DiscoverSDK
Tools for Developers

Linux VFS and Block Layers

Liran Ben Haim
liran@discoversdk.com

Rights to Copy

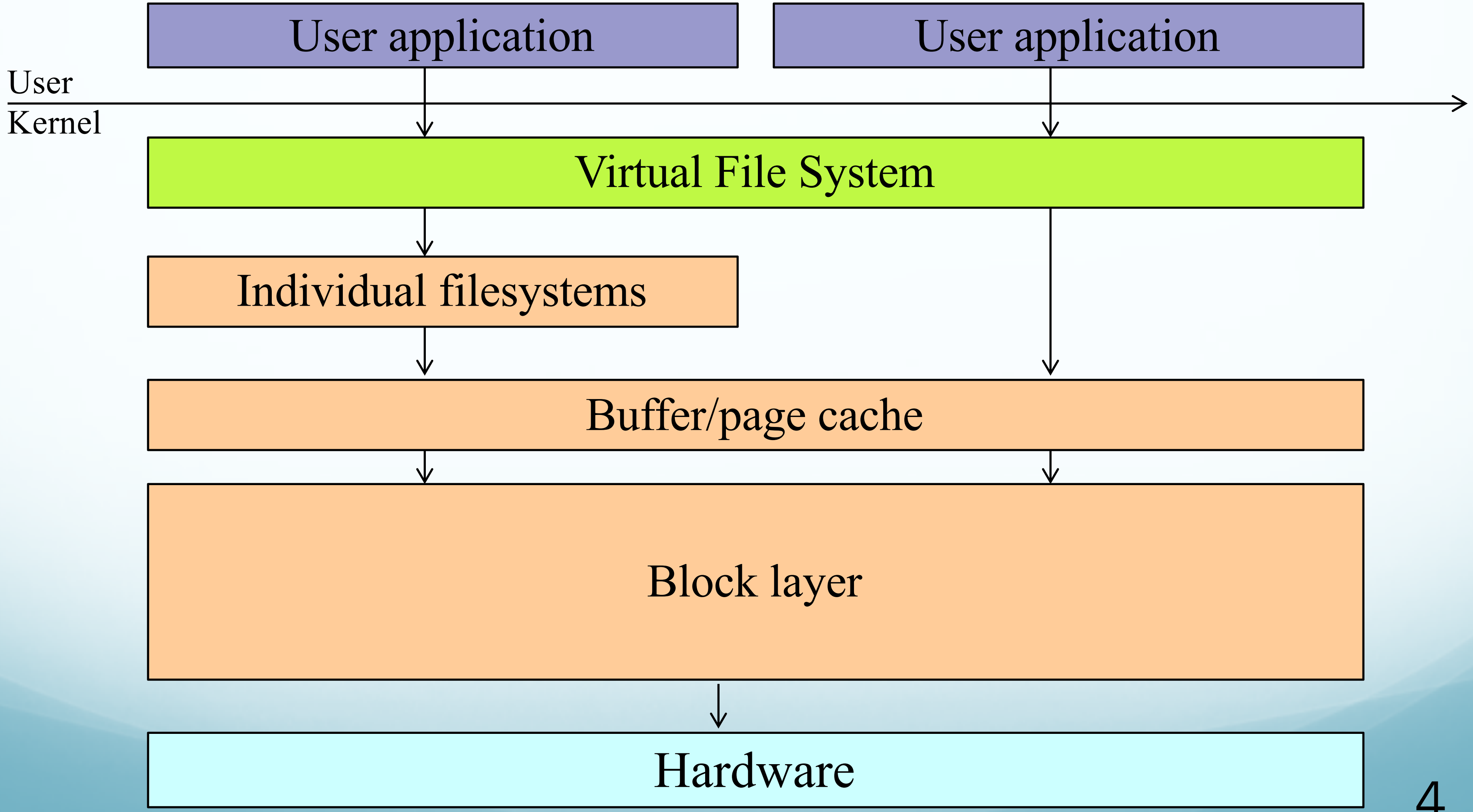
- **This kit contains work by the following authors:**
- © Copyright 2004-2009
  Michael Opdenacker /Free Electrons
  michael@free-electrons.com
  http://www.free-electrons.com
- © Copyright 2003-2006
  Oron Peled
  oron@actcom.co.il
  http://www.actcom.co.il/~oron
- © Copyright 2004–2008
  Codefidence ltd.
  info@codefidence.com
  http://www.codefidence.com
- © Copyright 2009–2017
  Bina ltd.
  info@bna.co.il
  http://www.bna.co.il

DiscoverSDK
Tools for Developers

Block Device Drivers

- Linux Drivers types:
  - Character Device Drivers
  - Block Device Drivers
  - Network Device Drivers
- Block Devices are used for storage
- The name “block device” comes from the fact that the corresponding hardware typically reads and writes a whole block at a time

Architecture



VFS

- Linux provides a unified Virtual File System interface:
  - The VFS layer supports abstract operations.
  - Specific file systems implements them.
- File operations always start with the VFS layer
  - Regular file
  - /dev/sda1
  - /proc/cpuinfo
  - ...

Example - read

- User space:

  x = read(fd, buffer, size);

- Kernel:

  sys_read(fd , buffer, size);

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
	struct fd f = fdget_pos(fd);
	ssize_t ret = -EBADF;

	if (f.file) {
		loff_t pos = file_pos_read(f.file);
		ret = vfs_read(f.file, buf, count, &pos);
		if (ret >= 0)
			file_pos_write(f.file, pos);
		fdput_pos(f);
	}
	return ret;
}
```

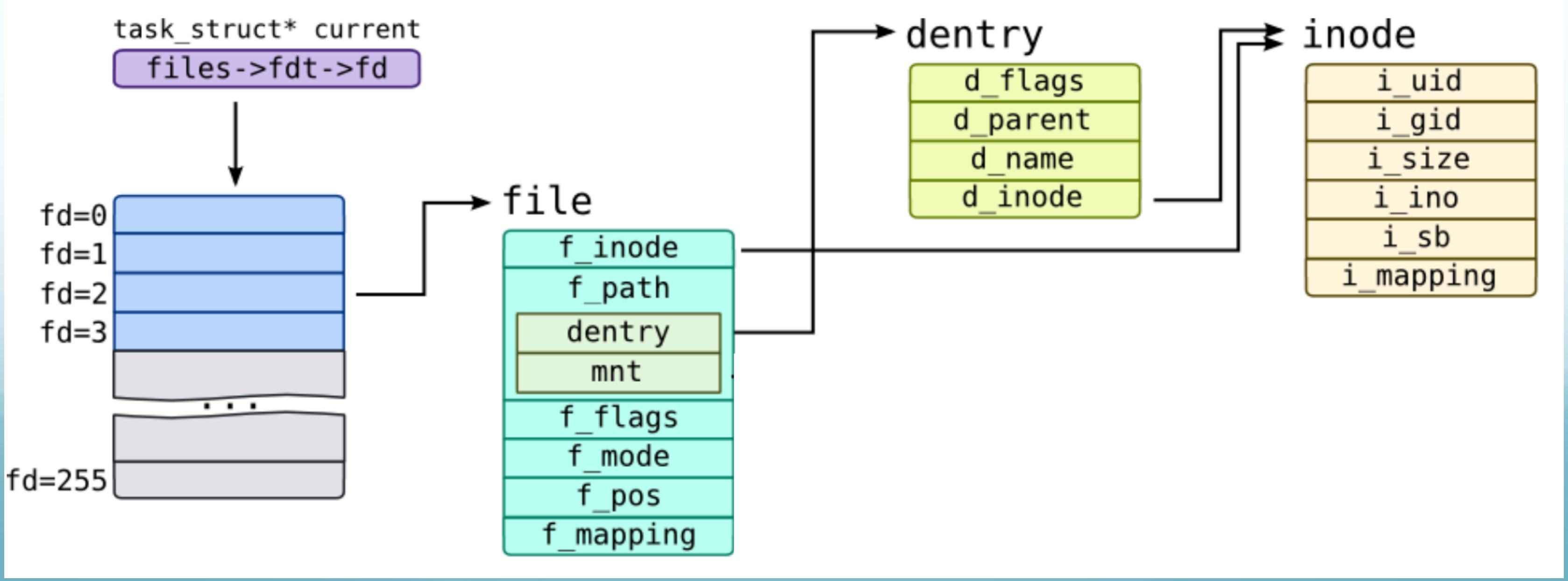
vfs_read

- Performs some checks and calls __vfs_read

```
ssize_t __vfs_read(struct file *file, char __user *buf, size_t count,
                   loff_t *pos)
{
        if (file->f_op->read)
                return file->f_op->read(file, buf, count, pos);
        else if (file->f_op->read_iter)
                return new_sync_read(file, buf, count, pos);
        else
                return -EINVAL;
}
```

VFS

- The major VFS abstract objects:
  - File - An open file (file descriptor).
  - Dentry - A directory entry. Maps names to inodes
  - Inode - A file inode. Contains persistent information
  - Superblock - descriptor of a mounted filesystem



File Object

- Information about an open file
  - Mode
  - Position
  - ...
- Per process
  - you can set the table size using ulimit -n

Dentry Object

- Information about a directory entry
  - Name
  - Pointer to the inode
- Multiple dentries can point to a single inode
  - Hard links

