The Virtual Filesystem

File Systems

- old days – "the" filesystem!
- now – many filesystem types, many instances
  - need to copy file from NTFS to Ext3
- original motivation – NFS support (Sun)
- idea – filesystem op abstraction layer (VFS)
  - Virtual File System (aka Virtual Filesystem Switch)
  - file-related ops determine filesystem type
  - dispatch (via function pointers) filesystem-specific op

File System Types

- lots and lots of filesystem types!
  - 2.6 has nearly 100 in the standard kernel tree
- examples
  - standard: ufs (Solaris), svfs (SysV), ffs (Berkeley)
  - network: RFS, NFS, Andrew, Coda, Samba, Novell
  - journaling: Ext3, Veritas, ReiserFS, XFS, JFS
  - media-specific: jffs, ISO9660 (cd), UDF (dvd)
  - special: /proc, tmpfs, sockfs, etc.
- proprietary
  - MSDOS, VFAT, NTFS, Mac, Amiga, etc.
- new generation for Linux
  - Ext3, ReiserFS, XFS, JFS

Common File Model

- standard api (basically UNIX file semantics)
  - doesn't fit perfectly with NT, etc.
  - example: directory is a file with specific structure
    - not true for some filesystems (MSDOS, etc.)
    - File Allocation Table (FAT)
- VFS layer just dispatches to fs-specific functions
  - libc read() -> sys_read()
    - what type of filesystem does this file belong to?
    - call filesystem (fs) specific read function
    - maintained in open file object (file)
  - example: file->f_op->read(…)
- similar to device abstraction model in UNIX

