The Linux Filesystem

Today’s lecture

Parts/figures of today’s lecture are adopted from Dan Porter’s excellent lectures available here : http://www.cs.unc.edu/~porter/courses/cse506/f12/syllabus.html besides the book! In addition some material from http://shop.oreilly.com/product/9780596005658.do which you can get via the library online

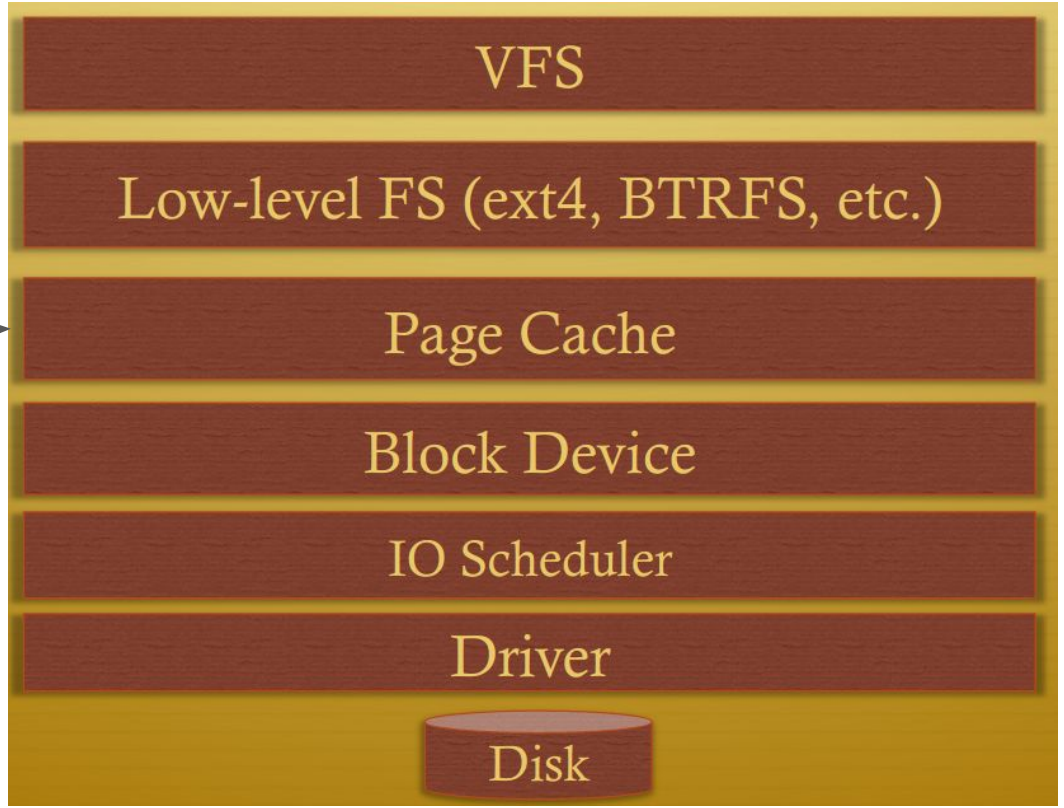
- The Linux Block I/O layer
- Linux File Systems
- Signals
- Pipes

Block devices

- Block devices are hardware devices distinguished by the random (that is, not necessarily sequential) access of fixed-size blocks of data.
  - Hard disks, blurays, flash memory, …
  - all devices on which you mount a filesystem
- Block devices are much harder to manage compared to character devices
  - Character devices have very few possible I/O states
  - Kenrel has no subsystem for managing character devices
- The main goals for the block device subsystem in linux are
  - Throughput
  - Latency
  - Safety against failures
  - fairness

