals
err: <optimized out>
tsk: <optimized out>
stack: <optimized out>
stack_vm_area: <optimized out>
node: -1
orig: 0xffff888003520000
Registers
CPU
Segs
FPU
Other Registers
SSE
WATCH
CALL STACK
PAUSED ON BREAKPOINT
dup_task_struct(int node, struct task_struct * orig) fork.c 881:1
copy_process(struct pid * pid, int trace, int node, struct kernel_clone_
kernel_clone(struct kernel_clone_args * args) fork.c 2565:1
__do_sys_clone(unsigned long clone_flags, unsigned long newsp, int * par
do_syscall_x64(int nr, struct pt_regs * regs) common.c 50:1
do_syscall_64(struct pt_regs * regs, int nr) common.c 80:1
entry_SYSCALL_64() entry_64.S 113:1
[Unknown/Just-In-Time compiled code] Unknown Source 0
BREAKPOINTS
All C++ Exceptions
fork.c kernel 874
fork.c kernel 874
fork.c 9+
common.c
kernel > fork.c > dup_task_struct(task_struct *, int)
868 unsigned long *stackend;
870 stackend = end_of_stack(tsk);
871 *stackend = STACK_END_MAGIC; /* for overflow detection */
872 }
874 static struct task_struct *dup_task_struct(struct task_struct *orig, int node)
875 {
876 struct task_struct *tsk;
877 unsigned long *stack;
878 struct vm_struct *stack_vm_area __maybe_unused;
879 int err;
881 if (node == NUMA_NO_NODE)
882 node = tsk_fork_get_node(orig);
883 tsk = alloc_task_struct_node(node);
884 if (!tsk)
885 return NULL;
887 stack = alloc_thread_stack_node(tsk, node);
888 if (!stack)
889 goto free_tsk;
891 if (memcg_charge_kernel_stack(tsk))
892 goto free_stack;
894 stack_vm_area = task_stack_vm_area(tsk);
PROBLEMS 250
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS 2
[2.588740] PM: Magic number: 14:562:333
[2.589157] tty tty54: hash matches
[2.589863] printk: console [netcon0] enabled
[2.590067] netconsole: network logging started
[2.593440] cfg80211: Loading compiled-in X.509 certificates for regulatory database
[2.627515] tsc: Refined TSC clocksource calibration: 1607.892 MHz
[2.627963] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x172d4430b0c, max_idle_ns: 440795221797 ns
[2.628509] clocksource: Switched to clocksource tsc
[2.671384] modprobe (64) used greatest stack depth: 14488 bytes left
[2.761915] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[2.764809] platform regulatory.0: Direct firmware load for regulatory.db failed with error -2
[2.765642] cfg80211: failed to load regulatory.db
[2.767459] ALSA device list:
[2.767658] No soundcards found.
[2.784558] Freeing unused kernel image (initmem) memory: 1252K
[2.785062] Write protecting the kernel read-only data: 22528k
[2.789961] Freeing unused kernel image (text/rodata gap) memory: 2032K
[2.792105] Freeing unused kernel image (rodata/data gap) memory: 1176K
[2.793031] Run /init as init process
Boot took 2.79 seconds
Operating Systems Engineering
Welcome to the Linux Kernel Playground
Type CTRL-A x to exit QEMU
/bin/sh: can't access tty; job control turned off
/ # [3.158838] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
ls
bin etc linuxrc root sys
dev init proc sbin usr
/ # ls
bash sources
(Advanced) Use vscode to debug Task
(debug) Start Kernel in QEMU Task