```
developer@:~/test$ ln -s file1 file3
developer@:~/test$ ln file1 file4
developer@:~/test$
developer@:~/test$ ls -l
total 168
-rw-rw-r-- 2 developer developer  10164 אוק 23 16:37 file1
-rw-rw-r-- 1 developer developer 143976 אוק 23 16:37 file2
lrwxrwxrwx 1 developer developer      5 אוק 23 16:37 file3 -> file1
-rw-rw-r-- 2 developer developer  10164 אוק 23 16:37 file4
developer@:~/test$ ls -li
total 168
2386855 -rw-rw-r-- 2 developer developer  10164 אוק 23 16:37 file1
2386856 -rw-rw-r-- 1 developer developer 143976 אוק 23 16:37 file2
2386857 lrwxrwxrwx 1 developer developer      5 אוק 23 16:37 file3 -> file1
2386855 -rw-rw-r-- 2 developer developer  10164 אוק 23 16:37 file4
developer@:~/test$
```

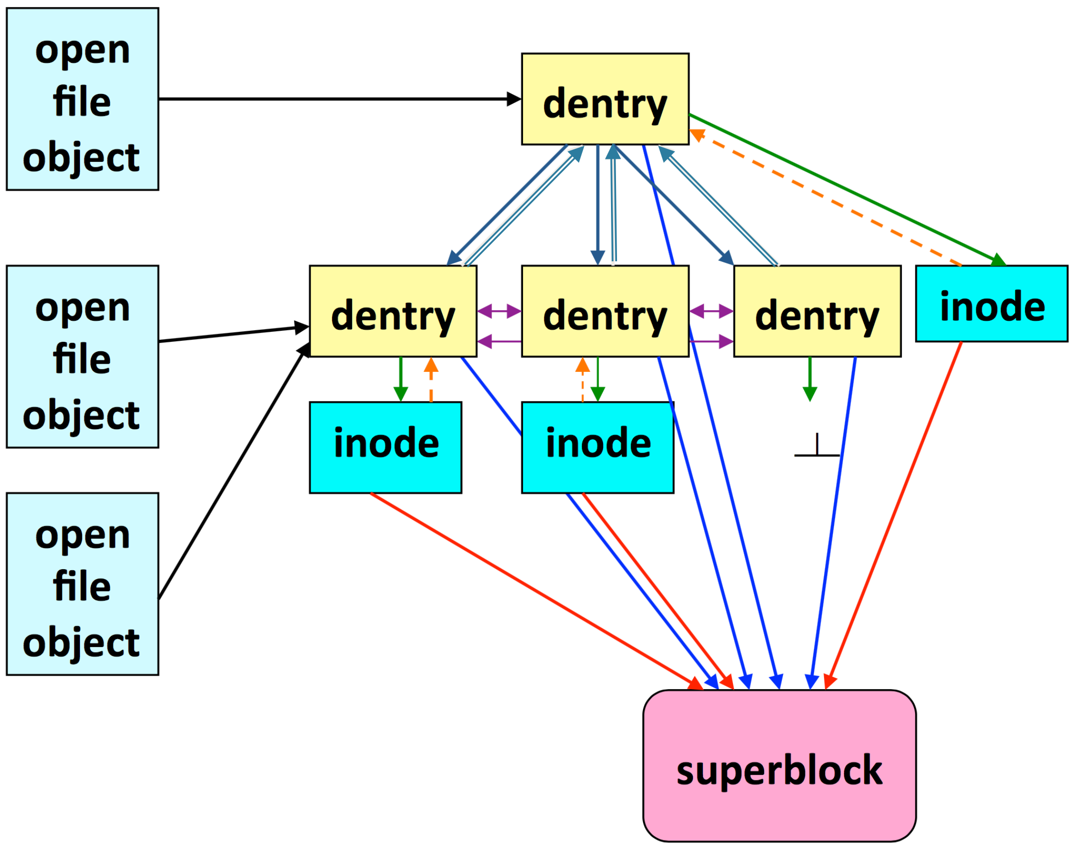
dcache

- The VFS implements the open(2), stat(2), chmod(2), and similar system calls.
  - The pathname argument that is passed to them is used by the VFS to search through the directory entry cache (dcache)
- This provides a very fast look-up mechanism to translate a pathname (filename) into a specific dentry.
- Dentries live in RAM and are never saved to disc

DiscoverSDK
Tools for Developers

Inode Object

- unique descriptor of a file or directory
- contains permissions, timestamps, block map (data)
- Usually stored in the a special block(s) on the disk
- inode#: integer (unique per mounted filesystem)
- Filesystem: fn(inode#) => data



f_dentry	
d_subdirs	
d_inode	
d_subdirs	
d_parent	
i_sb	
d_sb	
i_dentries	

Superblock

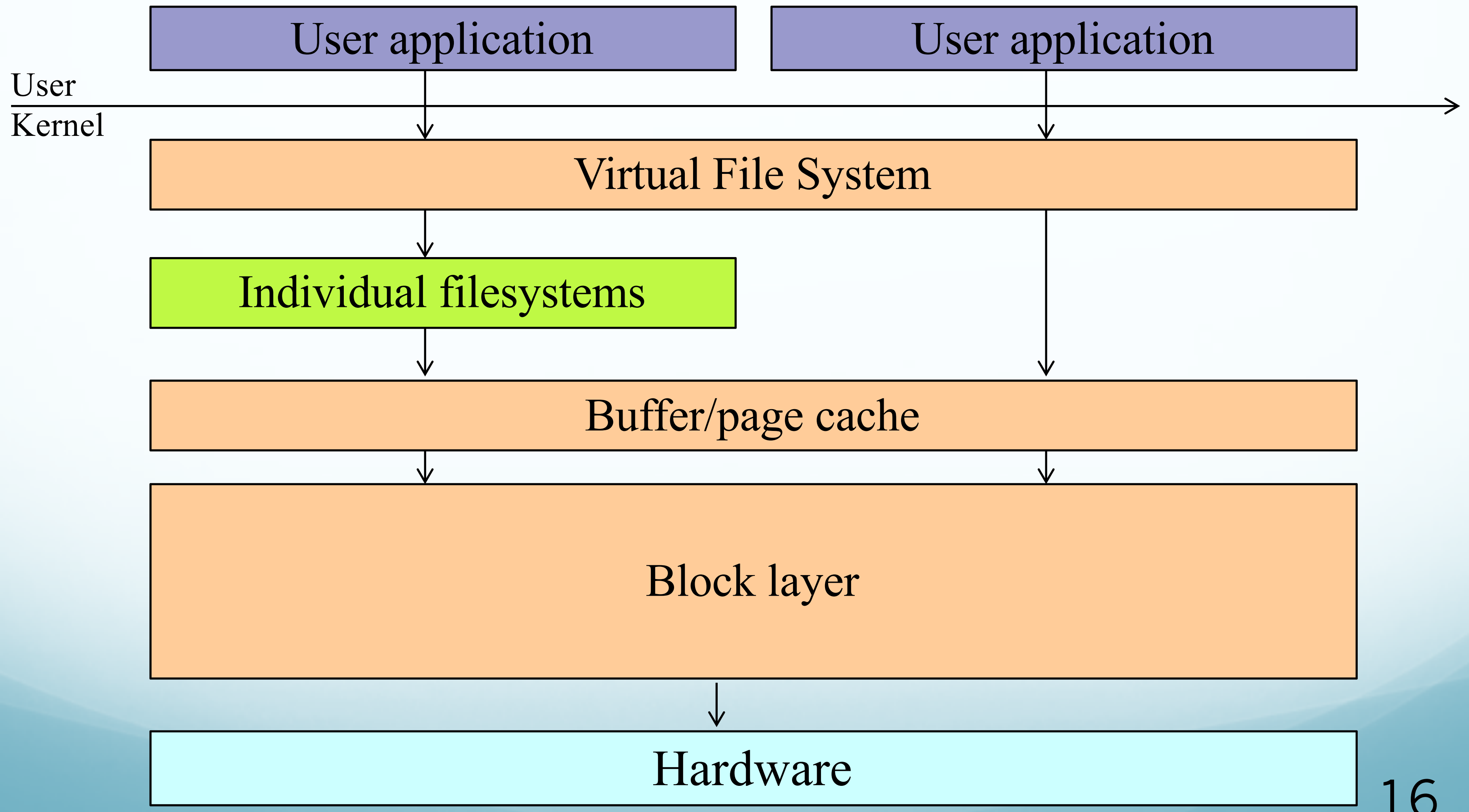
- The file system metadata
- Defines the file system type, size, status, and other information about other metadata structures
- Usually first block on disk (ater boot block)
- Copied into (similar) memory structure on mount

struct vfsmount

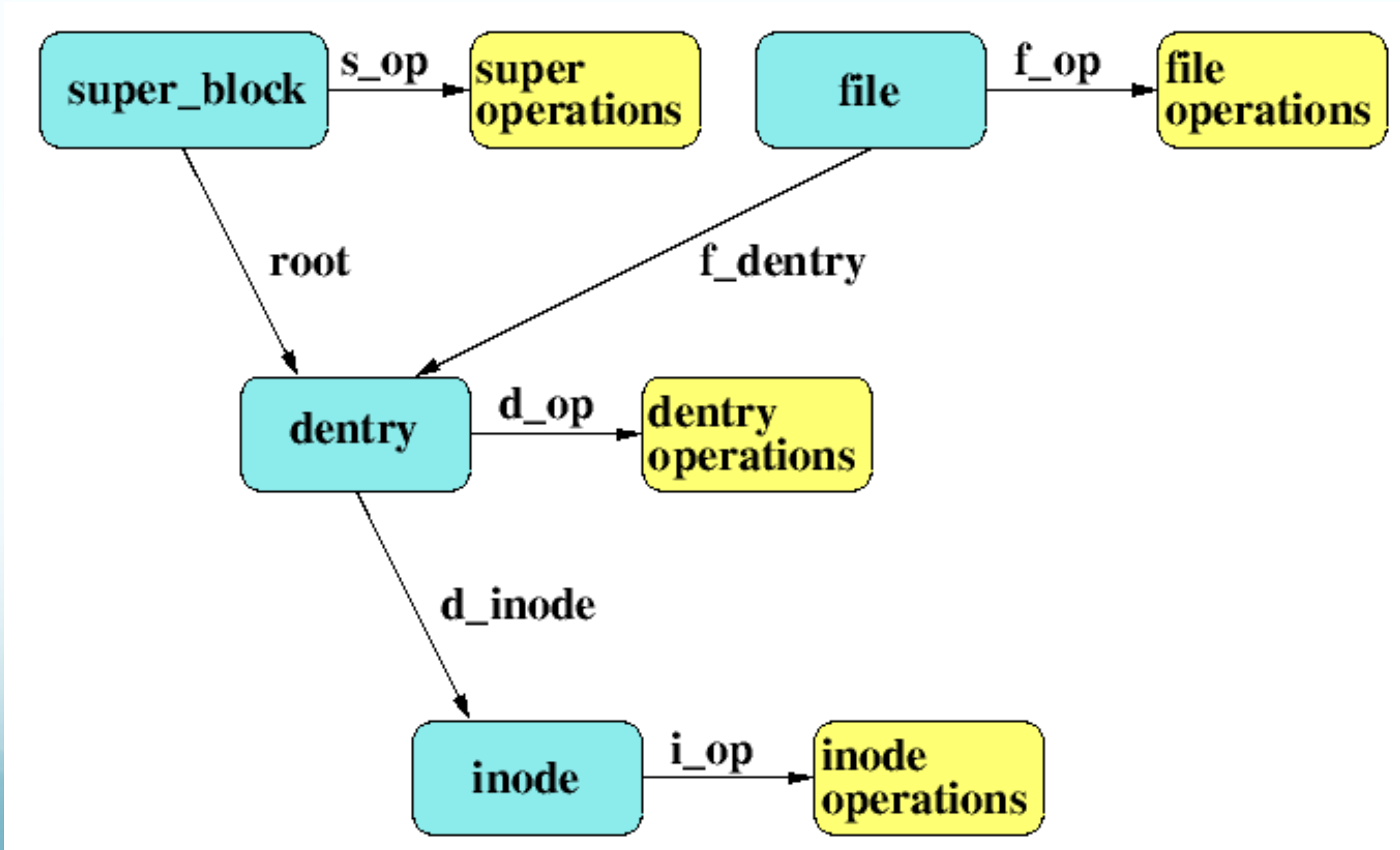
- Represents a mounted instance of a particular file system