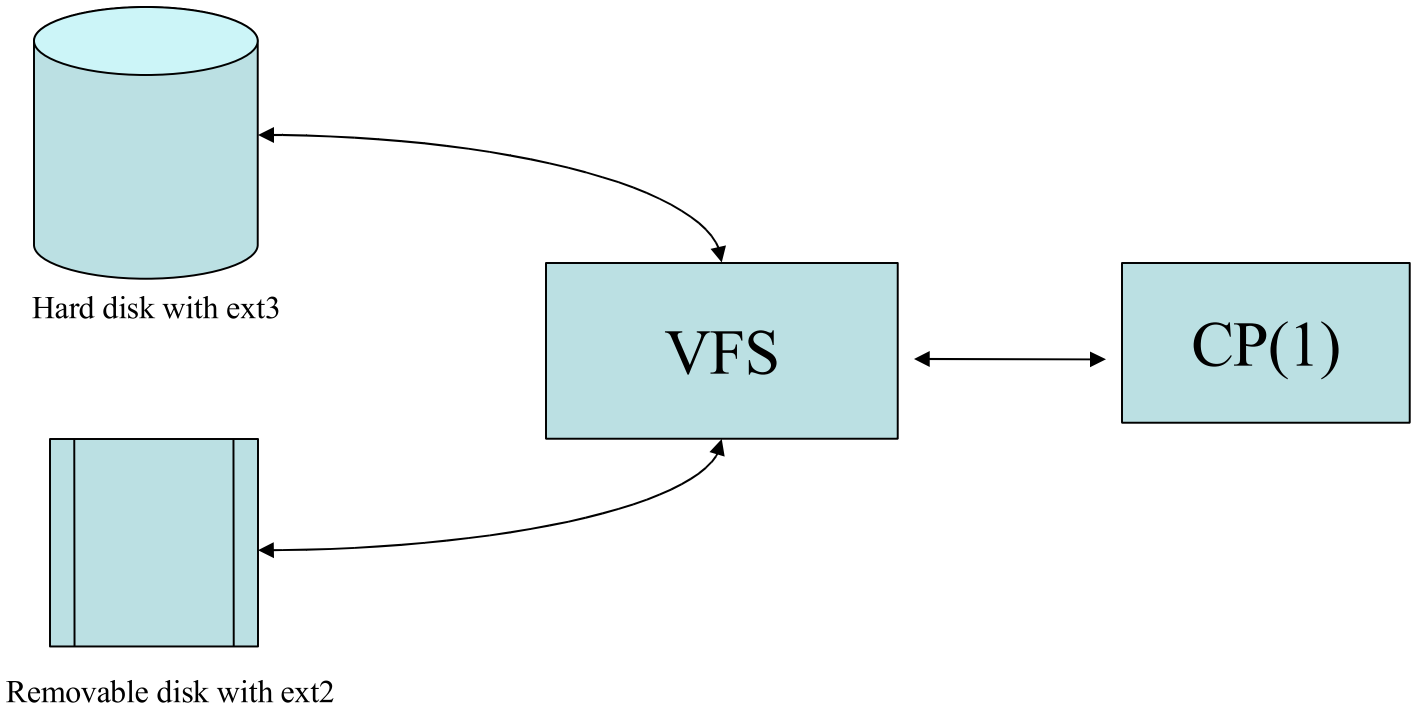
VFS System Calls

- fundamental UNIX abstractions
  - files (everything is a file)
    - ex: /dev/ttyS0 – device as a file
    - ex: /proc/123 – process as a file
  - processes
  - users
- lots of syscalls related to files (~100)
  - most dispatch to filesystem-specific calls
  - some require no filesystem action
    - example: lseek(pos) – change position in file
  - others have default VFS implementations

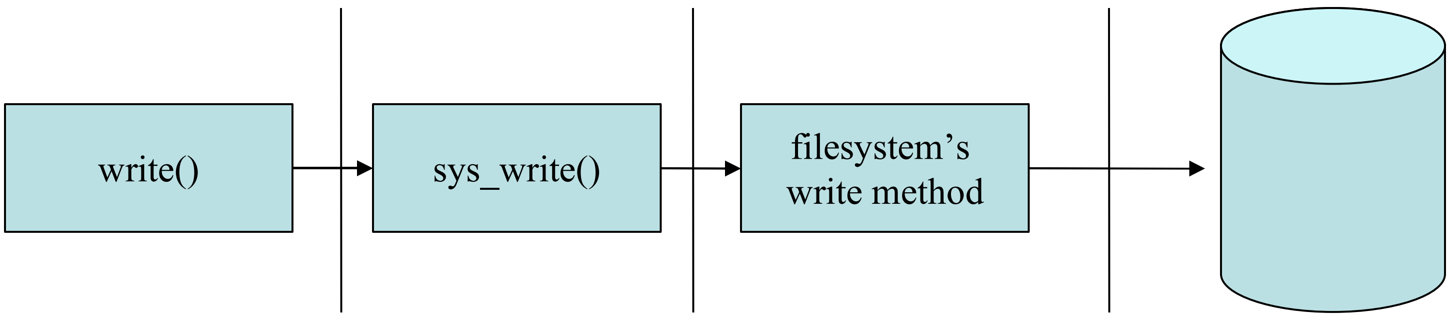
VFS System Calls (cont.)

- API :
  - filesystem ops – mounting, info, flushing, chroot, pivot_root
  - directory ops – chdir, getcwd, link, unlink, rename, symlink
  - file ops – open/close, read/write, stat, permissions, seek
    - chmod, chown, stat, creat, umask, dup, fcntl, truncate
    - read/write, readv/writev, pread/pwrite
  - memory mapping files – mmap, munmap, madvise, mlock
  - wait for input – poll, select
  - flushing – synch, fsync, msync, fdatasync
  - file locking – flock