Linux Storage Stack

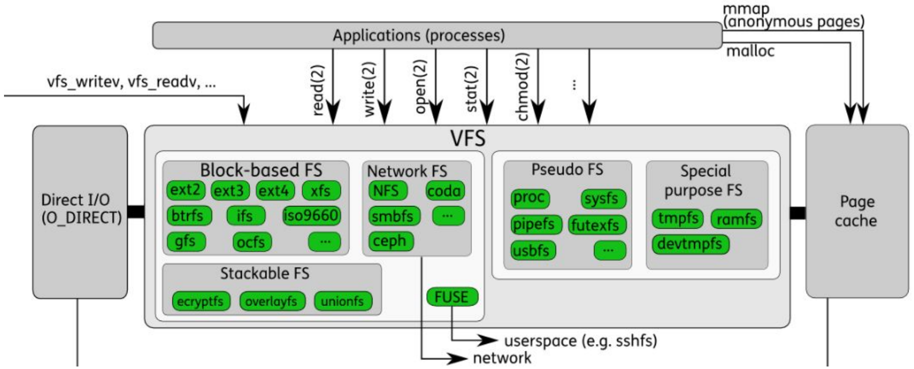
- Exhaustive and Modular



The Linux Storage Stack Diagram
version 4.10, 2017-03-10
outlines the Linux storage stack as of Kernel version 4.10
Fibre Channel over Ethernet
iSCSI
Fibre Channel
FireWire
USB
Virtual Host
LIO
tcm_fc
iscsi_target_mod
tcm_qla2xxx
sbp_target
tcm_usb_gadget
tcm_vhost
target_core_mod
target_core_file
target_core_iblock
target_core_pscsi
target_core_user
userspace
Applications (processes)
mmap (anonymous pages)
malloc
vfs_writev, vfs_readv, ...
read(2)
write(2)
open(2)
stat(2)
chmod(2)
VFS
Direct I/O (O_DIRECT)
Block-based FS
ext2 ext3 ext4 xfs
btrfs ifs iso9660
gfs ocfs ...
Network FS
NFS coda
smbfs ...
ceph
Pseudo FS
proc sysfs
pipefs futexfs
usbfs ...
Special purpose FS
tmpfs ramfs
devtmpfs
Stackable FS
ecryptfs overlayfs unionfs
FUSE
userspace (e.g. sshfs)
network
Page cache
struct bio
- sector on disk
- sector cnt
- bio_vec cnt
- bio_vec index
- bio_vec list
stackable
(optional)
Devices on top of "normal" block devices
drbd LVM
device mapper
mdraid
bcache
BIOs (block I/Os)
BIOs
Block Layer
I/O scheduler
Maps BIOs to requests
noop
cfq
deadline
Hardware dispatch queue
blkmq
multi queue
Software queues
Hardware dispatch queues
hooked in device drivers (they hook in like stacked devices do)
Request based drivers
BIO based drivers
Request-based device mapper targets
dm-multipath
sysfs (transport attributes)
Transport classes
scsi_transport_fc
scsi_transport_sas
scsi_transport_...
SCSI mid layer
scsi-mq
SCSI upper level drivers
/dev/sda /dev/sd*
/dev/sr* /dev/st*
/dev/zram*
/dev/ubiblock*
/dev/rbd*
/dev/nbd*
/dev/mmcblk*p*
/dev/loop*
/dev/nullb*
/dev/vd*
/dev/rssd*
/dev/nvme*n*
/dev/skd*
/dev/rsxx*
zram ubi rbd nbd mmc loop null_blk virtio_blk mtip32xx nvme skd rsxx
memory
network
SCSI low level drivers
libata megaraid_sas qla2xxx pm8001 iscsi_tcp virtio_scsi ufs ...
ahci ata_piix ... aacraid lpfc mpt3sas vmw_pvscsi
Physical devices
HDD SSD DVD drive LSI RAID Adaptec RAID Qlogic HBA Emulex HBA PMC-Sierra HBA LSI 12Gbs SAS HBA
para-virtualized SCSI
virtio_pci
VMware's para-virtualized SCSI
mobile device flash memory
SD-/MMC-Card
Micron PCIe card
nvme device
stec device
IBM flash adapter
The Linux Storage Stack Diagram
http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Created by Werner Fischer and Georg Schönberger
License: CC-BY-SA 3.0, see http://creativecommons.org/licenses/by-sa/3.0/
THOMAS KRENN

