UNIX File Management (continued)

THE UNIVERSITY OF NEW SOUTH WALES

OS storage stack (recap)

Application
FD table
OF table
VFS
FS
Buffer cache
Disk scheduler
Device driver

THE UNIVERSITY OF NEW SOUTH WALES

Virtual File System (VFS)

Application
FD table
OF table
VFS
FS
Buffer cache
Disk scheduler
Device driver

THE UNIVERSITY OF NEW SOUTH WALES

Older Systems only had a single file system

• They had file system specific open, close, read, write, … calls.

• However, modern systems need to support many file system types

– ISO9660 (CDROM), MSDOS (floppy), ext2fs, tmpfs

THE UNIVERSITY OF NEW SOUTH WALES
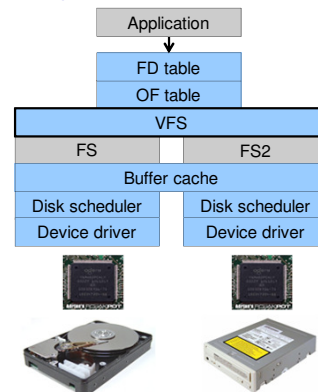
4

Supporting Multiple File Systems

Alternatives

• Change the file system code to understand different file system types
  – Prone to code bloat, complex, non-solution

• Provide a framework that separates file system independent and file system dependent code.
  – Allows different file systems to be "plugged in"

THE UNIVERSITY OF NEW SOUTH WALES

5

Virtual File System (VFS)

Application	
FD table	
OF table	
VFS	
FS	FS2
Buffer cache	
Disk scheduler	Disk scheduler
Device driver	Device driver

THE UNIVERSITY OF NEW SOUTH WALES

1

Virtual file system (VFS)

/

ext3

/home/leonidr

nfs

open("/home/leonidr/file", …);

Traversing the directory hierarchy may require VFS to issue requests to several underlying file systems

THE UNIVERSITY OF NEW SOUTH WALES

7

Virtual File System (VFS)

- Provides single system call interface for many file systems
  - E.g., UFS, Ext2, XFS, DOS, ISO9660,...
- Transparent handling of network file systems
  - E.g., NFS, AFS, CODA
- File-based interface to arbitrary device drivers (`/dev`)
- File-based interface to kernel data structures (`/proc`)
- Provides an indirection layer for system calls
  - File operation table set up at file open time
  - Points to actual handling code for particular type
  - Further file operations redirected to those functions

THE UNIVERSITY OF NEW SOUTH WALES

8

The file system independent code deals with vfs and vnodes

VFS   FS

vnode   inode

File Descriptor Tables

Open File Table

File system dependent code

THE UNIVERSITY OF NEW SOUTH WALES

VFS Interface

- Reference
  - S.R. Kleiman., *"Vnodes: An Architecture for Multiple File System Types in Sun Unix,"* USENIX Association: Summer Conference Proceedings, Atlanta, 1986
  - Linux and OS/161 differ slightly, but the principles are the same
- Two major data types
  - VFS
    - Represents all file system types
    - Contains pointers to functions to manipulate each file system as a whole (e.g. mount, unmount)
      - Form a standard interface to the file system
  - Vnode
    - Represents a file (inode) in the underlying filesystem
    - Points to the real inode
    - Contains pointers to functions to manipulate files/inodes (e.g. open, close, read, write,…)

