UNIX File Management

THE UNIVERSITY OF
NEW SOUTH WALES

UNIX File Management

- We will focus on two types of files
  – Ordinary files (stream of bytes)
  – Directories
- And mostly ignore the others
  – Character devices
  – Block devices
  – Named pipes
  – Sockets
  – Symbolic links

UNIX index node *(inode)*

- Each file is represented by an Inode
- Inode contains all of a file's metadata
  - Access rights, owner,accounting info
  - (partial) block index table of a file
- Each inode has a unique number (within a partition)
  - System oriented name
  - Try 'ls –i' on Unix (Linux)
- Directories map file names to inode numbers
  - Map human-oriented to system-oriented names
  - Mapping can be *many-to-one*
    - Hard links

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (10)
single indirect
double indirect
triple indirect

Inode Contents

- Mode
  - Type
    - Regular file or directory
  - Access mode
    - rwxrwxrwx
- Uid
  - User ID
- Gid
  - Group ID

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (10)
single indirect
double indirect
triple indirect

Inode Contents

- atime
  - Time of last access
- ctime
  - Time when file was created
- mtime
  - Time when file was last modified

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (10)
single indirect
double indirect
triple indirect

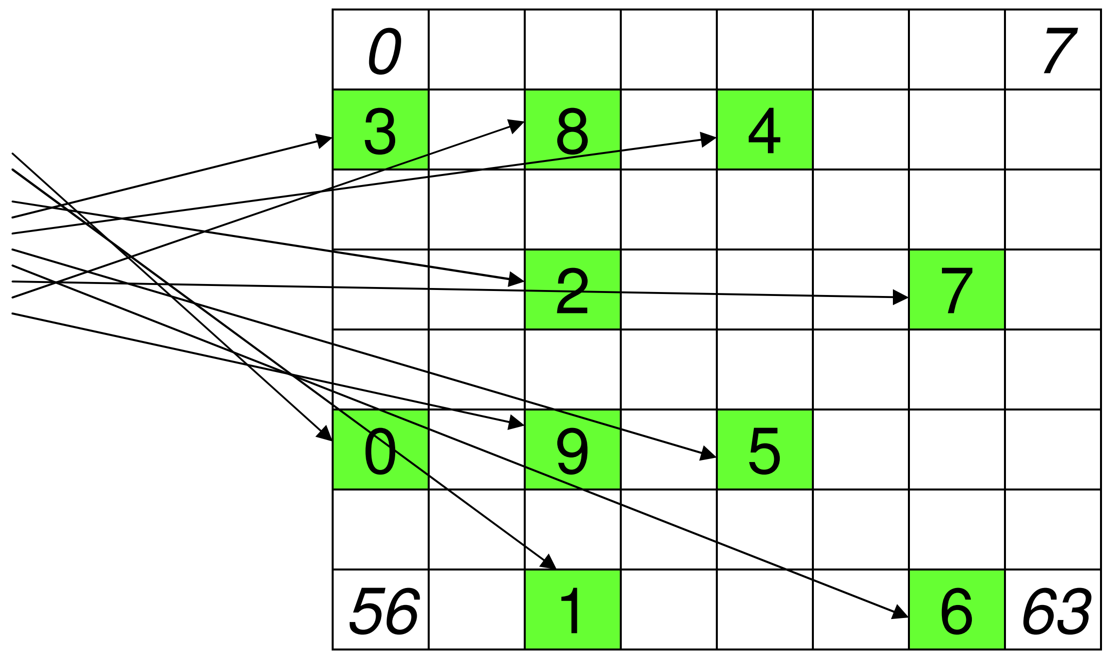
Inode Contents

- Size
  - Size of the file in bytes
- Block count
  - Number of disk blocks used by the file.
- Note that number of blocks can be much less than expected given the file size
  - Files can be sparsely populated
    - E.g. write(f,“hello”); lseek(f, 1000000); write(f, “world”);
    - Only needs to store the start an end of file, not all the empty blocks in between.
      - Size = 1000005
      - Blocks = 2 + overheads

Inode Contents

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (10) 40,58,26,8,12, 44,62,30,10,42
single indirect
double indirect
triple indirect

- Direct Blocks
  - Block numbers of first 10 blocks in the file
  - Most files are small
    - We can find blocks of file *directly* from the inode



Disk

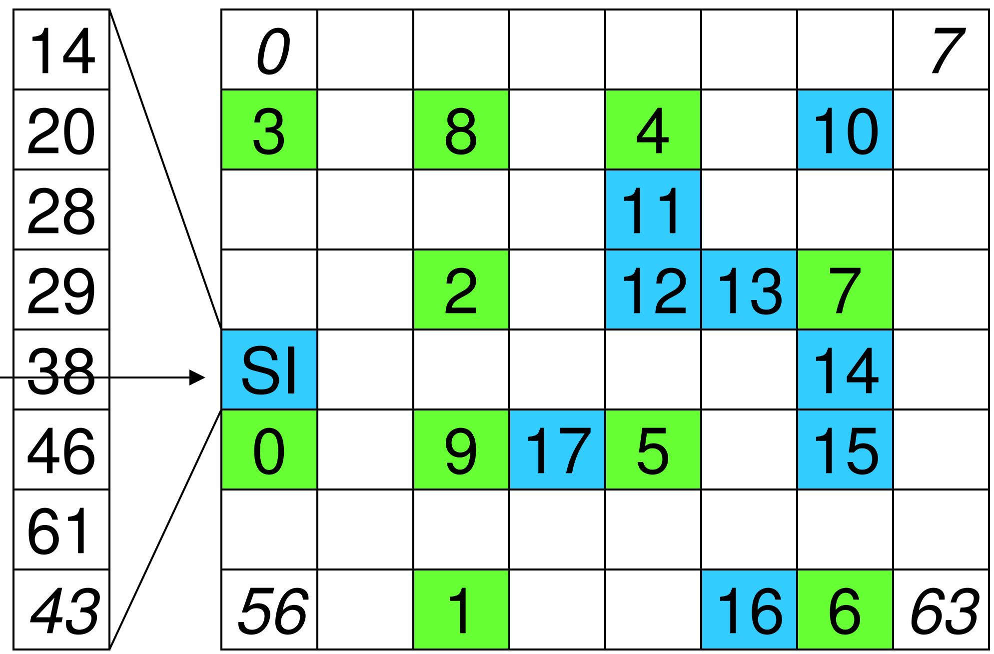
Problem

- How do we store files greater than 10 blocks in size?
  - Adding significantly more direct entries in the inode results in many unused entries most of the time.

Inode Contents

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (10) 40,58,26,8,12, 44,62,30,10,42
single indirect: 32
double indirect
triple indirect

- Single Indirect Block
  - Block number of a block containing block numbers
    - In this case 8



Disk

Single Indirection

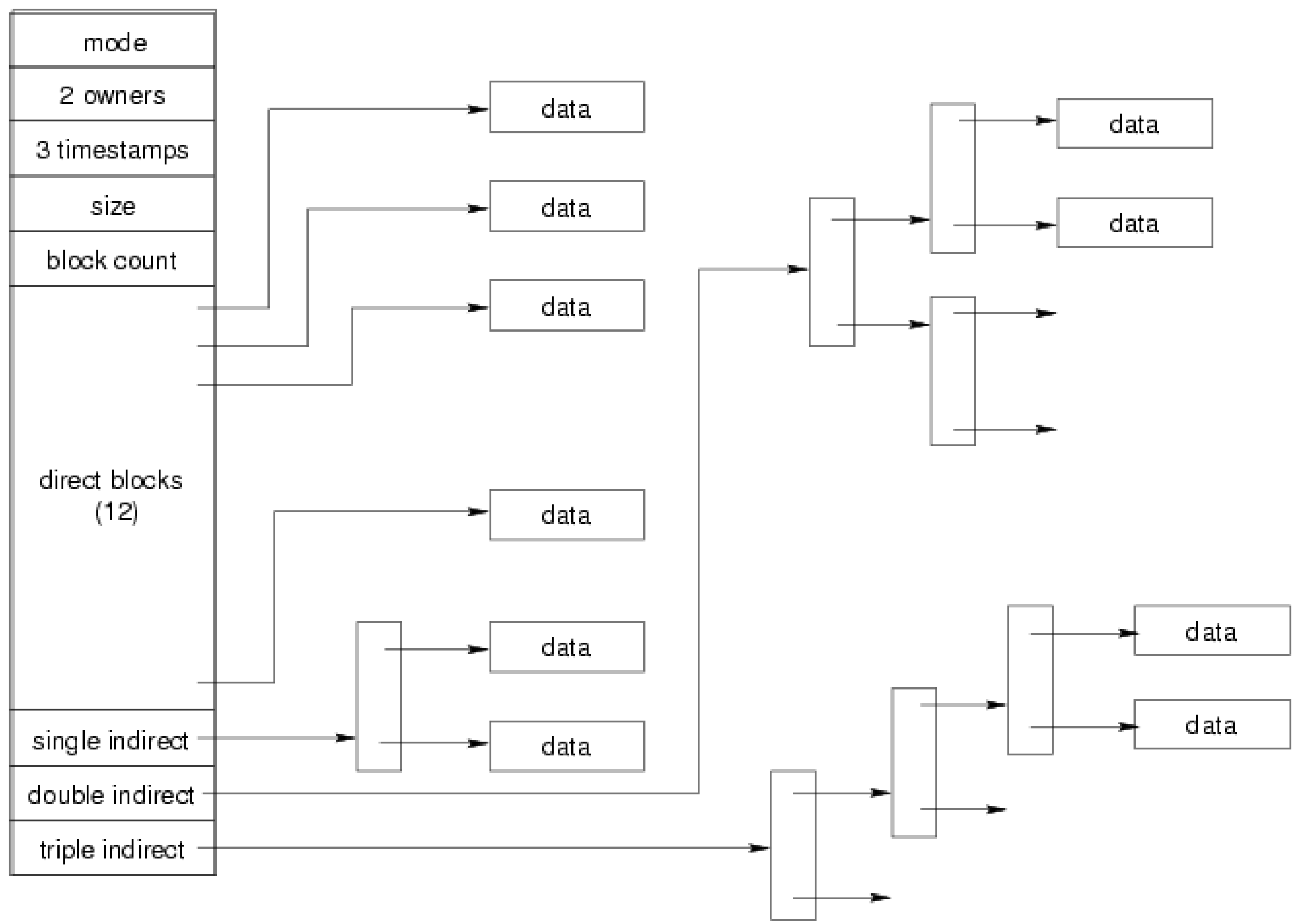
- Requires two disk access to read
  - One for the indirect block; one for the target block
- Max File Size
  - In previous example
    - 10 direct + 8 indirect = 18 block file
  - A more realistic example
    - Assume 1Kbyte block size, 4 byte block numbers
    - 10 * 1K + 1K/4 * 1K = 266 Kbytes
- For large majority of files (< 266 K), only one or two accesses required to read any block in file.

