

# File Systems

Abhijit A M  
abhijit.comp@coep.ac.in

# Introduction

- **Human end user's view of file system on a modern desktop operating system**
  - Files, directories(folders), hierarchy – acyclic graph like structure
  - Windows Vs Linux logical organization: multiple partitions (C:, D:,etc.), vs single logical namespace starting at “/”

# Introduction

## □ Secondary and Tertiary memory

- Hard disks, Pen drives, CD-ROMs, DVDs, Magnetic Tapes, Portable disks, etc. Used for storing files
- Each comes with a hardware “controller” that acts as an intermediary in the hardware/software boundary
- IDE, SATA, SCSI, SAS, etc. Protocols : Different types of cables, speeds, signaling mechanisms
- Controllers provide a block/sector based read/write

# Introduction

## □ OS and File system

- OS bridges the gap between end user and stoage hardware controller
- Provides data structure to map the logical view of end users onto disk storage
- Essentially an implementation of the acyclic graph on the sequential sector-based disk storage
  - Both in memory and on-disk
  - Provides system calls (open, read, write, etc.) to

# What we are going to learn

- The operating system interface (system calls, commands/utilities) for accessing files in a file-sysetm
- Design aspects of OS to implement the file system

# What is a file?

- **A sequence of bytes , with**
  - A name
  - Permissions
  - Owner
  - Timestamps,
  - Etc.
- **Types: Text files, binary files**

Text: All bytes are human readable

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

# What is a file?

- **The sequence of bytes can be interpreted to be**
  - Just a sequence of bytes
    - E.g. a text file
  - Sequence of records/structures
    - E.g. a file of student records
  - A complexly organized, collection of records and bytes
    - E.g. a “ODT” or “DOCX” file

# File attributes

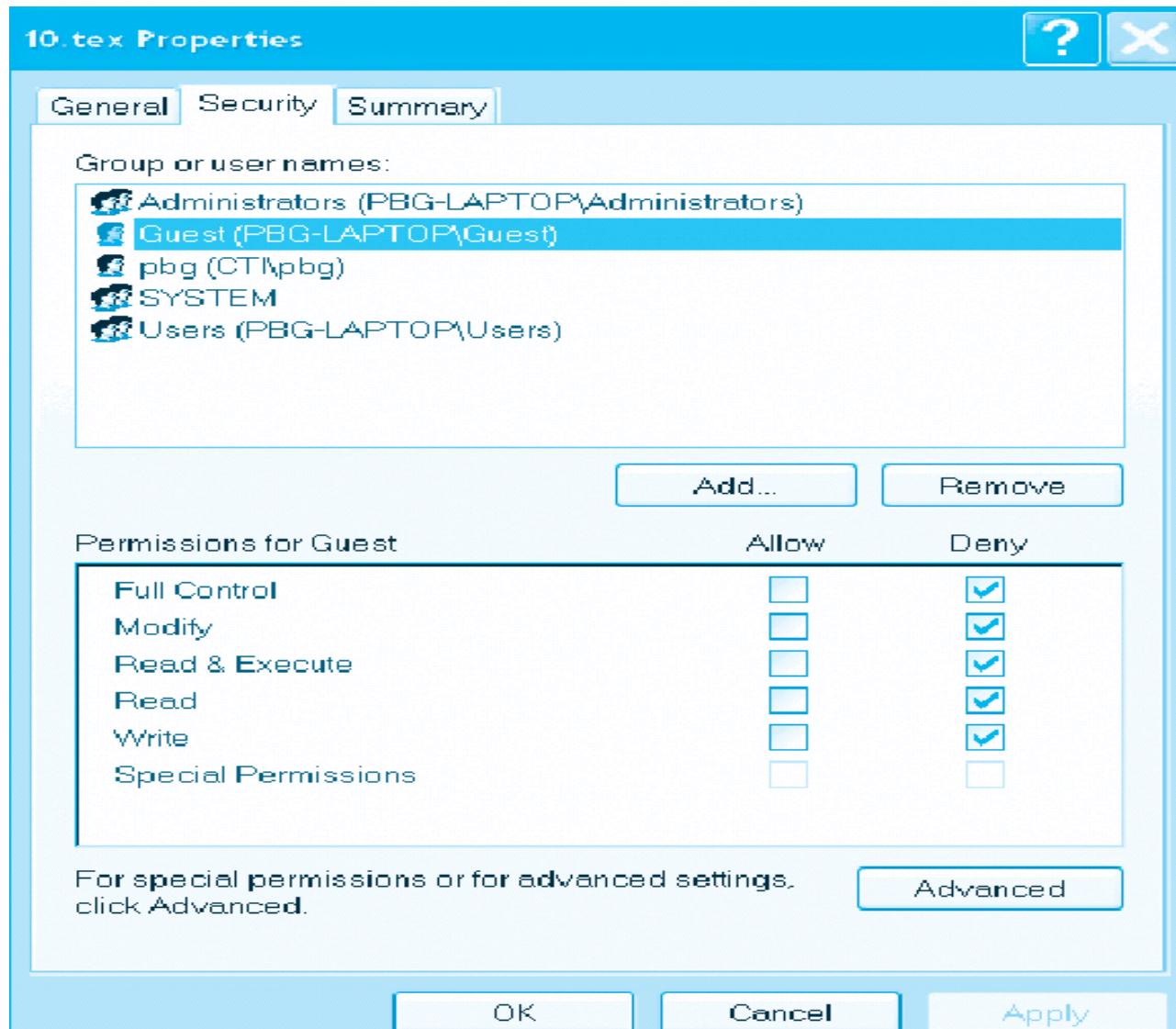
- **Run**
- **on Linux**
- **To see file listing with different attributes**
- **Different OSes and file-systems provide different sets of file attributes**
  - Some attributes are common to most, while some

# File protection attributes

- **File owner/creator should be able to control:**
  - what can be done
  - by whom
- **Types of access**
  - Read
  - Write
- **Linux file permissions**
  - For owner, group and others
  - Read, write and execute
  - Total 9 permissions

# A Sample UNIX Directory Listing

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/



# Windows XP Access List

# Access methods

- OS system calls may provide two types of access to files
  - Sequential Access
    - read next
    - write next
    - reset
    - no read after last write
  - Direct Access
  - $n$  = relative block number
    - read  $n$
    - write  $n$
    - position to  $n$
    - read next
    - write next
    - rewrite  $n$

# Device Drivers

- **Hardware manufacturers provide “hardware controllers” for their devices**
- **Hardware controllers can operate the hardware, as instructed**
- **Hardware controllers are instructed by writing to particular I/O ports using CPU’s machine instructions**
  - This bridges the hardware-software gap

# Disk device driver

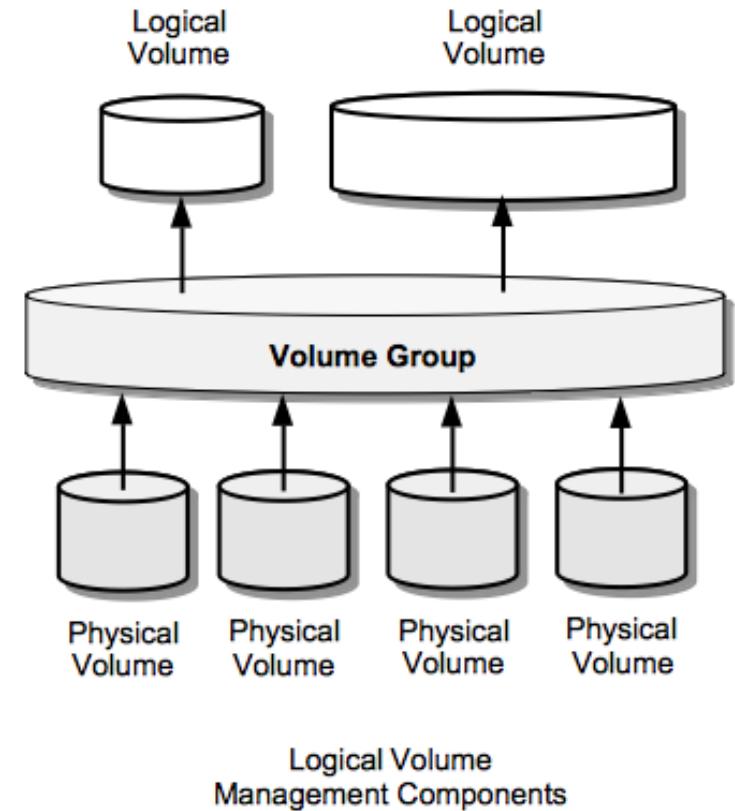
- OS views the disk as a logical sequence of blocks
  - OS's assumed block size may be > sector size
- OS Talks to disk controller
- Helps the OS Convert it's view of “logical block” of the disk, into physical sector numbers
- Acts as a translator between rest of OS and

# OS's job now

- To implement the logical view of file system as seen by end user
- Using the logical block-based view offered by the device driver

# Volume Managers

- Special type of kernel device drives, which reside on top of disk device drivers
- Provide a more abstract view of the underlying hardware
- E.g. Can combine two

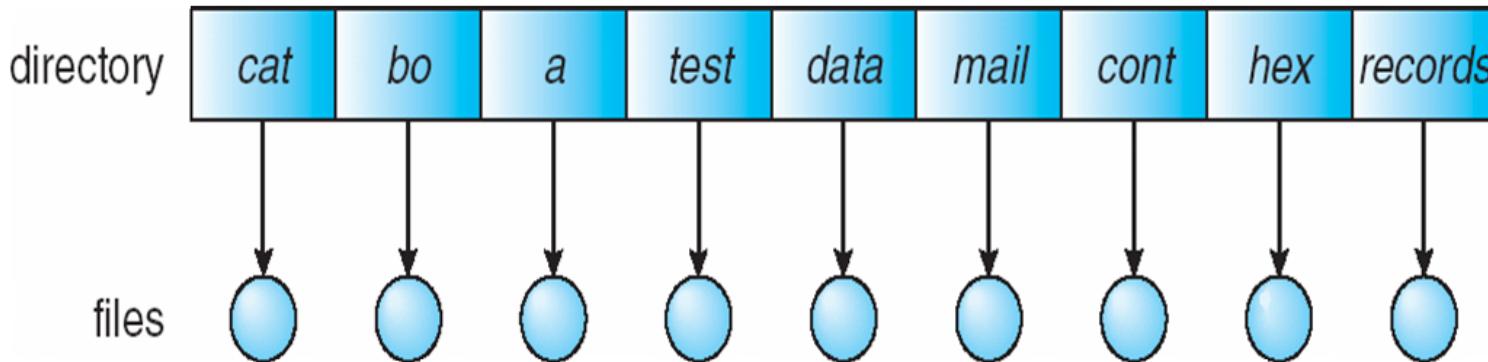


# Formatting

- **Physical hard disk divided into partitions**
  - Partitions also known as minidisks, slices
- **A raw disk partition is accessible using device driver – but no block contains any data !**
  - Like an un-initialized array, or sectors/blocks
- **Formatting**
  - Creating an initialized data structure on the