Common Filesystem Interface



Unix Filesystem



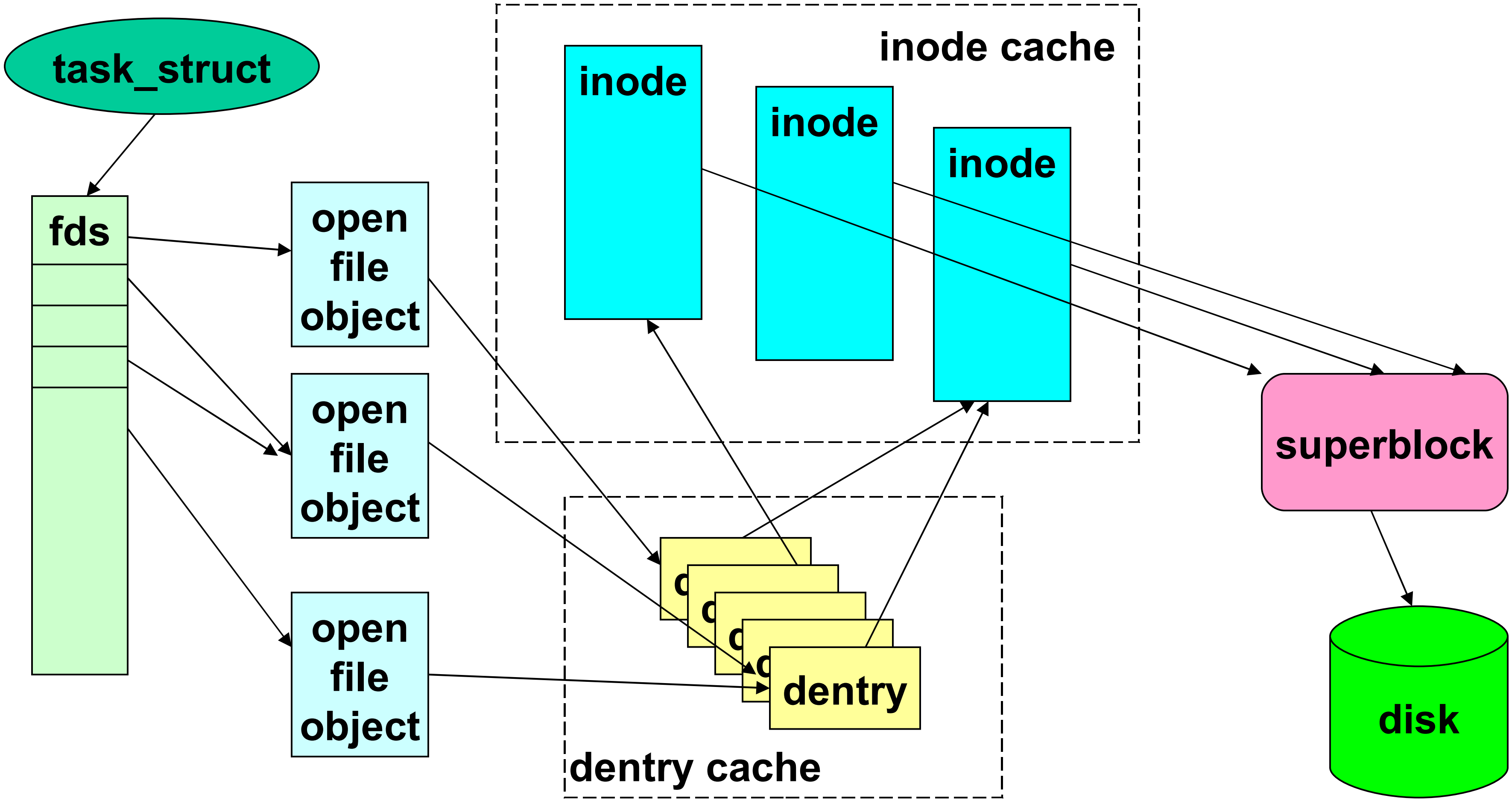
Big Four Data Structures

- **one** - open file object
  - information about an open file
  - includes current position (file pointer)
- **two** - dentry
  - information about a directory entry
  - includes name + inode#
- **three** - inode
  - unique descriptor of a file or directory
  - contains permissions, timestamps, block map (data)
  - inode#: integer (unique per mounted filesystem)
- **four** - superblock
  - descriptor of a mounted filesystem
- **ok, one more** - filesystem type
  - pointer to implementing module
  - including how to read a superblock

VFS Objects (Metadata Types)

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

Data Structure Relationships



Sharing Data Structures

- calling dup()
  - shares open file objects
  - example: 2>&1
- opening the same file twice
  - shares dentries
- opening same file via different hard links
  - shares inodes
- mounting same filesystem on different dirs
  - shares superblocks

VFS Objects

- The *super_operations* object
- The *inode_operations* object
- The *dentry_operations* object
- The *file_operations* object
- Others
  - *file_struct*
  - *fs_struct*
  - *namespace*

Superblock

- mounted filesystem descriptor
  - usually first block on disk (after boot block)
  - copied into (similar) memory structure on mount
    - distinction: disk superblock vs memory superblock
    - dirty bit (s_dirt), copied to disk frequently
- important fields
  - s_dev, s_bdev – device, device-driver
  - s_blocksize, s_maxbytes, s_type
  - s_flags, s_magic, s_count, s_root, s_dquot
  - s_dirty – dirty inodes for this filesystem
  - s_op – superblock operations
  - u – filesystem specific data