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (10) 40,58,26,8,12, 44,62,30,10,42
single indirect: 32
double indirect
triple indirect

Inode Contents

- Double Indirect Block
  - Block number of a block containing block numbers of blocks containing block numbers
- Triple Indirect
  - Block number of a block containing block numbers of blocks containing block numbers of blocks containing block numbers ☺

Unix Inode Block Addressing Scheme



Max File Size

- Assume 4 bytes block numbers and 1K blocks
- The number of addressable blocks
  - Direct Blocks = 12
  - Single Indirect Blocks = 256
  - Double Indirect Blocks = 256 * 256 = 65536
  - Triple Indirect Blocks = 256 * 256 * 256 = 16777216
- Max File Size
  - 12 + 256 + 65536 + 16777216 = 16843020 = 16 GB

Some Best and Worst Case Access Patterns

- To read 1 byte
  - Best:
    - 1 access via direct block
  - Worst:
    - 4 accesses via the triple indirect block
- To write 1 byte
  - Best:
    - 1 write via direct block (with no previous content)
  - Worst:
    - 4 reads (to get previous contents of block via triple indirect) + 1 write (to write modified block back)

Worst Case Access Patterns with Unallocated Indirect Blocks

- Worst to write 1 byte
  - 4 writes (3 indirect blocks; 1 data)
  - 1 read, 4 writes (read-write 1 indirect, write 2; write 1 data)
  - 2 reads, 3 writes (read 1 indirect, read-write 1 indirect, write 1; write 1 data)
  - 3 reads, 2 writes (read 2, read-write 1; write 1 data)
- Worst to read 1 byte
  - If reading writes an zero-filled block on disk
    - Worst case is same as write 1 byte
  - If not, worst-case depends on how deep is the current indirect block tree.

Inode Summary

- The inode contains the on disk data associated with a file
  - Contains mode, owner, and other bookkeeping
  - Efficient random and sequential access via *indexed allocation*
  - Small files (the majority of files) require only a single access
  - Larger files require progressively more disk accesses for random access
    - Sequential access is still efficient
  - Can support really large files via increasing levels of indirection

Where/How are Inodes Stored

Boot Block	Super Block	Inode Array	Data Blocks

- System V Disk Layout (s5fs)
  - Boot Block
    - contain code to bootstrap the OS
  - Super Block
    - Contains attributes of the file system itself
      - e.g. size, number of inodes, start block of inode array, start of data block area, free inode list, free data block list
  - Inode Array
  - Data blocks

Some problems with s5fs

- Inodes at start of disk; data blocks end
  - Long seek times
    - Must read inode before reading data blocks
- Only one superblock
  - Corrupt the superblock and entire file system is lost
- Block allocation suboptimal
  - Consecutive free block list created at FS format time
    - Allocation and de-allocation eventually randomises the list resulting the random allocation
- Inodes allocated randomly
  - Directory listing resulted in random inode access patterns

Berkeley Fast Filesystem (FFS)

- Historically followed s5fs
  - Addressed many limitations with s5fs
  - Linux mostly similar, so we will focus on Linux

The Linux Ext2 File System

- Second Extended Filesystem
  – Evolved from Minix filesystem (via "Extended Filesystem")
- Features
  – Block size (1024, 2048, and 4096) configured at FS creation
  – Pre-allocated inodes (max number also configured at FS creation)
  – Block groups to increase locality of reference (from BSD FFS)
  – Symbolic links < 60 characters stored within inode
- Main Problem: unclean unmount →**`e2fsck`**
  – Ext3fs keeps a journal of (meta-data) updates
  – Journal is a file where updated are logged
  – Compatible with ext2fs

Layout of an Ext2 Partition

Boot Block	Block Group 0	….	Block Group *n*

- Disk divided into one or more *partitions*
- Partition:
  - Reserved boot block,
  - Collection of equally sized *block groups*
  - All block groups have the same structure

Layout of a Block Group

Super Block	Group Descrip-tors	Data Block Bitmap	Inode Bitmap	Inode Table	Data blocks
1 blk	*n* blks	1 blk	1 blk	*m* blks	*k* blks

- *Replicated* super block
  - For e2fsck
- Group descriptors
- Bitmaps identify used inodes/blocks
- All block have the same number of data blocks
- Advantages of this structure:
  - Replication simplifies recovery
  - Proximity of inode tables and data blocks (reduces seek time)

Superblocks

- Size of the file system, block size and similar parameters
- Overall free inode and block counters
- Data indicating whether file system check is needed:
  - Uncleanly unmounted
  - Inconsistency
  - Certain number of mounts since last check
  - Certain time expired since last check
- Replicated to provide redundancy to add recoverability

Group Descriptors

- Location of the bitmaps
- Counter for free blocks and inodes in this group
- Number of directories in the group

Performance considerations

- EXT2 optimisations
  - Read-ahead for directories
    - For directory searching
  - Block groups cluster related inodes and data blocks
  - Pre-allocation of blocks on write (up to 8 blocks)
    - 8 bits in bit tables
    - Better contiguity when there are concurrent writes
- FFS optimisations
  - Files within a directory in the same group

Thus far…

- Inodes representing files laid out on disk.
- Inodes are referred to by number!!!
  - How do users name files? By number?
  - Try ls –i to see how useful inode numbers are….

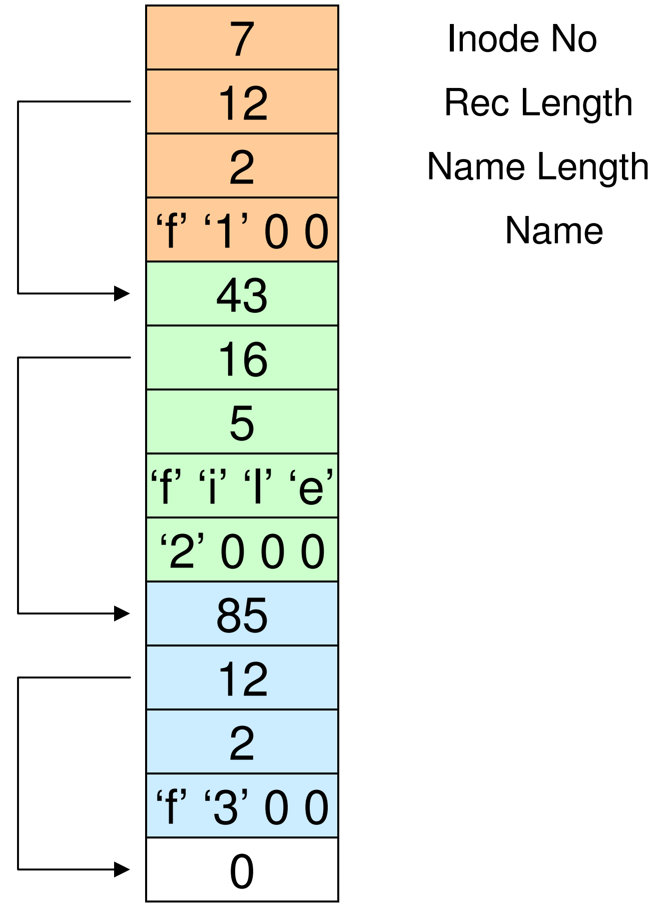
Ext2fs Directories

inode	rec_len	name_len	type	name…

- Directories are files of a special type
  - Consider it a file of special format, managed by the kernel, that uses most of the same machinery to implement it
    - Inodes, etc…
- Directories translate names to inode numbers
- Directory entries are of variable length
- Entries can be deleted in place
  - inode = 0
  - Add to length of previous entry
  - use null terminated strings for names

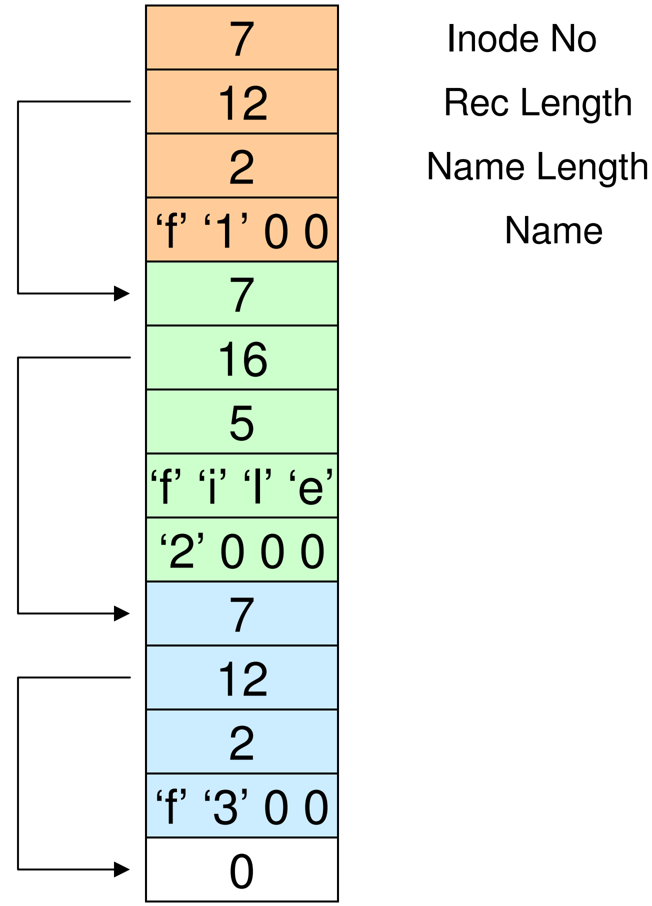
Ext2fs Directories

- "f1" = inode 7
- "file2" = inode 43
- "f3" = inode 85



Ext2fs Directories

- Note that inodes can have more than one name
  - Called a *Hard Link*
  - Inode (file) 7 has three names
    - "f1" = inode 7
    - "file2" = inode 7
    - "f3" = inode 7