```
struct vfsmount {
        struct dentry *mnt_root;        /* root of the mounted tree */
        struct super_block *mnt_sb;     /* pointer to superblock */
        int mnt_flags;
} __randomize_layout;
```

Architecture



VFS Structures



Registering a new FS

- Call register_file_system and pass a pointer to:
  - struct file_system_type
- Fields:
  - Name (/proc/filesystems)
  - Flags
  - Mount callback

Mount

- mount –t myfs /dev/myblk /myfs
- The mount callback is called
- Typical implementation:
  - mount_bdev/mount_nodev/mount_mtd

```
static struct dentry *efs_mount(struct file_system_type *fs_type,
        int flags, const char *dev_name, void *data)
{
        return mount_bdev(fs_type, flags, dev_name, data, efs_fill_super);
}
```

Filling the super block

- Extents the super block to add private data

```
asb = kzalloc(sizeof(*asb), GFP_KERNEL);
if (!asb)
        return -ENOMEM;
sb->s_fs_info = asb;
```

- Set the block size
- Read the super block data from the device
  - sb_bread(sb, block_num)
- Set super_operations field

Filling the super block(2)

- Create a root inode
  - Set inode_operations
  - Set file_operations
  - Set address_space_operations
- Create a root dentry
  - d_make_root

```
root = adfs_iget(sb, &root_obj);
sb->s_root = d_make_root(root);
```

super_operations

- alloc/read/write/clear/delete inode
- put_super (release)
- freeze/unfreeze/remount/sync the file system
- show_options (/proc/[pid]/mounts)
- statfs – statfs(2)

inode_operations

- create – new inode for regular file
- link/unlink/rename – add/remove/modify dir entry
- symlink, readlink, get_link – sot link ops
- mkdir/rmdir – new inode for directory file
- mknod – new inode for device file
- permission – check access permissions
- lookup – called when the VFS needs to look up an inode in a parent directory

file_operations

- open/release
- read/write
- read_iter/write_iter – async ops
- iterate – directory content(ls)
- mmap/lock/sync/poll
- *_ioctl
- ...

dentry_operations

- The filesystem can overload the standard dentry operations
- Special cases
  - Msdos 8.3 limit
  - fat case insensitive

ls /myfs

- stat(2) the path
- open(2) the path for read with O_DIRECORY
- getdents(2) to get all dentries (multiple time until it returns 0)
  - iterate_dir
  - Calls iterate callback in file_operations for that inode

```
if (!IS_DEADDIR(inode)) {
        ctx->pos = file->f_pos;
        if (shared)
                res = file->f_op->iterate_shared(file, ctx);
        else
                res = file->f_op->iterate(file, ctx);
```

iterate

- Checks the requested position (the user buffer can be smaller than the directory content)
- Read the data from the device (sb_sread)
- Call dir_emit* for each directory entry

Simple example

```
static int simpfs_iterate(struct file *file, struct dir_context *ctx)
{
        struct inode *inode = file_inode(file);
        struct dentry *de = file->f_path.dentry;
        int parent = inode->i_ino;

        if(ctx->pos == 4)
                return 0;

        if (ctx->pos < 2) {
                if (!dir_emit_dots(file, ctx)) // create the . and .. directories
                        return 0;
        }


        dir_emit(ctx,"Dir1",4,parent,DT_DIR);

        dir_emit(ctx, "testfs", 6, parent, DT_REG);
        ctx->pos = 4;

        return 0;

}
```

inode_operations lookup

- For each directory entry name, we need to find and fill dentry object and inode object
- Called when the VFS needs to look up an inode in a parent directory. The name to look for is found in the dentry
  - Get/allocate inode (maybe read from device)
  - Call d_add(dentry,inode) ;

DiscoverSDK
Tools for Developers

Open a file

- sys_open
- do_sys_open
- do_flip_open
- path_openat
  - get_empty_flip
  - do_last
    - vfs_open
      - do_dentry_open
        - Set the file_operations (fops_get)
        - Call the open callback if exists

Read/Write

- sys_read -> vfs_read -> __vfs_read
- sys_write -> vfs_write -> __vfs_write