THE UNIVERSITY OF NEW SOUTH WALES

10

Vfs and Vnode Structures

struct vnode

Generic (FS-independent) fields
fs_data
vnode ops

- size
- uid, gid
- ctime, atime, mtime
- …

ext2fs_read
ext2fs_write
…

FS-specific implementation of vnode operations

FS-specific fields

- Block group number
- Data block list
- …

THE UNIVERSITY OF NEW SOUTH WALES

11

Vfs and Vnode Structures

struct vfs

Generic (FS-independent) fields
fs_data
vfs ops

- Block size
- Max file size
- …

ext2_unmount
ext2_getroot
…

FS-specific implementation of FS operations

FS-specific fields

- i-nodes per group
- Superblock address
- …

THE UNIVERSITY OF NEW SOUTH WALES

12

A look at OS/161's VFS

The OS161's file system type
Represents interface to a mounted filesystem

```
struct fs {
    int        (*fs_sync)(struct fs *);
    const char *(*fs_getvolname)(struct fs *);
    struct vnode *(*fs_getroot)(struct fs *);
    int        (*fs_unmount)(struct fs *);

    void *fs_data;
};
```

Force the filesystem to flush its content to disk

Retrieve the volume name

Retrieve the vnode associated with the root of the filesystem

Unmount the filesystem
Note: mount called via function ptr passed to **vfs_mount**

Private file system specific data

THE UNIVERSITY OF NEW SOUTH WALES

13

Vnode

Count the number of "references" to this vnode

Lock for mutual exclusive access to counts

```
struct vnode {
    int vn_refcount;
    struct spinlock vn_countlock;k
    struct fs *vn_fs;
    void *vn_data;

    const struct vnode_ops *vn_ops;
};
```

Pointer to FS specific vnode data (e.g. inode)

Pointer to FS containing the vnode

Array of pointers to functions operating on vnodes

THE UNIVERSITY OF NEW SOUTH WALES

14

Vnode Ops

```
struct vnode_ops {
    unsigned long vop_magic;        /* should always be VOP_MAGIC */

    int (*vop_eachopen)(struct vnode *object, int flags_from_open);
    int (*vop_reclaim)(struct vnode *vnode);


    int (*vop_read)(struct vnode *file, struct uio *uio);
    int (*vop_readlink)(struct vnode *link, struct uio *uio);
    int (*vop_getdirentry)(struct vnode *dir, struct uio *uio);
    int (*vop_write)(struct vnode *file, struct uio *uio);
    int (*vop_ioctl)(struct vnode *object, int op, userptr_t data);
    int (*vop_stat)(struct vnode *object, struct stat *statbuf);
    int (*vop_gettype)(struct vnode *object, int *result);
    int (*vop_isseekable)(struct vnode *object, off_t pos);
    int (*vop_fsync)(struct vnode *object);
    int (*vop_mmap)(struct vnode *file /* add stuff */);
    int (*vop_truncate)(struct vnode *file, off_t len);
    int (*vop_namefile)(struct vnode *file, struct uio *uio);
```

THE UNIVERSITY OF NEW SOUTH WALES

15

Vnode Ops

```
    int (*vop_creat)(struct vnode *dir,
                const char *name, int excl,
                struct vnode **result);
    int (*vop_symlink)(struct vnode *dir,
                const char *contents, const char *name);
    int (*vop_mkdir)(struct vnode *parentdir,
                const char *name);
    int (*vop_link)(struct vnode *dir,
                const char *name, struct vnode *file);
    int (*vop_remove)(struct vnode *dir,
                const char *name);
    int (*vop_rmdir)(struct vnode *dir,
                const char *name);

    int (*vop_rename)(struct vnode *vn1, const char *name1,
                struct vnode *vn2, const char *name2);

    int (*vop_lookup)(struct vnode *dir,
                char *pathname, struct vnode **result);
    int (*vop_lookparent)(struct vnode *dir,
                char *pathname, struct vnode **result,
                char *buf, size_t len);
};
```

THE UNIVERSITY OF NEW SOUTH WALES

16

Vnode Ops

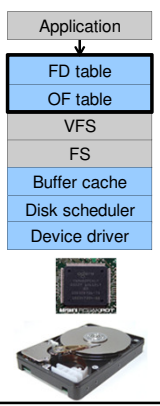• Note that most operations are on vnodes. How do we operate on file names?