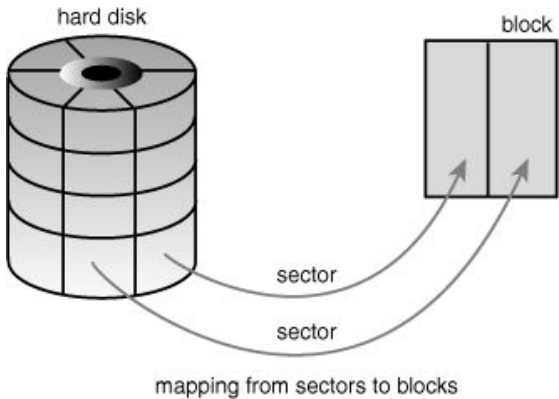
The big picture of today's lecture

Already covered



Anatomy of a block device

- Smallest addressable unit on a device is called a sector
  - Sector size varies between devices and architectures, but typically 512 Bytes or 4KBs
  - Also called “device blocks” or “hard sectors”
- Software however does not have the same HW limitations
  - imposes its own smallest logically addressable unit, which is the block
  - the kernel performs all disk operations in terms of blocks
  - the block size can be no smaller than the sector and must be a multiple of a sector.
  - kernel also requires that a block be no larger than the page size
  - Sometimes referred to as “filesystem blocks” or “I/O blocks



mapping from sectors to blocks

Buffers and buffer heads

- a block is stored in memory—say, after a read or pending a write—it is stored in a buffer.
  - The buffer serves as the object that represents a disk block in memory
  - a single page can hold one or more blocks in memory
  - the kernel requires some associated control information to accompany the data (such as from which block device and which specific block the buffer is)
  - each buffer is associated with a descriptor or a “buffer head”, defined in https://github.com/torvalds/linux/blob/master/include/linux/buffer_head.h
  - The purpose of a buffer head is to describe this mapping between the on-disk block and the physical in-memory buffer. Acting as a descriptor of this buffer-to-block mapping is the data structure’s only role in the kernel

Buffers

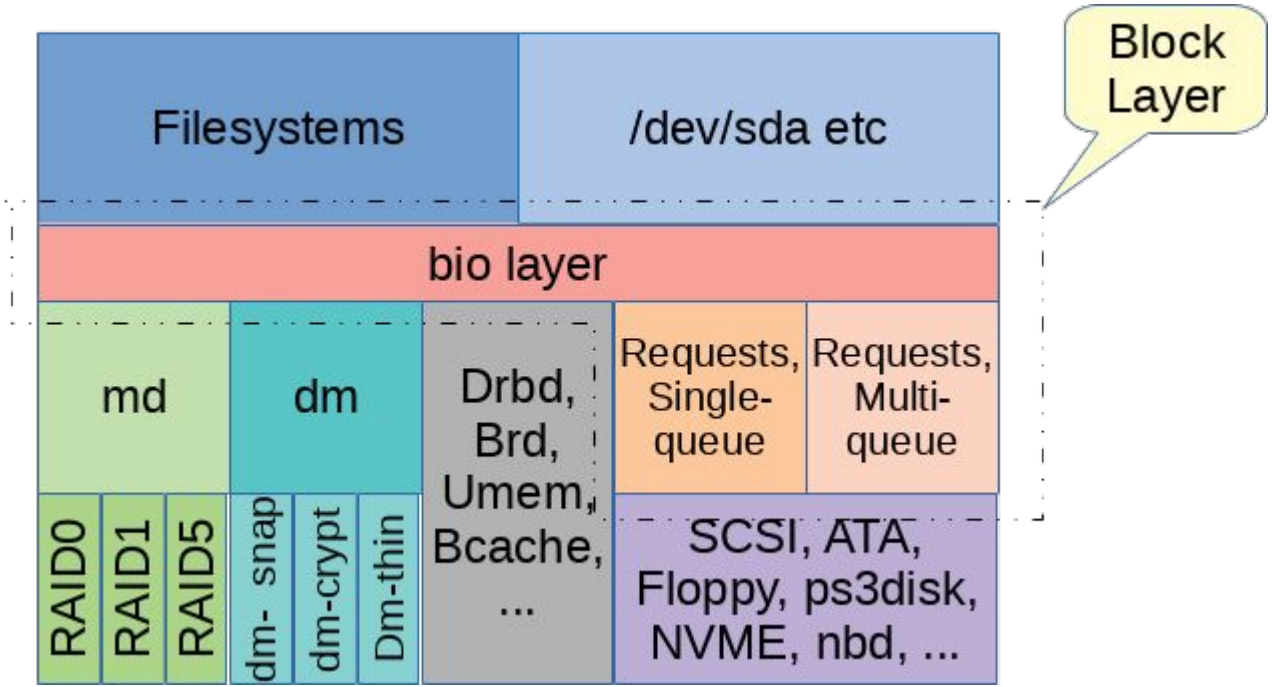
- a block –it is stored in a buff
  - The
  - a sir
  - the l (such as from whic
  - each
    https
- Was the

```c
/*
 * 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, and buffer_heads are used for extracting block
 * mappings (via a get_block_t call), for tracking state within
 * a page (via a page_mapping) and for wrapping bio submission
 * for backward compatibility reasons (e.g. submit_bh).
 */
struct buffer_head {
        unsigned long b_state;          /* buffer state bitmap (see above) */
        struct buffer_head *b_this_page;/* circular list of page's buffers */
        struct page *b_page;            /* the page this bh is mapped to */

        sector_t b_blocknr;             /* start block number */
        size_t b_size;                  /* size of mapping */
        char *b_data;                   /* pointer to data within the page */

        struct block_device *b_bdev;
        bh_end_io_t *b_end_io;          /* I/O completion */
        void *b_private;                /* reserved for b_end_io */
        struct list_head b_assoc_buffers; /* associated with another mapping *
        struct address_space *b_assoc_map;      /* mapping this buffer is
                                                   associated with */
        atomic_t b_count;               /* users using this buffer_head */
};
```