THE UNIVERSITY OF NEW SOUTH WALES

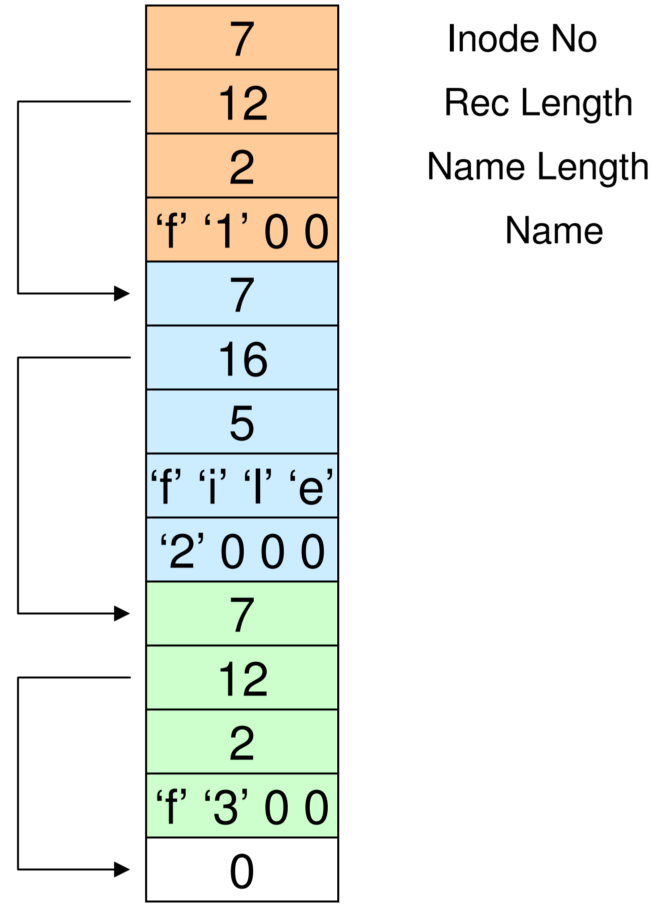
mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (10) 40,58,26,8,12, 44,62,30,10,42
single indirect: 32
double indirect
triple indirect

Inode Contents

- We can have many name for the same inode.
- When we delete a file by name, i.e. remove the directory entry (link), how does the file system know when to delete the underlying inode?
  - Keep a *reference count* in the inode
    - Adding a name (directory entry) increments the count
    - Removing a name decrements the count
    - If the reference count == 0, then we have no names for the inode (it is unreachable), we can delete the inode (underlying file or directory)

Ext2fs Directories

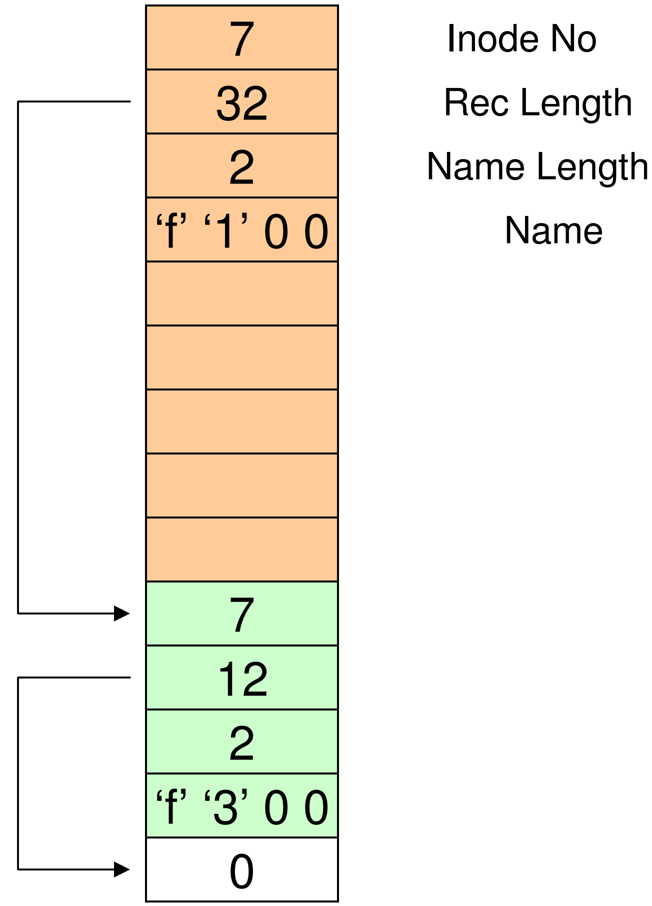
- Deleting a filename
  - rm file2



THE UNIVERSITY OF NEW SOUTH WALES

Ext2fs Directories

- Deleting a filename
  - rm file2
- Adjust the record length to skip to next valid entry



THE UNIVERSITY OF NEW SOUTH WALES

Kernel File-related Data Structures and Interfaces

- We have reviewed how files and directories are stored on disk
- We know the UNIX file system-call interface
  - open, close, read, write, lseek,…..
- What is in between?

What do we need to keep track of?

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

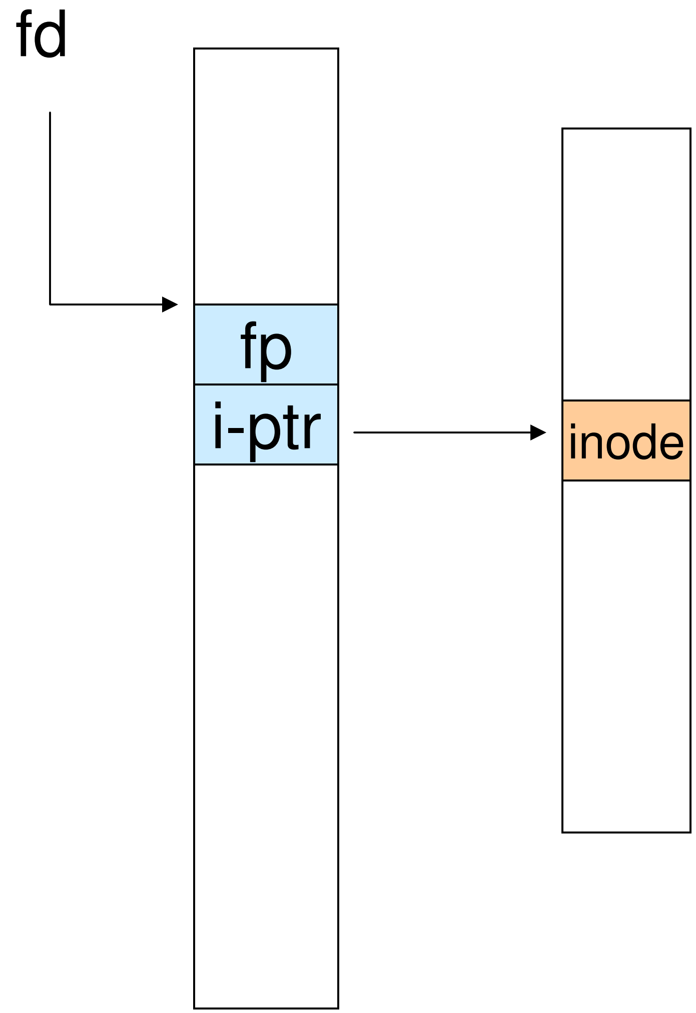
An Option?

- Use inode numbers as file descriptors and add a file pointer to the inode

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

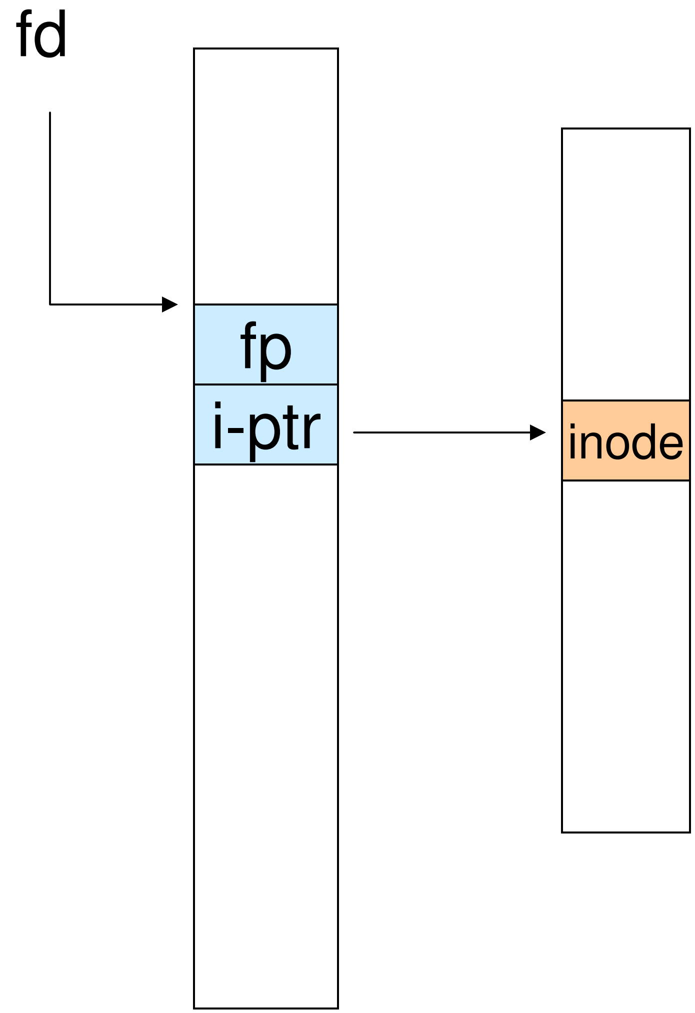
An Option?