– Higher level API on names that uses the internal VOP_* functions

```
int vfs_open(char *path, int openflags, struct vnode **ret);
void vfs_close(struct vnode *vn);
int vfs_readlink(char *path, struct uio *data);
int vfs_symlink(const char *contents, char *path);
int vfs_mkdir(char *path);
int vfs_link(char *oldpath, char *newpath);
int vfs_remove(char *path);
int vfs_rmdir(char *path);
int vfs_rename(char *oldpath, char *newpath);

int vfs_chdir(char *path);
int vfs_getcwd(struct uio *buf);
```

THE UNIVERSITY OF NEW SOUTH WALES

17

Example: OS/161 emufs vnode ops

```
/*
 * Function table for emufs
 files.
 */
static const struct vnode_ops
    emufs_fileops = {
    VOP_MAGIC,  /* mark this a
    valid vnode ops table */

    emufs_eachopen,
    emufs_reclaim,

    emufs_read,
    NOTDIR,  /* readlink */
    NOTDIR,  /* getdirentry */
    emufs_write,
    emufs_ioctl,
    emufs_stat,

    emufs_file_gettype,
    emufs_tryseek,
    emufs_fsync,
    UNIMP,   /* mmap */
    emufs_truncate,
    NOTDIR,  /* namefile */

    NOTDIR,  /* creat */
    NOTDIR,  /* symlink */
    NOTDIR,  /* mkdir */
    NOTDIR,  /* link */
    NOTDIR,  /* remove */
    NOTDIR,  /* rmdir */
    NOTDIR,  /* rename */

    NOTDIR,  /* lookup */
    NOTDIR,  /* lookparent */
};
```
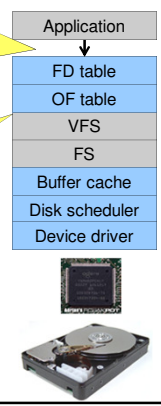
3

File Descriptor & Open File Tables

Application
FD table
OF table
VFS
FS
Buffer cache
Disk scheduler
Device driver

THE UNIVERSITY OF
NEW SOUTH WALES

Motivation

System call interface:
fd = open("file",…);
read(fd,…);write(fd,…);lseek(fd,…);
close(fd);

VFS interface:
vnode = vfs_open("file",…);
vop_read(vnode,uio);
vop_write(vnode,uio);
vop_close(vnode);

Application
FD table
OF table
VFS
FS
Buffer cache
Disk scheduler
Device driver

THE UNIVERSITY OF
NEW SOUTH WALES

File Descriptors

- File descriptors
  - Each open file has a file descriptor
  - Read/Write/lseek/…. use them to specify which file to operate on.
- State associated with a file descriptor
  - File pointer
    - Determines where in the file the next read or write is performed
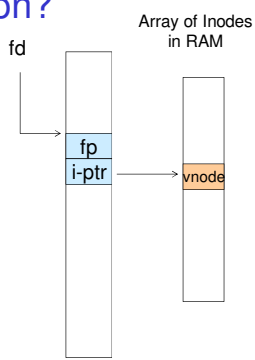 - Mode
    - Was the file opened read-only, etc….

THE UNIVERSITY OF
NEW SOUTH WALES

21

An Option?

•Use vnode numbers as file descriptors and add a file pointer to the vnode

•Problems
–What happens when we concurrently open the same file twice?
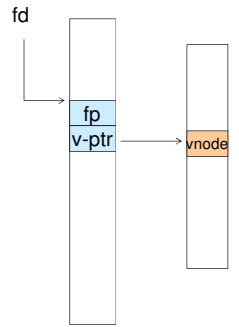•We should get two separate file descriptors and file pointers….

THE UNIVERSITY OF
NEW SOUTH WALES

22

An Option?