The bio layer

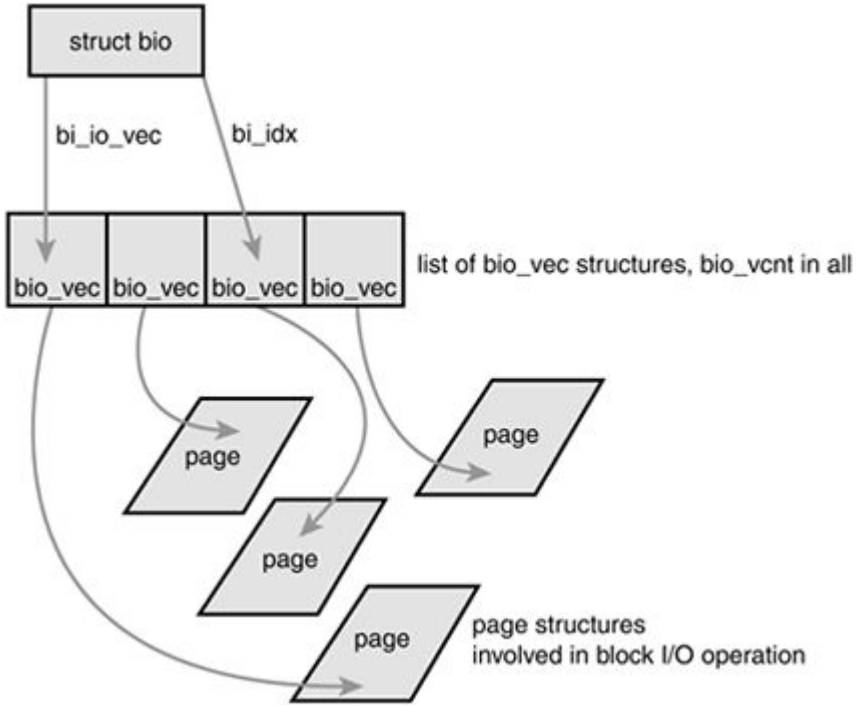


The bio layer

- All block devices in Linux are represented by struct gendisk — a "generic disk" defined in https://github.com/torvalds/linux/blob/master/include/linux/genhd.h
- carries read and write requests, and assorted other control requests, from the block_device, past the gendisk, and on to the driver.
- A bio identifies a target device, an offset in the linear address space of the device, a request (typically READ or WRITE), a size, and some memory where data will be copied to or from
- Each block I/O request is represented by a bio structure `struct bio` data structure

The bio structure

- The basic container for block I/O
- represents block I/O operations that are active as a list of segments.
  - A segment is a chunk of a buffer that is contiguous in memory
  - individual buffers need not be contiguous in memory
  - the bio structure provides the capability for the kernel to perform block I/O operations of even a single buffer from multiple locations in memory