- Single global open file array
  - *fd* is an index into the array
  - Entries contain file pointer and pointer to an inode



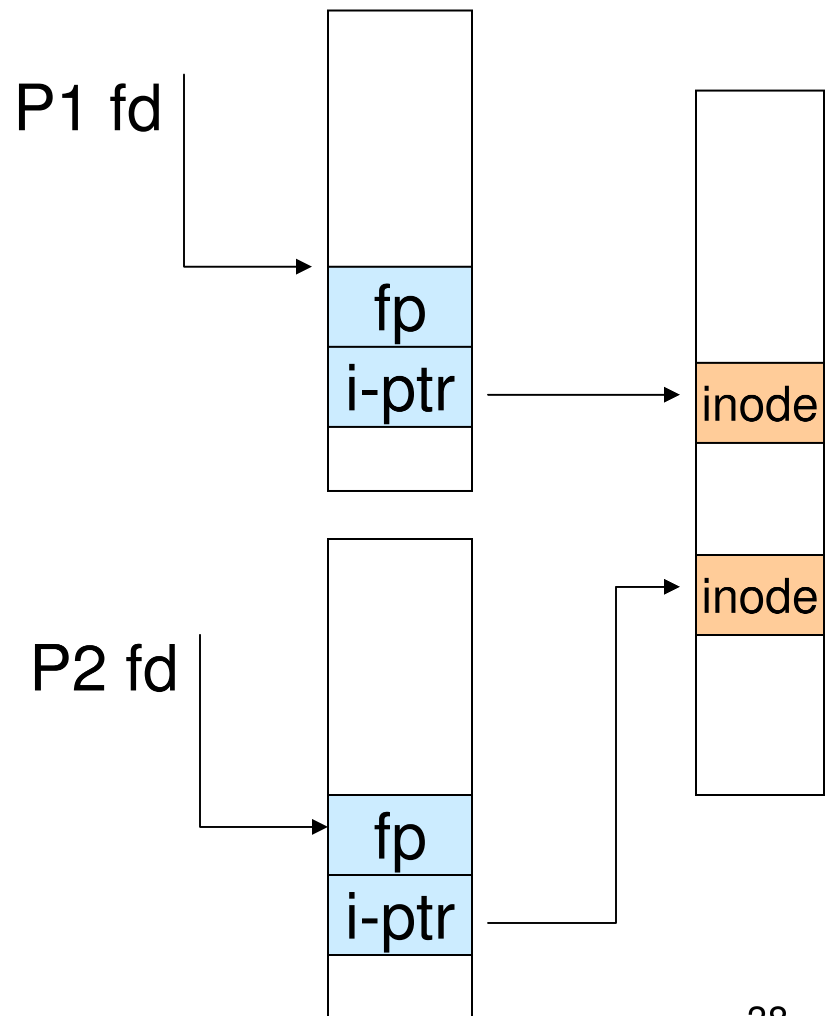
Issues

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



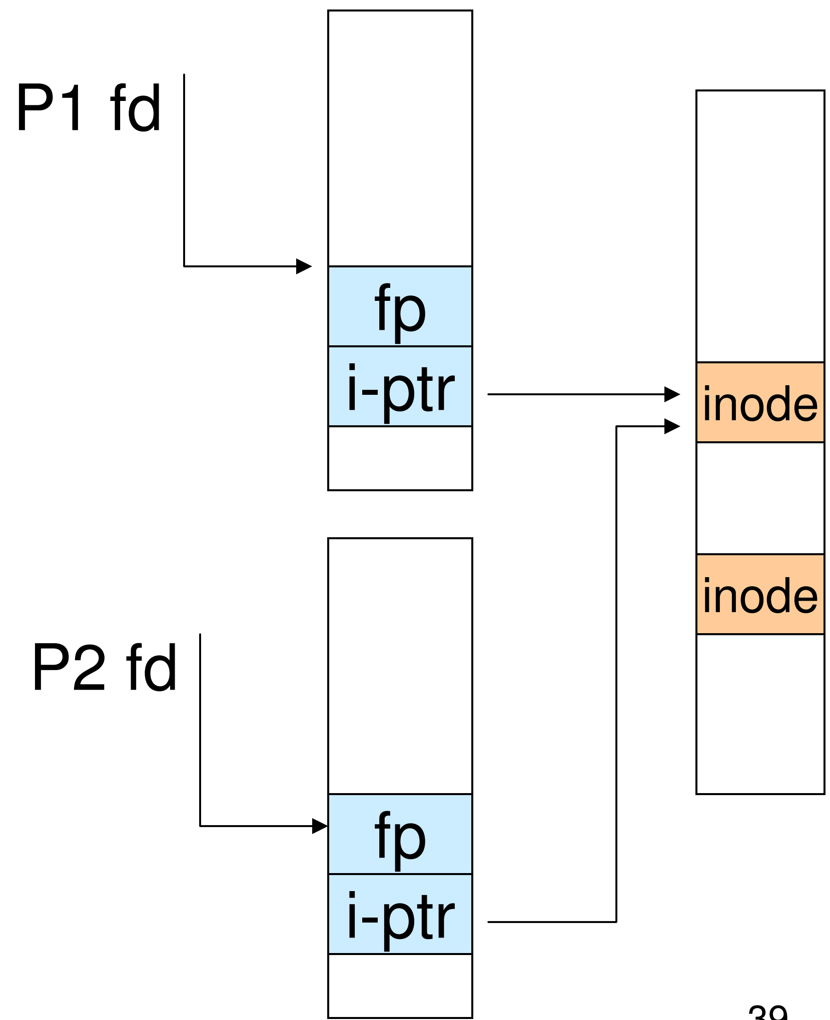
Per-process File Descriptor Array

- Each process has its own open file array
  - Contains fp, i-ptr etc.
  - *Fd* 1 can be any inode for each process (console, log file).



Issue

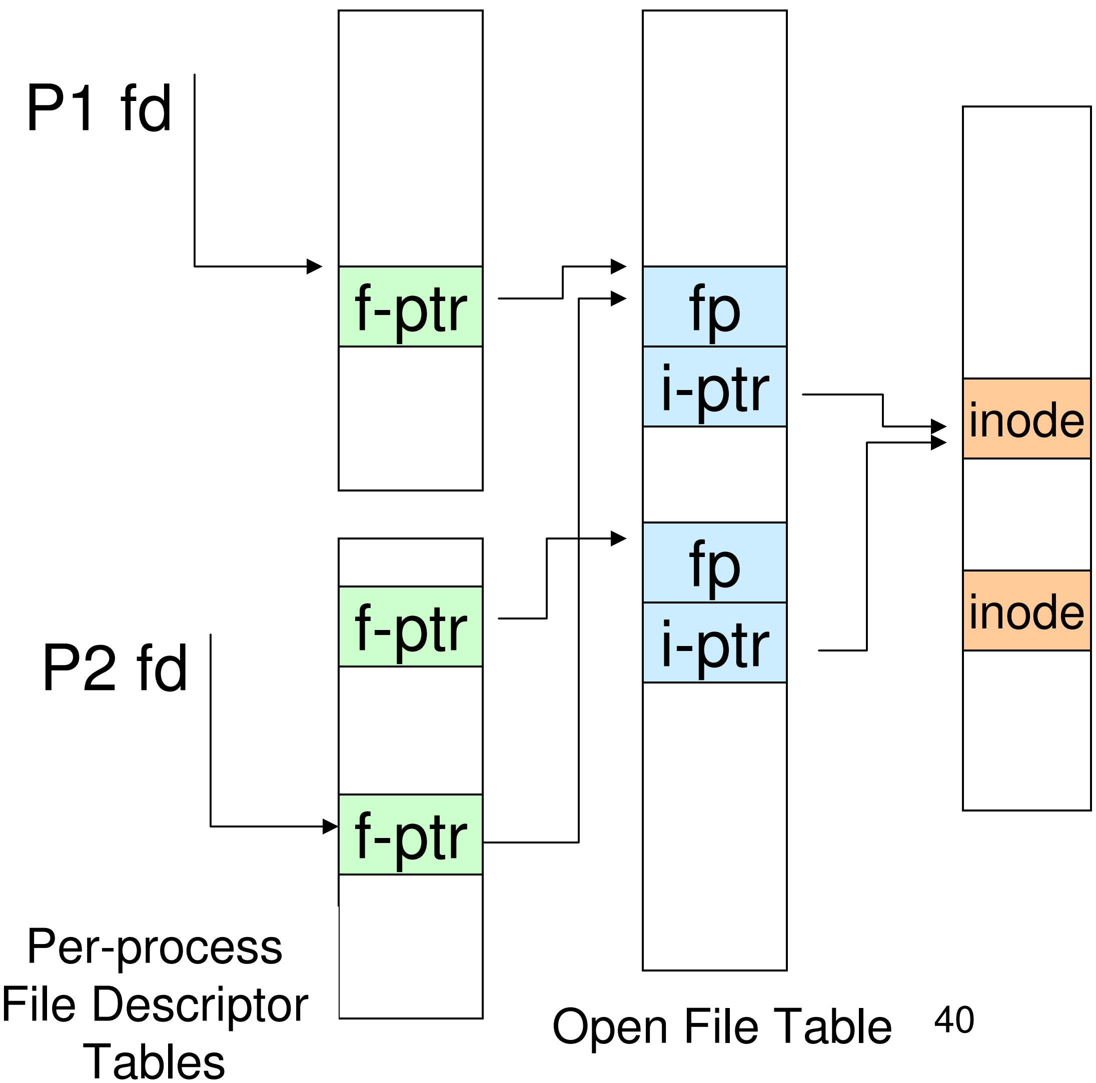
- Fork
  - Fork defines that the child shares the file pointer with the parent
- Dup2
  - Also defines the file descriptors share the file pointer
- With per-process table, we can only have independent file pointers
  - Even when accessing the same file