Superblock Operations

- filesystem-specific operations
  - read/write/clear/delete inode
  - write_super, put_super (release)
    - no get_super()
      - It is in file_system_type descriptor
  - write_super, lockfs, unlockfs, statfs
  - file_handle ops (NFS-related)
  - show_options

The Superblock Object

```
struct super_block {
      …
}
```

- `linux/fs.h`
- Created via `alloc_super()`
- Filled from the disk when mounted

Superblock Operations

- Writing to its superblock:
  `sb->s_op->write_super(sb)`
- Creating a new inode under the given superblock:
  `sb->s_op->alloc_inode(sb)`
- Deallocating the given inode:
  `sb->s_op->destroy_inode(inode)`
- Reading the inode from the disk:
  `sb->s_op->read_inode(inode)`
- Writing the inode to the disk
  `sb->s_op->write_inode(inode)`
- Others manipulating inodes

Inode

- "index" node – unique file or directory descriptor
  - meta-data: permissions, owner, timestamps, size, link count
  - data: pointers to disk blocks containing actual data
    - data pointers are "indices" into file contents (hence "inode")
- inode # - unique integer (per-mounted filesystem)
- what about names and paths?
  - high-level fluff on top of a "flat-filesystem"
  - implemented by directory files (directories)
  - directory contents: name + inode

File Links

- UNIX link semantics
  - hard links – multiple dir entries with same inode #
    - equal status; first is not "real" entry
    - file deleted when link count goes to 0
    - restrictions
      - can't hard link to directories (avoids cycles)
      - or across filesystems
  - soft (symbolic) links – little files with pathnames
    - just aliases for another pathname
    - no restrictions, cycles possible, dangling links possible

Inode Fields

- large struct (~50 fields)
- linux/fs.h
- important fields
  - i_sb, i_ino (number), i_nlink (link count)
  - metadata: i_mode, i_uid, i_gid, i_size, i_times
  - i_flock (lock list), i_wait (waitq – for blocking ops)
  - linkage: i_hash, i_list, i_dentry (aliases)
  - i_op (inode ops), i_fop (default file ops)
  - u (filesystem specific data – includes block map)

Inode Operations

- create – new inode for regular file
- link/unlink/rename –
  – add/remove/modify dir entry
- symlink, readlink, follow_link – soft link ops
- mkdir/rmdir – new inode for directory file
- mknod – new inode for device file
- truncate – modify file size
- permission – check access permissions

The Inode Object

```
struct inode {
  struct hlist_node   i_hash; // hash list
  struct list_head    i_list; // linked list
  struct list_head    i_dentry; // dentry list
  unsigned long       i_ino;
  atomic_t            i_count;
  umode_t             i_mode;
  i_uid, i_gid, i_size;
  struct inode_operations *i_op;
  struct file_operations *i_fop;
  struct super_block   *i_sb;
  kdev_t                i_rdev; // real device node
  struct block_device  *i_bdev; // bdev driver
  struct address_space *i_mapping, *i_data;
  …
}
```

Inode Operations

```
create(struct inode *, struct dentry *, int mode)
lookup(struct inode *, struct dentry *)
link(old_dentry, dir, dentry)
unlink(dir, dentry)
mkdir(dir, dentry, mode)
rmdir(dir, dentry) // remove dentry from dir
mnod() // device file, named pipe, socket, etc
rename()
readlink(dentry, buffer, buflen) // man readlink
follow_link() // translating a symbolic
link to the inode it points to
truncate(struct inode *inode) // modify file size
…
```

Dentry

- abstraction of directory entry
  - ex: line from ls -l
  - either files (hard links) or soft links or subdirectories
  - every dentry has a parent dentry (except root)
  - sibling dentries – other entries in the same directory
- directory api: dentry iterators
  - posix: opendir(), readdir(), scandir(), seekdir(), rewinddir()
  - syscall: getdents()
- why an abstraction?
  - UNIX: directories are really files with directory "records"
  - MSDOS, etc.: directory is just a big table on disk (FAT)
    - no such thing as subdirectories!
    - just fields in table (file->parentdir), (dir->parentdir)

Dentry (cont.)