•Single global open file array
–*fd* is an index into the array
–Entries contain file pointer and pointer to a vnode

fd

Array of Inodes in RAM

fp
i-ptr → vnode

THE UNIVERSITY OF
NEW SOUTH WALES

23
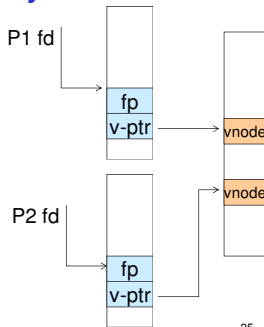
Issues

•File descriptor 1 is stdout
–Stdout is
•console for some processes
•A file for others
•Entry 1 needs to be different per process!

fd

fp
v-ptr → vnode

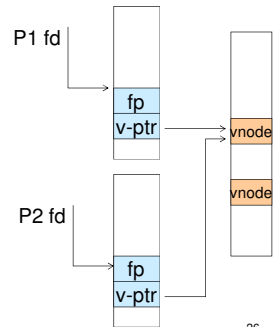THE UNIVERSITY OF
NEW SOUTH WALES

24

Per-process File Descriptor Array

- Each process has its own open file array
  - Contains fp, v-ptr etc.
  - *Fd* 1 can point to any vnode for each process (console, log file).

P1 fd
fp
v-ptr → vnode

vnode

P2 fd
fp
v-ptr

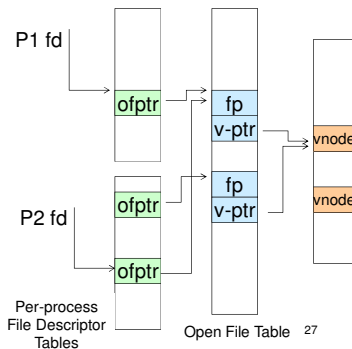THE UNIVERSITY OF NEW SOUTH WALES

25

Issue

- Fork
  - Fork defines that the child shares the file pointer with the parent
- Dup2
  - Also defines the file descriptors share the file pointer
- With per-process table, we can only have independent file pointers
  - Even when accessing the same file

P1 fd
fp
v-ptr → vnode

vnode

P2 fd
fp
v-ptr

THE UNIVERSITY OF NEW SOUTH WALES

26

Per-Process *fd* table with global open file table

- Per-process file descriptor array
  - Contains pointers to *open file table entry*
- Open file table array
  - Contain entries with a fp and pointer to an vnode.
- Provides
  - Shared file pointers if required
  - Independent file pointers if required
- Example:
  - All three *fds* refer to the same file, two share a file pointer, one has an independent file pointer
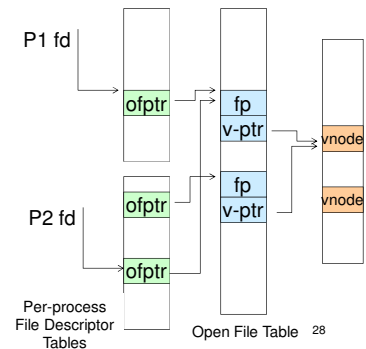
P1 fd
ofptr → fp / v-ptr → vnode
ofptr → fp / v-ptr → vnode

P2 fd
ofptr

Per-process File Descriptor Tables

Open File Table   27

Per-Process *fd* table with global open file table

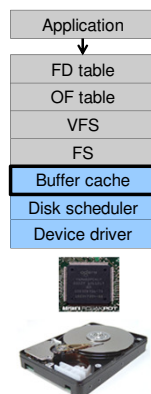- Used by Linux and most other Unix operating systems

P1 fd
ofptr → fp / v-ptr → vnode
ofptr → fp / v-ptr → vnode

P2 fd
ofptr

Per-process File Descriptor Tables

Open File Table   28

Buffer Cache

Application
FD table
OF table
VFS
FS
Buffer cache
Disk scheduler
Device driver

THE UNIVERSITY OF NEW SOUTH WALES

Buffer

- Buffer:
- Temporary storage used when transferring data between two entities
- Especially when the entities work at different rates
- Or when the unit of transfer is incompatible
- Example: between application program and disk

THE UNIVERSITY OF NEW SOUTH WALES

30

5

Slide 31

Buffering Disk Blocks

•Allow applications to work with arbitrarily sized region of a file
–However, apps can still optimise for a particular block size