```
ssize_t __vfs_write(struct file *file, const char __user *p, size_t count,
                    loff_t *pos)
{
        if (file->f_op->write)
                return file->f_op->write(file, p, count, pos);
        else if (file->f_op->write_iter)
                return new_sync_write(file, p, count, pos);
        else
                return -EINVAL;
}
```

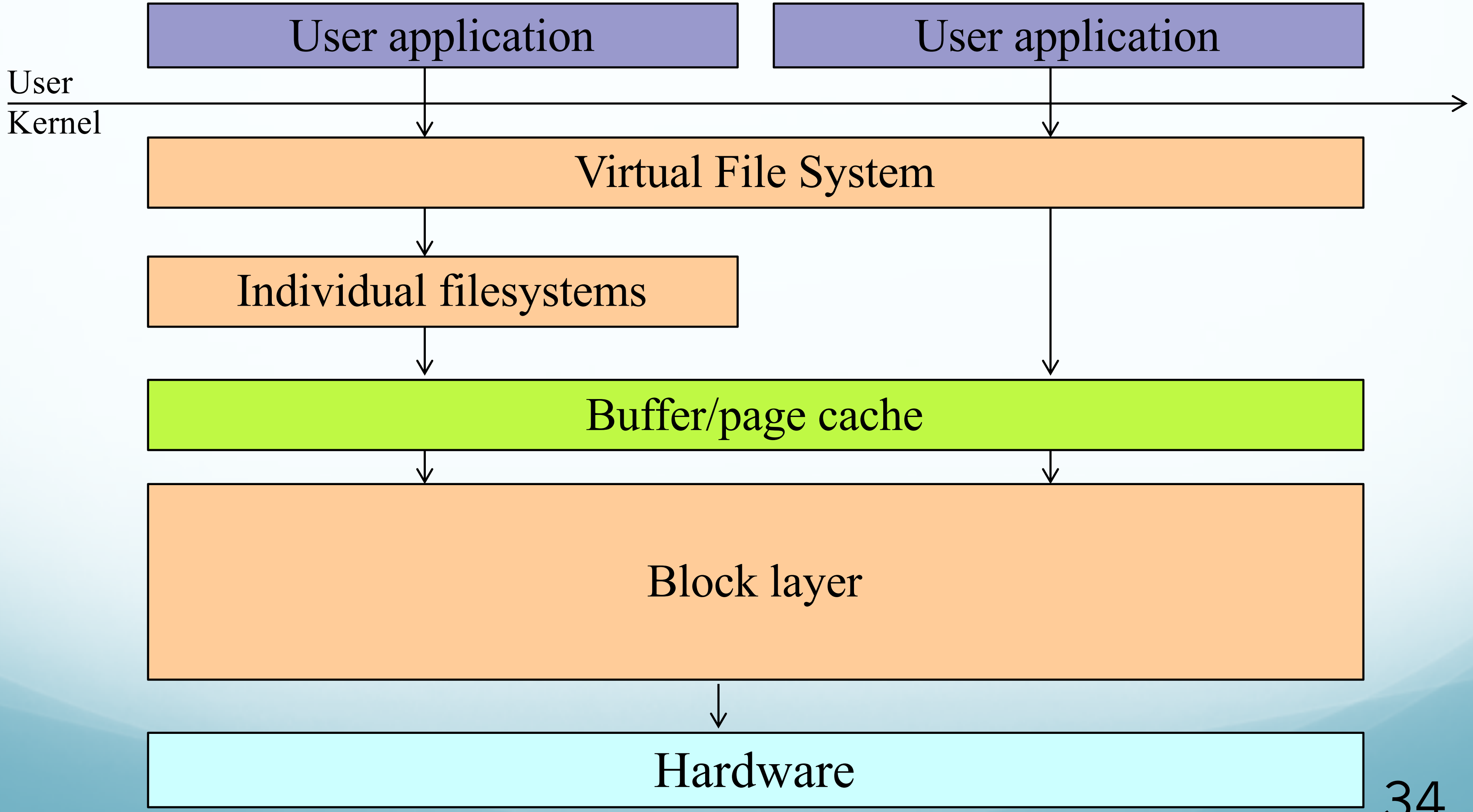
Generic Functions

```
int generic_file_mmap(struct file *, struct vm_area_struct *);
int generic_file_readonly_mmap(struct file *, struct vm_area_struct *);
ssize_t generic_write_checks(struct kiocb *, struct iov_iter *);
ssize_t generic_file_read_iter(struct kiocb *, struct iov_iter *);
ssize_t __generic_file_write_iter(struct kiocb *, struct iov_iter *);
ssize_t generic_file_write_iter(struct kiocb *, struct iov_iter *);
ssize_t generic_file_direct_write(struct kiocb *, struct iov_iter *);
ssize_t generic_perform_write(struct file *, struct iov_iter *, loff_t);
```

libfs

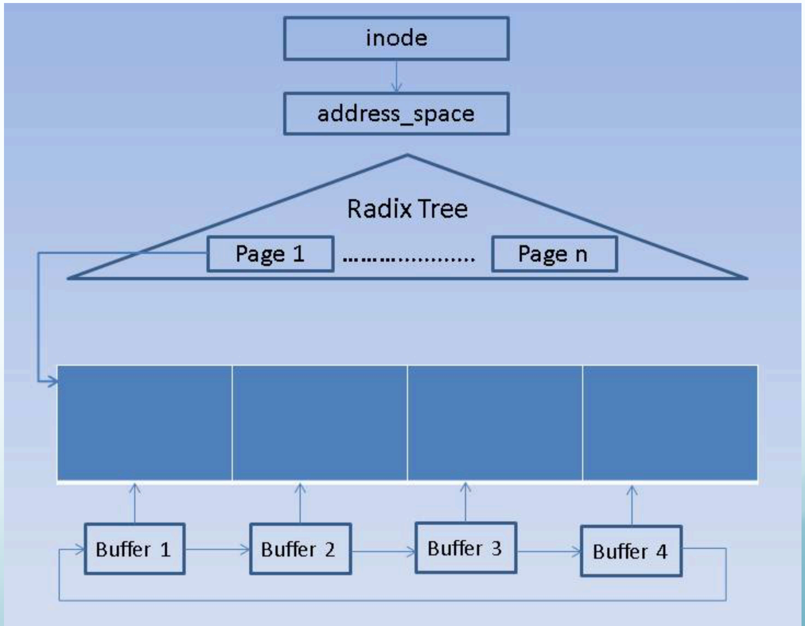
- /fs/libfs.c – library for filesystem writer
- simple_* functions
  - simple_lookup, simple_mkdir, ...
- Simple file_operations
- Simple inode_operations

Architecture



Integration with Memory Subsystem

- The address_space object
  - Used to group and manage pages in the page cache
  - One per file
  - The “physical analogue” to the virtual vm_area_struct
  - Radix tree enables quick searching for the desired page, given only the file offset
- The address_space_operation structure
  - Implement reading and writing pages
- You can choose not to use it
  - Read and write directly
  - Examples: efivarfs , openpromfs, pseudo filesystems like proc, sysfs

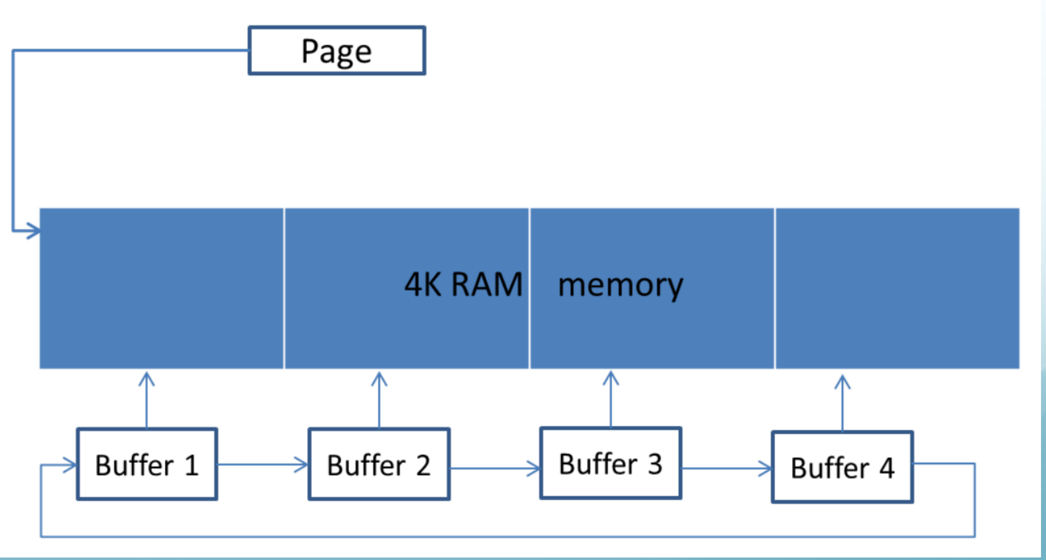


address_space_operations

- readpage – read a page from backing store
- writepage - write a dirty page to backing store
- readpages/writepages
- set_page_dirty
- write_begin - Called by the generic buffered write code to ask the filesystem to prepare to write len bytes at the given offset in the file
- write_end– called after a successful write_begin, and data copy

The Page Cache

- Page cache can read individual disk blocks whose size is determined by the filesystem
- Use sb_bread to read the corresponding block from the block device and store the block in a buffer
  - or just return it from memory
- The block device specified in the super block



DiscoverSDK
Tools for Developers

The Page Cache

- Use mark_buffer_dirty to flag the buffer as dirty
  - Need their data to be synced to disk
- After sb_bread, the buffer_head and the data block contents are pinned in memory. The page cache won't remove them
- Use brelse to release it and let the kernel free the buffer_head (only frees when there is a memory pressure)
- If kernel decides to free a buffer_head, it will sync its data to disk, **but only if the buffer_head marked dirty**

DiscoverSDK Tools for Developers

Read/Write Examples

```
if (!(bh = sb_bread(info->vfs_sb, block)))
{
        return -EIO;
}
memcpy(buf, bh->b_data + offset, len);
brelse(bh);
```

```
if (!(bh = sb_bread(info->vfs_sb, block)))
{
        return -EIO;
}
memcpy(bh->b_data + offset, buf, len);
mark_buffer_dirty(bh);
brelse(bh);
```

DiscoverSDK
Tools for Developers

Simple readpage()

```c
static int simp_readpage(struct file *file, struct page *page)
{
        return mpage_readpage(page, find_and_map_block_fn);
}
```