# Different types of “layouts”

## Single level directory

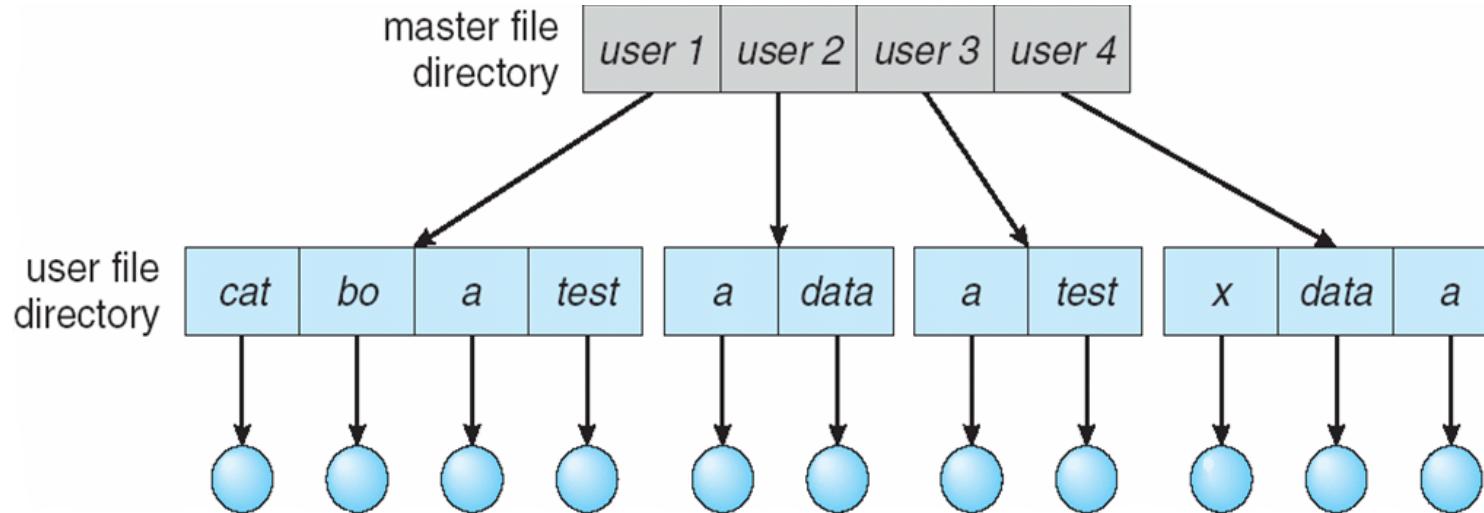


Naming problem

Grouping problem

# Different types of “layouts”

## Two level directory



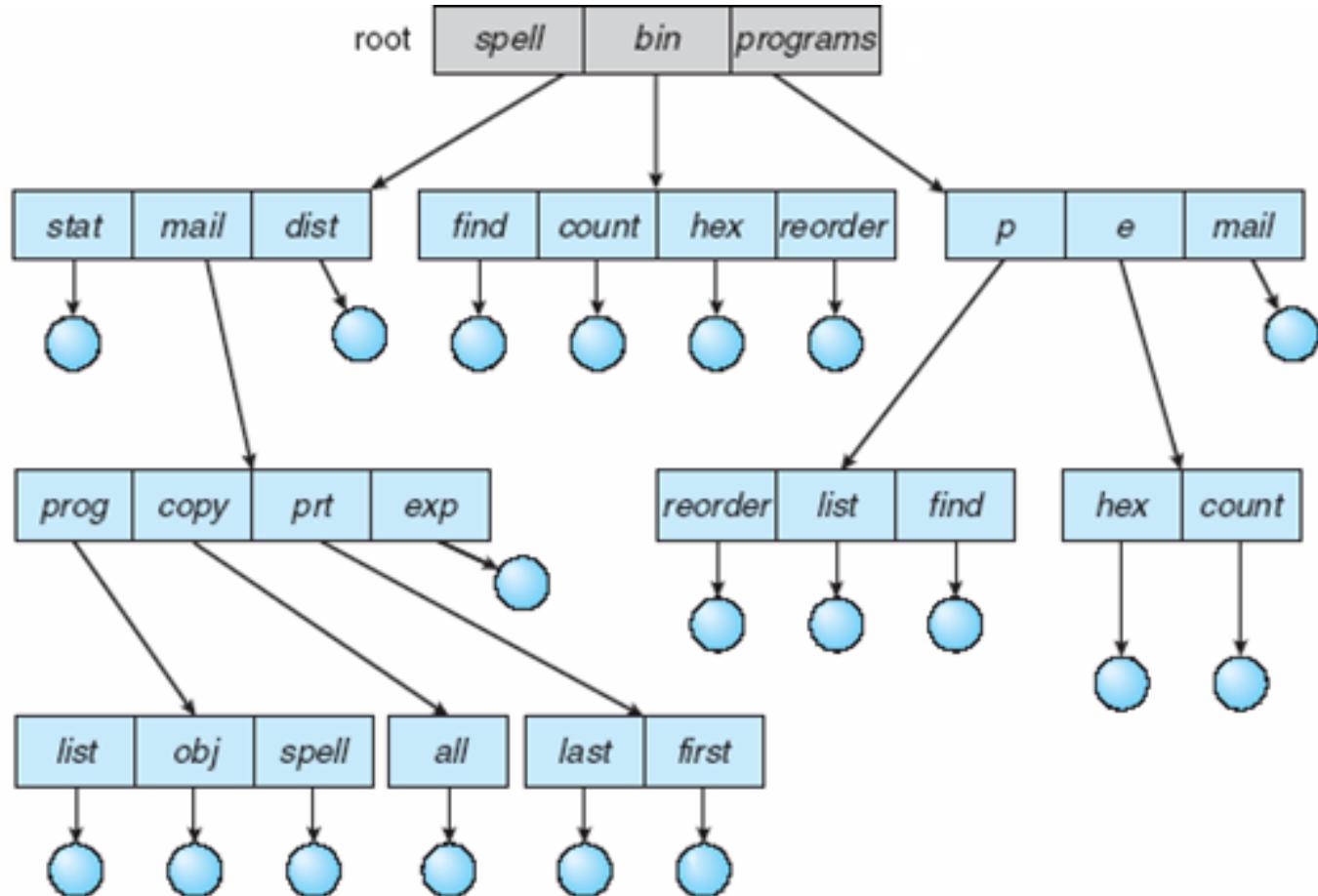
Path name

Can have the same file name for different user

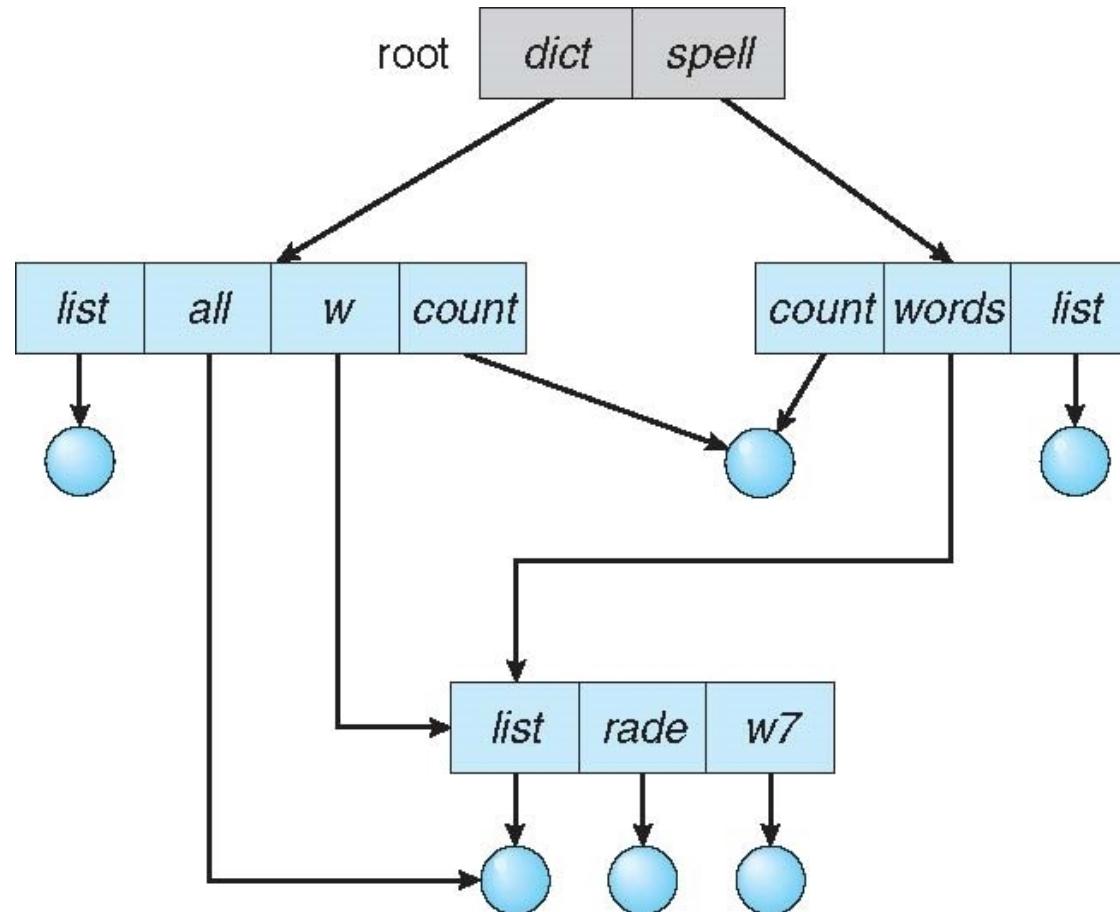
Efficient searching

No grouping capability

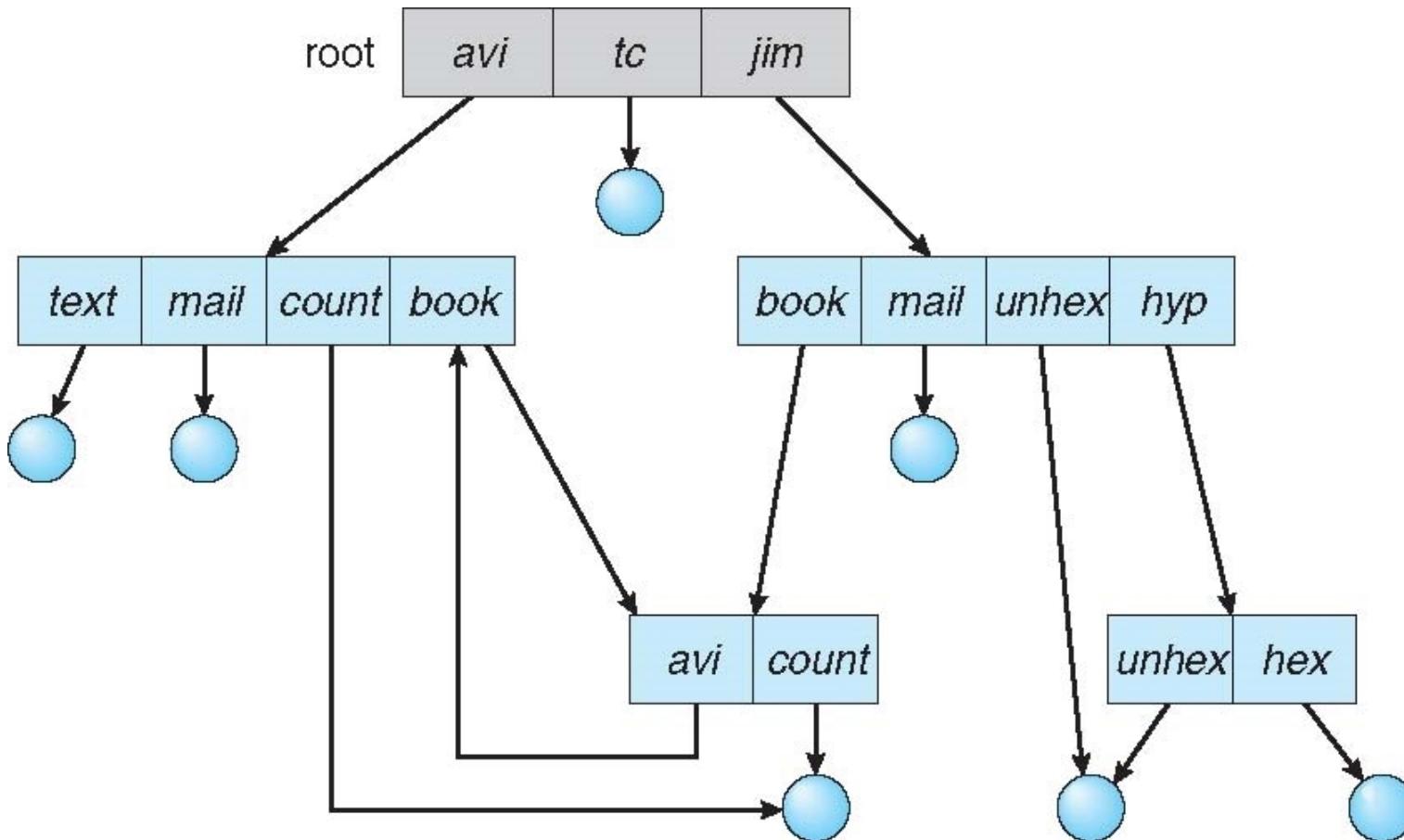
# Tree Structured directories



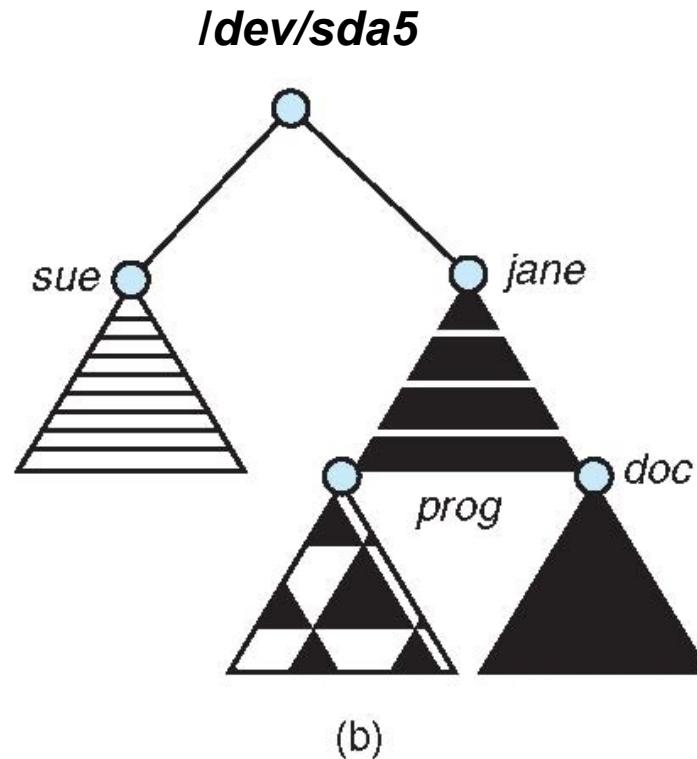
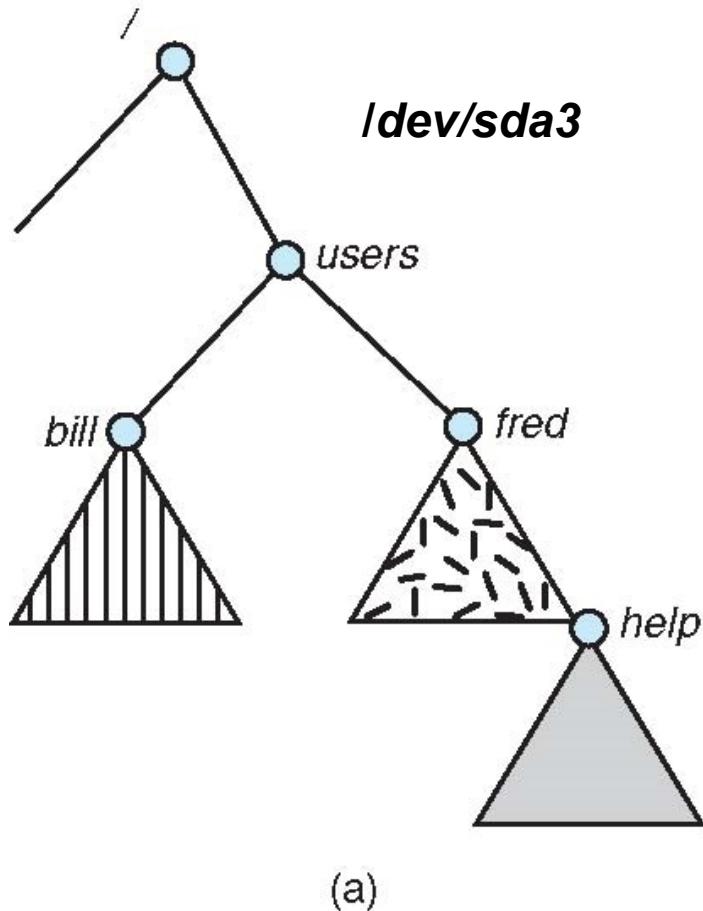
# Acyclic Graph Directories



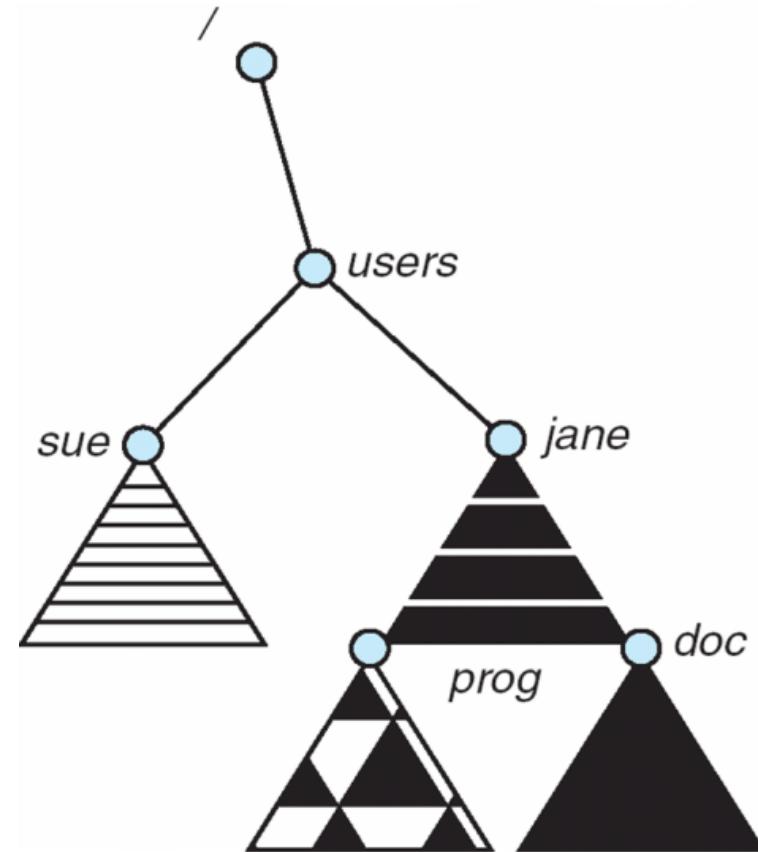
# General Graph directory



# Mounting of a file system: before



# Mounting of a file system: after



```
$sudo mount /dev/sda5 /users
```

# Remote mounting: NFS

- **Network file system**
- **\$ sudo mount 10.2.1.2:/x/y /a/b**
  - The /x/y partition on 10.2.1.2 will be made available under the folder /a/b on this computer