The `bio struct`

- Defined in https://github.com/torvalds/linux/blob/master/include/linux/blk_types.h
  - Controlled by https://github.com/torvalds/linux/blob/master/include/linux/bio.h
- The one described in the book is obsolete

```
/*
 * main unit of I/O for the block layer and lower layers (ie drivers and
 * stacking drivers)
 */
struct bio {
        struct bio              *bi_next;       /* request queue link */
        struct gendisk          *bi_disk;
        unsigned int            bi_opf;         /* bottom bits req flags,
                                                 * top bits REQ_OP. Use
                                                 * accessors.
                                                 */
        unsigned short          bi_flags;       /* status, etc and bvec pool number */
        unsigned short          bi_ioprio;
        unsigned short          bi_write_hint;
        blk_status_t            bi_status;
        u8                      bi_partno;
        atomic_t                __bi_remaining;

        struct bvec_iter        bi_iter;

        bio_end_io_t            *bi_end_io;

        void                    *bi_private;
#ifdef CONFIG_BLK_CGROUP
        /*
         * Represents the association of the css and request_queue for the bio.
         * If a bio goes direct to device, it will not have a blkg as it will
         * not have a request_queue associated with it.  The reference is put
         * on release of the bio.
         */
        struct blkcg_gq         *bi_blkg;
        struct bio_issue        bi_issue;
#ifdef CONFIG_BLK_CGROUP_IOCOST
        u64                     bi_iocost_cost;
```

Important fields in the struct

- `Bi_io_vec:` an array of bio_vec structures. These structures are used as lists of individual segments in this specific block I/O operation
    - Each bio_vec is treated as a vector of the form <page, offset, len>,
        - describes a specific segment:
        - the physical page on which it lies
        - the location of the block as an offset into the page
        - the length of the block starting from the given offset

A deeper bio understanding

- Beyond the scope of this lecture
- Readings
  - https://lwn.net/Articles/736534/

Request Queues

- Block devices maintain request queues to store their pending block I/O requests
  - Each block device request is represented by a request descriptor
- The request queues are represented by the `request_queue` structure
  - Defined in https://github.com/torvalds/linux/blob/master/include/linux/blkdev.h
- A request is represented by a `request` structure
  - Again defined in https://github.com/torvalds/linux/blob/master/include/linux/blkdev.h
- The request queue contains a doubly linked list of requests and associated control information
  - As long as the request queue is nonempty, the block device driver associated with the queue grabs the request from the head of the queue and submits it to its associated block device

Single versus multiple request queues (1)

- Traditionally, most storage devices were made up of a set of spinning circular platters with magnetic coating and a single head
  - Such a device can only process a single request at a time, and has a substantial cost in moving from one location on the platters to another.
- The three key tasks for a single-queue scheduler are:
  - collect multiple bios representing contiguous operations into a smaller number of requests that are large enough to make best use of the hardware but not so large that they exceed any limitations of the device.
  - To queue these requests in an order that minimizes seek time while not delaying important requests unduly. Relying on heuristics, aim to provide an optimal solution to this problem is the source of all the complexity.
  - To make these requests available to the underlying driver so it can pluck them off the queue when it is ready and to provide a mechanism for notification when those requests are complete.

Single versus multiple request queues (2)

- Many reasons for having multiple request queues today
  - many devices can accept multiple I/O requests at once, e.g., many of today's SSDs
  - As we get more and more processing cores in our systems, the locking overhead required to place requests from all cores into a single queue increases.
- Linux introduced software staging queues
  - struct blk_mq_ctx - State for a software queue facing the submitting CPUs
    - Defined in https://github.com/torvalds/linux/blob/master/block/blk-mq.h
  - Requests are added to these queues, controlled by a spinlock that should mostly be uncontended
  - Hardware dispatch queues are allocated based on the target hardware
    - there may be just one, or there may be as many as 2048
    - These queues are the responsibility of the driver
- More info here: https://lwn.net/Articles/738449/

I/O Schedulers

- Between page cache and disk, you have a queue of pending requests
- Requests are a tuple of (block #, read/write, buffer addr)
- These write requests can be reordered based on some heuristic
  - Also called IO Scheduling
- Rationale
  - I/O is very slow compared to most other system components (seek time, rotational time, etc)
  - Even on SSD’s you need a multi-queue I/O scheduler
- The I/O scheduler divides the resource of disk I/O among the pending block I/O requests in the system.
  - merging and sorting of pending requests in the request queue
  - Requires some understanding of the disk model
  - Unfair to some requests at the expense of improving the overall performance of the system

Multiple I/O schedulers available

- Many of the I/O schedulers are now obsolete!
  - We describe some old and new
- They operate on the I/O request queues
- The elevator algorithm was developed by Linus for kernel v2.4
  - When a request is added to the queue, it is first checked against every other pending request to see whether it is a possible candidate for merging
  - If a suitable location sector-wise is in the queue, the new request is inserted there. This keeps the queue sorted by physical location on disk.
  - Finally, if no such suitable insertion point exists, the request is inserted at the tail of the queue.