- not-disk based (no dirty bit)
  - dentry_cache – slab cache
  - consistency maintenance using version numbers (later)
- important fields
  - d_name, d_count, d_flags
  - d_inode – associated inode
  - d_parent – parent dentry
  - d_child – siblings list
  - d_subdirs – my children (if i'm a subdirectory)
  - d_alias – other names (links) for the same object (inode)?
  - d_lru – unused state linkage
  - d_op – dentry operations (function pointer table)
  - d_fsdata – filesystem-specific data

Dentry Cache

- very important cache for filesystem performance
  - every file access causes multiple dentry accesses!
  - example: /tmp/foo
    - dentries for "/", "/tmp", "/tmp/foo" (path components)
- dentry cache "controls" inode cache
  - inodes released only when dentry is released
- dentry cache accessed via hash table
  - hash(dir, filename) -> dentry

Dentry Cache (cont.)

- dentry states
  - free (not valid; maintained by slab cache)
  - in-use (associated with valid open inode)
  - unused (valid but not being used; LRU list)
  - negative (invalid inode)
    - example: bad symbolic link (link exists but not file/inode)
- dentry ops
  - just a few, mostly default actions
  - ex: d_compare(dir, name1, name2)
    - case-insensitive for MSDOS

The Dentry Object

```
struct dentry {
  atomic_t   d_count; // usage count
  struct inode *d_inode
  struct dentry_operations *d_op;
  struct super_block *d_sb;
  void *d_fsdata; // filesystem-specific data
  struct qstr d_name; //dentry name
  unsigned char d_iname[]; // short filenames
  struct list_head d_lru; // unused list
  struct hlist_node d_hash; // hash list
  struct hlist_head *d_bucket;  // hash bucket
}
```

Dentry State:

- Used, d_inode points to an inode
- Unused, d_inode, d_count = 0
- Negative, d_inode = NULL

The Dentry Cache (*dcache*)

The dentry cache consists of three parts:

- Lists of “used” dentries that are linked off their associated inode via the i_dentry field of the inode object.
- A doubly linked “least recently used” list of unused and negative dentry objects.
- A hash table and hashing function used to quickly resolve a given path into the associated dentry object.

Dentry Operations

`d_revalidate(dentry, flags)`

`d_hash(dentry, name)`: creates a hash value from the given dentry. Called when to add a dentry to the hash table

`d_compare(dentry, name1, name2)`

`d_delete()`

`de_release()`

`de_iput()`

Icache

- The dentry cache also acts as a controller for the *inode cache*
- Inodes in kernel memory associated with unused dentries are not discarded since i_count is not null
- Thus inode objects are kept in RAM and can be referenced by corresponding dentries.

(Open) File Object

- struct file (usual variable name - filp)
  - association between file and process
  - no disk representation
  - created for each open (multiple possible, even same file)
  - most important info: file pointer
- file descriptor (small ints)
  - index into array of pointers to open file objects
- file object states
  - unused (memory cache + root reserve (10))
    - get_empty_filp()
  - inuse (per-superblock lists)
- system-wide max on open file objects (~8K)
  - /proc/sys/fs/file-max

File Object Fields

- important fields
  - f_dentry (associated dentry)
  - f_vfsmnt (fs mount point)
  - f_op (fs-specific functions – table of function pointers)
  - f_count, f_flags, f_mode (r/w, permissions, etc.)
  - f_pos (current position – file pointer)
  - info for read-ahead (more later)
  - f_uid, f_gid, f_owner
  - f_version (for consistency maintenance)
  - private_data (fs-specific data)

File Object Operations

- f_op field – table of function pointers
  - copied from inode (i_fop) initially (fs-specific)
  - possible to change to customize (per-open)
    - device-drivers do some tricks like this sometimes
- important operations
  - llseek(), read(), write(), readdir(), poll()
  - ioctl() – "wildcard" function for per-fs semantics
  - mmap(), open(), flush(), release(), fsync()
  - fasync() – turn on/off asynchronous i/o notifications
  - lock() – file-locks (more later)
  - readv(), writev() – "scatter/gather i/o"
    - read/write with discontiguous buffers (e.g. packets)
  - sendpage() – page-optimized socket transfer

The File Object

```
struct file {
  struct list_head f_list;
  struct dentry *f_dentry; // associated dentry
  struct vfsmount *f_vfsmnt; // assoc mounted fsys
  struct file_operations *f_op;
  atomic_t f_count;
  unsigned int f_flags; // flags specified on open
  mode_t f_mode;
  loff_t f_pos; // file offset
  …
}
```