THE UNIVERSITY OF NEW SOUTH WALES

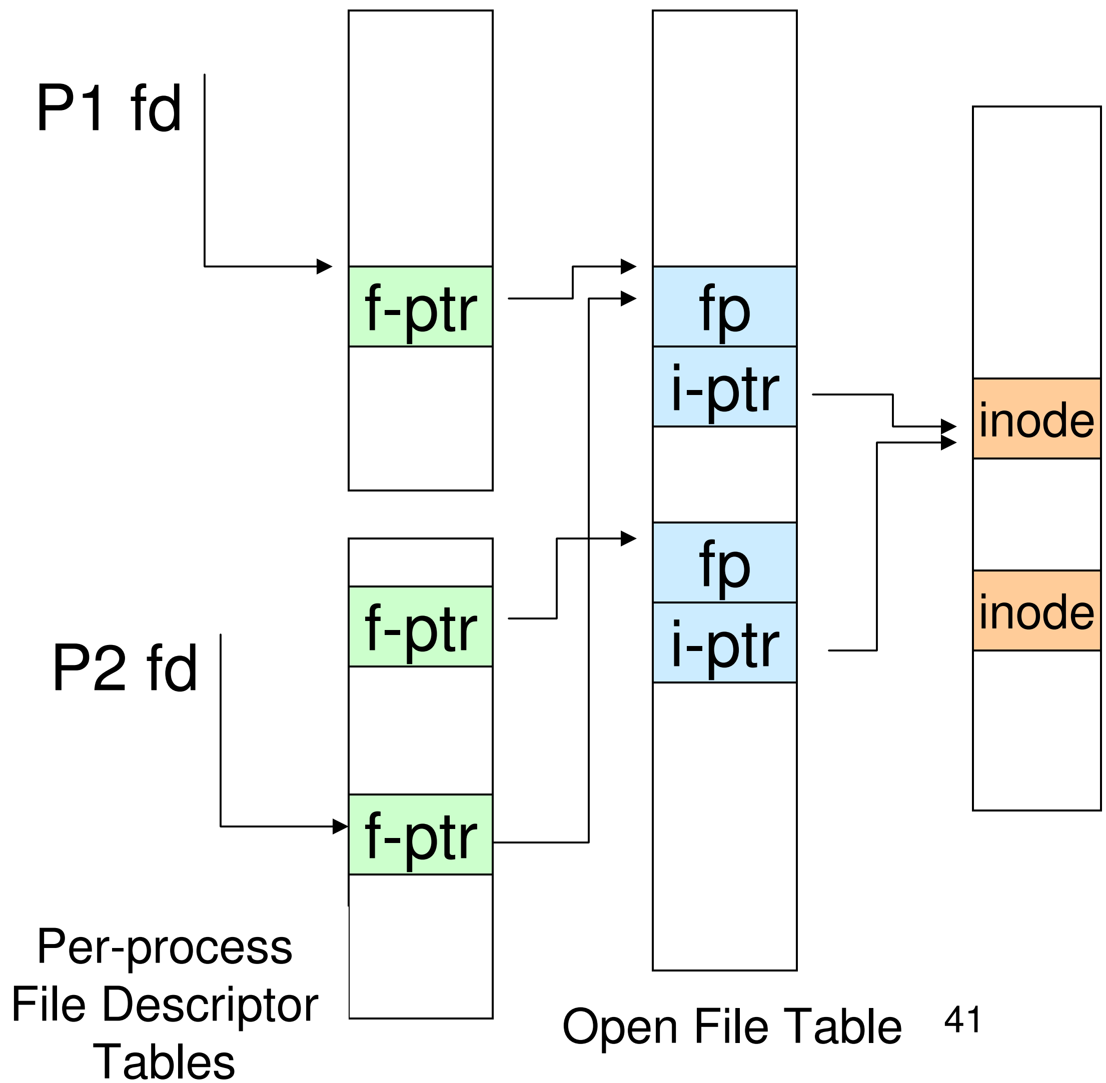
Per-Process *fd* table with global open file table

- 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 inode.
- 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



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

- Used by Linux and most other Unix operating systems



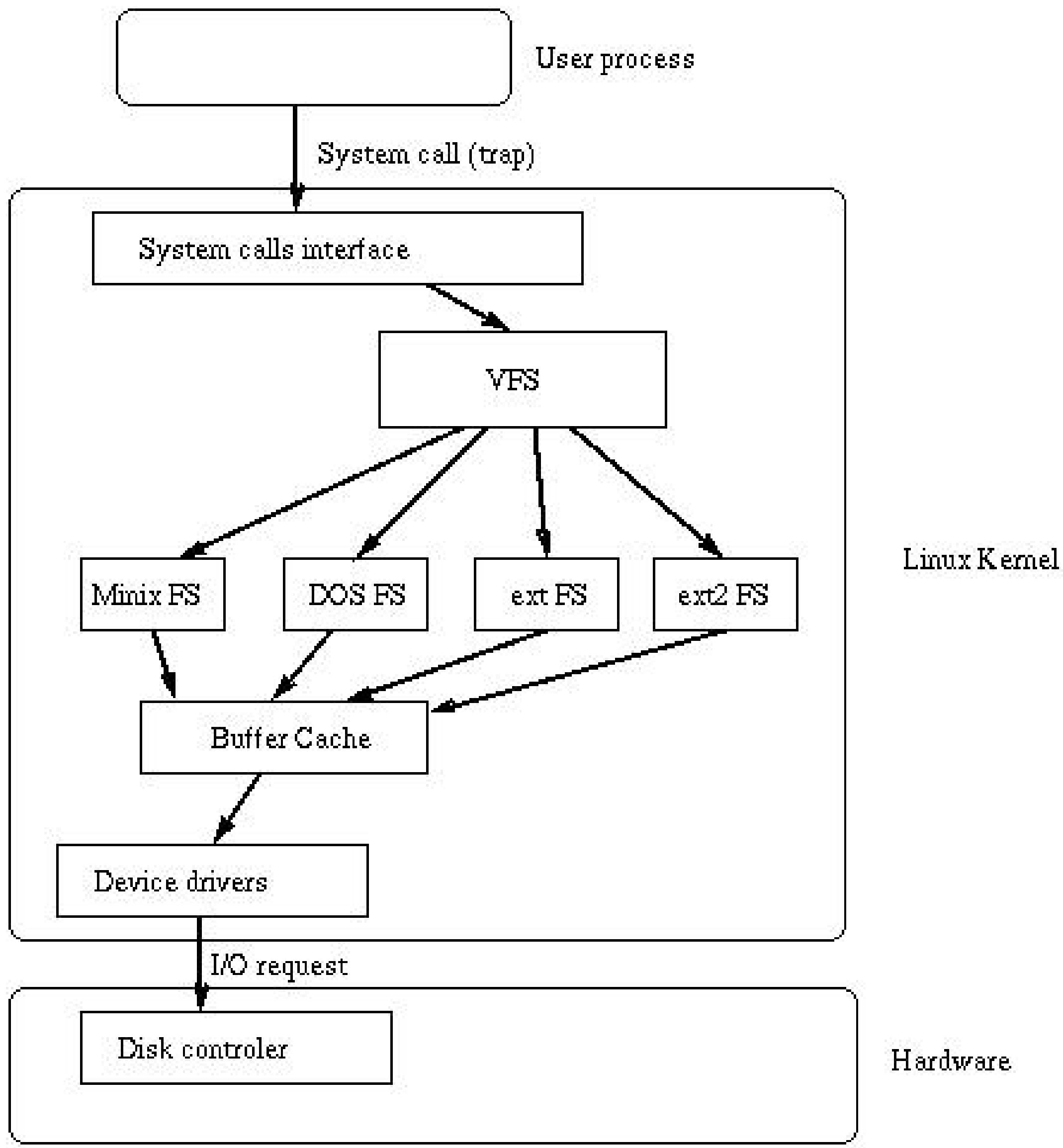
Older Systems only had a single file system

- They had file system specific open, close, read, write, … calls.
- The open file table pointed to an in-memory representation of the inode
  - inode format was specific to the file system used (s5fs, Berkley FFS, etc)
- However, modern systems need to support many file system types
  - ISO9660 (CDROM), MSDOS (floppy), ext2fs, tmpfs

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"
    - File descriptor, open file table and other parts of the kernel can be independent of underlying file system