# File sharing semantics

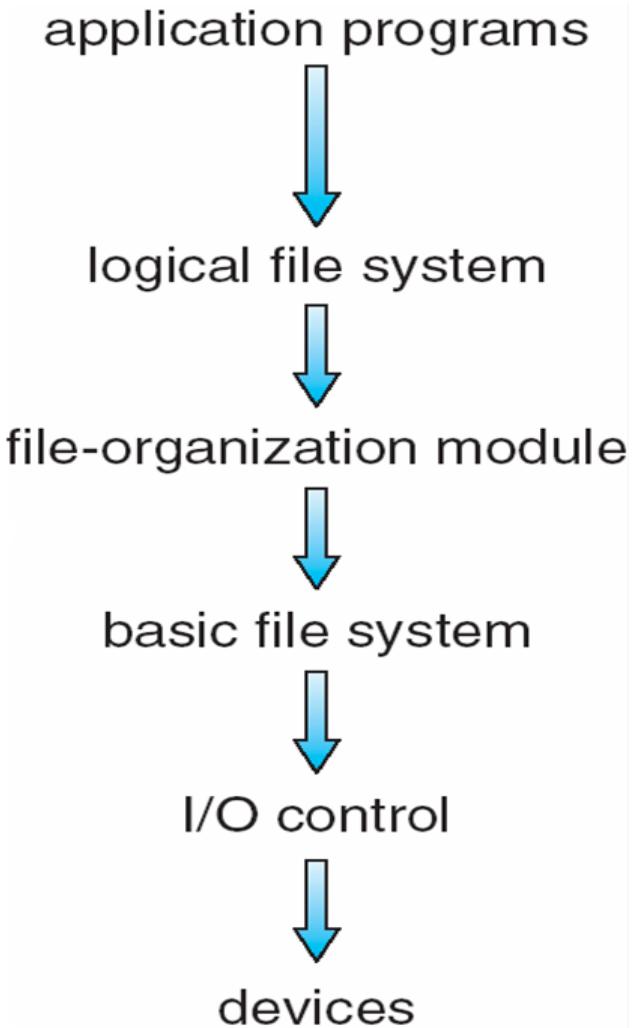
- **Consistency semantics specify how multiple users are to access a shared file simultaneously**
- **Unix file system (UFS) implements:**
  - Writes to an open file visible immediately to other users of the same open file
  - One mode of sharing file pointer to allow multiple users to read and write concurrently

# **Implementing file systems**

# **File system on disk**

- **Disk I/O in terms of sectors (512 bytes)**
- **File system: implementation of acyclic graph using the linear sequence of sectors**
- **Device driver: available to rest of the OS code to access disk using a block number**

# File system implementation: layering



# File system: Layering

- **Device drivers manage I/O devices at the I/O control layer**
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level data.
- **File organization module understands files, logical address, and physical blocks**
  - Translates logical block # to physical block #
  - Manages free space.

## Application programs

```
int main() {  
    char buf[128]; int count;  
    fd = open(...);  
    read(fd, buf, count);  
}
```

-----

## OS

### Logical file system:

```
sys_read(int fd, char *buf, int count) {  
    file *fp = currproc->fdarray[fd];  
    file_read(fp, ...);  
}
```

### File organization module:

```
file_read(file *fp, char *buf, int count) {  
    offset = fp->current_offset;  
    translate offset into blockno;  
    basic_read(blockno, buf, count);
```

### Basic File system:

```
basic_read(int blockno, char *buf, ...) {  
    os_buffer *bp;  
    sectorno = calculation on blockno;  
    disk_driver_read(sectorno, bp );  
    move-process-to-wait-queue;  
    copy-data-to-user-buffer(bp, buf);  
}
```

### IO Control, Device driver:

```
disk_driver_read(sectorno) {  
    issue instructions to disk controller (often assembly)  
    to read sectorno into specific location;  
}
```

*XV6 does it slightly differently, but following the same steps*

# Layering advantages

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32)

# **File system implementation: Different problems to be solved**

- What to do at boot time ?**
- How to store directories and files on the partition ?**
  - Complex problem. Hierarchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)
- How to manage list of free sectors/blocks?**
- How to store the summary information**

# File system implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures. Let's see some of the important ones.
  - Boot control block contains info needed by system to boot OS from that volume
    - Not always needed. Needed if volume contains OS, usually first block of volume
    - Volume control block (superblock, master file table)

# A typical file control block (inode)

file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

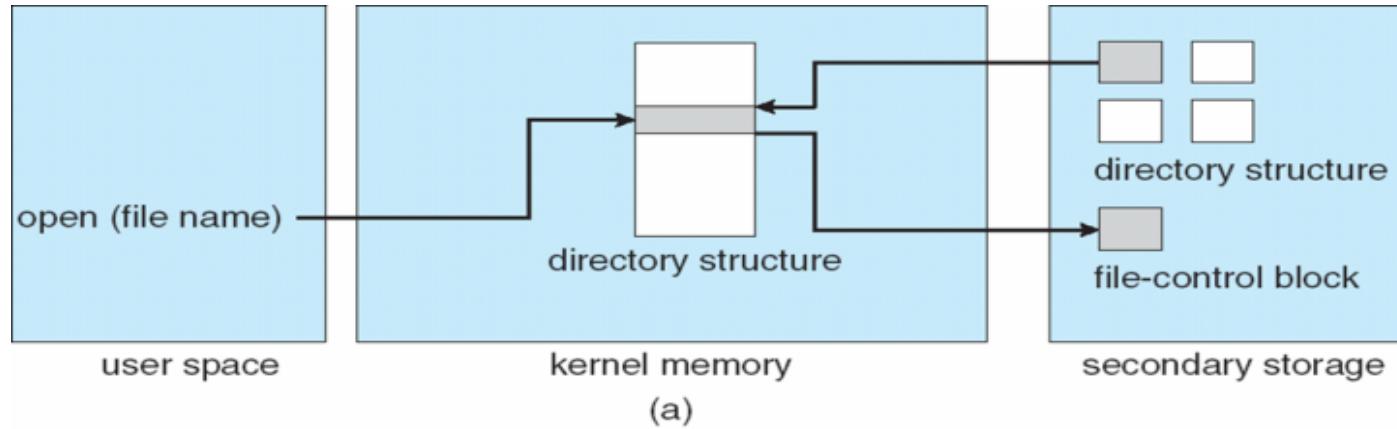
Why does it NOT contain the

Name of the file ?

# In memory data structures

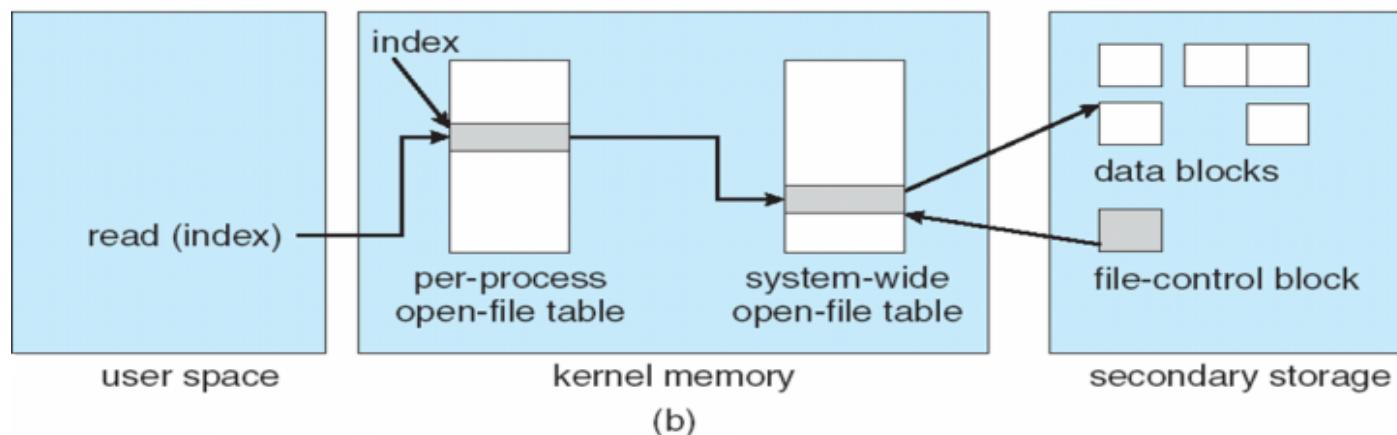
- **Mount table**
  - storing file system mounts, mount points, file system types
- **See next slide for “file” realated data structures**
- **Buffers**
  - hold data blocks from secondary storage

# In memory data structures: for open,read,write, ...



Open returns a file handle for the file-control block

Data from read eventually comes from secondary storage



# At boot time

- **Root partition**
  - Contains the file system hosting OS
  - “mounted” at boot time – contains “/”
    - Normally can’t be unmounted!
- **Check all other partitions**
  - Specified in `/etc/fstab` on Linux
  - Check if the data structure on them is consistent
    - Consistent != perfect/accurate/complete

# Directory Implementation

## □ Problem

- Directory contains files and/or subdirectories
- Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
- Directory needs to give location of each file on disk

# Directory Implementation

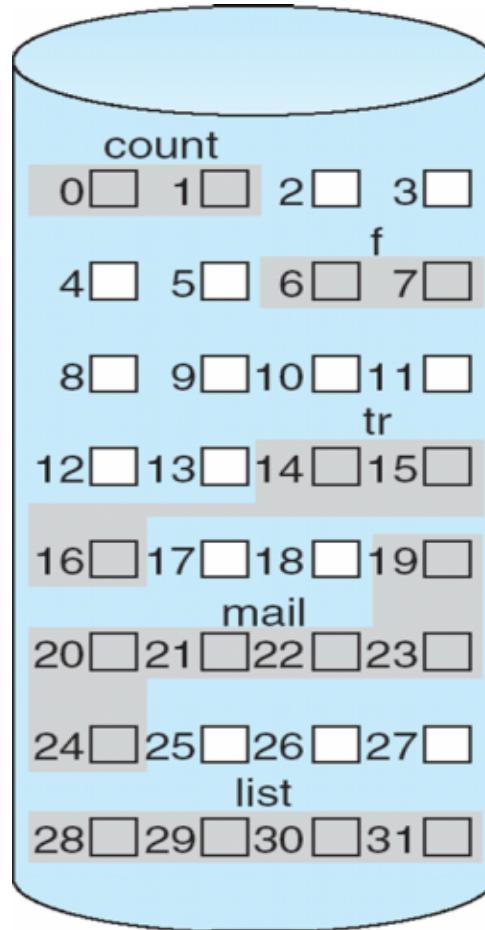
- **Linear list of file names with pointer to the data blocks**
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.

**Hash Table - Linear list with hash data**

# Disk space allocation for files

- **File contain data and need disk blocks/sectors for storing it**
- **File system layer does the allocation of blocks on disk to files**
- **Files need to**
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

# Contiguous Allocation of Disk Space



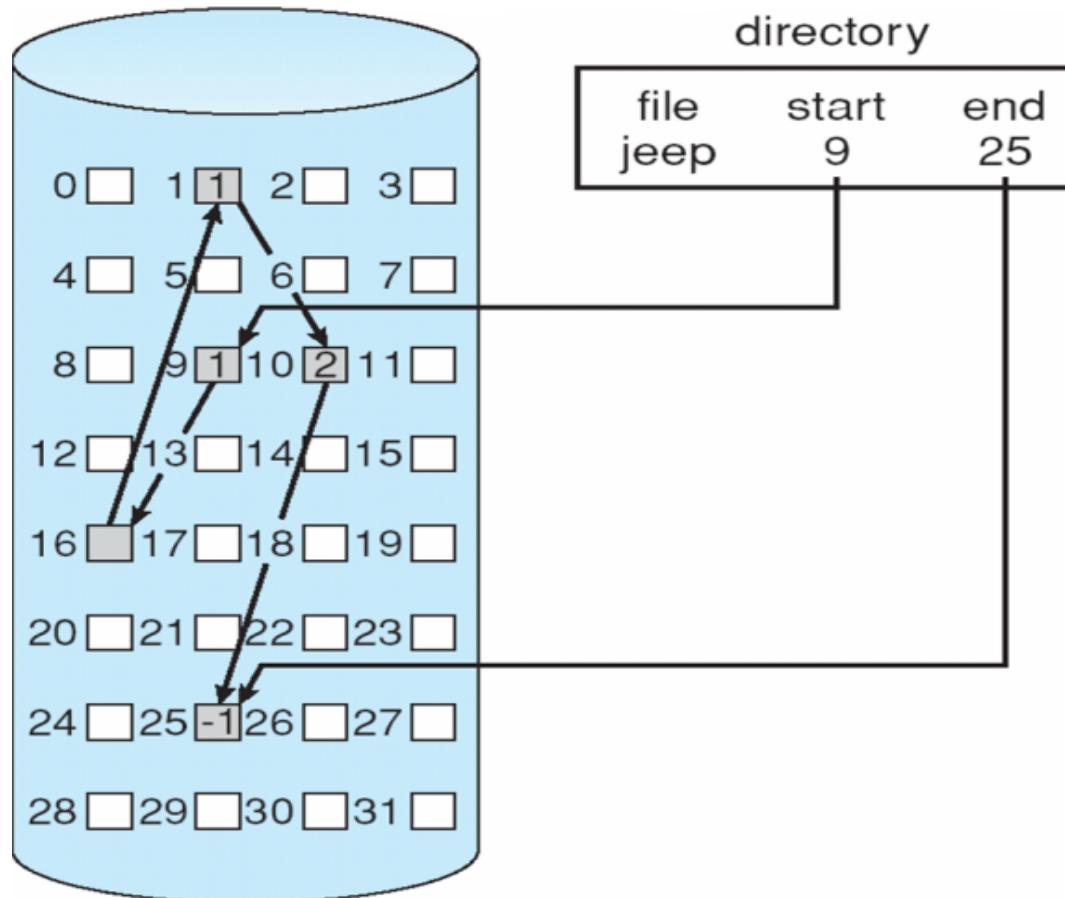
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

# Contiguous allocation

- **Each file occupies set of contiguous blocks**
- **Best performance in most cases**
- **Simple – only starting location (block #) and length (number of blocks) are required**
- **Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line**

# Linked allocation of blocks to a file



# Linked allocation of blocks to a file

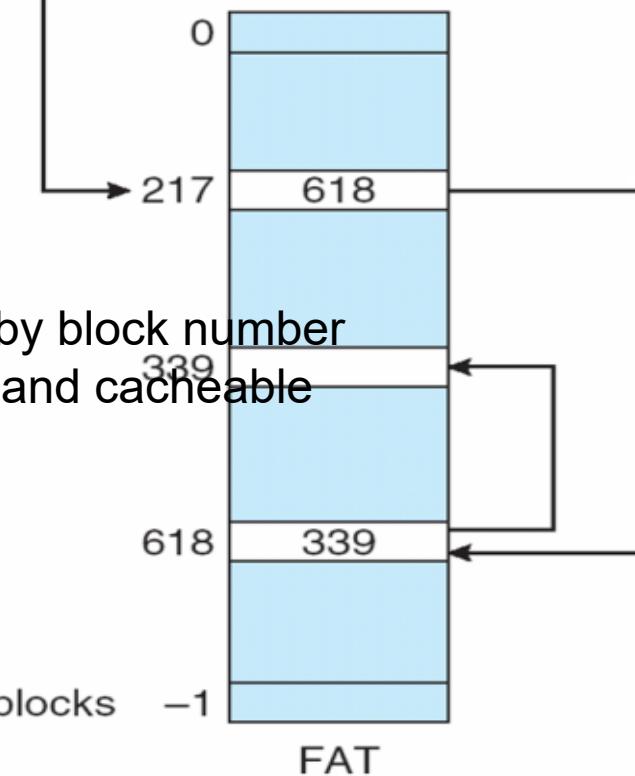
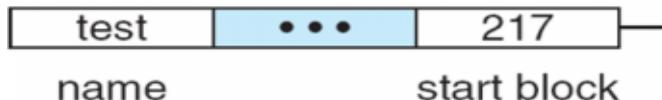
## □ **Linked allocation**