File Operations

- llseek()
- read(), readv()
- aio_read()
- write(), writev()
- aio_write()
- poll()
- ioctl()
- mmap()
- open()
- flush()
- fsync()
- aio_fsync()
- fasync()
- sendfile(), sendpage()
- get_unmapped_area(): gets unused address space to map the given file

Filesystem Types

- Linux must "know about" filesystem before mount
  - multiple (mounted) instances of each type possible
- special (virtual) filesystems (like /proc)
  - structuring technique to touch kernel data
  - examples:
    - /proc, /dev (devfs)
    - sockfs, pipefs, tmpfs, rootfs, shmfs
  - associated with fictitious block device (major# 0)
    - minor# distinguishes special filesystem types

Registering a Filesystem Type

- must register before mount
  - static (compile-time) or dynamic (modules)
- register_filesystem() / unregister_filesystem
  - adds file_system_type object to linked-list
    - file_systems (head; kernel global variable)
    - file_systems_lock (rw spinlock to protect list)
- file_system_type descriptor
  - name, flags, pointer to implementing module
  - list of superblocks (mounted instances)
  - read_super() – pointer to method for reading superblock
    - most important thing! filesystem specific

Data Structures Associated with Filesystems

```
struct file_system_type {
  const char *name;
  struct subsystem subsys;
  int fs_flags;
  struct super_block *(*get_sb)();
  void (*kill_sb)(struct super_block *);
  struct module *owner; // assoc module if any
  struct file_system_type *next;
  struct list_head fs_supers; // sb list
 }
There is only one above struct per filesystem.
```

Mounting a Filesystem

- Vfsmount is used to represent a specific instance of a filesystem—a mount point

```
struct vfsmount {
 struct dentry *mnt_mountpoint; // mnt point
 dentry
 struct dentry *mnt_root;// fs root dentry
 struct super_block *mnt_sb;
 atomic_t mnt_count; //usage count
 char *mnt_devname; // device file name
 …
}
```

VFS-related Task Fields

- task_struct fields
  - fs – includes root, pwd
    - pointers to dentries
  - files – includes file descriptor array fd[]
    - pointers to open file objects

Data structures associated with a process

- `struct files_struct,` the "files" field in task_struct
- `struct fs_struct` contains filesystem information related to a process and is pointed by the "fs" field in task_struct

```
struct fs_struct {
  struct dentry *root, *pwd, *altroot,
  struct vfsmount *rootmnt, *pwdmnt, *altrootmnt
  …
}
```

- `struct namespace,` enables each process to have a unique view of the mounted filesystems on the system (not in 2.2 kernels)

```
struct namespace {
  atomic_t count
  struct vfsmont *root; // mount obj of root directory
  struct list_head list; // list of mount points
  struct rw_semphore sem; // semaphore for namespace
}
```

Process-related Files

- current->fs (fs_struct)
  - root (for chroot jails)
  - pwd
  - umask (default file permissions)
- current->files (files_struct)
  - fd[] (file descriptor array – pointers to file objects)
    - 0, 1, 2 – stdin, stdout, stderr
  - originally 32, growable to 1,024 (RLIMIT_NOFILE)
    - complex structure for growing …
  - close_on_exec memory (bitmap)
- open files normally inherited across exec

Accessing FileSystem Data

- mmap()
  - Gives application direct memory-mapped access to the kernel's page cache data.
- Direct block I/O (*read, write*)
  - The read() system call reads data from block device into the kernel cache, then copies data from the kernel cached copy onto the application address space.

Linux Page-cache and Buffer-cache

- Buffer cache:
  - Holding individual disk blocks copies.
  - Using device and block No. indexes the cache entries.
  - Using Linked-list (unused, free, clean, dirty, locked, etc.) to minimize management overhead.
  - Using hash table to speed up cache finding.
  - Grouping several writes together (*dirty buffers*).

Linux Page-cache and Buffer-cache (cont.)

- Page cache:
  - 4K / page
  - Page cache entries are partially indexed by the file i-node number and its offset within the file.

Integration of page and buffer cache

- If the system become short on memory, the page cache tends to be easier to deal with to reclaim memory from.
- The individual blocks of a page cache entry are still managed through the buffer cache.
- Linux stores the file data only in the page cache to reduce the inefficiencies of double copies.