```c
/*
 * This is the worker routine which does all the work of mapping the disk
 * blocks and constructs largest possible bios, submits them for IO if the
 * blocks are not contiguous on the disk.
 *
 * We pass a buffer_head back and forth and use its buffer_mapped() flag to
 * represent the validity of its disk mapping and to decide when to do the next
 * get_block() call.
 */
static struct bio *
do_mpage_readpage(struct bio *bio, struct page *page, unsigned nr_pages,
                sector_t *last_block_in_bio, struct buffer_head *map_bh,
                unsigned long *first_logical_block, get_block_t get_block,
                gfp_t gfp)
{
```

DiscoverSDK
Tools for Developers

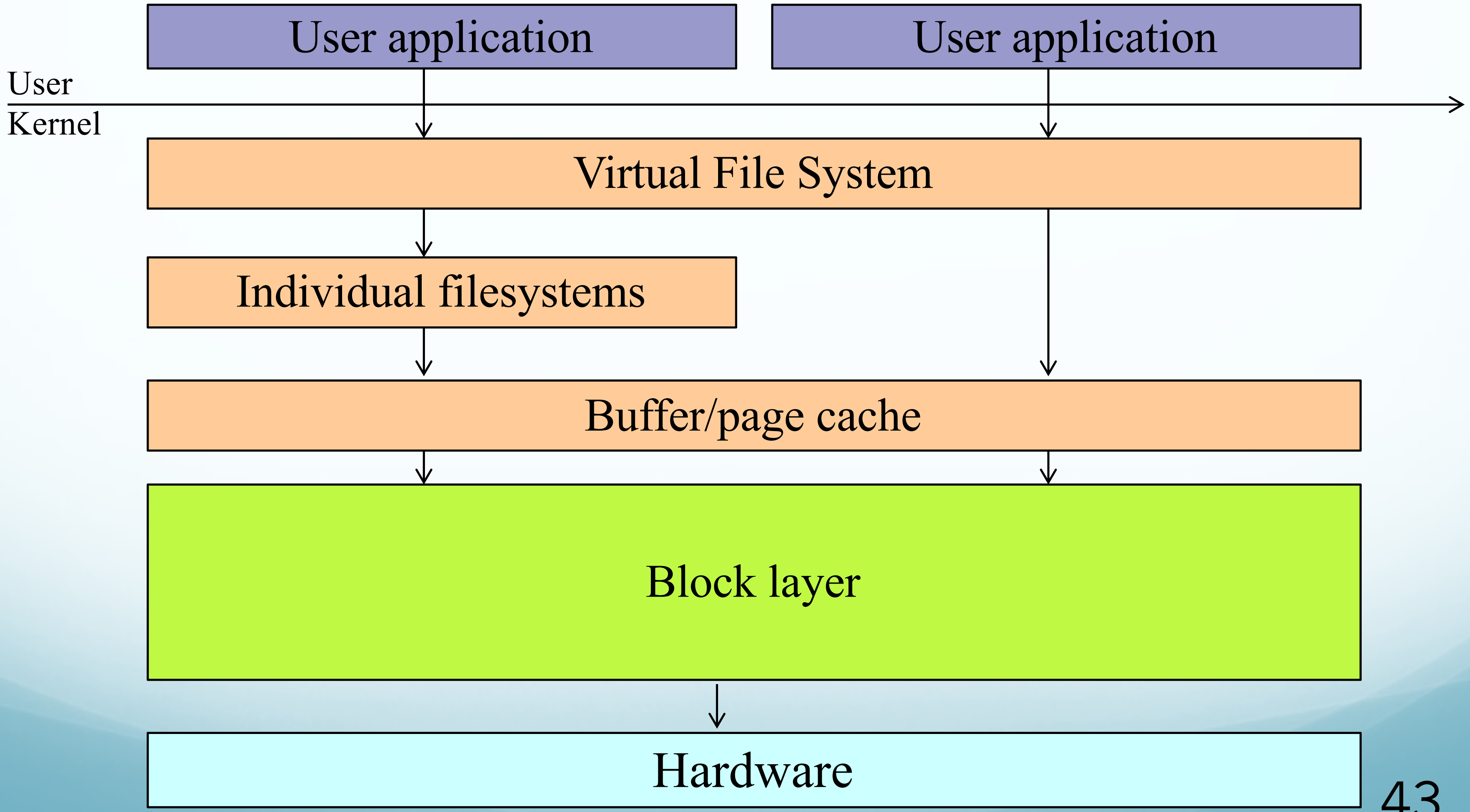
bio – IO Request

- Historically, a buffer_head was used to map a single block within a page, and of course as the unit of I/O through the filesystem and block layers.
- Nowadays the basic I/O unit is the bio
  - See EXT4 readpages for example (submit_bio)
- buffer_heads are used for:
  - extracting block mappings (via a get_block_t call),
  - tracking state within a page (via a page_mapping)
  - wrapping bio submission for backward compatibility reasons (e.g. submit_bh).

submit_bh

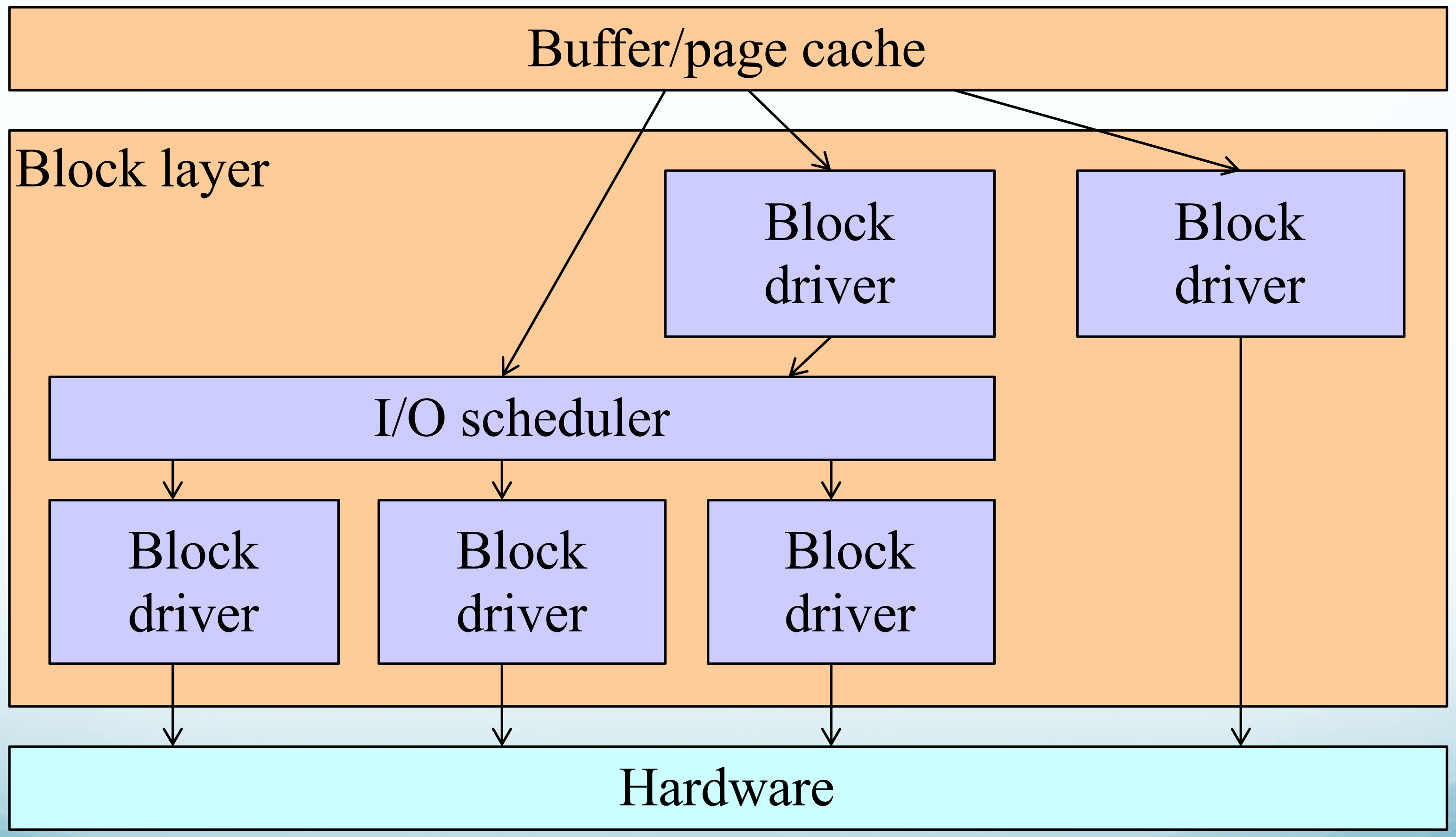
- Calls submit_bh_wbc
- From here on down, it's all bio
  - bio_alloc
  - bio_add_page
  - ...
  - submit_bio
    - To the request layer
- sb_bread -> .... -> submit_bh -> submit_bio
- See code in fs/buffer.c

Architecture



DiscoverSDK
Tools for Developers

Inside the block layer



Inside the block layer (2)

- The block layer allows block device drivers to receive I/O requests, and is in charge of I/O scheduling
- I/O scheduling allows to
  - Merge requests so that they are of greater size
  - Re-order requests so that the disk head movement is as optimized as possible
- Several I/O schedulers with different policies are available in Linux.
- A block device driver can handle the requests before or after they go through the I/O scheduler

DiscoverSDK
Tools for Developers

Two main types of drivers

- Most of the block device drivers are implemented below the I/O scheduler, to take advantage of the I/O scheduling
  - Hard disk drivers, CD-ROM drivers, etc.
- For some drivers however, it doesn't make sense to use the IO scheduler
  - RAID and volume manager, like md
  - The special loop driver
  - Memory-based block devices

DiscoverSDK
Tools for Developers

Available I/O schedulers

- I/O schedulers in current kernels
  - Noop, for non-disk based block devices
  - Deadline, tries to guarantee that an I/O will be served within a deadline
  - CFQ, the Complete Fairness Queuing, the default scheduler, tries to guarantee fairness between users of a block device
- The current scheduler for a device can be get and set in /sys/block/<dev>/queue/scheduler

Looking at the code

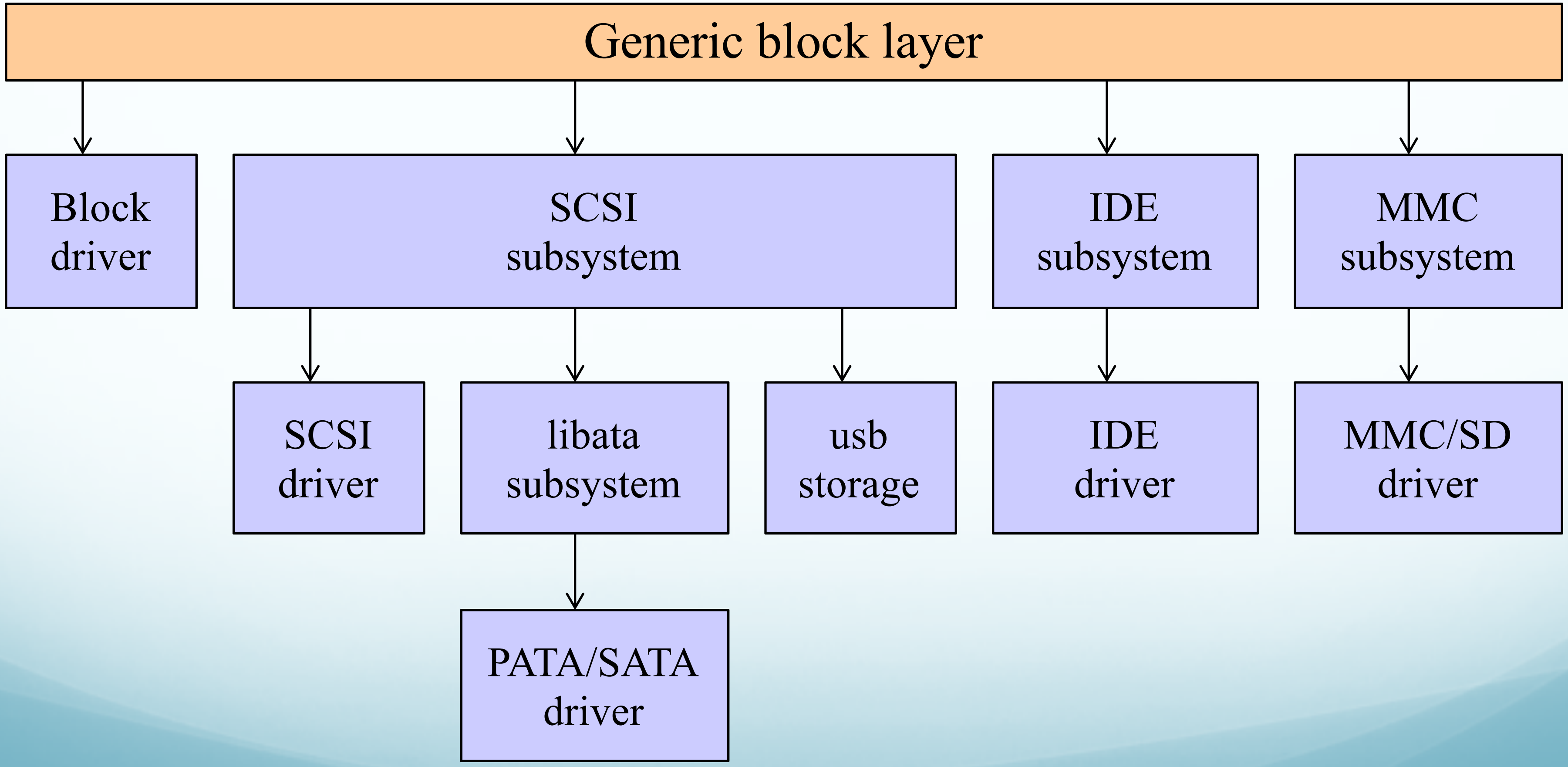
- The block device layer is implemented in the block/ directory of the kernel source tree
  - This directory also contains the I/O scheduler code, in the
    *-iosched.c files.
- A few simple block device drivers are implemented in drivers/block/, including
  - loop.c, the loop driver that allows to see a regular file as a block device
  - brd.c, a ramdisk driver
  - nbd.c, a network-based block device driver

DiscoverSDK
Tools for Developers

Implementing a block device driver

- A block device driver must implement a set of operations to be registered in the block layer and receive requests from the kernel