- Each file a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block (i.e. data + pointer to

- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem

# FAT: File Allocation Table

directory entry



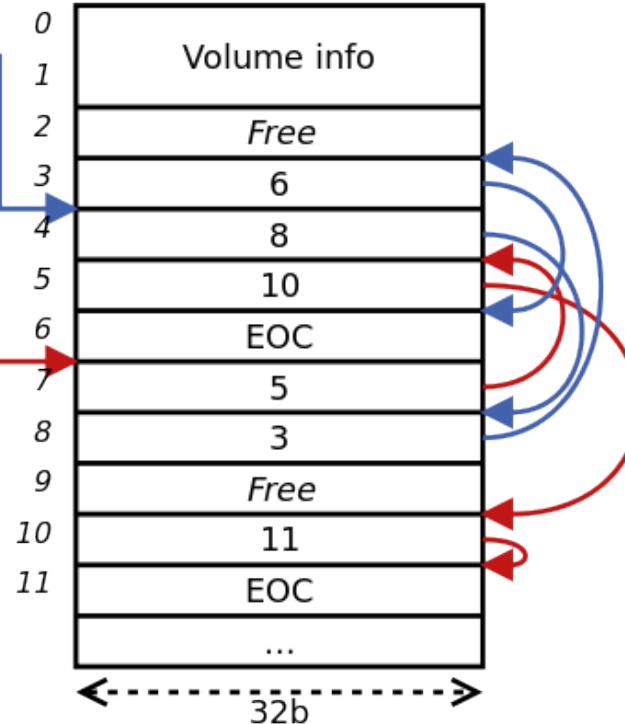
- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

# FAT: File Allocation Table

Directory table entry (32B)

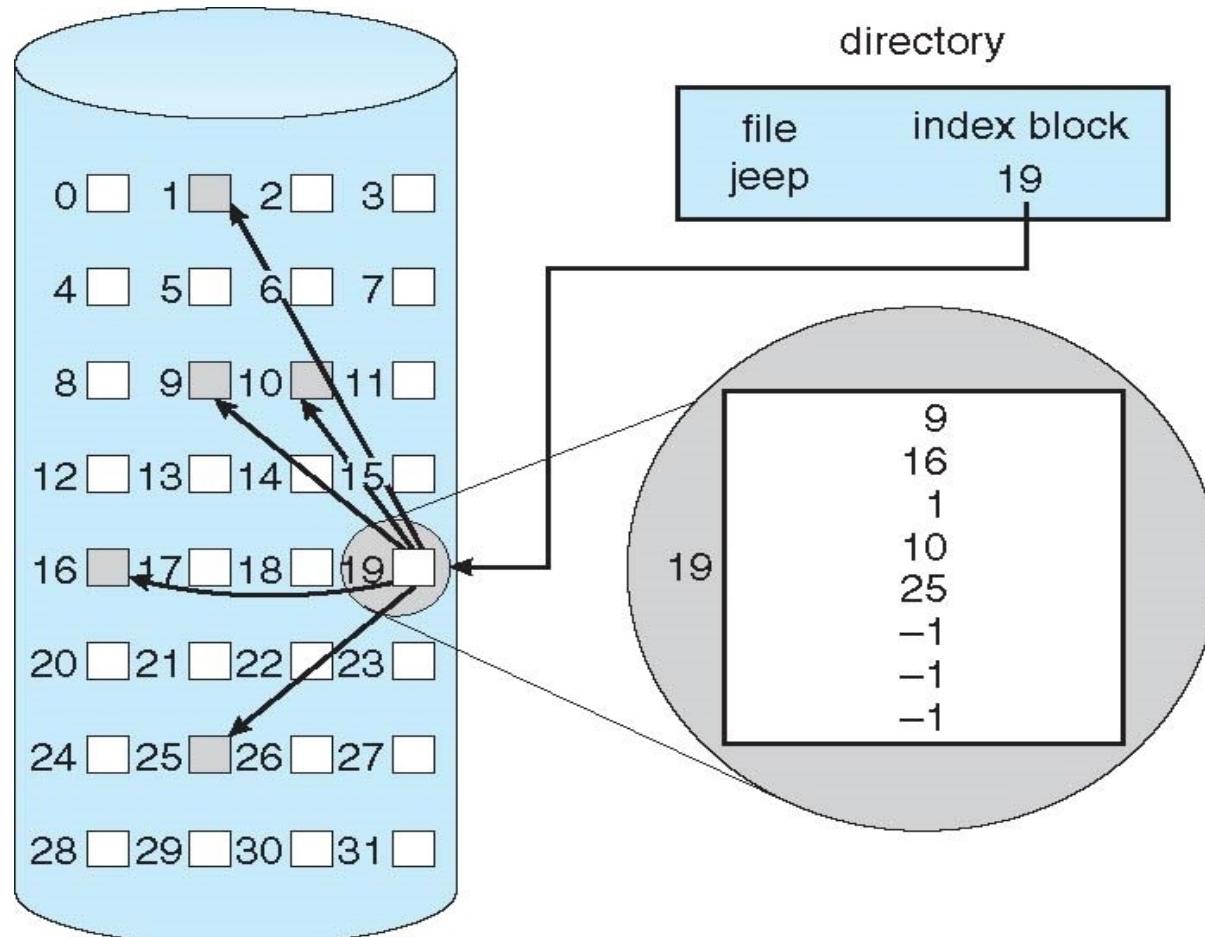
Filename (8B)
Extension (3B)
Attributes (1B)
Reserved (1B)
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)

File allocation table



Variants: FAT8,  
FAT12, FAT16,  
FAT32, VFAT, ...

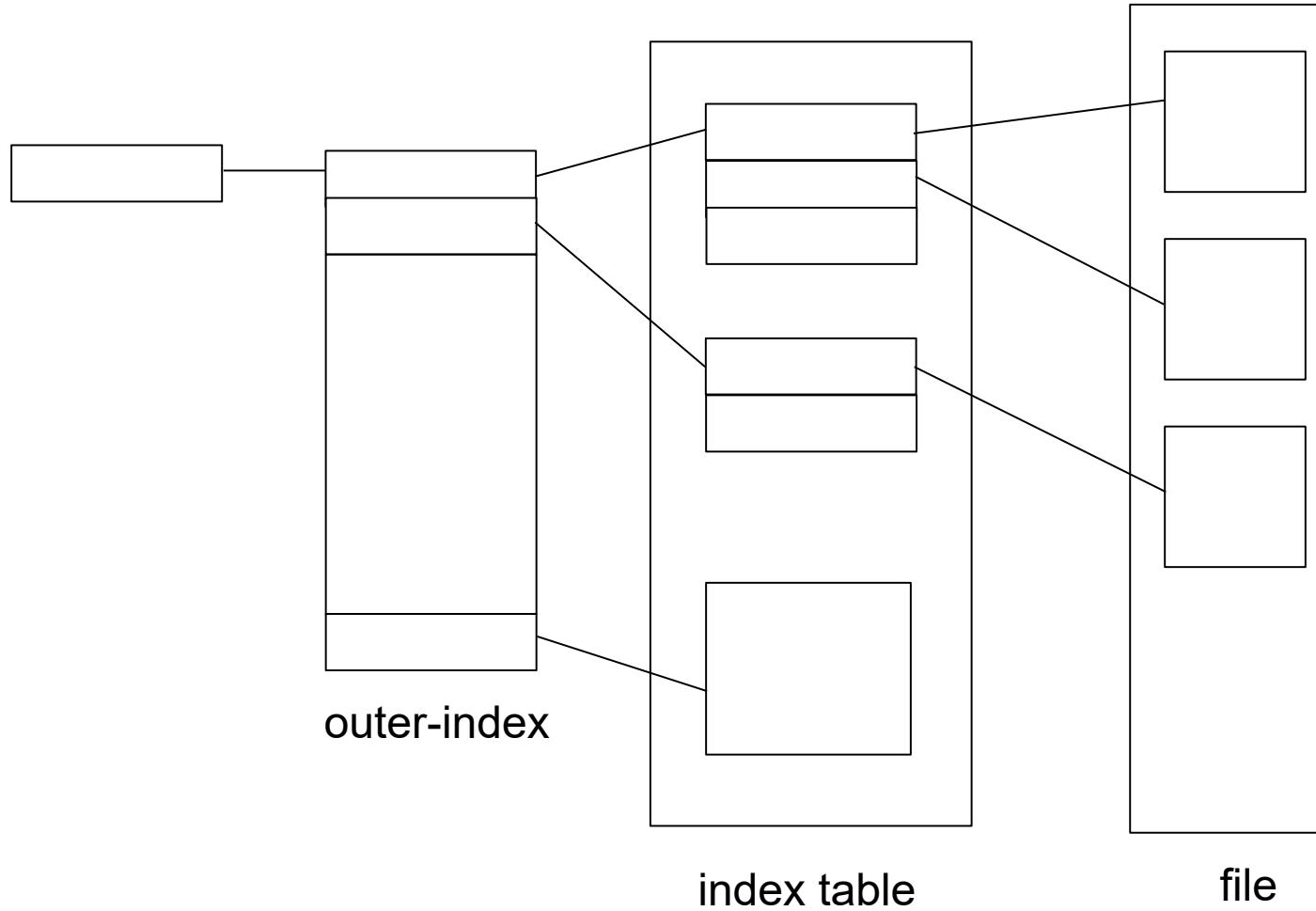
# Indexed allocation



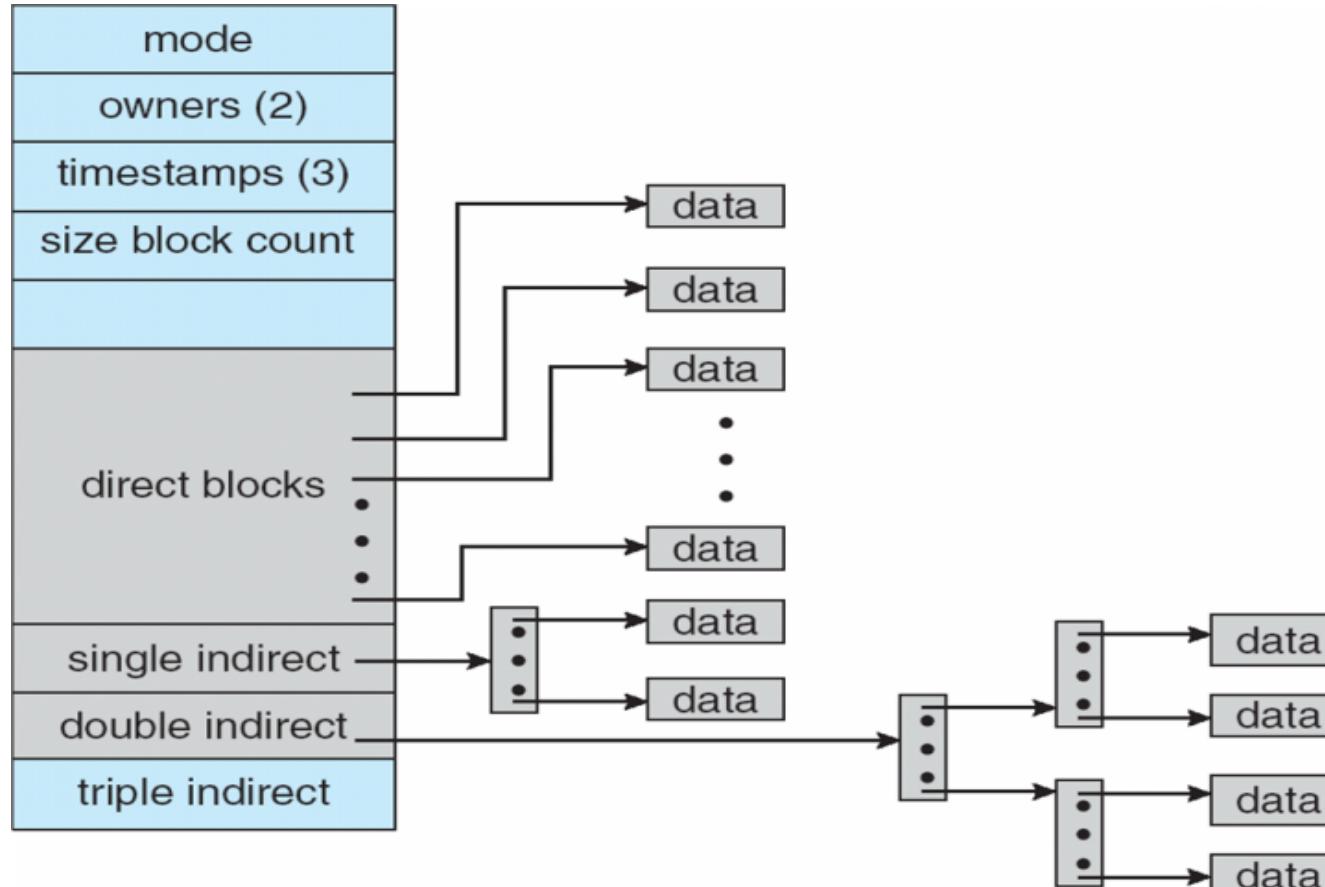
# Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index

# Multi level indexing



# Unix UFS: combined scheme for block allocation



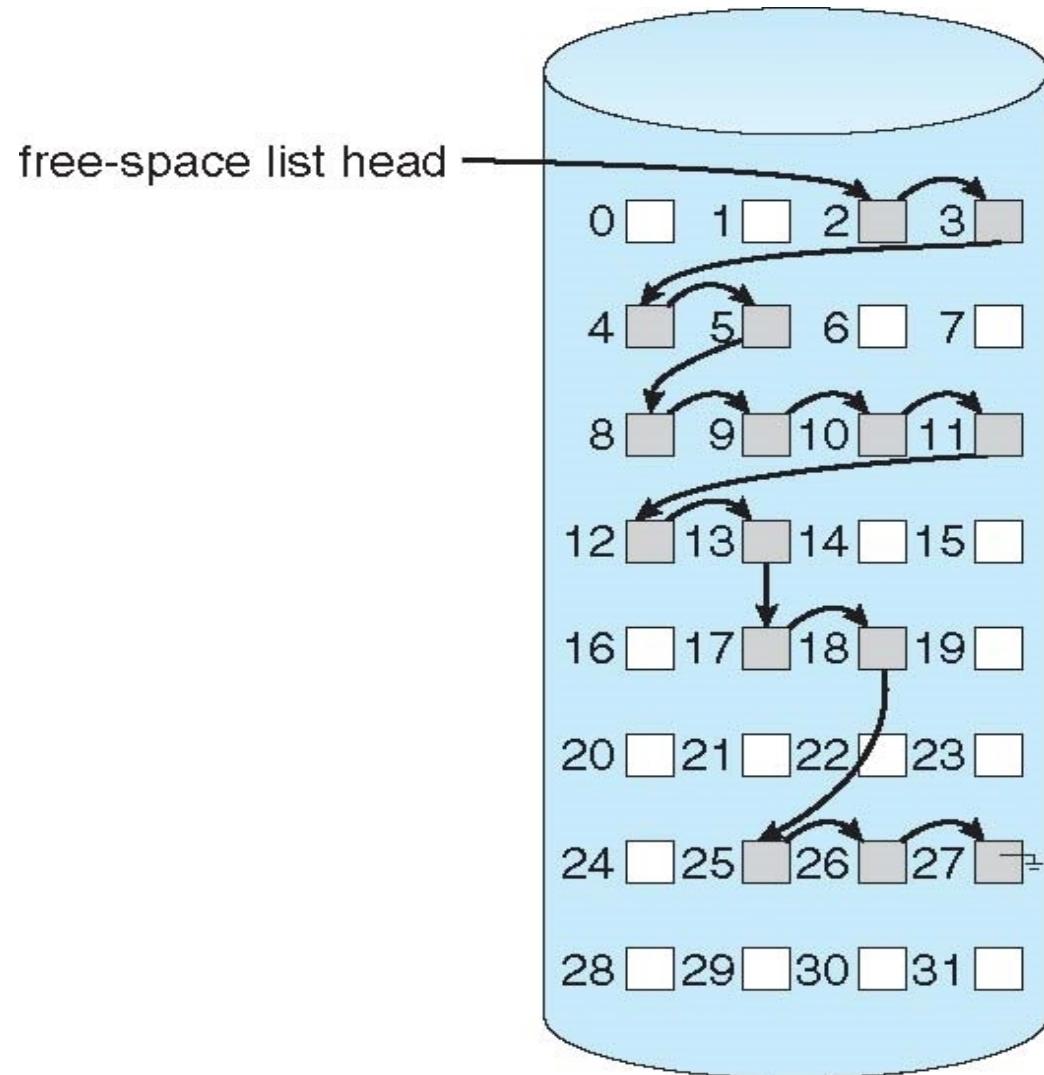
# Free Space Management

- File system maintains free-space list to track available blocks/clusters
  - Bit vector or bit map (n blocks)
  - Or Linked list

# Free Space Management: bit vector

- **Each block is represented by 1 bit.**
- **If the block is free, the bit is 1; if the block is allocated, the bit is 0.**
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 001111001111110001100000011100000 ...