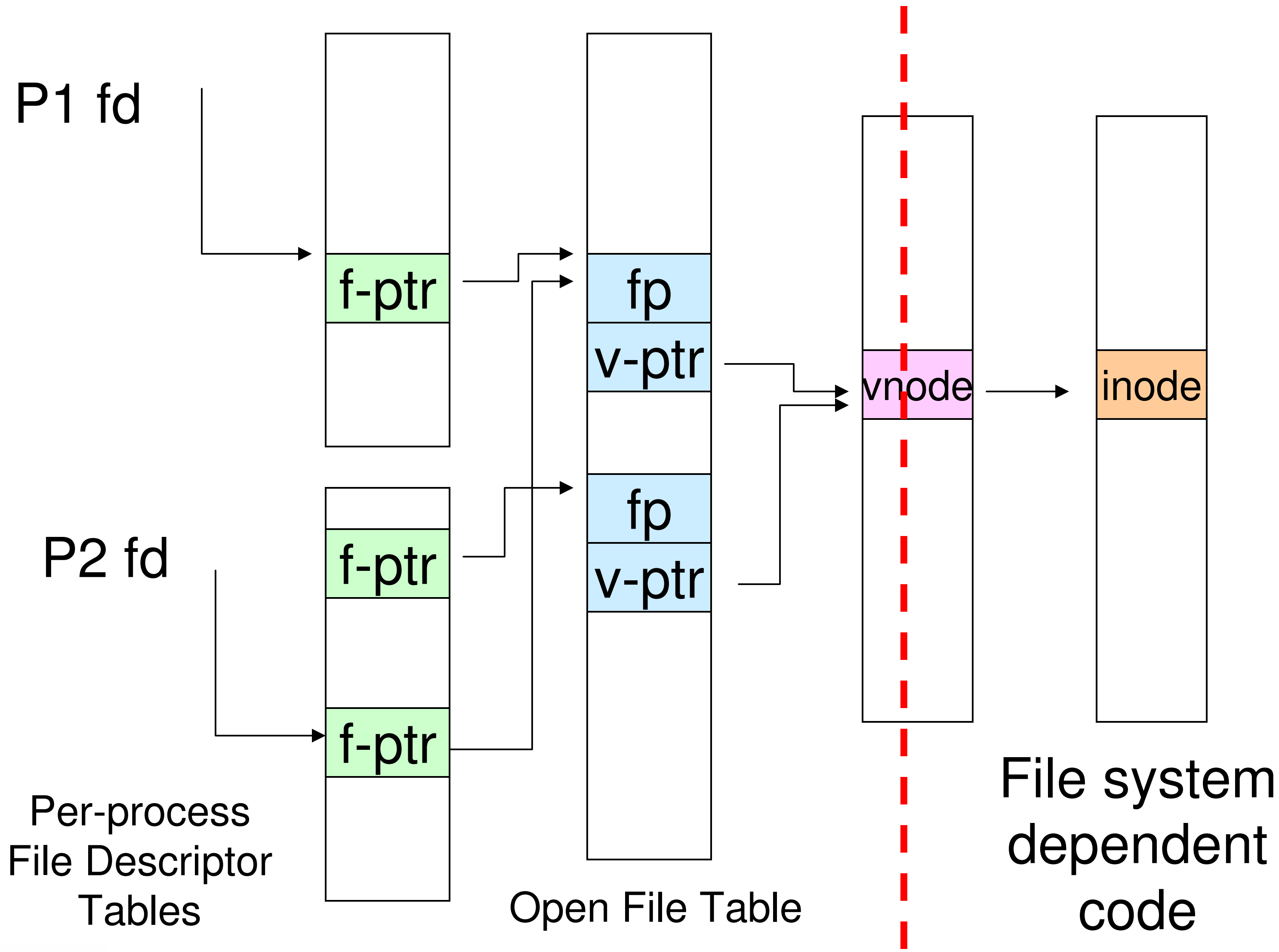
VFS architecture



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 file system independent code deals with vfs and vnodes



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,…)

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 associates with the root of the filesystem

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

Private file system specific date

THE UNIVERSITY OF NEW SOUTH WALES

Vnode

```
struct vnode {
  int vn_refcount;
  int vn_opencount;
  struct lock *vn_countlock;
  struct fs *vn_fs;
  void *vn_data;

  const struct vnode_ops *vn_ops;
};
```

Count the number of “references” to this vnode

Number of times vnode is currently open

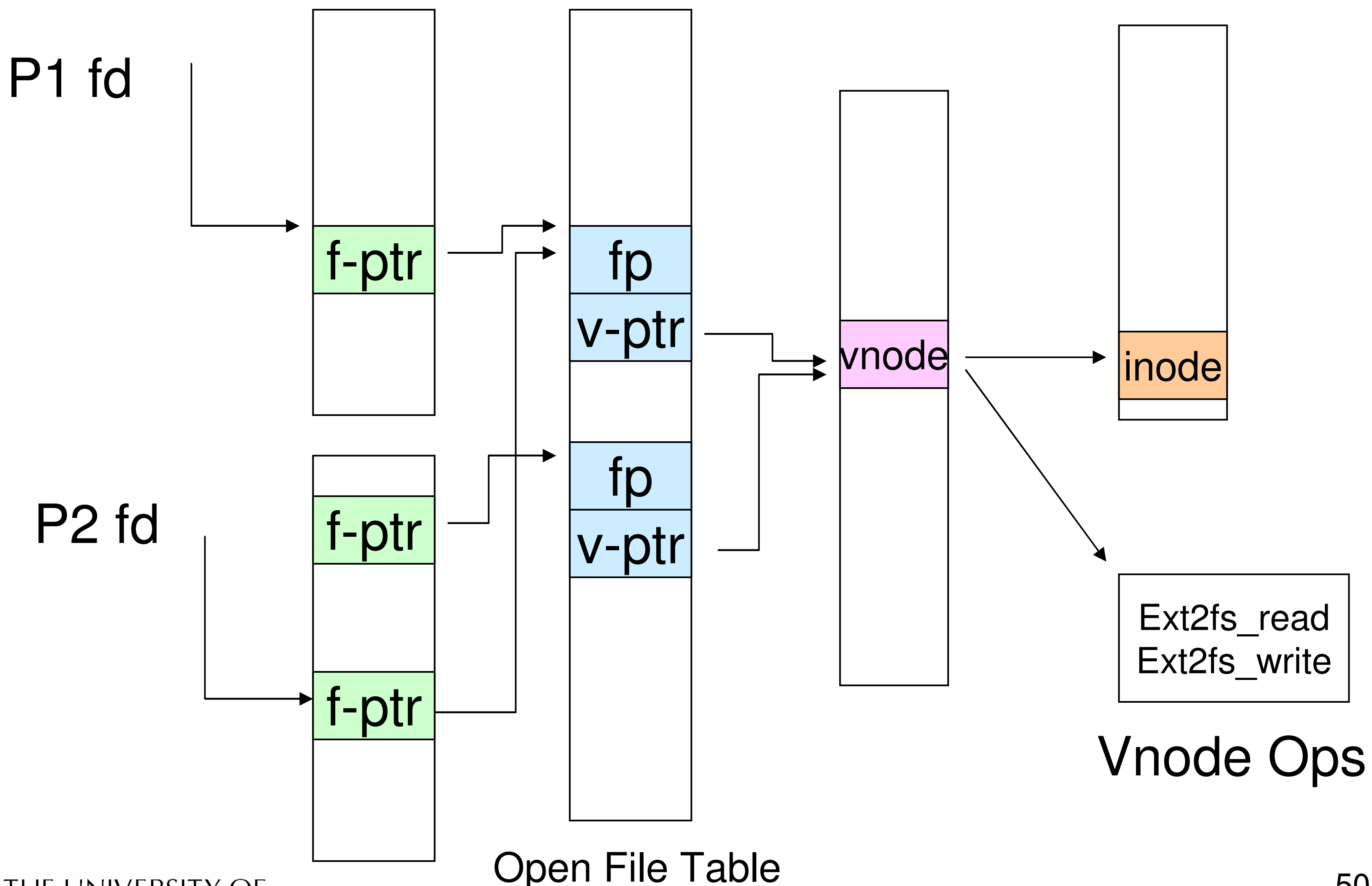
Lock for mutual exclusive access to counts

Pointer to FS containing the vnode

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

Array of pointers to functions operating on vnodes

Access Vnodes via Vnode Operations



Vnode Ops

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

   int (*vop_open)(struct vnode *object, int flags_from_open);
   int (*vop_close)(struct vnode *object);
   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_tryseek)(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);
```

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);
};
```

Vnode Ops

- Note that most operation are on vnodes. How do we operate on file names?
  - Higher level API on names that uses the internal VOP_* functions

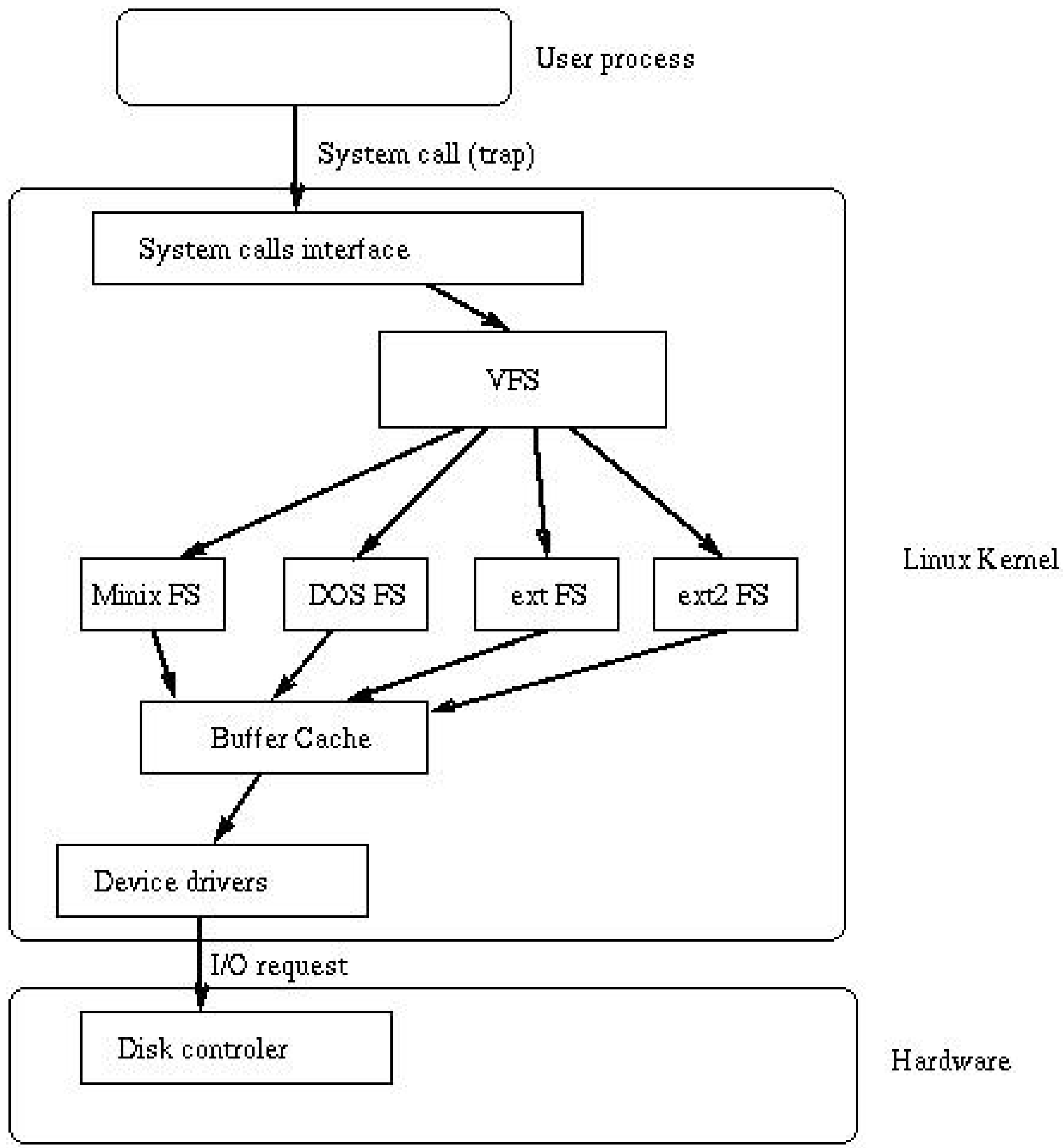
```
int vfs_open(char *path, int openflags, struct vnode **ret);
void vfs_close(struct vnode *vn);
int vfs_readlink(char *path, struct uio *data);
int vfs_symlink(const char *contents, char *path);
int vfs_mkdir(char *path);
int vfs_link(char *oldpath, char *newpath);
int vfs_remove(char *path);
int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

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_open,
   emufs_close,
   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 */
};
```