Application Program

Buffers in Kernel RAM

Transfer of arbitrarily sized regions of file

Transfer of whole blocks

4		10
11		
12	13	7
		14
5		15
	16	6

THE UNIVERSITY OF NEW SOUTH WALES

Disk   31

Slide 32

Buffering Disk Blocks

•Writes can return immediately after copying to kernel buffer
–Avoids waiting until write to disk is complete
–Write is scheduled in the background

Application Program

Buffers in Kernel RAM

Transfer of arbitrarily sized regions of file

Transfer of whole blocks

4		10
11		
12	13	7
		14
5		15
	16	6

THE UNIVERSITY OF NEW SOUTH WALES

Disk   32

Slide 33

Buffering Disk Blocks

•Can implement read-ahead by pre-loading next block on disk into kernel buffer
–Avoids having to wait until next read is issued

Application Program

Buffers in Kernel RAM

Transfer of arbitrarily sized regions of file

Transfer of whole blocks

4		10
11		
12	13	7
		14
5		15
	16	6

THE UNIVERSITY OF NEW SOUTH WALES

Disk   33

Slide 34

Cache

•Cache:
–Fast storage used to temporarily hold data to speed up repeated access to the data
•Example: Main memory can cache disk blocks

THE UNIVERSITY OF NEW SOUTH WALES

34

Slide 35

Caching Disk Blocks

•On access
–Before loading block from disk, check if it is in cache first
•Avoids disk accesses
•Can optimise for repeated access for single or several processes

Application Program

Cached blocks in Kernel RAM

Transfer of arbitrarily sized regions of file

Transfer of whole blocks

4		10
11		
12	13	7
		14
5		15
	16	6

THE UNIVERSITY OF NEW SOUTH WALES

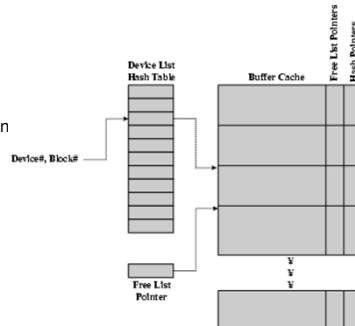Disk   35

Slide 36

Buffering and caching are related

•Data is read into buffer; an extra independent cache copy would be wasteful
•After use, block should be cached
•Future access may hit cached copy
•Cache utilises unused kernel memory space;
–may have to shrink, depending on memory demand

THE UNIVERSITY OF NEW SOUTH WALES

36

Unix Buffer Cache

On read
- Hash the device#, block#
- Check if match in buffer cache
- Yes, simply use in-memory copy
- No, follow the collision chain
- If not found, we load block from disk into buffer cache

THE UNIVERSITY OF
NEW SOUTH WALES

Replacement

- What happens when the buffer cache is full and we need to read another block into memory?
  - We must choose an existing entry to replace
  - Need a policy to choose a victim
    - Can use First-in First-out
    - Least Recently Used, or others.
      - Timestamps required for LRU implementation
    - However, is strict LRU what we want?

THE UNIVERSITY OF
NEW SOUTH WALES

38

File System Consistency

- File data is expected to survive
- Strict LRU could keep modified critical data in memory forever if it is frequently used.

THE UNIVERSITY OF
NEW SOUTH WALES

39

File System Consistency

- Generally, cached disk blocks are prioritised in terms of how critical they are to file system consistency
- Directory blocks, inode blocks if lost can corrupt entire filesystem
- E.g. imagine losing the root directory
- These blocks are usually scheduled for immediate write to disk
- Data blocks if lost corrupt only the file that they are associated with
- These blocks are only scheduled for write back to disk periodically
- In UNIX, flushd (*flush daemon*) flushes all modified blocks to disk every 30 seconds