# Free Space Management: Linked list (not in memory, on disk!)

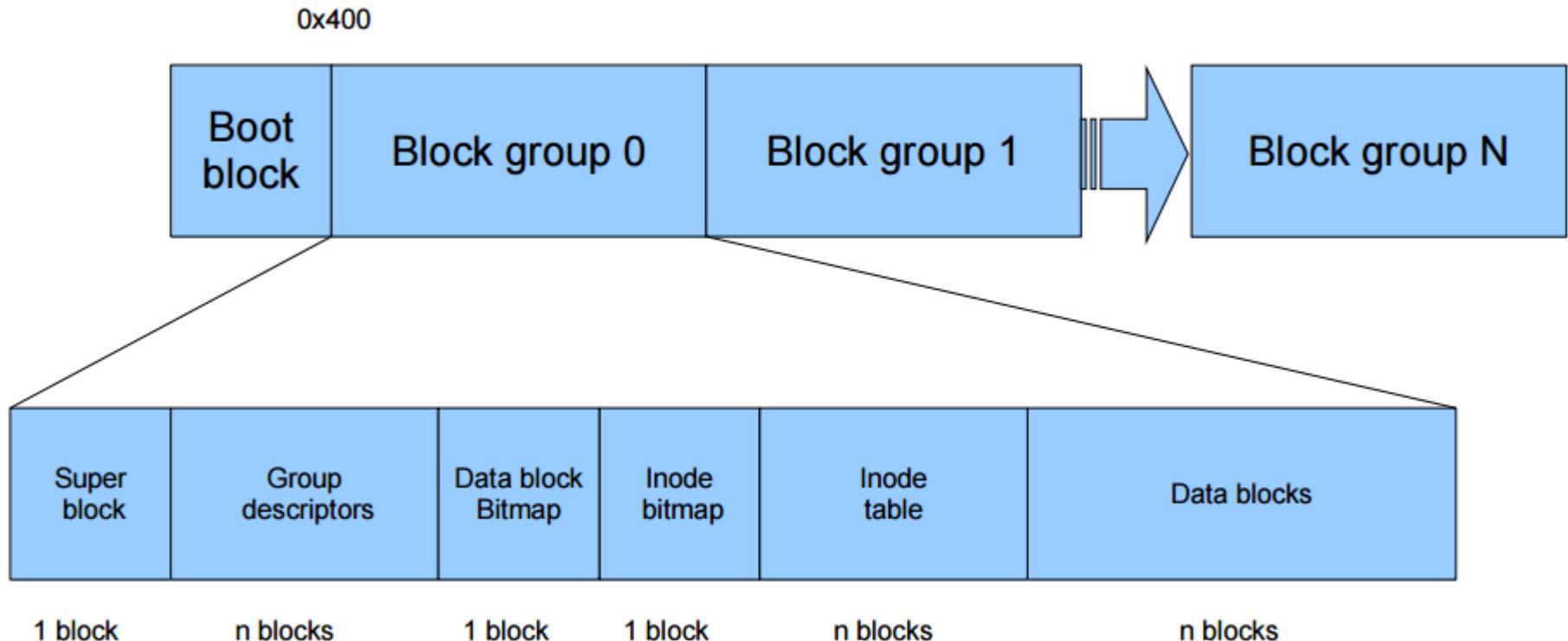


# Further improvements on link list method of free-blocks

- **Grouping**
- **Counting**
- **Space Maps (ZFS)**
- **Read as homework**

# Ext2 FS layout

# Ext2 FS Layout



```
struct ext2_super_block {
    __le32 s_inodes_count; /* Inodes count */
    __le32 s_blocks_count; /* Blocks count */
    __le32 s_r_blocks_count; /* Reserved blocks count */
    __le32 s_free_blocks_count; /* Free blocks count */
    __le32 s_free_inodes_count; /* Free inodes count */
    __le32 s_first_data_block; /* First Data Block */
    __le32 s_log_block_size; /* Block size */
    __le32 s_log_frag_size; /* Fragment size */
    __le32 s_blocks_per_group; /* # Blocks per group */
    __le32 s_frags_per_group; /* # Fragments per group */
    __le32 s_inodes_per_group; /* # Inodes per group */
    __le32 s_mtime; /* Mount time */
    __le32 s_wtime; /* Write time */
    __le16 s_mnt_count; /* Mount count */
    __le16 s_max_mnt_count; /* Maximal mount count */
    __le16 s_magic; /* Magic signature */
    __le16 s_state; /* File system state */
    __le16 s_errors; /* Behaviour when detecting errors */
```

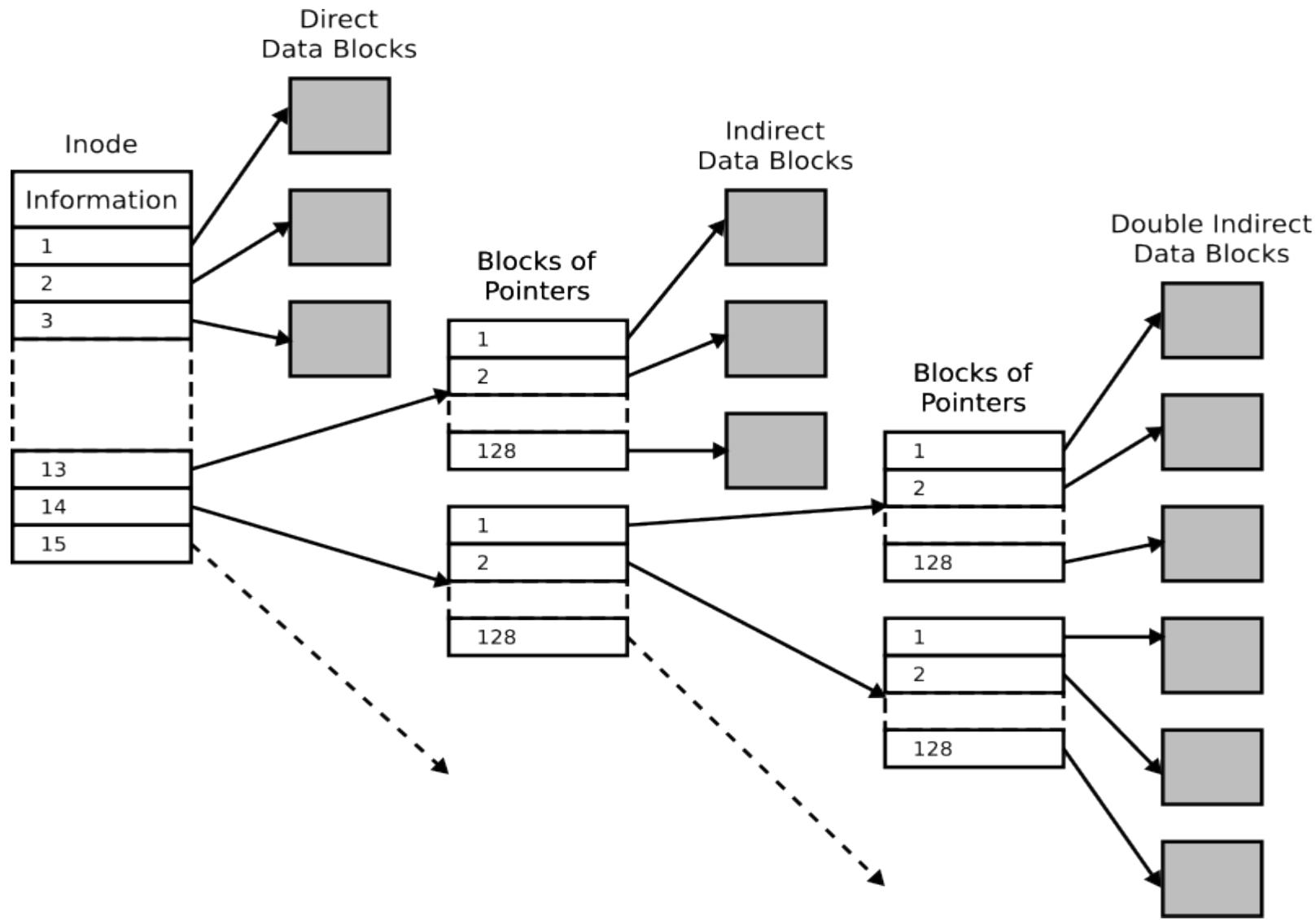
```
struct ext2_super_block {  
...  
    __le16 s_minor_rev_level; /* minor revision level */  
    __le32 s_lastcheck; /* time of last check */  
    __le32 s_checkinterval; /* max. time between checks */  
    __le32 s_creator_os; /* OS */  
    __le32 s_rev_level; /* Revision level */  
    __le16 s_def_resuid; /* Default uid for reserved blocks */  
    __le16 s_def_resgid; /* Default gid for reserved blocks */  
    __le32 s_first_ino; /* First non-reserved inode */  
    __le16 s_inode_size; /* size of inode structure */  
    __le16 s_block_group_nr; /* block group # of this superblock */  
    __le32 s_feature_compat; /* compatible feature set */  
    __le32 s_feature_incompat; /* incompatible feature set */  
    __le32 s_feature_ro_compat; /* readonly-compatible feature set */  
    __u8 s_uuid[16]; /* 128-bit uuid for volume */  
    char s_volume_name[16]; /* volume name */  
    char s_last_mounted[64]; /* directory where last mounted */  
    __le32 s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {  
...  
    __u8 s_prealloc_blocks; /* Nr of blocks to try to preallocate*/  
    __u8 s_prealloc_dir_blocks; /* Nr to preallocate for dirs */  
    __u16 s_padding1;  
/*  
 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.  
 */  
    __u8 s_journal_uuid[16]; /* uuid of journal superblock */  
    __u32 s_journal_inum; /* inode number of journal file */  
    __u32 s_journal_dev; /* device number of journal file */  
    __u32 s_last_orphan; /* start of list of inodes to delete */  
    __u32 s_hash_seed[4]; /* HTREE hash seed */  
    __u8 s_def_hash_version; /* Default hash version to use */  
    __u8 s_reserved_char_pad;  
    __u16 s_reserved_word_pad;  
    __le32 s_default_mount_opts;  
    __le32 s_first_meta_bg; /* First metablock block group */  
    __u32 s_reserved[190]; /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;      /* Blocks bitmap block */
    __le32 bg_inode_bitmap;     /* Inodes bitmap block */
    __le32 bg_inode_table;     /* Inodes table block */
    __le16 bg_free_blocks_count; /* Free blocks count */
    __le16 bg_free_inodes_count; /* Free inodes count */
    __le16 bg_used_dirs_count; /* Directories count */
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

```
struct ext2_inode {  
    __le16 i_mode; /* File mode */  
    __le16 i_uid; /* Low 16 bits of Owner Uid */  
    __le32 i_size; /* Size in bytes */  
    __le32 i_atime; /* Access time */  
    __le32 i_ctime; /* Creation time */  
    __le32 i_mtime; /* Modification time */  
    __le32 i_dtime; /* Deletion Time */  
    __le16 i_gid; /* Low 16 bits of Group Id */  
    __le16 i_links_count; /* Links count */  
    __le32 i_blocks; /* Blocks count */  
    __le32 i_flags; /* File flags */
```

# Inode in ext2



```
struct ext2_inode {  
...  
union {  
    struct {  
        __le32 l_i_reserved1;  
    } linux1;  
    struct {  
        __le32 h_i_translator;  
    } hurd1;  
    struct {  
        __le32 m_i_reserved1;  
    } masix1;  
} osd1; /* OS dependent 1 */  
    __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */  
    __le32 i_generation; /* File version (for NFS) */  
    __le32 i_file_acl; /* File ACL */  
    __le32 i_dir_acl; /* Directory ACL */  
    __le32 i_faddr; /* Fragment address */
```