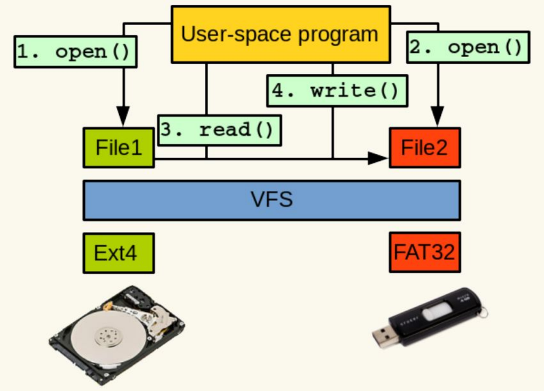

The Linux Storage Stack Diagram
version 4.10, 2017-03-10
outlines the Linux storage stack as of Kernel version 4.10
Syscall
VFS
FileSystems
Page Cache
Block Scheduler
Device Drivers
Hardware devices
Applications (processes)
mmap (anonymous pages)
malloc
vfs_writev, vfs_readv, ...
Direct I/O (O_DIRECT)
Block-based FS
ext2 ext3 ext4 xfs
btrfs ifs iso9660
gfs ocfs
Pseudo FS
sysfs
futexfs
usbfs
Special purpose FS
tmpfs ramfs
devtmpfs
ecryptfs
ceph
userspace (e.g. sshfs)
network
LIO
Fibre Channel over Ethernet
iSCSI
Fibre Channel
FireWire
USB
Virtual Host
tcm_fc
iscsi_target_mod
tcm_qla2xxx
sbp_target
tcm_usb_gadget
tcm_vhost
target_core_mod
target_core_file
target_core_iblock
target_core_pscsi
target_core_user
userspace
BIOs (block I/Os)
stackable
(optional)
Devices on top of "normal" block devices
drbd
LVM
device mapper
dm-crypt dm-mirror
dm-cache dm-thin
dm-raid dm-delay
mdraid
bcache
struct bio
- sector on disk
- sector cnt
- bio_vec cnt
- bio_vec index
- bio_vec list
BIOs
Maps
noop
hooked in device drivers (they hook in like stacked devices do)
Request based drivers
BIO based drivers
Request-based device mapper targets
dm-multipath
sysfs (transport attributes)
Transport classes
scsi_transport_fc
scsi_transport_sas
scsi_transport_...
scsi-mq
/dev/sda
/dev/sr*
/dev/st*
/dev/skd*
/dev/rsxx*
skd
rsxx
SCSI low level drivers
libata
megaraid_sas
qla2xxx
pm8001
iscsi_tcp
virtio_scsi
ufs
ahci
ata_piix
aacraid
lpfc
mpt3sas
vmw_pvscsi
network
HDD
SSD
DVD drive
LSI RAID
Qlogic HBA
Adaptec RAID
stec device
IBM flash adapter
Physical devices
The Linux Storage Stack Diagram
http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Created by Werner Fischer and Georg Schönberger
License: CC-BY-SA 3.0, see http://creativecommons.org/licenses/by-sa/3.0/
THOMAS KRENN

VFS



- Virtual File System (~22K SLOC).
- Everything is a File!!
  - E.g., Network file system! sshfs!?
- ~42 File Systems supported in Linux!!

VFS to Applications

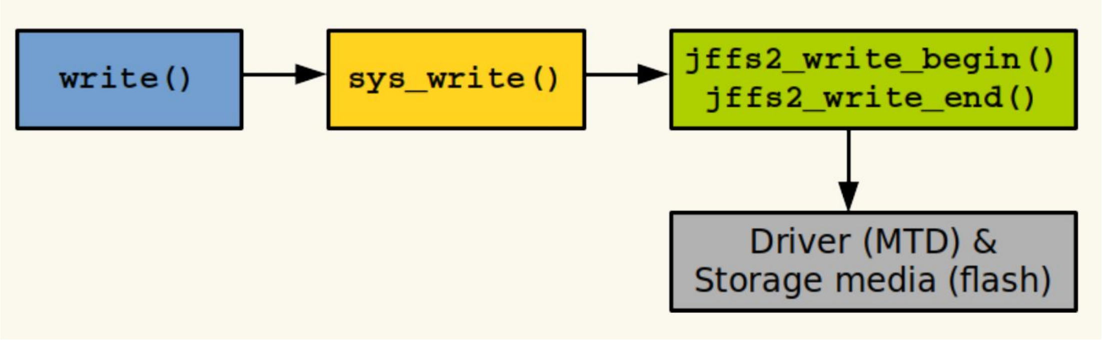


- Common interface for accessing files irrespective of file systems.
- File systems no need to worry about interface to user.