- A block device driver can directly implement this set of operation. However, as in many areas in the kernel, subsystems have been created to factorize common code of drivers for devices of the same type
  - SCSI devices
  - PATA/SATA devices
  - MMC/SD devices
  - etc.

Implementing a block device driver (2)



Registering the major

- The first step in the initialization of a block device driver is the registration of the major number
  - int register_blkdev(unsigned int major,
    const char *name);
  - Major can be 0, in which case it is dynamically allocated
  - Once registered, the driver appears in /proc/devices with the other block device drivers
- Of course, at cleanup time, the major must be unregistered
  - void unregister_blkdev(unsigned int major, const char *name);
- The prototypes of these functions are in <linux/fs.h>

struct gendisk

- The structure representing a single block device, defined in <linux/genhd.h>
  - int major, major of the device driver
  - int first_minor, minor of this device. A block device can have several minors when it is partitionned
  - int minors, number of minors. 1 for non-partitionable devices
  - struct block_device_operations *fops, pointer to the list of block device operations
  - struct request_queue *queue, the queue of requests
  - sector_t capacity, size of the block device in sectors

Initializing a disk

- Allocate a gendisk structure
  struct gendisk *alloc_disk(int minors)

  minors tells the number of minors to be allocated for this disk. Usually 1, unless your device can be partitionned

- Allocate a request queue
  struct request_queue *blk_init_queue
  (request_fn_proc *rfn, spinlock_t *lock)

  rfn is the request function (covered later). lock is a optional spinlock needed to protect the request queue against concurrent access. If NULL, a default spinlock is used

Initializing a disk (2)

- Initialize the gendisk structure
  Fields major, first_minor, fops, disk_name and queue should at the minimum be initialized
  private_data can be used to store a pointer to some private information for the disk
- Set the capacity
  void set_capacity(struct gendisk *disk, sector_t size)

  The size is a number of 512-bytes sectors. sector_t is 64 bits wide on 64 bits architectures, 32 bits on 32 bits architecture, unless CONFIG_LBD (large block devices) has been selected

Initializing a disk (3)

- Add the disk to the system
  void add_disk(struct gendisk *disk);

  The block device can now be accessed by the system, so the driver must be fully ready to handle I/O requests before calling add_disk(). I/O requests can even take place during the call to add_disk().

Unregistering a disk

- Unregister the disk
  void del_gendisk(struct gendisk *gp);

- Free the request queue
  void blk_cleanup_queue(struct request_queue *);

- Drop the reference taken in alloc_disk()
  void put_disk(struct gendisk *disk);

block_device_operations

- A set of function pointers
  - open() and release(), called when a device handled by the driver is opened and closed
  - ioctl() for driver specific operations. unlocked_ioctl() is the non-BKL variant, and compat_ioctl() for 32 bits processes running on a 64 bits kernel
  - direct_access() required for XIP support, see http://lwn.net/Articles/135472/
  - media_changed() and revalidate() required for removable media support
  - getgeo(), to provide geometry informations to userspace

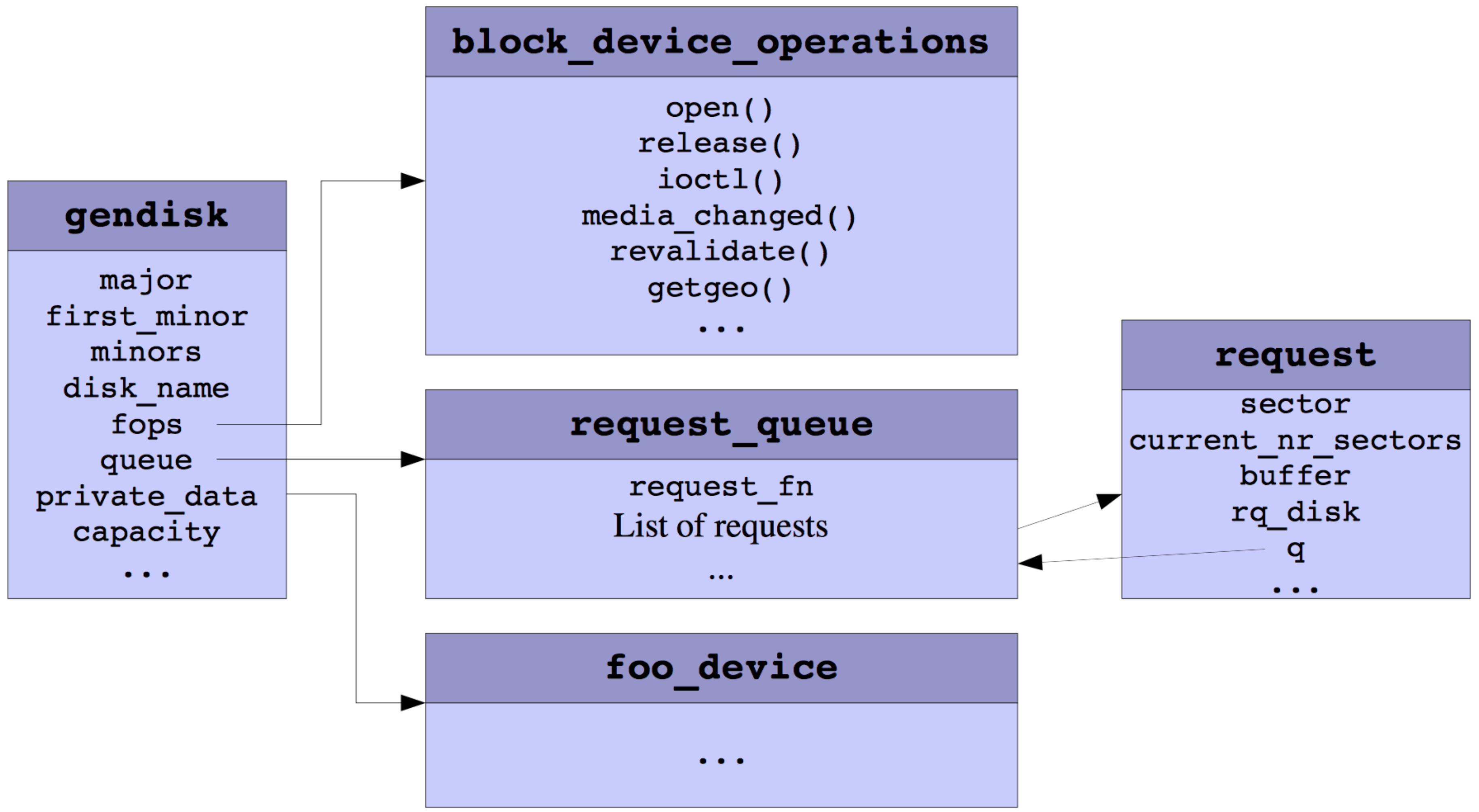
A simple request() function