```
struct ext2_inode {  
...  
union {  
struct {  
__u8 l_i_frag; /* Fragment number */ __u8 l_i_fsize; /* Fragment size */  
__u16 l_i_pad1; __le16 l_i_uid_high; /* these 2 fields */  
__le16 l_i_gid_high; /* were reserved2[0] */  
__u32 l_i_reserved2;  
} linux2;  
struct {  
__u8 h_i_frag; /* Fragment number */ __u8 h_i_fsize; /* Fragment size */  
__le16 h_i_mode_high; __le16 h_i_uid_high;  
__le16 h_i_gid_high;  
__le32 h_i_author;  
} hurd2;  
struct {  
__u8 m_i_frag; /* Fragment number */ __u8 m_i_fsize; /* Fragment size */  
__u16 m_pad1; __u32 m_i_reserved2[2];  
} masix2;  
} osd2; /* OS dependent 2 */
```

# Ext2 FS Layout: Directory entry

	inode	rec_len	file_type	name_len	name						
0	21	12	1	2	.	\0	\0	\0			
12	22	12	2	2	.	.	\0	\0			
24	53	16	5	2	h	o	m	e	1	\0	\0
40	67	28	3	2	u	s	r	\0			
52	0	16	7	1	o	l	d	f	i	l	e
68	34	12	4	2	s	b	i	n			

Let's see a program to read superblock of an ext2  
file system.

**Efficiency and Performance  
(and the risks created  
while trying to achieve it!)**

# Efficiency

- **Efficiency dependent on:**
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

# Performance

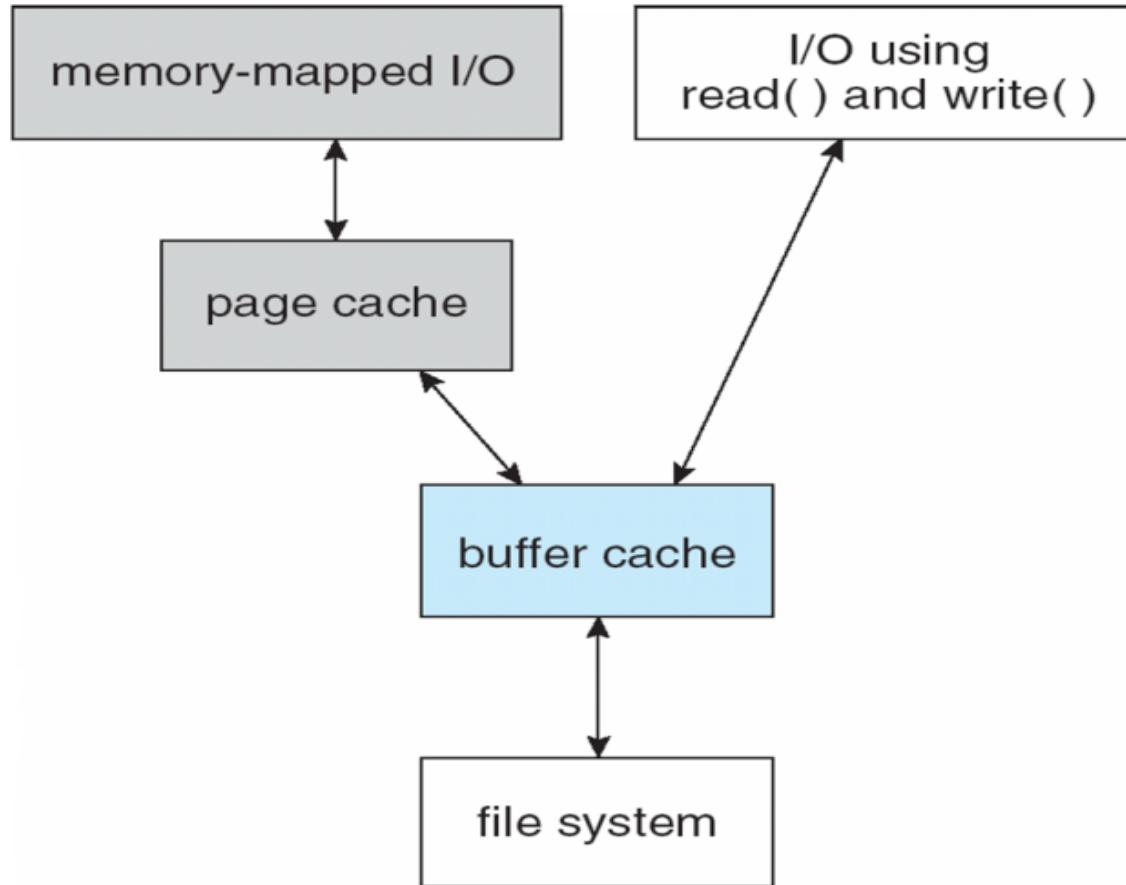
- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement

Asynchronous writes more common, buffer

# Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

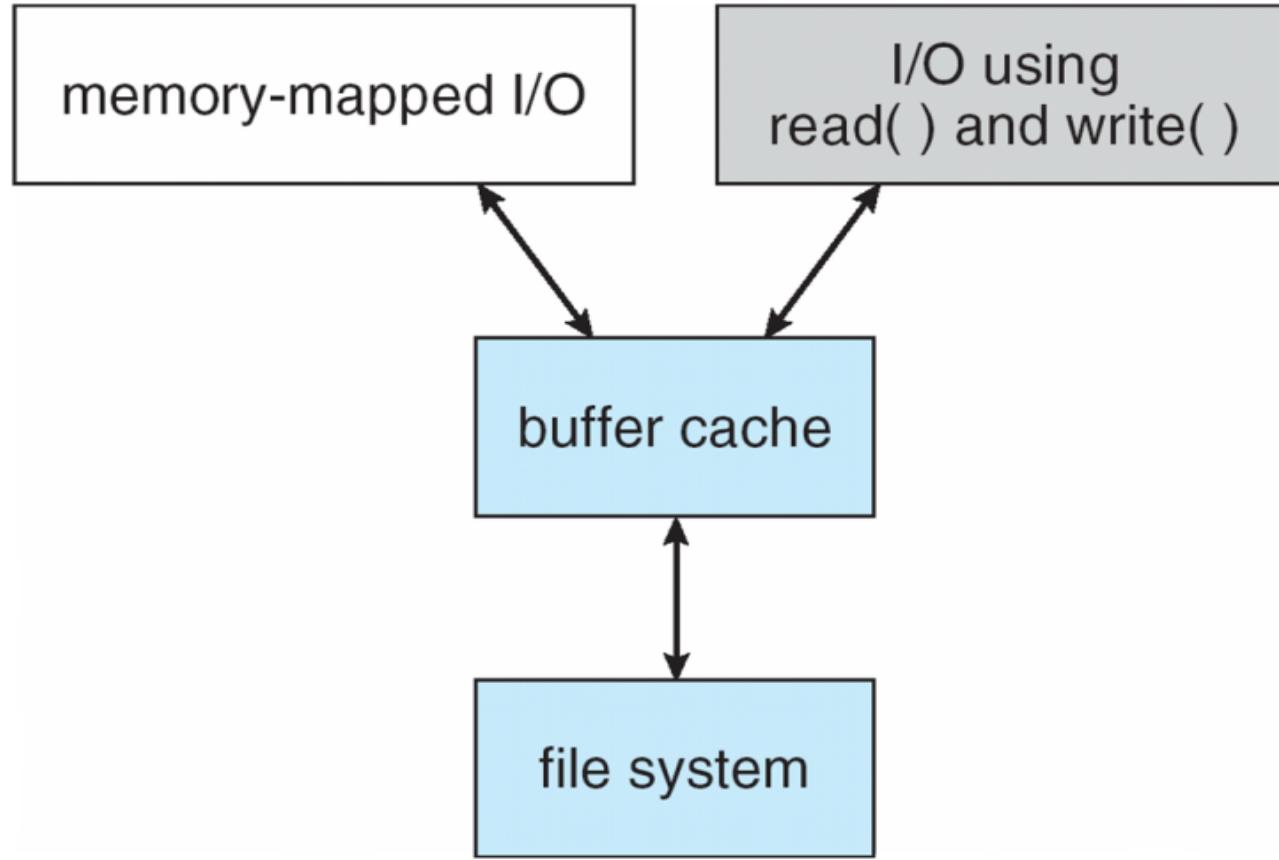
# I/O Without a Unified Buffer Cache



# Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

# I/O Using a Unified Buffer Cache



# Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS

# Recovery

- **Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
  - Can be slow and sometimes fails
- **Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)**
- **Recover lost file or disk by restoring data**

# Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated

# Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!















# File Systems

Abhijit A M  
abhijit.comp@coep.ac.in

# **System calls related to files/file-system**

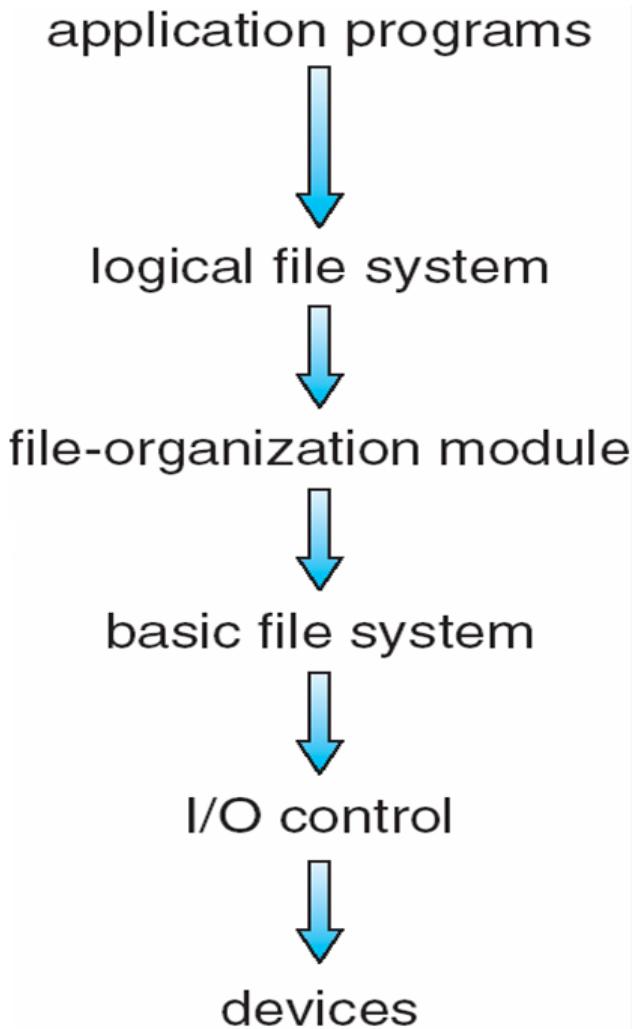
- **Open(2), chmod(2), chown(2), close(2), dup(2), fcntl(2), link(2), lseek(2), mknod(2), mmap(2), mount(2), read(2), stat(2), umask(2), unlink(2), write(2), fstat(2), access(2), readlink(2), ...**

# **Implementing file systems**

# **File system on disk**

- **Disk I/O in terms of sectors (512 bytes)**
- **File system: implementation of acyclic graph using the linear sequence of sectors**
- **Device driver: available to rest of the OS code to access disk using a block number**

# File system implementation: layering



## Application programs

```
int main() {  
    char buf[128]; int count;  
    fd = open(...);  
    read(fd, buf, count);  
}
```

-----

## OS

### Logical file system:

```
sys_read(int fd, char *buf, int count) {  
    file *fp = currproc->fdarray[fd];  
    file_read(fp, ...);  
}
```

### File organization module:

```
file_read(file *fp, char *buf, int count) {  
    offset = fp->current_offset;  
    translate offset into blockno;  
    basic_read(blockno, buf, count);
```

### Basic File system:

```
basic_read(int blockno, char *buf, ...) {  
    os_buffer *bp;  
    sectorno = calculation on blockno;  
    disk_driver_read(sectorno, bp );  
    move-process-to-wait-queue;  
    copy-data-to-user-buffer(bp, buf);  
}
```

### IO Control, Device driver:

```
disk_driver_read(sectorno) {  
    issue instructions to disk controller (often assembly)  
    to read sectorno into specific location;  
}
```

*XV6 does it slightly differently, but following the same steps*

# A typical file control block (inode)

file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

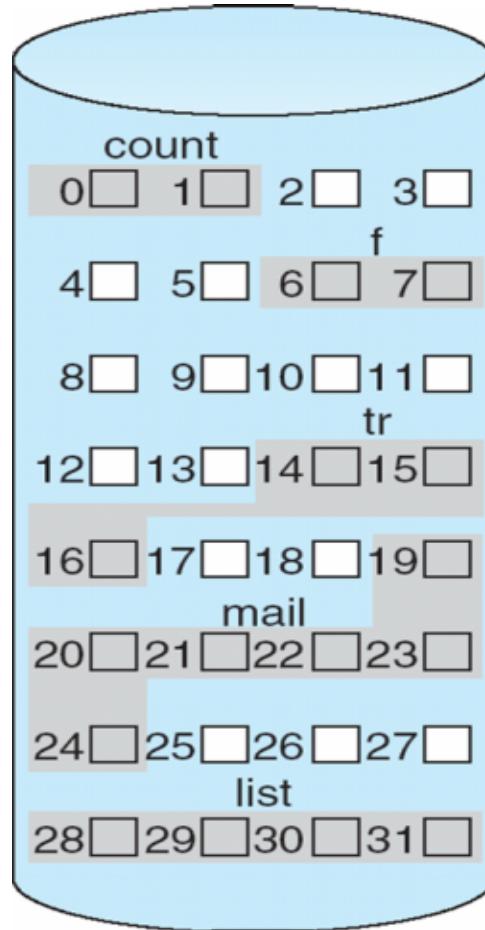
Why does it NOT contain the

Name of the file ?

# Disk space allocation for files

- **File contain data and need disk blocks/sectors for storing it**
- **File system layer does the allocation of blocks on disk to files**
- **Files need to**
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

# Contiguous Allocation of Disk Space



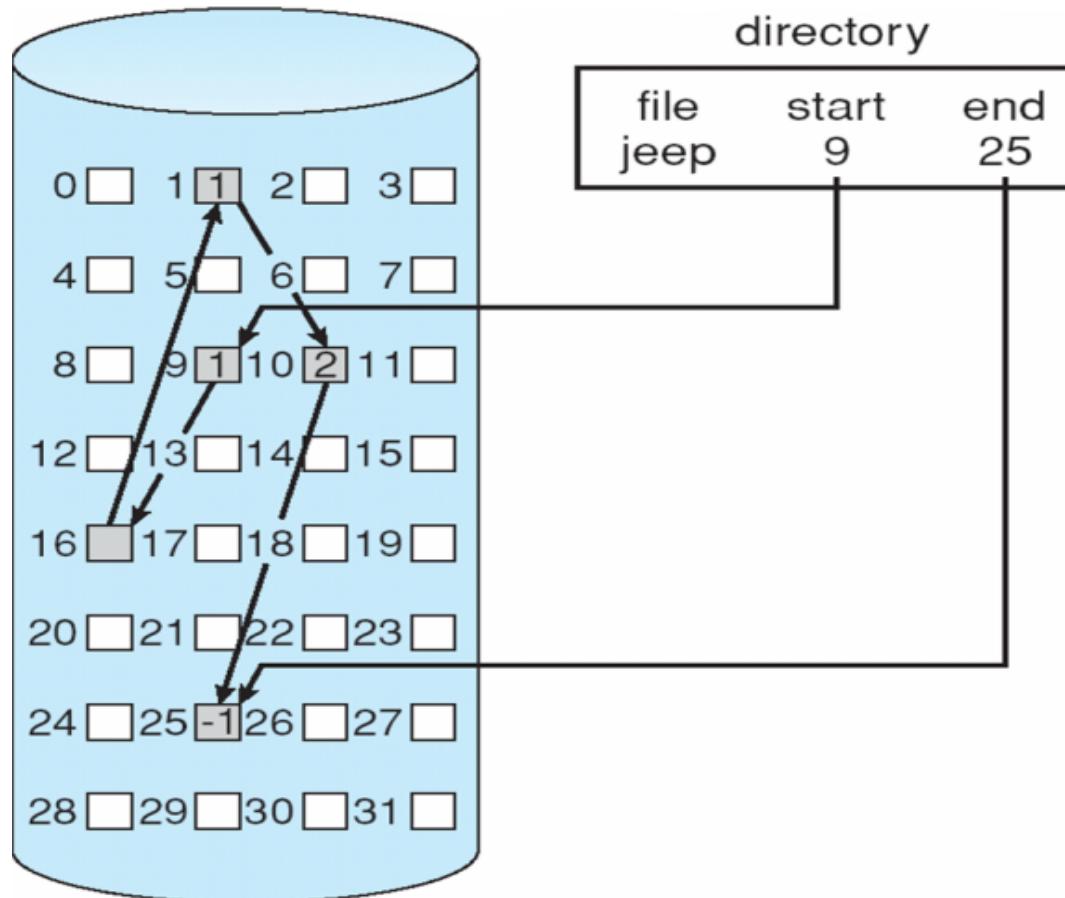
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

# Contiguous allocation

- **Each file occupies set of contiguous blocks**
- **Best performance in most cases**
- **Simple – only starting location (block #) and length (number of blocks) are required**
- **Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line**

# Linked allocation of blocks to a file



# Linked allocation of blocks to a file

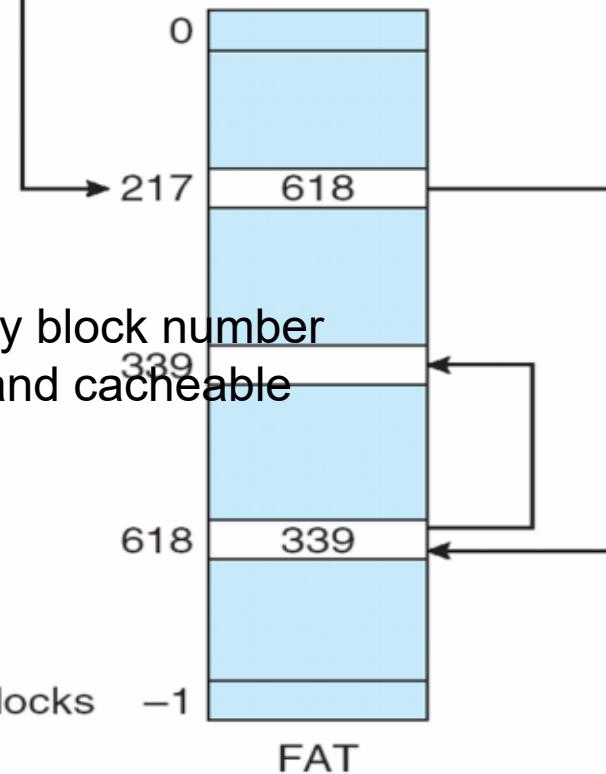
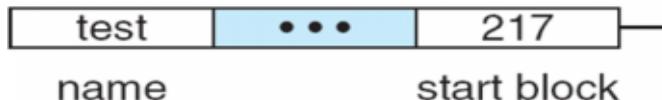
## □ **Linked allocation**

- Each file a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block (i.e. data + pointer to

- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem

# FAT: File Allocation Table

directory entry



- FAT (File Allocation Table), a variation

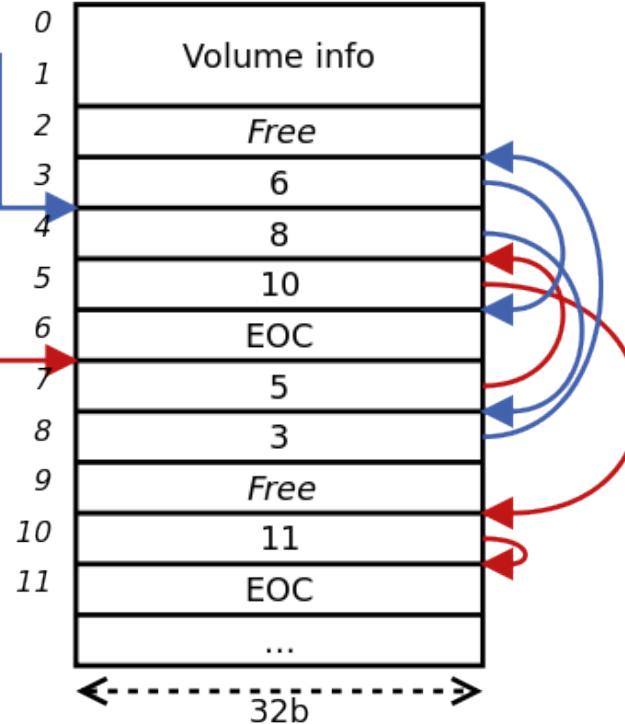
- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple

# FAT: File Allocation Table

Directory table entry (32B)

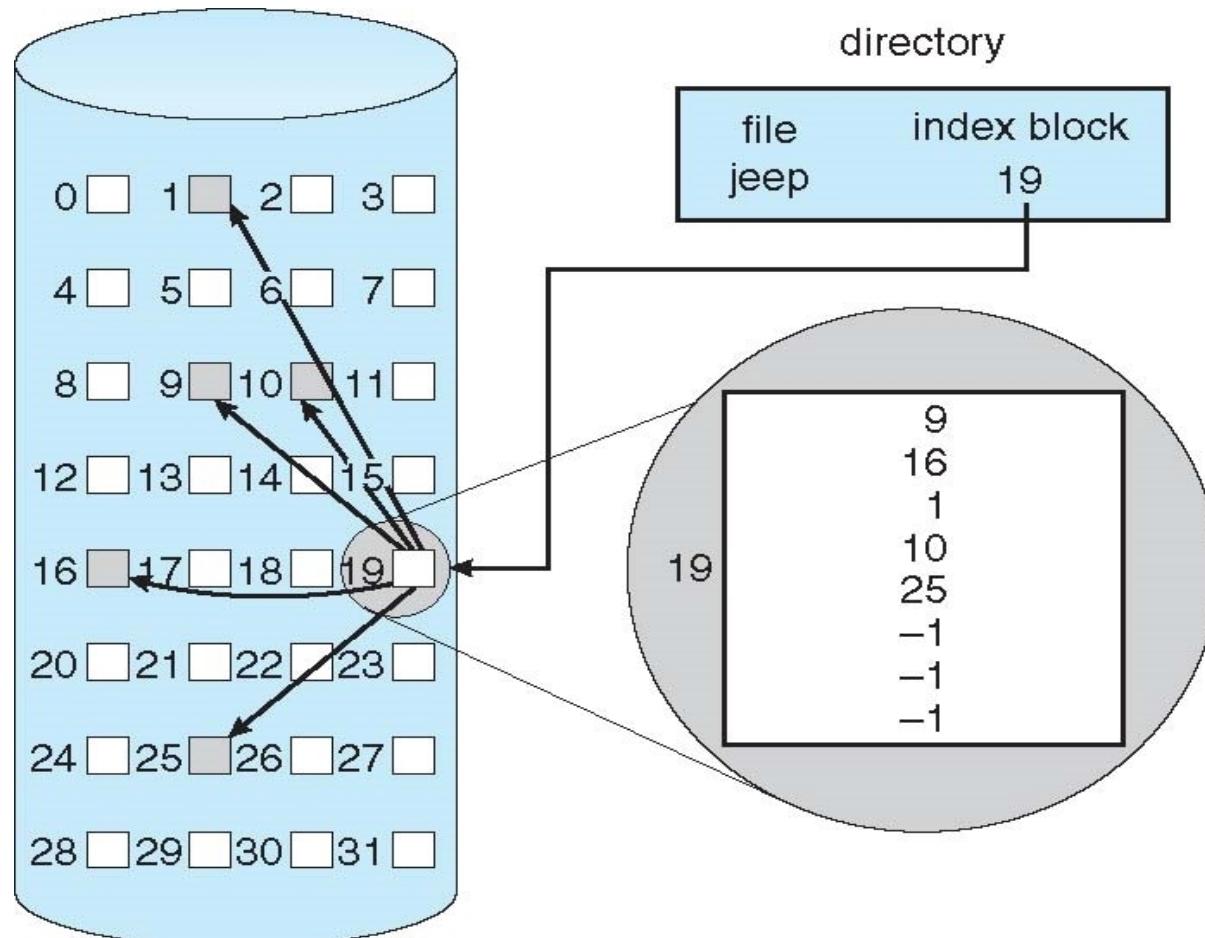
Filename (8B)
Extension (3B)
Attributes (1B)
Reserved (1B)
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)

File allocation table



Variants: FAT8,  
FAT12, FAT16,  
FAT32, VFAT, ...

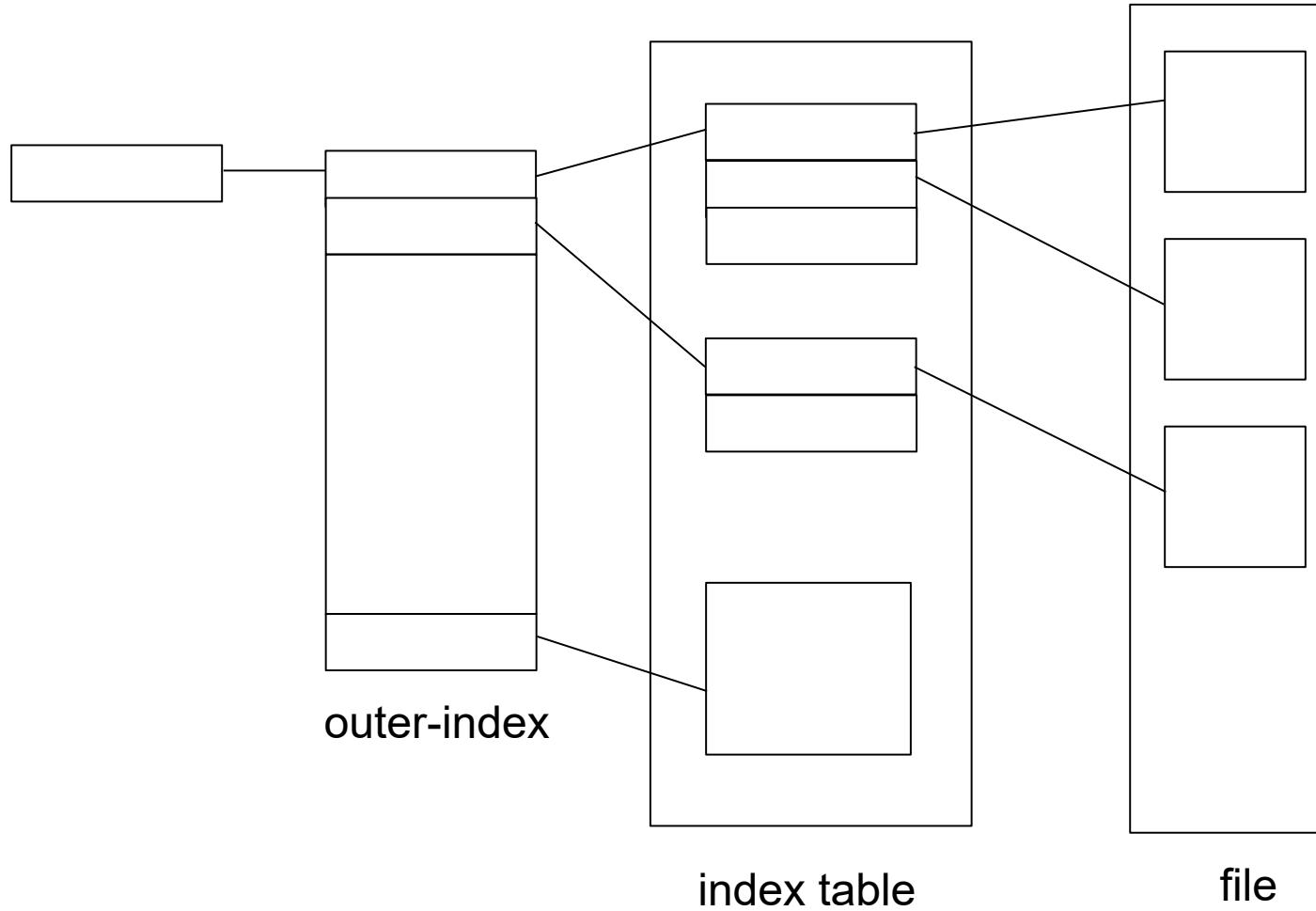
# Indexed allocation



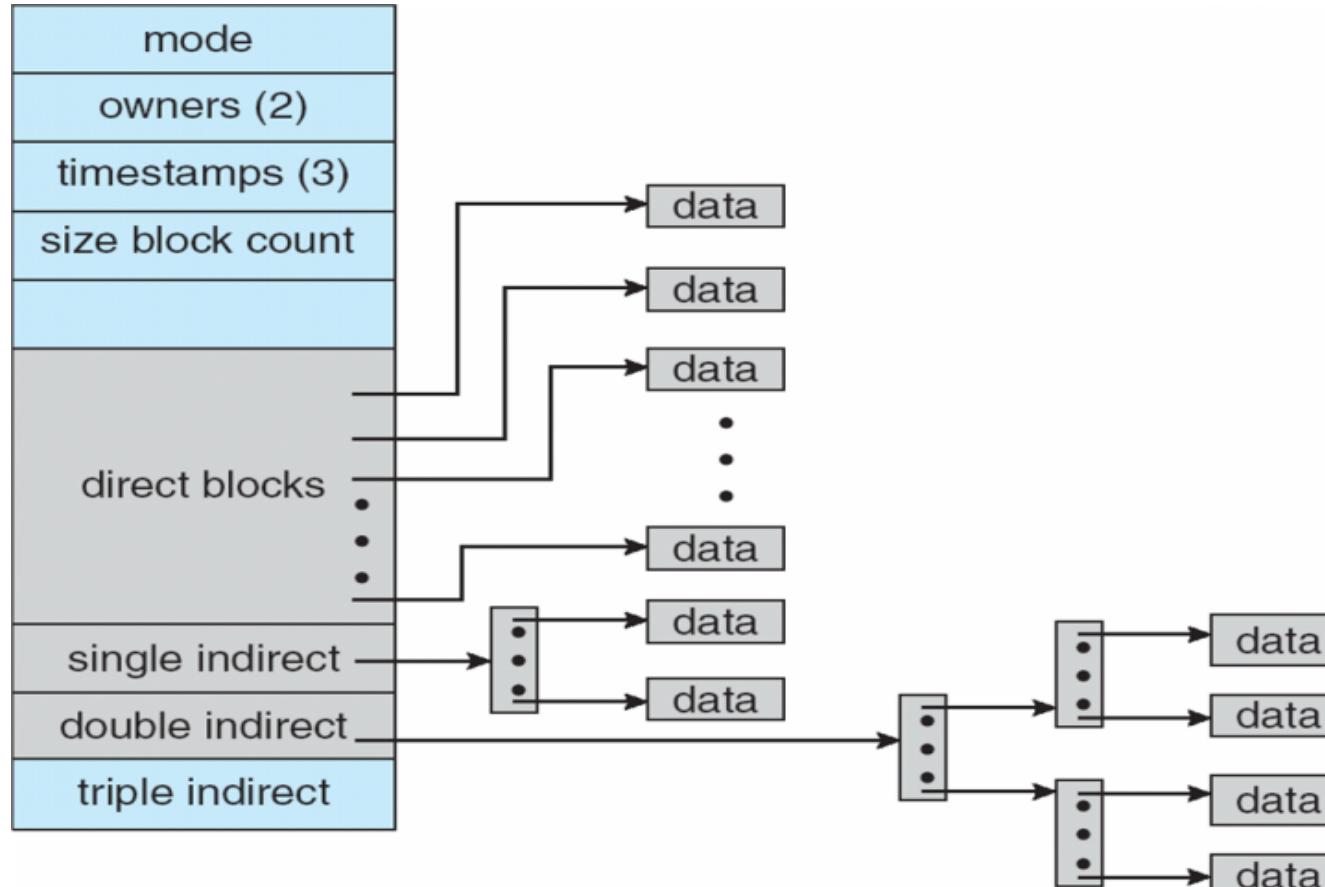
# Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index

# Multi level indexing



# Unix UFS: combined scheme for block allocation



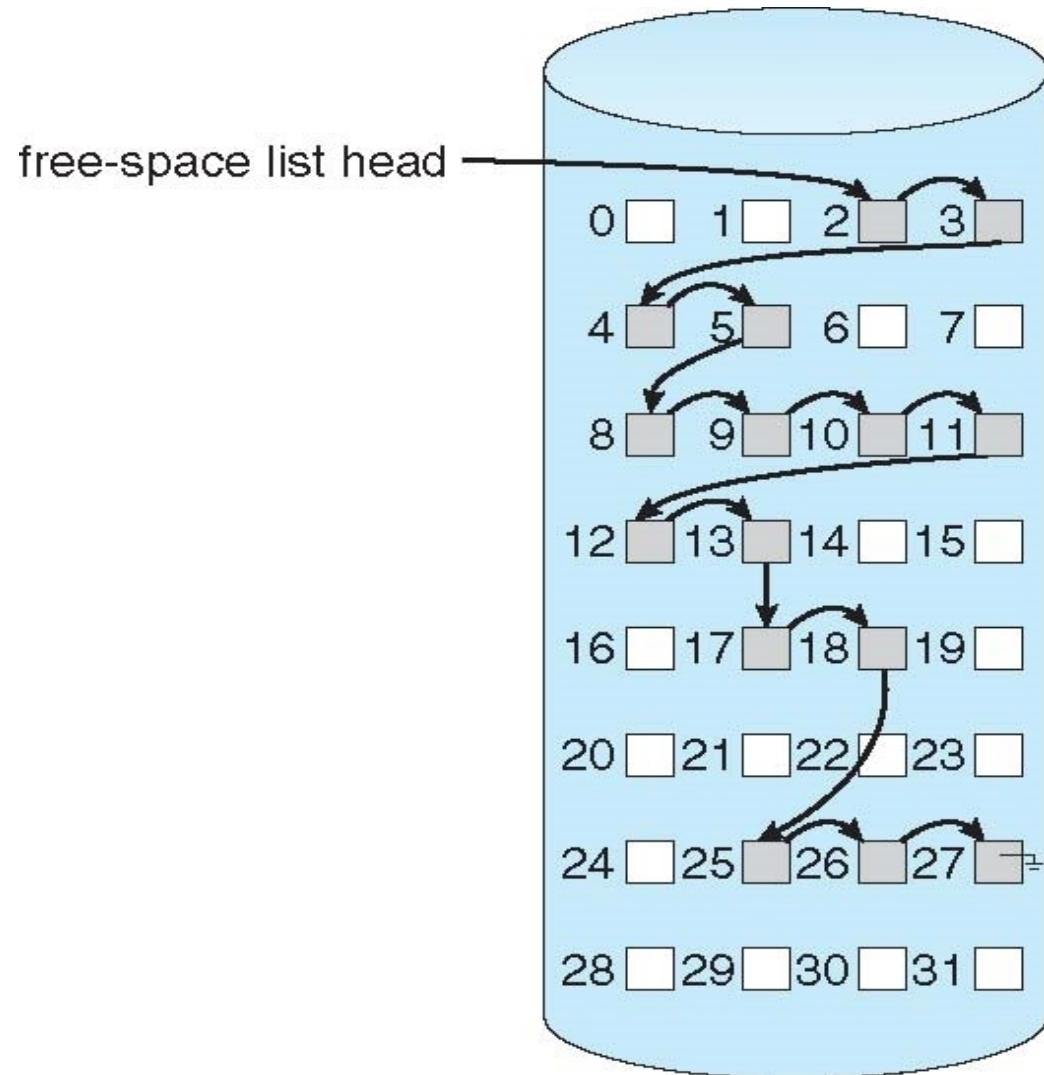
# Free Space Management

- File system maintains free-space list to track available blocks/clusters
  - Bit vector or bit map (n blocks)
  - Or Linked list

# Free Space Management: bit vector

- **Each block is represented by 1 bit.**
- **If the block is free, the bit is 1; if the block is allocated, the bit is 0.**
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 001111001111110001100000011100000 ...

# Free Space Management: Linked list (not in memory, on disk!)



# Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

# Directory Implementation

## □ Problem

- Directory contains files and/or subdirectories
- Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
- Directory needs to give location of each file on disk

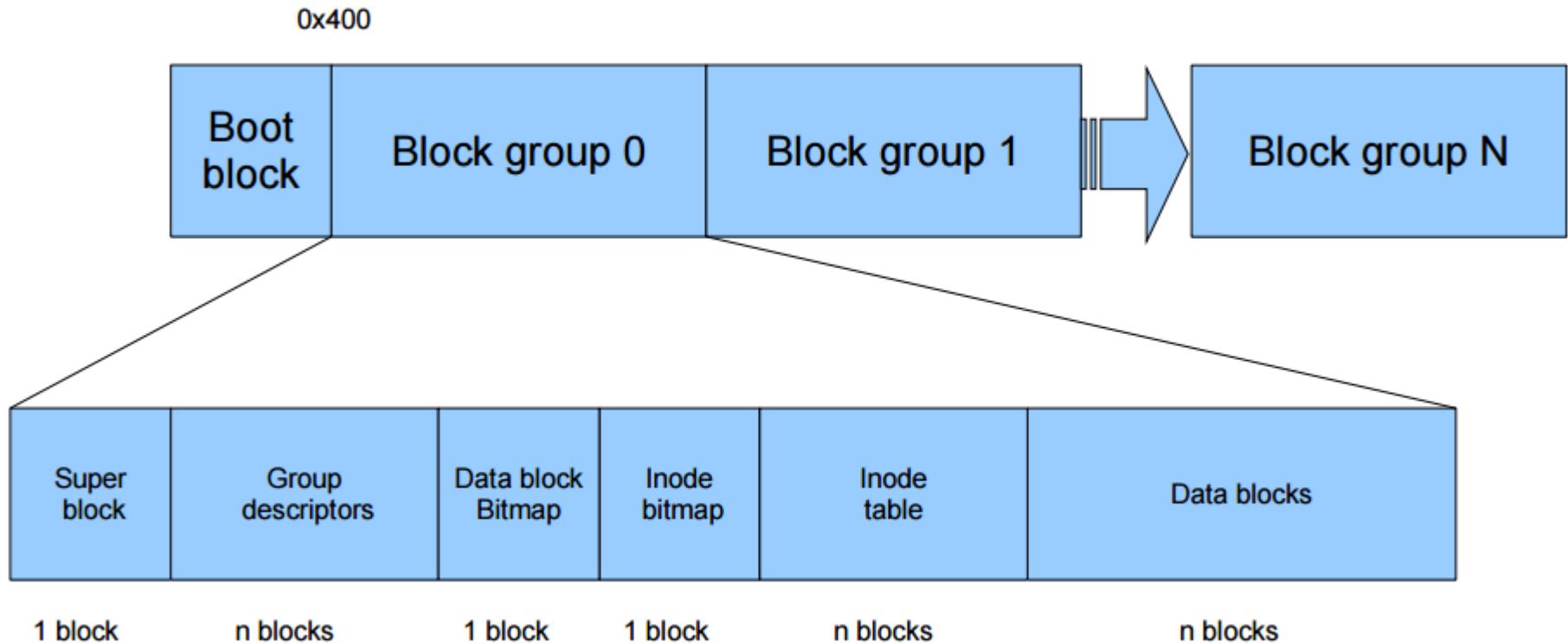
# Directory Implementation

- **Linear list of file names with pointer to the data blocks**
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.

**Hash Table - Linear list with hash data**

# Ext2 FS layout

# Ext2 FS Layout