Linux Page-cache and Buffer-cache

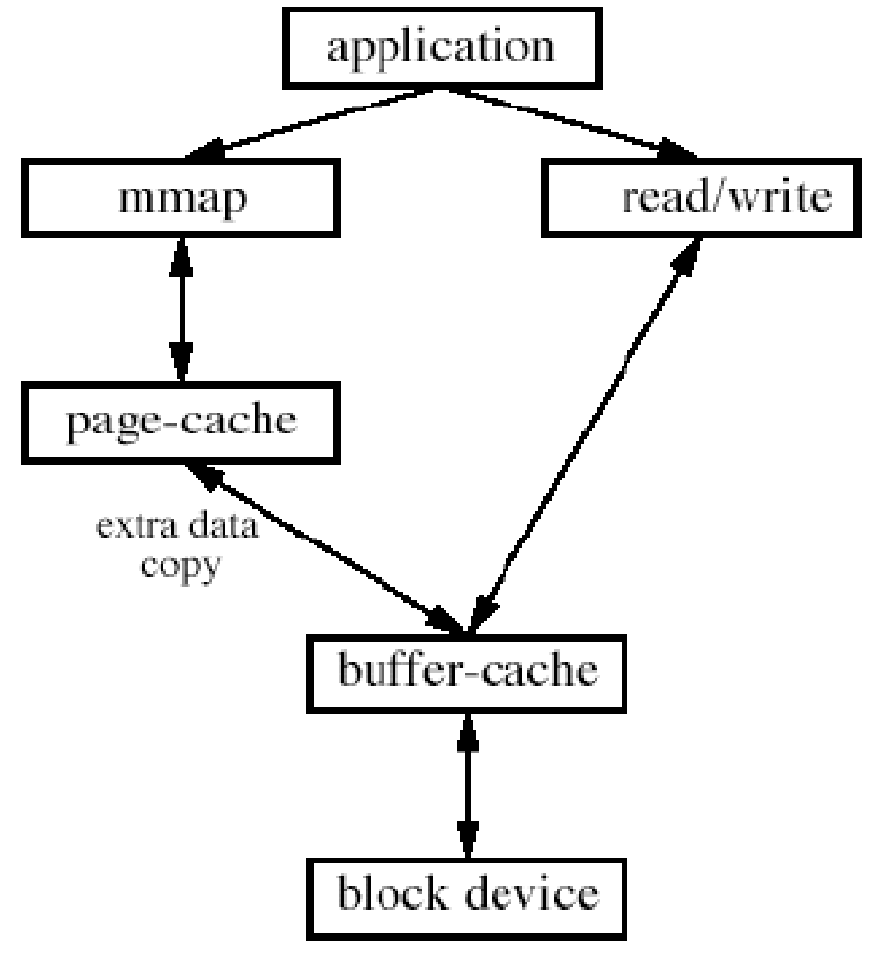


Fig. 1: Buffer cache and page cache

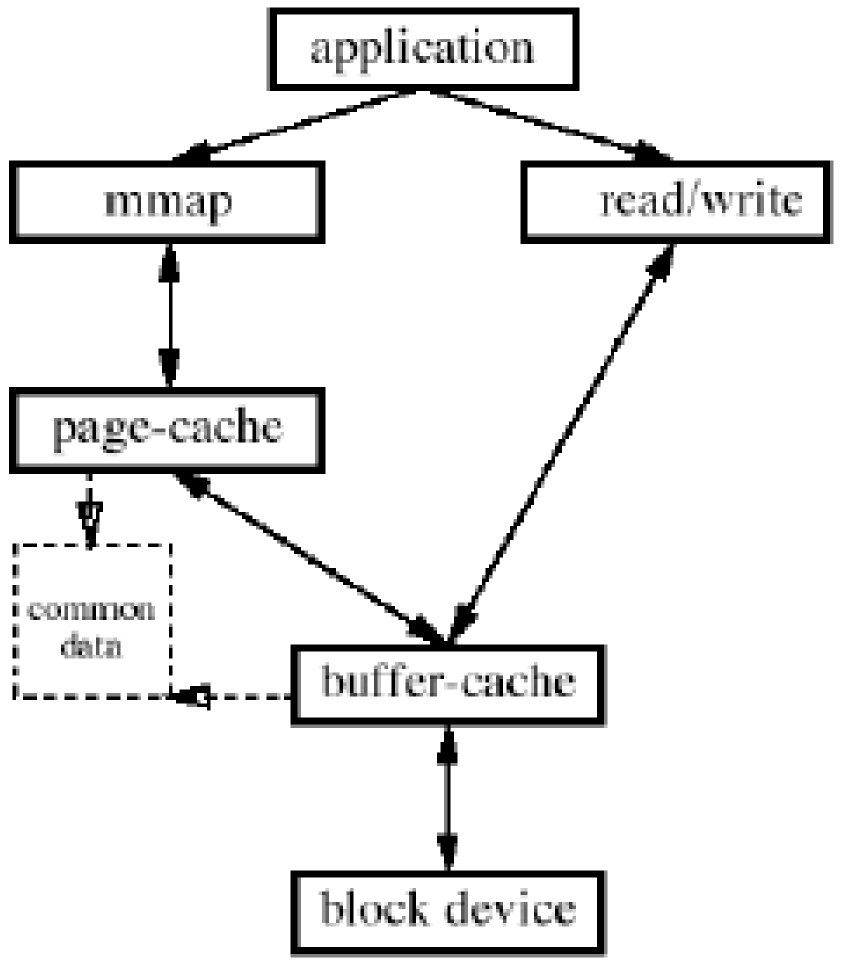