Buffer Cache

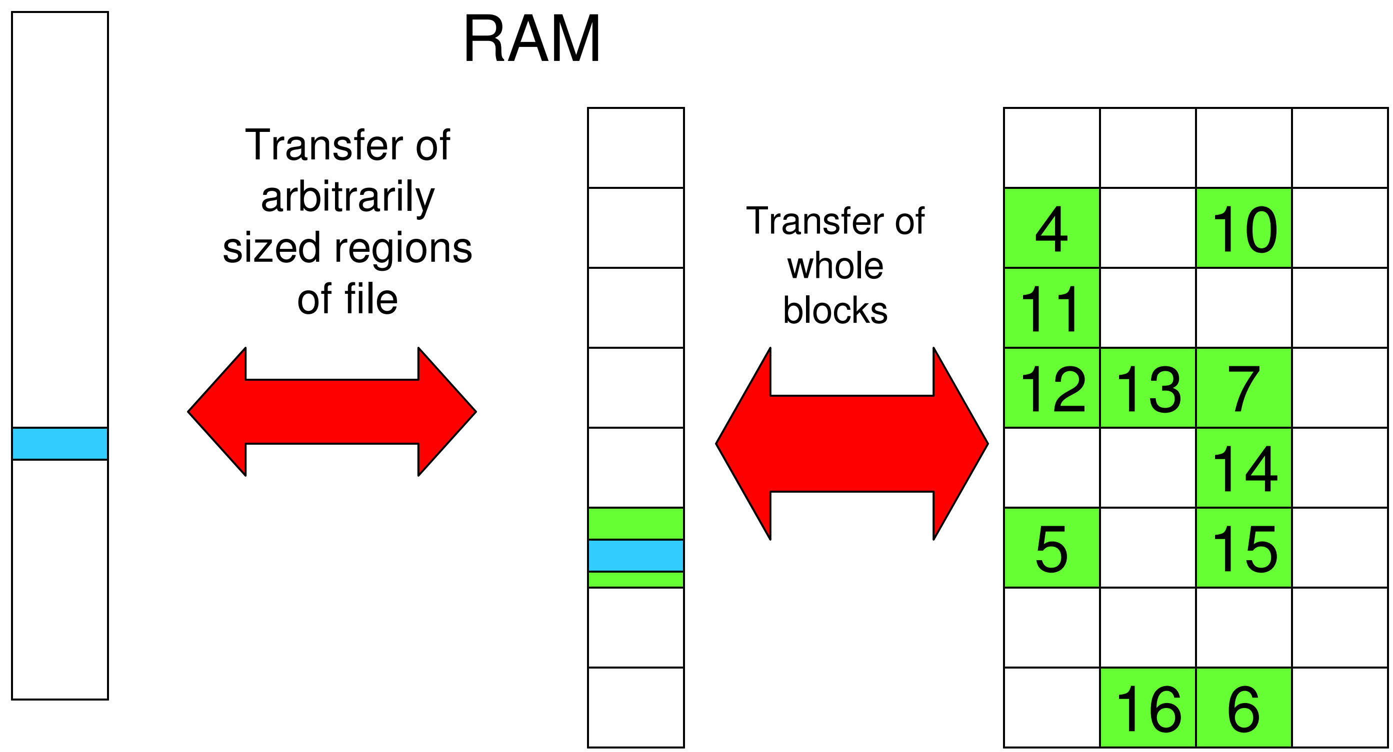


Buffer

- 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

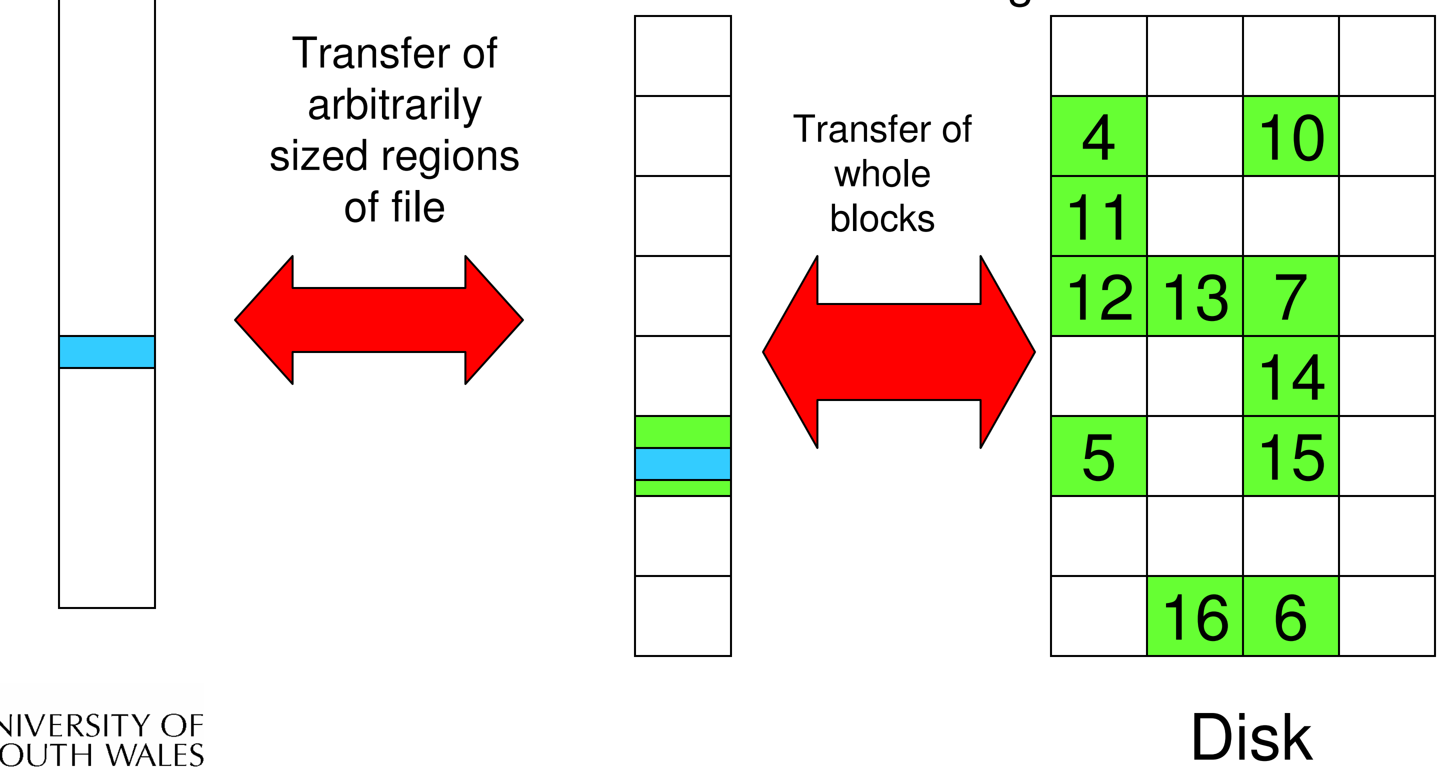
Buffering Disk Blocks

- Allow applications to work with arbitrarily sized region of a file
  - Apps can still optimise for a particular block size