  Code available in
  https://github.com/torvalds/linux/blob/master/block/elevator.c

-

The (obsolete) NOOP scheduler

- The simplest available scheduler is the NOOP scheduler
  - provides minimal sorting of requests, never allowing a read to be moved ahead of a write or vice-versa,
  - allowing one request to overtake another if, in line with the elevator algorithm
  - the I/O head is likely to reach the new one before the old one.
  - Apart from this simple sorting, "noop" provides first-in-first-out queuing.

Code here: https://github.com/spotify/linux/blob/master/block/noop-iosched.c

The Deadline I/O Scheduler

- "deadline" collects batches of either read or write requests that were all submitted at close to the same time.
  - Maintains a request queue sorted by physical location on disk that is used for scheduling as long as time did not expire, once filled, this queue is used to schedule operations
  - In addition, Read requests are sorted into a special read FIFO queue, and write requests are inserted into a special write FIFO queue.
    - If the request at the head of either the write FIFO queue or the read FIFO queue expires (default 500 ms), the scheduler then begins servicing requests from the FIFO queue
  - Fixes possible starvation in the elevator and NOOP algorithms
  - Now exists for multi-queue scheduling

Code here: https://github.com/torvalds/linux/blob/master/block/mq-deadline.c

The BFQ I/O Scheduler

- Replaced a scheduler called cfq scheduler described in the book
  - Lots of very good material here: https://algo.ing.unimo.it/people/paolo/disk_sched/
- The Budget Fair Queueing (BFQ) I/O Scheduler is a proportional-share (measured in number of sectors) storage-I/O schedule where each process/thread to be assigned a fraction of the I/O throughput
  - explicitly privileges the I/O of two classes of time-sensitive applications: interactive and soft real-time.
    - BFQ constantly tries to detect whether the I/O requests in a bfq_queue come from an interactive or a soft real-time application.

Code here: https://github.com/torvalds/linux/blob/master/block/bfq-iosched.c

The Kyber I/O Scheduler

- Intended for fast multiqueue devices
- I/O requests passing through Kyber are split into two primary queues, one for synchronous requests (reads) and one for asynchronous requests (writes)
- the number of operations (both reads and writes) sent to the dispatch queues (the queues that feed operations directly to the device) is strictly limited

Very clean code:
https://github.com/torvalds/linux/blob/master/block/kyber-iosched.c