Fig. 2: Data is shared by page cache and buffer cache

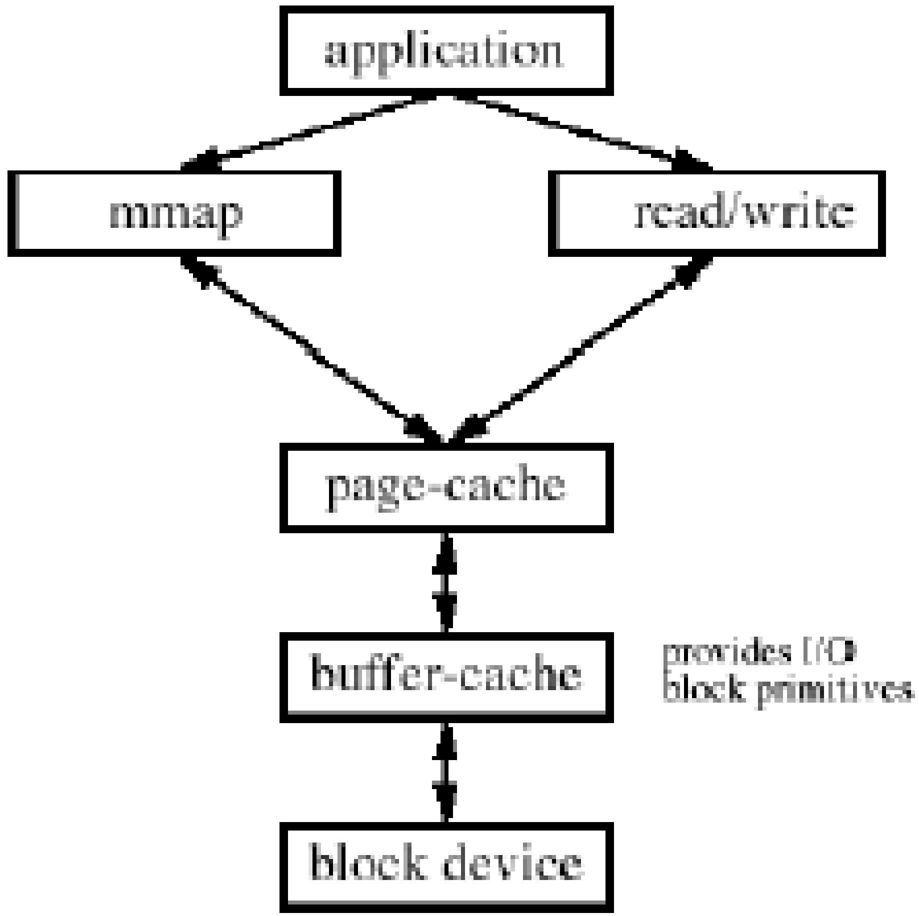


Fig. 3: Unified to page cache