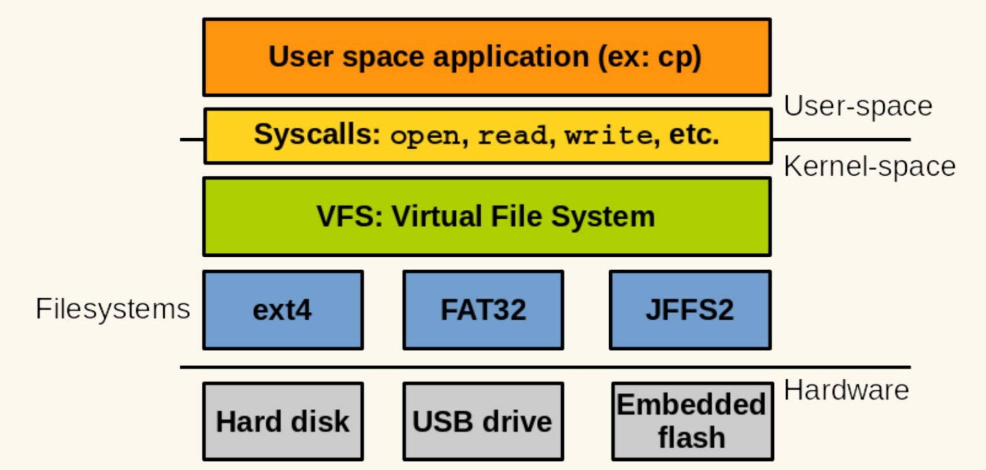
VFS to File System Implementers

- Exposes common optimization logic. E.g., Page cache, Path lookup.

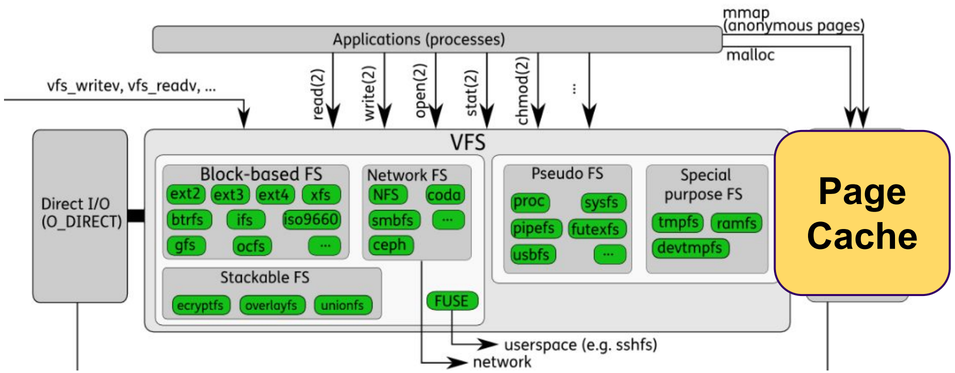
- Define functions to be implemented by the filesystems.



What does File System Implementers do?