THE UNIVERSITY OF
NEW SOUTH WALES

40

File System Consistency

- Alternatively, use a write-through cache
- All modified blocks are written immediately to disk
- Generates much more disk traffic
  - Temporary files written back
  - Multiple updates not combined
- Used by DOS
- Gave okay consistency when
  - Floppies were removed from drives
  - Users were constantly resetting (or crashing) their machines
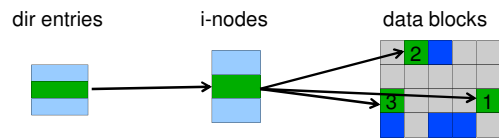- Still used, e.g. USB storage devices

THE UNIVERSITY OF
NEW SOUTH WALES

41

Consistency Case study: ext3 FS

THE UNIVERSITY OF
NEW SOUTH WALES

ext2fs reliability

- Disk writes are buffered in RAM
  - OS crash or power outage ==> lost data
  - Commit writes to disk periodically (e.g., every 30 sec)
  - Use the sync command to force a FS flush
- FS operations are non-atomic
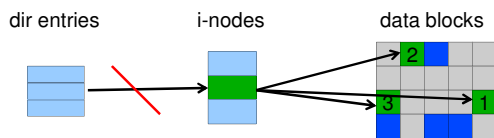 - Incomplete transaction can leave the FS in an inconsistent state

THE UNIVERSITY OF
NEW SOUTH WALES

---

ext2fs reliability

dir entries   i-nodes   data blocks

- Example: deleting a file
  1. Remove the directory entry
  2. Mark the i-node as free
  3. Mark disk blocks as free

THE UNIVERSITY OF
NEW SOUTH WALES

---

ext2fs reliability
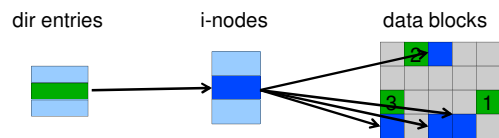
dir entries   i-nodes   data blocks

- Example: deleting a file
  1. Remove the directory entry--> crash
  2. Mark the i-node as free
  3. Mark disk blocks as free

The i-node and data blocks are lost

THE UNIVERSITY OF
NEW SOUTH WALES

---

ext2fs reliability

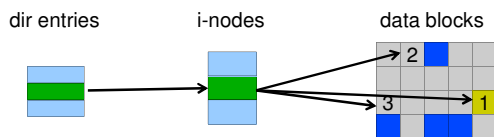dir entries   i-nodes   data blocks

- Example: deleting a file
  1. Mark the i-node as free --> crash
  2. Remove the directory entry
  3. Mark disk blocks as free

The dir entry points to the wrong file

THE UNIVERSITY OF
NEW SOUTH WALES

---

ext2fs reliability

dir entries   i-nodes   data blocks

- Example: deleting a file
  1. Mark disk blocks as free --> crash
  2. Remove the directory entry
  3. Mark the i-node as free

The file randomly shares disk blocks with other files

THE UNIVERSITY OF
NEW SOUTH WALES

---

ext2fs reliability

- e2fsck
  - Scans the disk after an unclean shutdown and attempts to restore FS invariants
- Journaling file systems
  - Keep a journal of FS updates
  - Before performing an atomic update sequence,
  - write it to the journal
  - Replay the last journal entries upon an unclean shutdown
  - Example: ext3fs

THE UNIVERSITY OF
NEW SOUTH WALES

The ext3 file system

- Design goals
  - Add journaling capability to the ext2 FS
  - Backward and forward compatibility with ext2
    - Existing ext2 partitions can be mounted as ext3
  - Leverage the proven ext2 performance
  - Reuse most of the ext2 code base
  - Reuse ext2 tools, including e2fsck

THE UNIVERSITY OF
NEW SOUTH WALES   49

The ext3 journal

Option1: Journal FS data structure updates

- Example:
  - **Start transaction**
  - Delete dir entry
  - Delete i-node
  - Release blocks 32, 17, 60
  - **End transaction**

Option2: Journal disk block updates

- Example:
  - **Start transaction**
  - Update block #n1 *(contains the dir entry)*
  - Update block #n2 *(i-node allocation bitmap)*
  - Update block #n3 *(data block allocation bitmap)*
  - **Add transaction**

Question: which approach is better?