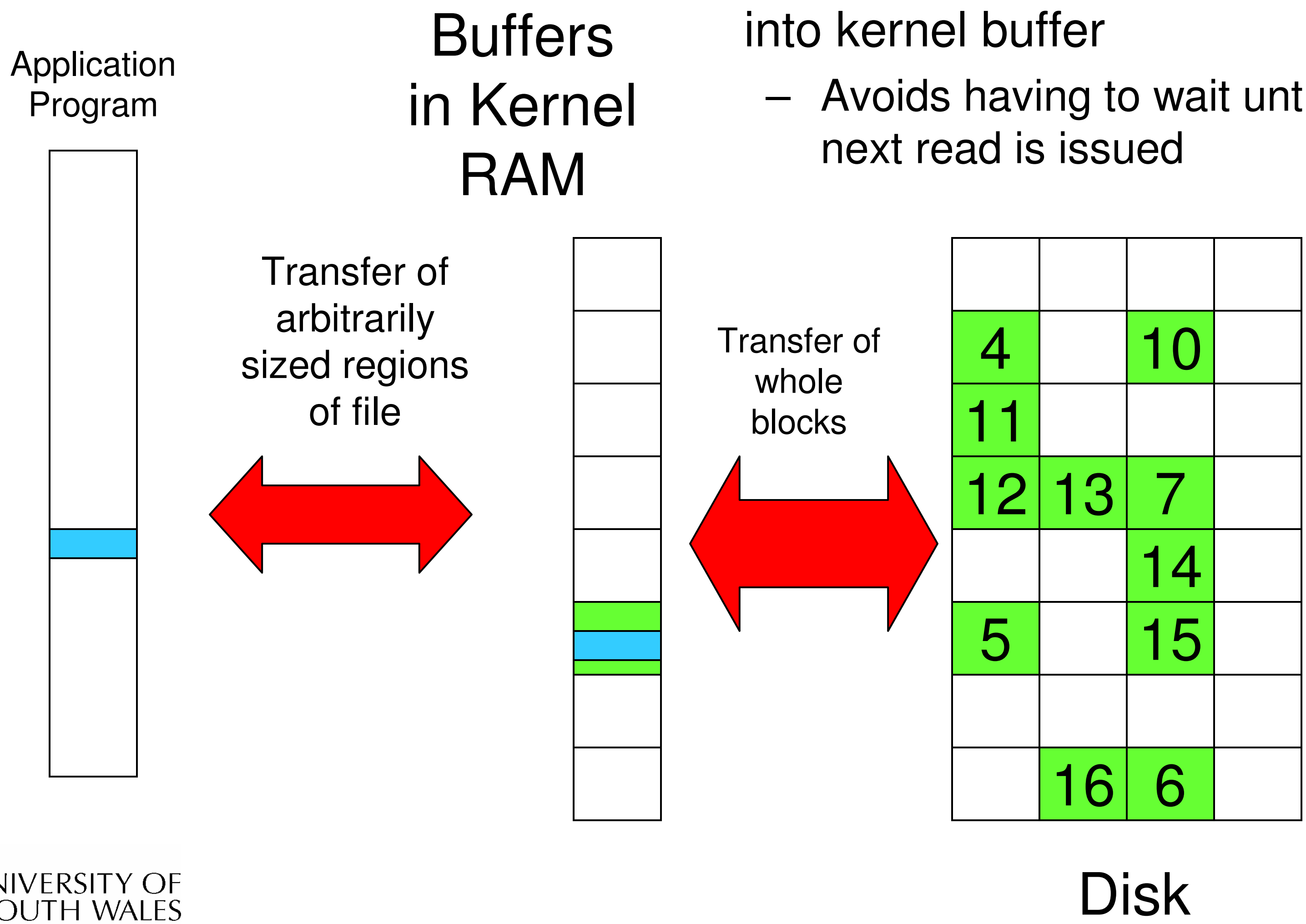
Buffering Disk Blocks

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



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

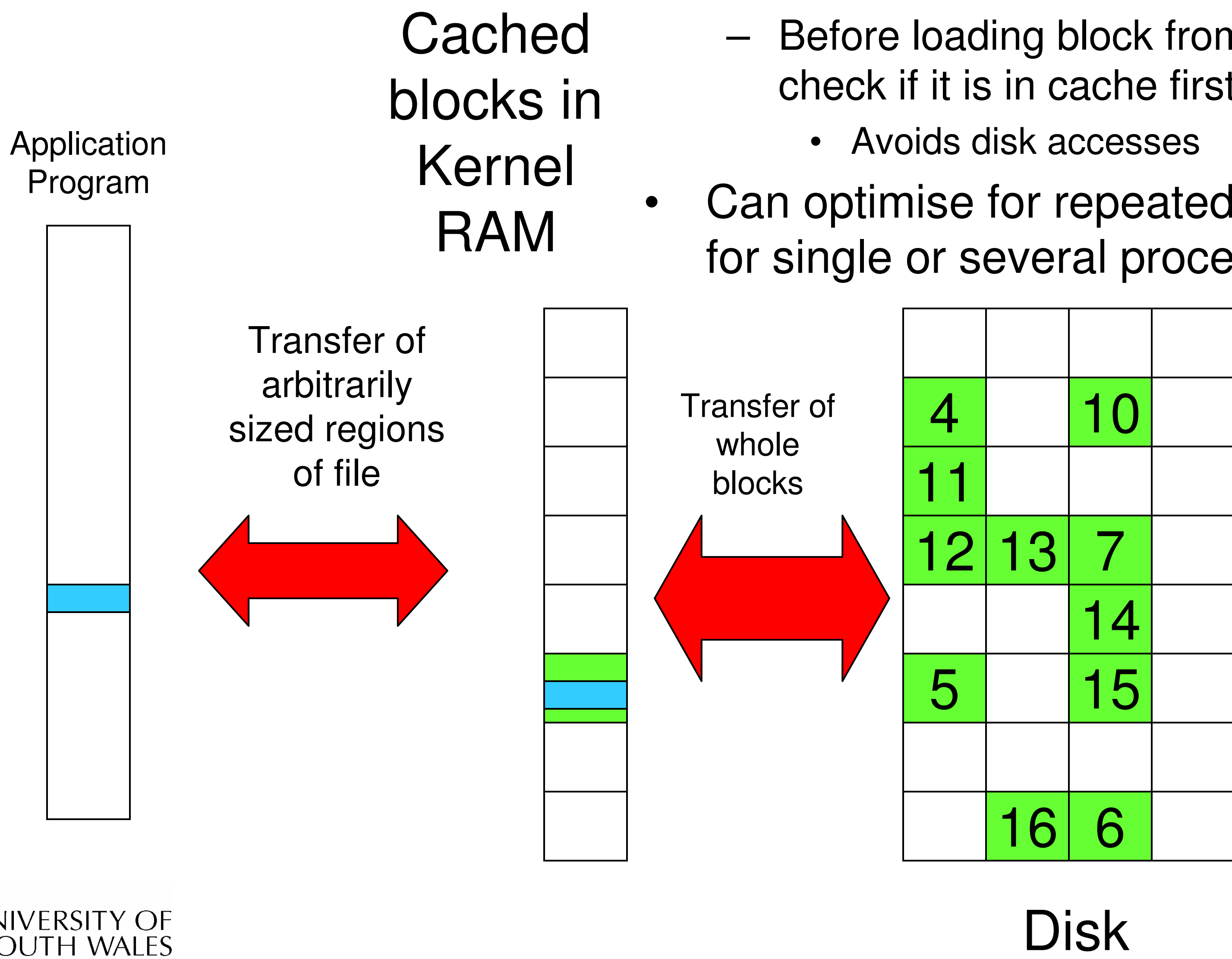


Cache

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

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



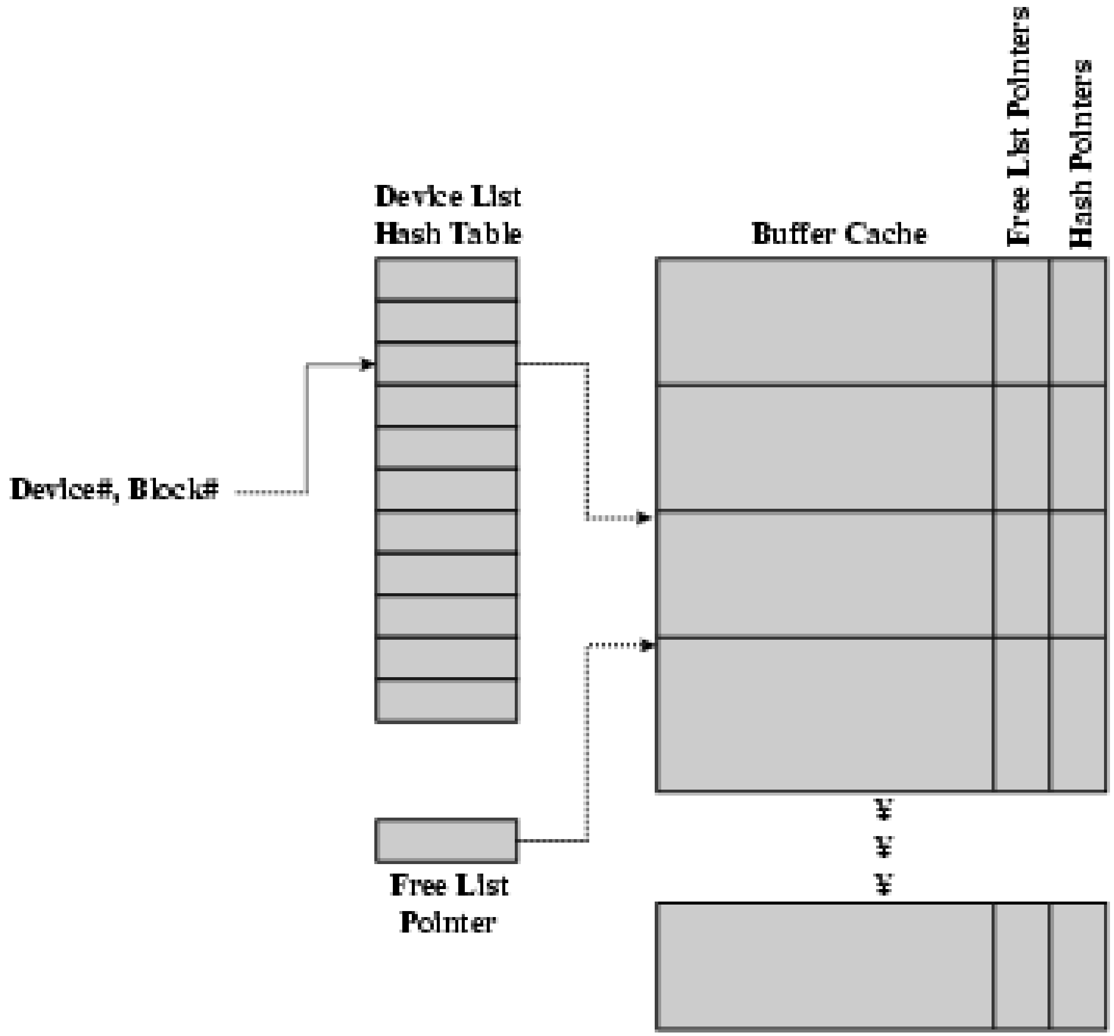
Buffering and caching are related

- Data is read into buffer; extra cache copy would be wasteful
- After use, block should be put in cache
- Future access may hit cached copy
- Cache utilises unused kernel memory space; may have to shrink

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 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
  - Similar to page replacement policy
    - Can use FIFO, Clock, LRU, etc.
    - Except disk accesses are much less frequent and take longer than memory references, so LRU is possible
    - However, is strict LRU what we want?
      - What is different between paged data in RAM and file data in RAM?

File System Consistency

- Paged data is not expected to survive crashes or power failures
- File data is expected to survive
- Strict LRU could keep critical data in memory forever if it is frequently used.

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 the 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 block are only scheduled for write back to disk periodically
    - In UNIX, flushd (*flush daemon*) flushes all modified blocks to disk every 30 seconds

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