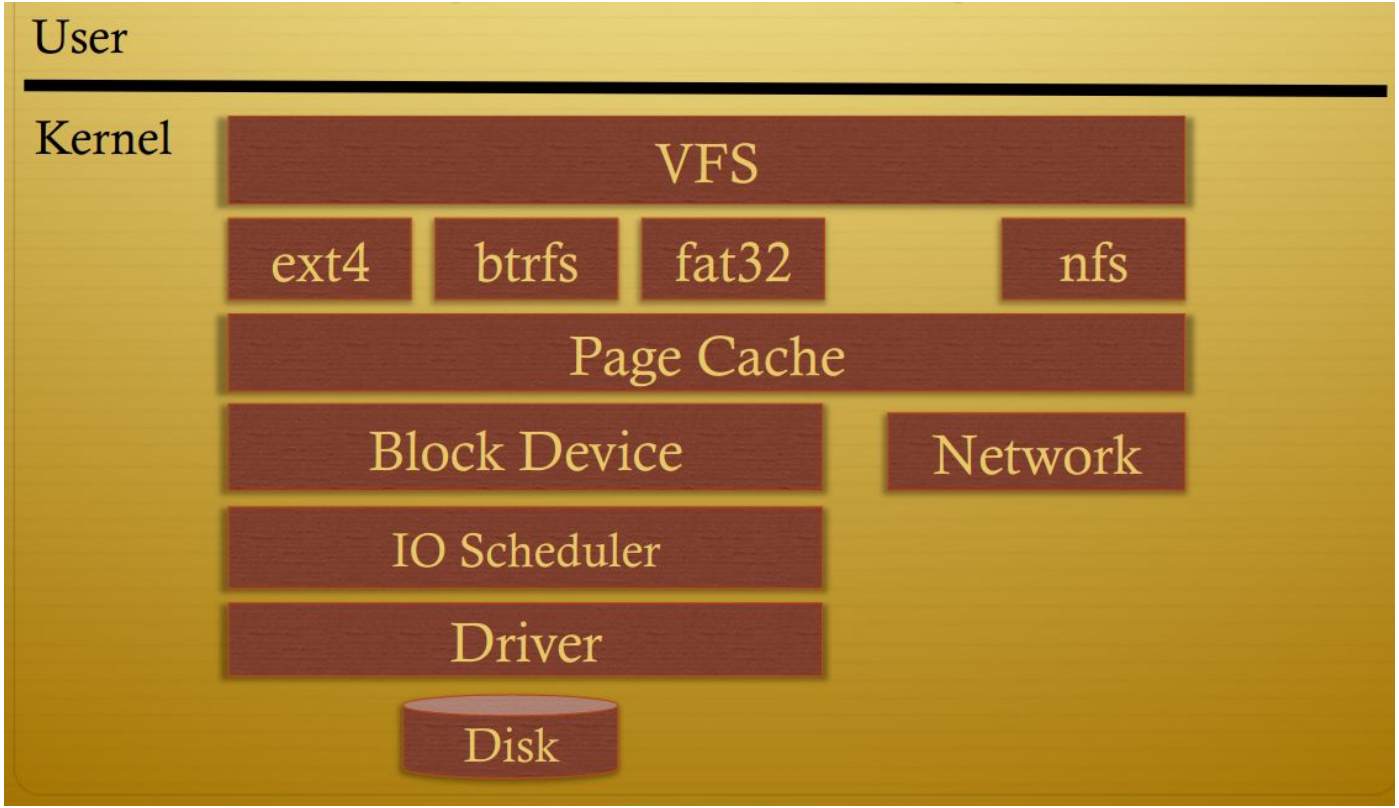
Which I/O Scheduler to use?

- Benchmark your applications
- One scheduler can be better than the other., but for most applications, the performance differences can be minute

  See for example: https://www.phoronix.com/scan.php?page=article&item=linux-50hdd-io&num=2

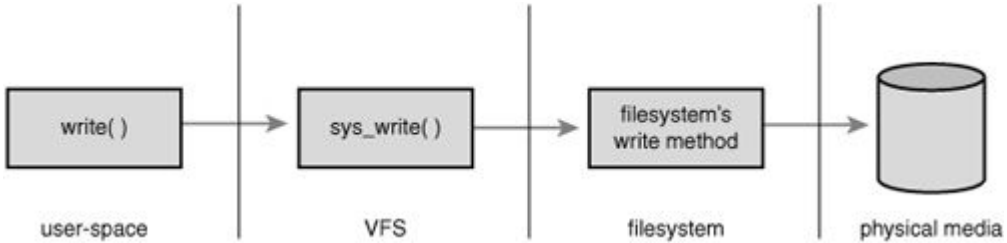
The Linux (virtual) Filesystem

Supports multiple Filesystems



What the VFS does

- The VFS is a substantial piece of code, not just an API wrapper
  - Caches file system metadata (e.g., file names, attributes)
  - Coordinates data caching with the page cache
  - Enforces a common access control model
  - Implements complex, common routines, such as path lookup, file opening, and file handle management



Why have different filesystems?

- A filesystem is a hierarchical storage of data adhering to a specific structure
- In essence, how to represent and manipulate data, files, and folders in the computer
  - Optimizations for each filesystem depends on the goal
  - Differences in their capabilities
  - Some are for more specialized HW
  - A good list of filesystems: https://en.wikipedia.org/wiki/List_of_file_systems
  - A good comparison of filesystems: https://en.wikipedia.org/wiki/Comparison_of_file_systems
- Filesystems are mounted at a specific mount point in a global hierarchy known as a namespace

Primary object types of the VFS

- The superblock object, which represents a specific mounted filesystem.
- The inode object, which represents a specific file.
- The dentry object, which represents a directory entry, which is a single component of a path.
- The file object, which represents an open file as associated with a process.

Lots of similarities with Minix and with your undergrad class!