Page Cache

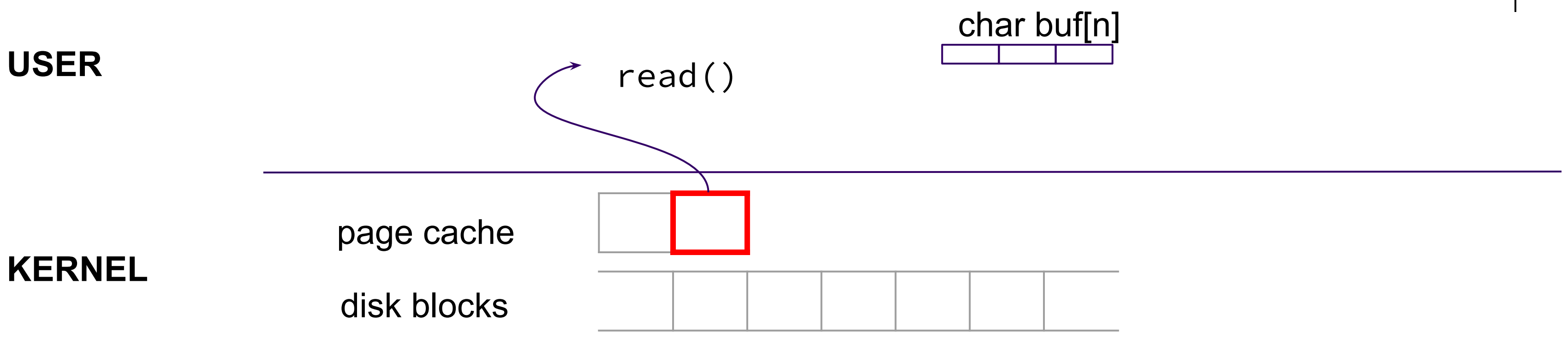


- Reduce Disk IO
- Memory pages maintained by the kernel for storing contents to/from disks.
- Disk block <-> Page

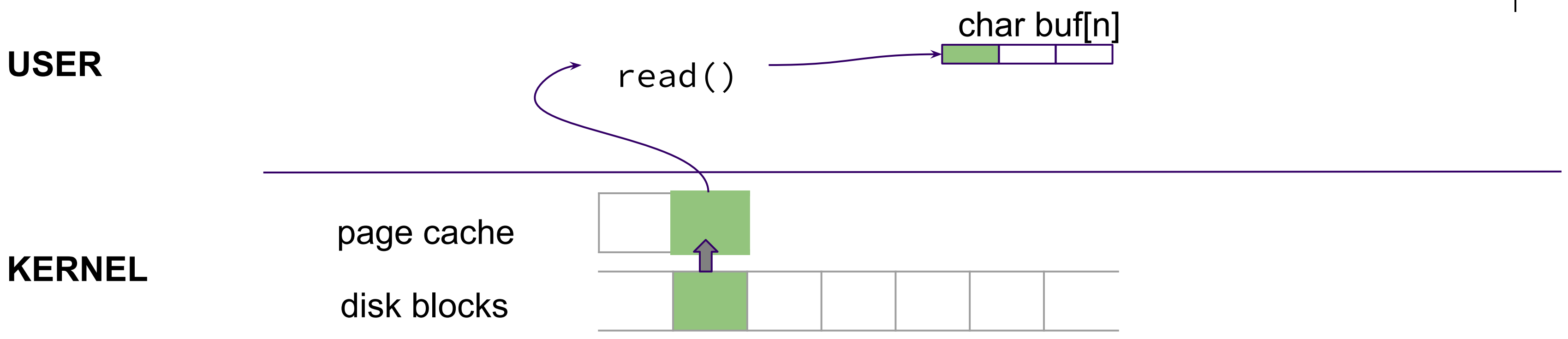
File IO with Page Cache

- `read()`: Serviced by Page Cache!
  - Optimization: Read ahead!
- `write()`: Dirty pages; will be written to disk later!
  - Can loose data!?
- `sync()`: Flush all writes to files.
  - Synchronous

File IO with Page Cache



File IO with Page Cache



Page Cache Implementation

- For each file (inode):
  - Has addr space.
  - File offset -> Page cache.
- For each page:
  - A reference to the file/process.
  - The offset with in the file.

The mmap system call

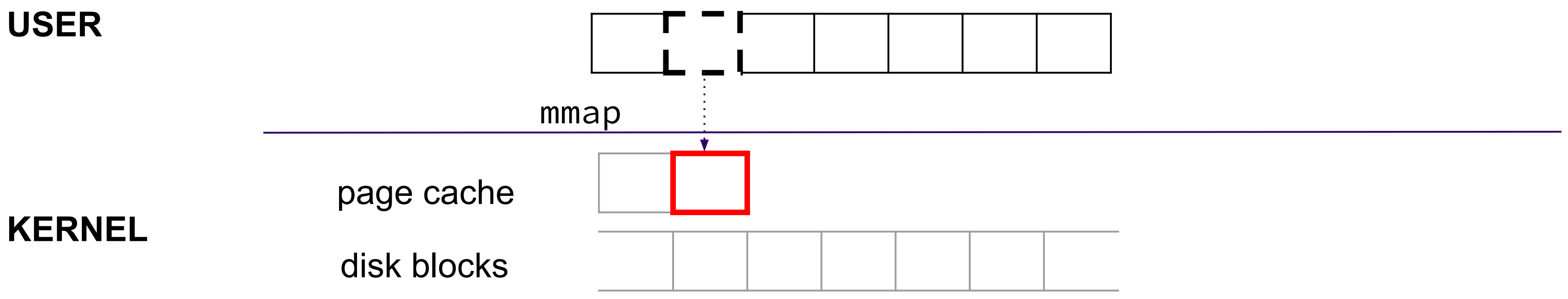
- Bind virtual memory to file blocks.

```
fd = open("hello.txt", O_RDWR);

// map 4k from offset 0 into virtual address space of the
process.
char *data = mmap(..,fd, 0);

// read 7th character from file.
char c = data[6];

// write 101th character into file.
data[100] = 'a'
```