THE UNIVERSITY OF
NEW SOUTH WALES   50

The ext3 journal

Option1: Journal FS data structure updates

✓ Efficient use of journal space; hence faster journaling

✗ Individual updates are applied separately

✗ The journaling layer must understand FS semantics

Option2: Journal disk block updates

✗ Even a small update adds a whole block to the journal

✓ Multiple updates to the same block can be aggregated into a single update

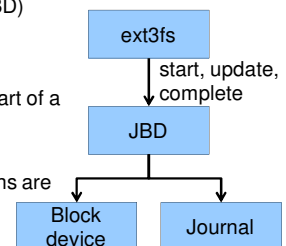✓ The journaling layer is FS-independent (easier to implement)

Ext3 implements Option 2

THE UNIVERSITY OF
NEW SOUTH WALES   51

Journaling Block Device (JBD)

- The ext3 journaling layer is called Journaling Block Device (JBD)
- JBD interface
  - Start a new transaction
  - Update a disk block as part of a transaction
  - Complete a transaction
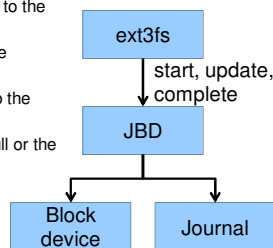   - Completed transactions are cached in RAM

ext3fs → start, update, complete → JBD → Block device / Journal

THE UNIVERSITY OF
NEW SOUTH WALES   52
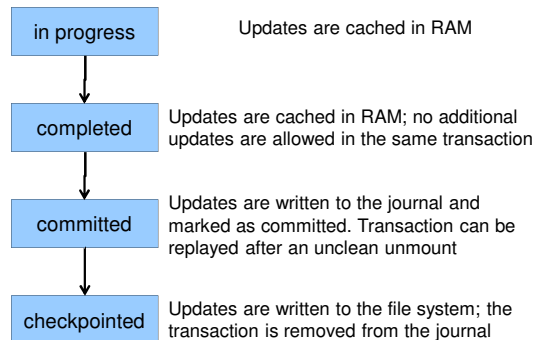
Journaling Block Device (JBD)

- JBD interface (continued)
  - Commit: write transaction data to the journal (persistent storage)
    - Multiple FS transactions are committed in one go
  - Checkpoint: flush the journal to the disk
    - Used when the journal is full or the FS is being unmounted

ext3fs → start, update, complete → JBD → Block device / Journal

THE UNIVERSITY OF
NEW SOUTH WALES   53

Transaction lifecycle

in progress — Updates are cached in RAM

↓

completed — Updates are cached in RAM; no additional updates are allowed in the same transaction

↓

committed — Updates are written to the journal and marked as committed. Transaction can be replayed after an unclean unmount

↓

checkpointed — Updates are written to the file system; the transaction is removed from the journal

THE UNIVERSITY OF
NEW SOUTH WALES   54

Journaling modes

- Ext3 supports two journaling modes
  - Metadata+data
    - Enforces atomicity of all FS operations
  - Metadata journaling
    - Metadata is journaled
    - Data blocks are written directly to the disk
    - Improves performance
    - Enforces file system integrity
    - Does not enforce atomicity of write's
      - New file content can be stale blocks

THE UNIVERSITY OF
NEW SOUTH WALES

55

JBD

- JBD can keep the journal on a block device or in a file
  - Enables compatibility with ext2 (the journal is just a normal file)
- JBD is independent of ext3-specific data structures
  - Separation of concerns
    - The FS maintains on-disk data and metadata
    - JBD takes care of journaling
  - Code reuse
    - JBD can be used by any other FS that requires journaling

THE UNIVERSITY OF
NEW SOUTH WALES

56