```
static void foo_request(struct request_queue *q)
{
    struct request *req;
    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            __blk_end_request(req, 1, req->nr_sectors << 9);
            continue;
        }
        /* Do the transfer here */
        __blk_end_request(req, 0, req->nr_sectors << 9);
    }
}
```

A simple request() function (2)

- Information about the transfer are available in the struct request
  - sector, the position in the device at which the transfer should be made
  - current_nr_sectors, the number of sectors to transfer
  - buffer, the location in memory where the data should be read or written to
  - rq_data_dir(), the type of transfer, either READ or WRITE
- __blk_end_request() or blk_end_request() is used to notify the completion of a request. __blk_end_request() must be used when the queue lock is already held

Data structures



Request queue configuration (1)

- blk_queue_bounce_limit(queue, u64)
  Tells the kernel the highest physical address that the device can handle. Above that address, bouncing will be made. BLK_BOUNCE_HIGH, BLK_BOUNCE_ISA and BLK_BOUNCE_ANY are special values
  - HIGH: will bounce if the pages are in high-memory
  - ISA: will bounce if the pages are not in the ISA 16 Mb zone
  - ANY: will not bounce

Request queue configuration (2)

- blk_queue_max_sectors(queue, unsigned int)
  Tell the kernel the maximum number of 512 bytes sectors for each request.

- blk_queue_max_phys_segments(queue, unsigned short)
  blk_queue_max_hw_segments(queue, unsigned short)
  Tell the kernel the maximum number of non-memory-adjacent segments that the driver can handle in a single request (default 128).

- blk_queue_max_segment_size(queue, unsigned int)
  Tell the kernel how large a single request segment can be

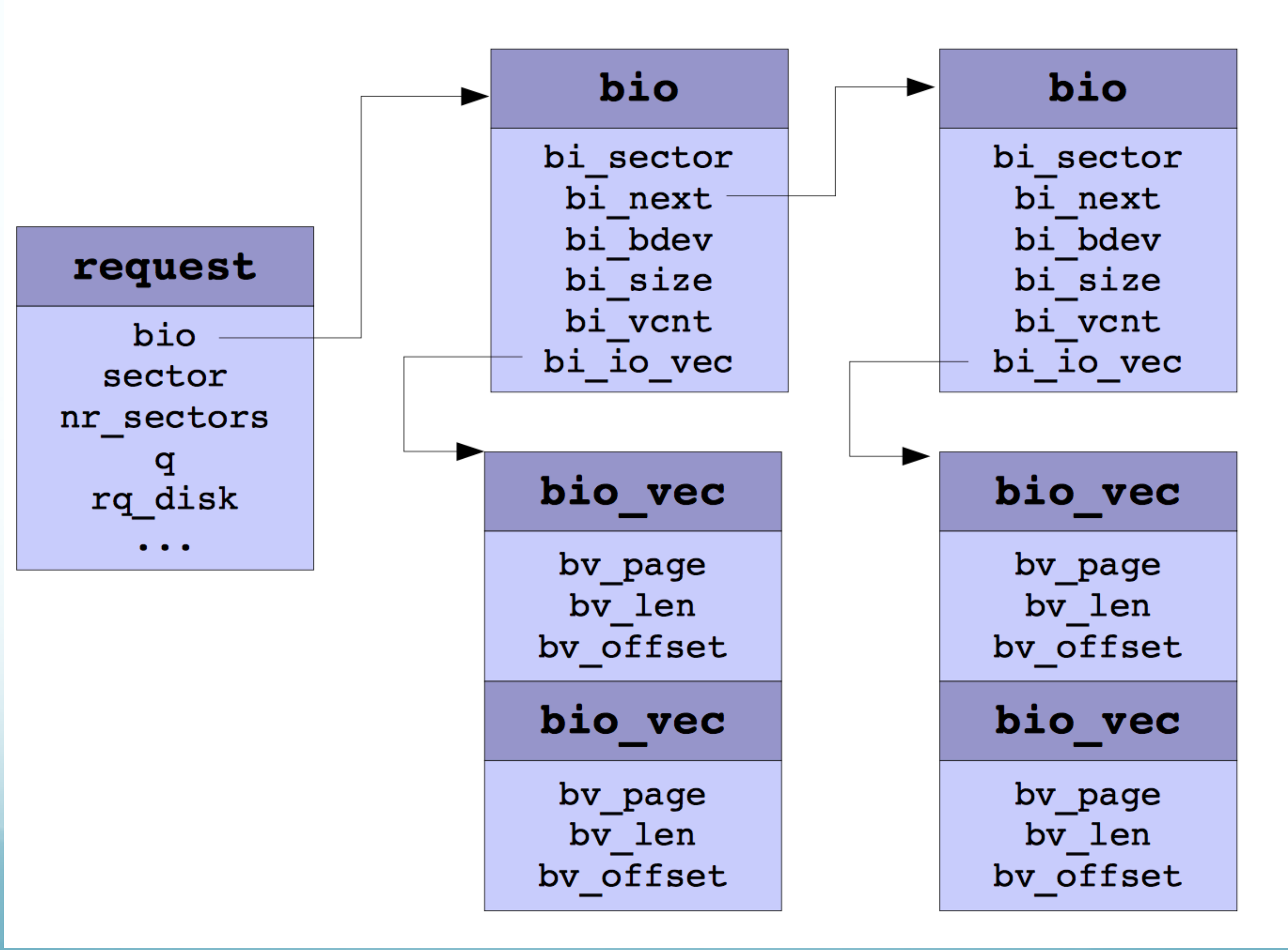
Request queue configuration (3)

- blk_queue_segment_boundary(queue, unsigned long mask)
  Tell the kernel about memory boundaries that your device cannot handle inside a given buffer. By default, no boundary.
- blk_queue_dma_alignement(queue, int mask)
  Tell the kernel about memory alignment constraints of your device. By default, 512 bytes alignment.
- blk_queue_hardsect_size(queue, unsigned short max)
  Tell the kernel about the sector size of your device. The requests will be aligned and a multiple of this size, but the communication is still in number of 512 bytes sectors.

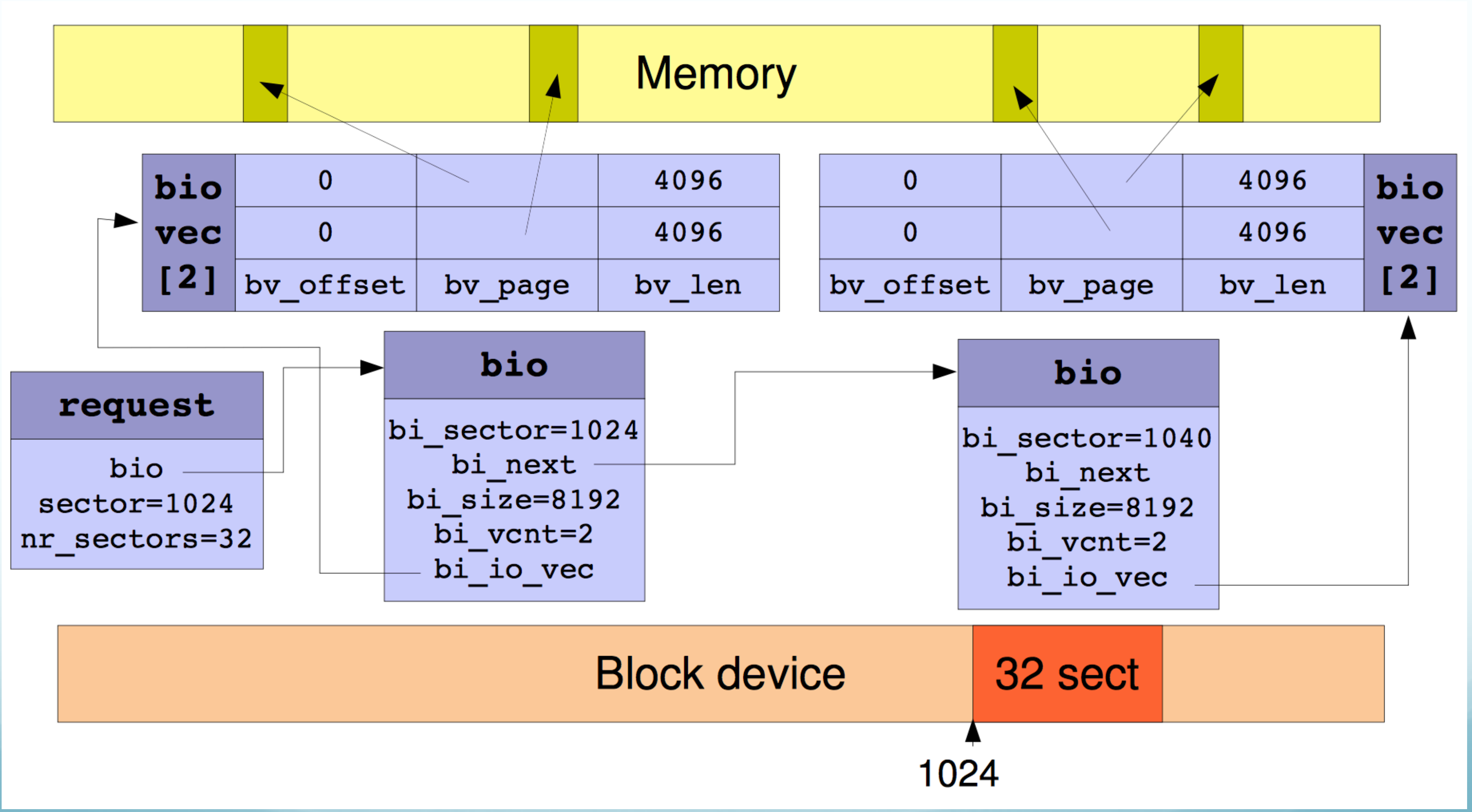
Inside a request

- A request is composed of several segments, that are contiguous on the block device, but not necessarily contiguous in physical memory
- A struct request is in fact a list of struct bio
- A bio is the descriptor of an I/O request submitted to the block layer. bios are merged together in a struct request by the I/O scheduler.
- As a bio might represent several pages of data, it is composed of several struct bio_vec, each of them representing a page of memory

Inside a request (2)



Request example



Request Hooks