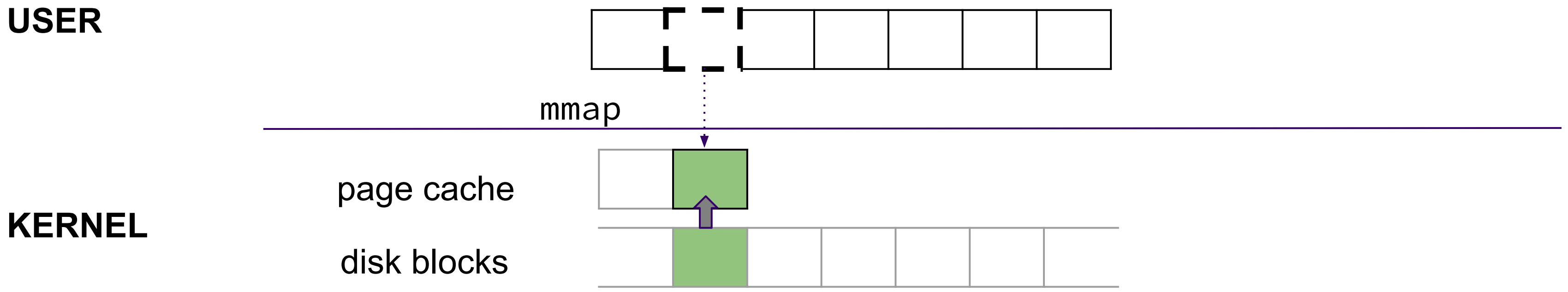
Flushing mmap region to file

```
MSYNC(2)

NAME
       msync - synchronize a file with a memory map

SYNOPSIS
       #include <sys/mman.h>

       int msync(void *addr, size_t length, int flags);

DESCRIPTION
       msync()  flushes  changes  made  to the in-core copy of a file that was mapped i
       part of the file that corresponds to the memory area starting at addr and having
```

Memory RW with Page Cache

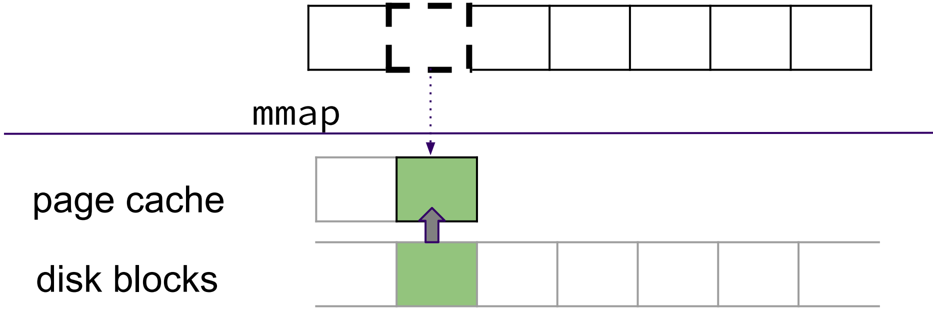


Memory RW with Page Cache



Mmap v/s Explicit IO

- Mmap:
  - No syscalls on each access.
  - Page cache <-> Disk.
  - Dynamic paging.
  - Extra PTEs.
  - Mapping large files? IO Errors?



- File IO
  - Universal.
  - app buffer <-> page cache <-> Disk.