Special filesystems

- Enables an administrator to manipulate some of the kernel data structures nd to implement special features of the operating system. For example:
  - The bdev filesystem for managing block device drivers
  - The pipefs special filesystem (for Pipes, more later)
  - The proc filesystem mounted on /proc, which acts as a General access point to kernel data structures

Example special FS: The pipefs

- Pipes are an interprocess communication mechanism
- A pipe is a one-way flow of data between processes: all data written by a process to the pipe is routed by the kernel to another process
  - Pipe: a pair of file descriptors, one for read, one for write
  - Parent create a pipe, fork, then communicate with child
- Linux support for pipe built on pipefs
  - A special file system type for pipe (and named pipe)– Kernel mounted, but no mount point
  - Implementation in: https://github.com/torvalds/linux/blob/master/fs/pipe.c
- A pipe is implemented as a set of VFS objects, which have no corresponding disk images

Pipes

- Pipes may be considered open files that have no corresponding image in the mounted filesystems
  - A process creates a new pipe by means of the pipe( ) system call, which returns a pair of file descriptors
  - the process may then pass these descriptors to its descendants through fork( ) thus sharing the pipe with them.
  - The processes can read from the pipe by using the read( ) system call with the first file descriptor;
  - likewise, they can write into the pipe by using the write( ) system call with the second file descriptor.

Anatomy of a pipe: ls | more

- When the command shell interprets the ls|more statement, it essentially performs the following actions:
  a. Invokes the pipe( ) system call; let's assume that pipe( ) returns the file descriptors 3 (the pipe's read channel) and 4 (the write channel).
  b. Invokes the fork( ) system call twice.
  c. Invokes the close( ) system call twice to release file descriptors 3 and 4.
- The first child process, which must execute the ls program, performs the following operations:
  a. Invokes dup2(4,1) to copy file descriptor 4 to file descriptor 1. From now on, file descriptor 1 refers to the pipe's write channel.
  b. Invokes the close( ) system call twice to release file descriptors 3 and 4.
  c. Invokes the execve( ) system call to execute the ls program
  d. The program writes its output to the file that has file descriptor 1 (the standard output); i.e., it writes into the pipe.

Anatomy of a pipe: ls | more

- The second child process must execute the more program, therefore, it performs the following operations:
  - Invokes dup2(3,0) to copy file descriptor 3 to file descriptor 0. From now on, file descriptor 0 refers to the pipe's read channel.
  - Invokes the close( ) system call twice to release file descriptors 3 and 4.
  - Invokes the execve( ) system call to execute more. By default, that program reads its input from the file that has file descriptor 0 (the standard input); i.e., it reads from the pipe.

Example Linux FS: Ext3

- Some features (some common with Ext2)
  - Support for immutable files (they cannot be modified, deleted, or renamed) and for append-only files
  - The filesystem preallocates disk data blocks to regular files before they are actually used. Thus, when the file increases in size, several blocks are already reserved at physically adjacent positions, reducing file fragmentation.
  - The filesystem partitions disk blocks into groups. Each group includes data blocks and inodes stored in adjacent tracks. Thanks to this structure, files stored in a single block group can be accessed with a lower average disk seek time.
  - Journaling avoids the time-consuming check that is automatically performed on a filesystem when it is abruptly unmounted — for instance, as a consequence of a system crash.
    - In fact, most of the differences between Ext3 and Ext2 is in failure handling!

Signals

The Role of Signals

- A signal is a very short message that may be sent to a process or a group of processes.
  - The only information given to the process is usually a number identifying the signal;
  - there is no room in standard signals for arguments, a message, or other accompanying information.
  - A set of macros whose names start with the prefix SIG is used to identify signals
- Signals serve two main purposes:
  - To make a process aware that a specific event has occurred
  - To force a process to execute a signal handler function included in its code
- Process can change how it handle certain signal
  - Use a different signal-handler function (user-provided)
  - Use kernel default (set signal handler to 0x0)–
  - Ignore the signal (set signal handler to 0x1)
  - Block the signal (will be pending until unblocked later)