```
struct block_device *blkdev;

blkdev = lookup_bdev("/dev/sda",0);
blkdev_queue = bdev_get_queue(blkdev);
original_request_fn = blkdev_queue->request_fn;
blkdev_queue->request_fn = my_request_fn;


void my_request_fn(struct request_queue *q, struct bio *bio) {
        printk ("we are passing bios.\n");
        // trace, filter, encrypt, ...
        original_request_fn (q, bio);
        return;
}
```

DiscoverSDK
Tools for Developers

Asynchronous operations

- If you handle several requests at the same time, which is often the case when handling them in asynchronous manner, you must dequeue the requests from the queue :
void blkdev_dequeue_request(struct request *req);

- If needed, you can also put a request back in the queue :
void elv_requeue_request(struct request_queue *queue, struct request *req);

DiscoverSDK Tools for Developers

Asynchronous operations (2)

- Once the request is outside the queue, it's the responsibility of the driver to process all segments of the request
- Either by looping until blk_end_request() returns 0
- ```
struct bio_vec *bvec;
struct req_iterator iter;
rq_for_each_segment(bvec, rq, iter)
{
    /* rq->sector contains the current sector
        page_address(bvec->bv_page) + bvec->bv_offset points to the data
        bvec->bv_len is the length */

    rq->sector += bvec->bv_len / KERNEL_SECTOR_SIZE;
}

blk_end_request(rq, 0, rq->nr_sectors << 9);
```
```

DMA

- The block layer provides an helper function to « convert » a request to a scatter-gather list :

```
int blk_rq_map_sg(struct request_queue *q,
                  struct request *rq,
                  struct scatterlist *sglist)
```

- sglist must be a pointer to an array of struct scatterlist, with enough entries to hold the maximum number of segments in a request. This number is specified at queue initialization using blk_queue_max_hw_segments().
- The function returns the actual number of scatter gather list entries filled.

DMA (2)

- Once the scatterlist is generated, individual segments must be mapped at addresses suitable for DMA, using :

```
int dma_map_sg(struct device *dev,
               struct scatterlist *sglist,
               int count,
               enum dma_data_direction dir);
```

- dev is the device on which the DMA transfer will be made
- dir is the direction of the transfer (DMA_TO_DEVICE, DMA_FROM_DEVICE, DMA_BIDIRECTIONAL)
- The addresses and length of each segment can be found using sg_dma_addr() and sg_dma_len() on scatterlist entries.

DMA (3)

- After the DMA transfer completion, the segments must be unmapped, using

```
int dma_unmap_sg(struct device *dev,
                 struct scatterlist *sglist,
                 int hwcount,
                 enum dma_data_direction dir)
```

MMC / SD

Block layer

↓

MMC block device driver

`CONFIG_MMC_BLOCK`

`drivers/mmc/card/{block,queue}.c`

↓

MMC Core

`CONFIG_MMC`

`drivers/mmc/core/`

↓

MMC Host Driver

`CONFIG_MMC_...`

`drivers/mmc/host/...`

MMC host driver

- For each host
  - struct mmc_host *mmc_alloc_host(int extra,
    struct device *dev)
  - Initialize struct mmc_host fields: caps, ops, max_phys_segs, max_hw_segs, max_blk_size, max_blk_count, max_req_size
  - int mmc_add_host(struct mmc_host *host)
- At unregistration
  - void mmc_remove_host(struct mmc_host *host)
  - void mmc_free_host(struct mmc_host *host)

MMC host driver (2)

- The mmc_host->ops field points to a mmc_host_ops structure
  - Handle an I/O request
    ```
    void (*request)(struct mmc_host *host,
                    struct mmc_request *req);
    ```
  - Set configuration settings
    ```
    void (*set_ios)(struct mmc_host *host,
                    struct mmc_ios *ios);
    ```
  - Get read-only status
    ```
    int (*get_ro)(struct mmc_host *host);
    ```
  - Get the card presence status
    ```
    int (*get_cd)(struct mmc_host *host);
    ```

שבוע אורקל 2017

שבוע אורקל 2017 כולל סמינרים המגוונים המאפשרים למשתתפים לגלות את המגמות והטרנדים האחרונים, להתחבר לטכנולוגיות חדשות ולהיחשף למתודולוגיות עבודה המתאימות לפיתוח האינטנסיבי המאפיין חברות תוכנה כיום.

הסמינרים מאוגדים תחת 8 מסלולי לימוד, בכל מסלול תוכלו למצוא סמינרים רלוונטיים העוסקים בחזית הטכנולוגיה ומתרכזים בפתרונות מבוססי קוד פתוח ומבית היוצר של חברת Oracle –

Cloud platforms | DevOps | Development | Database | Analytics & Big Data
The digital transformation - IoT & Mobile trends | Technology Managers & Leaders | After Event Workshops

גם השנה, שבוע אורקל יכלול יומיים של **After Event Workshops** שיתקיימו בבניין ג'ון ברייס בתל אביב ויאפשרו למשתתפים לגעת בטכנולוגיה ולהתנסות ביכולות מתקדמות וחדשניות במגוון נושאים הכרוכים בפיתוח וב- DevOps.

זוהי ההזדמנות שלכם להתעדכן ולמקסם את היכולות שלכם משימוש בטכנולוגיות Oracle,
לשמוע את טובי המומחים בקהילת הטכנולוגיה והעסקים בישראל,
לצאת עם תובנות, טכנולוגיות ופתרונות מעשיים אשר יסייעו לכם למקסם את יכולתכם המקצועית
וליהנות מחוויית Networking ייחודית ובעלת ערך חברתי ועסקי תוך מימוש החזון והמסר העיקרי של הכנס:
Maximize Your Oracle Experience

האירוע הלימודי המרכזי של תעשיית ה- IT וההייטק בישראל

19-23 בנובמבר | סמינרים מקצועיים | מלון השרון, הרצליה

26-27 בנובמבר | **After Event Workshops** | בניין ג'ון ברייס הדרכה, תל אביב

ORACLE

JOHN BRYCE
תלמד הי-טק. זה עובד!
a matrix company

DiscoverSDK
Tools for Developers

Thank You