#	Signal name	Default action	Comment
1	SIGHUP	Terminate	Hang up controlling terminal or process
2	SIGINT	Terminate	Interrupt from keyboard
3	SIGQUIT	Dump	Quit from keyboard
4	SIGILL	Dump	Illegal instruction
5	SIGTRAP	Dump	Breakpoint for debugging
6	SIGABRT	Dump	Abnormal termination
6	SIGIOT	Dump	Equivalent to SIGABRT
7	SIGBUS	Dump	Bus error
8	SIGFPE	Dump	Floating-point exception
24	SIGXCPU	Dump	CPU time limit exceeded
25	SIGXFSZ	Dump	File size limit exceeded
26	SIGVTALRM	Terminate	Virtual timer clock
27	SIGPROF	Terminate	Profile timer clock
28	SIGWINCH	Ignore	Window resizing
29	SIGIO	Terminate	I/O now possible
29	SIGPOLL	Terminate	Equivalent to SIGIO
30	SIGPWR	Terminate	Power supply failure
31	SIGSYS	Dump	Bad system call

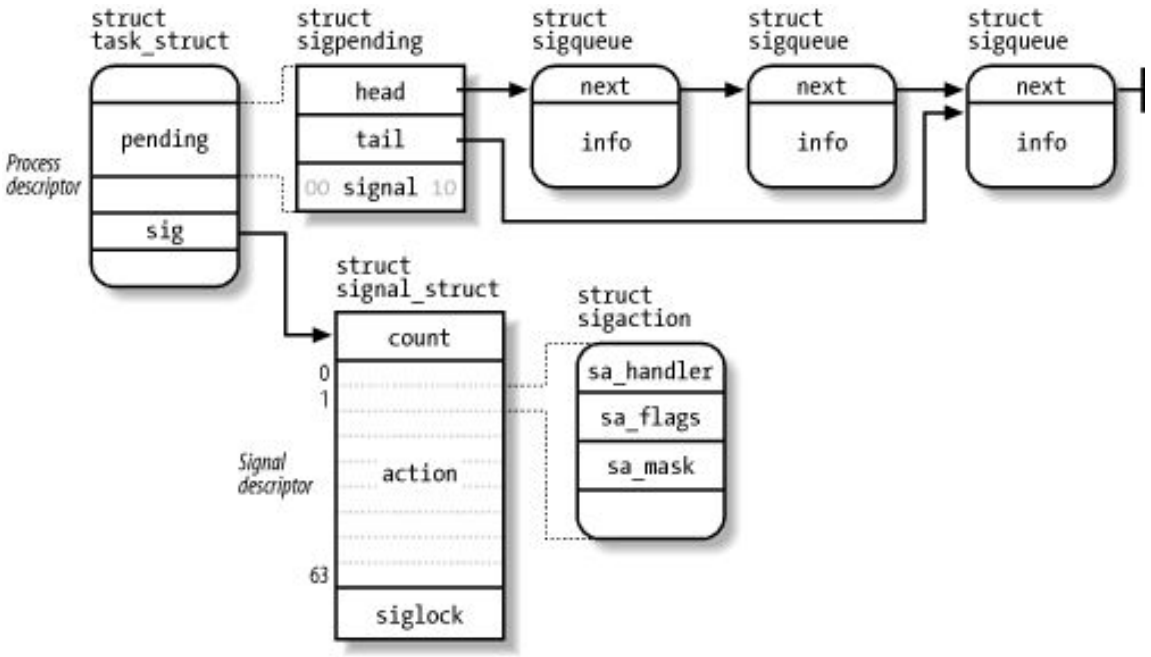
Some signal features

- Signals may be sent at any time to a process whose state is usually unpredictable.
  - Signals sent to a process that is not currently executing must be saved by the kernel until that process resumes execution.
  - kernel distinguishes two different phases related to signal transmission:
    - Signal generation: The kernel updates a data structure of the destination process to represent that a new signal has been sent.
    - Signal delivery: The kernel forces the destination process to react to the signal by changing its execution state, by starting the execution of a specified signal handler, or both.
- Signals are consumable resources: once they have been delivered, all process descriptor information that refers to their previous existence is canceled.

kernel must

- Remember which signals are blocked by each process.
- When switching from Kernel Mode to User Mode, check whether a signal for any process has arrived. This happens at almost every timer interrupt (roughly every 10 ms).
- Determine whether the signal can be ignored.
- Handle the signal, which may require switching the process to a handler function at any point during its execution and restoring the original execution context after the function returns.

Kernel data-structures for signals



Definitions in: https://github.com/torvalds/linux/blob/master/include/linux/signal.h and https://github.com/torvalds/linux/blob/master/include/linux/signal_types.h