```
struct ext2_super_block {
    __le32 s_inodes_count; /* Inodes count */
    __le32 s_blocks_count; /* Blocks count */
    __le32 s_r_blocks_count; /* Reserved blocks count */
    __le32 s_free_blocks_count; /* Free blocks count */
    __le32 s_free_inodes_count; /* Free inodes count */
    __le32 s_first_data_block; /* First Data Block */
    __le32 s_log_block_size; /* Block size */
    __le32 s_log_frag_size; /* Fragment size */
    __le32 s_blocks_per_group; /* # Blocks per group */
    __le32 s_frags_per_group; /* # Fragments per group */
    __le32 s_inodes_per_group; /* # Inodes per group */
    __le32 s_mtime; /* Mount time */
    __le32 s_wtime; /* Write time */
    __le16 s_mnt_count; /* Mount count */
    __le16 s_max_mnt_count; /* Maximal mount count */
    __le16 s_magic; /* Magic signature */
    __le16 s_state; /* File system state */
    __le16 s_errors; /* Behaviour when detecting errors */
```

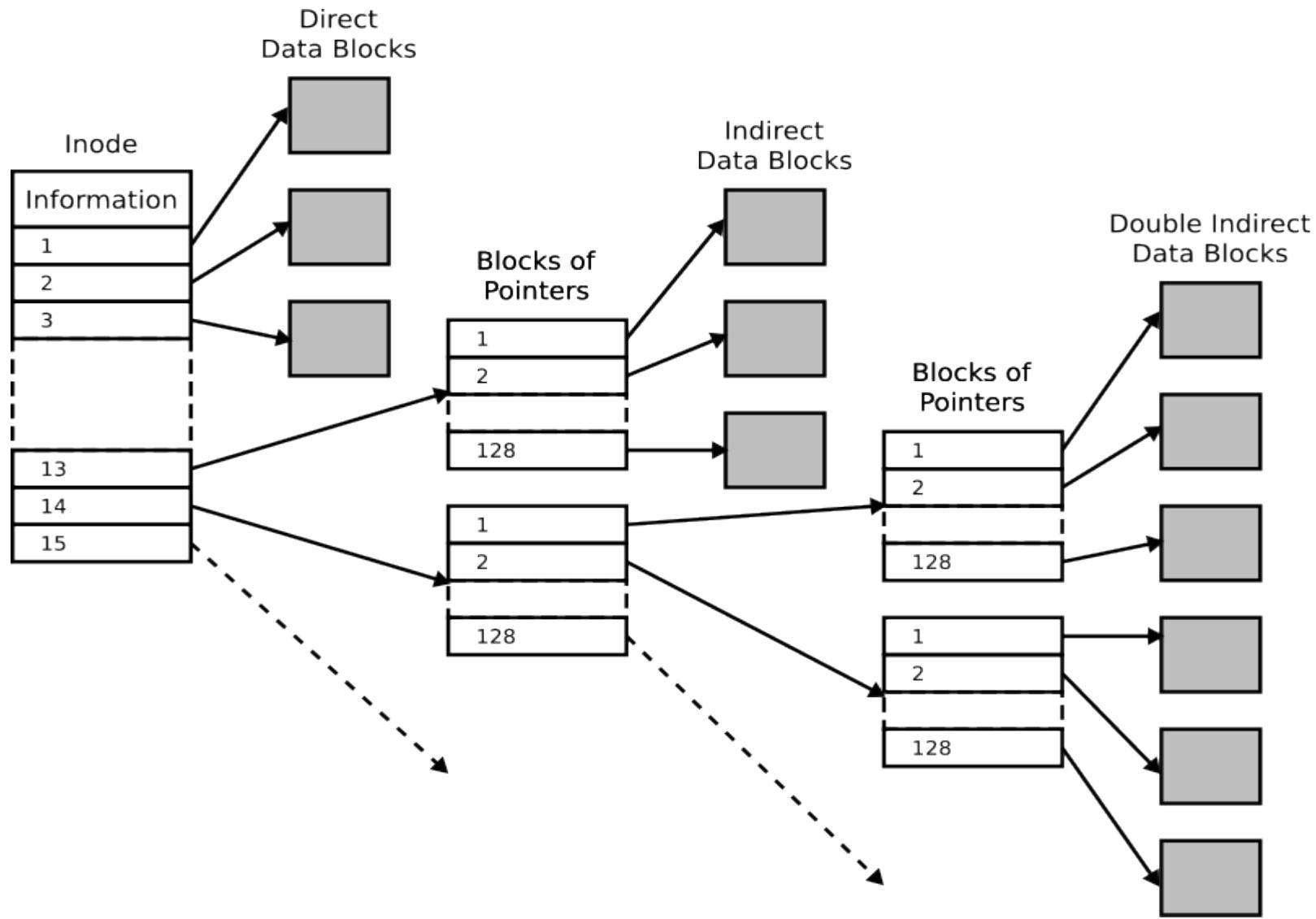
```
struct ext2_super_block {  
...  
    __le16 s_minor_rev_level; /* minor revision level */  
    __le32 s_lastcheck; /* time of last check */  
    __le32 s_checkinterval; /* max. time between checks */  
    __le32 s_creator_os; /* OS */  
    __le32 s_rev_level; /* Revision level */  
    __le16 s_def_resuid; /* Default uid for reserved blocks */  
    __le16 s_def_resgid; /* Default gid for reserved blocks */  
    __le32 s_first_ino; /* First non-reserved inode */  
    __le16 s_inode_size; /* size of inode structure */  
    __le16 s_block_group_nr; /* block group # of this superblock */  
    __le32 s_feature_compat; /* compatible feature set */  
    __le32 s_feature_incompat; /* incompatible feature set */  
    __le32 s_feature_ro_compat; /* readonly-compatible feature set */  
    __u8 s_uuid[16]; /* 128-bit uuid for volume */  
    char s_volume_name[16]; /* volume name */  
    char s_last_mounted[64]; /* directory where last mounted */  
    __le32 s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {  
...  
    __u8 s_prealloc_blocks; /* Nr of blocks to try to preallocate*/  
    __u8 s_prealloc_dir_blocks; /* Nr to preallocate for dirs */  
    __u16 s_padding1;  
/*  
 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.  
 */  
    __u8 s_journal_uuid[16]; /* uuid of journal superblock */  
    __u32 s_journal_inum; /* inode number of journal file */  
    __u32 s_journal_dev; /* device number of journal file */  
    __u32 s_last_orphan; /* start of list of inodes to delete */  
    __u32 s_hash_seed[4]; /* HTREE hash seed */  
    __u8 s_def_hash_version; /* Default hash version to use */  
    __u8 s_reserved_char_pad;  
    __u16 s_reserved_word_pad;  
    __le32 s_default_mount_opts;  
    __le32 s_first_meta_bg; /* First metablock block group */  
    __u32 s_reserved[190]; /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;      /* Blocks bitmap block */
    __le32 bg_inode_bitmap;     /* Inodes bitmap block */
    __le32 bg_inode_table;     /* Inodes table block */
    __le16 bg_free_blocks_count; /* Free blocks count */
    __le16 bg_free_inodes_count; /* Free inodes count */
    __le16 bg_used_dirs_count; /* Directories count */
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

```
struct ext2_inode {  
    __le16 i_mode; /* File mode */  
    __le16 i_uid; /* Low 16 bits of Owner Uid */  
    __le32 i_size; /* Size in bytes */  
    __le32 i_atime; /* Access time */  
    __le32 i_ctime; /* Creation time */  
    __le32 i_mtime; /* Modification time */  
    __le32 i_dtime; /* Deletion Time */  
    __le16 i_gid; /* Low 16 bits of Group Id */  
    __le16 i_links_count; /* Links count */  
    __le32 i_blocks; /* Blocks count */  
    __le32 i_flags; /* File flags */
```

# Inode in ext2



```
struct ext2_inode {  
...  
union {  
    struct {  
        __le32 l_i_reserved1;  
    } linux1;  
    struct {  
        __le32 h_i_translator;  
    } hurd1;  
    struct {  
        __le32 m_i_reserved1;  
    } masix1;  
} osd1; /* OS dependent 1 */  
    __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */  
    __le32 i_generation; /* File version (for NFS) */  
    __le32 i_file_acl; /* File ACL */  
    __le32 i_dir_acl; /* Directory ACL */  
    __le32 i_faddr; /* Fragment address */
```

```
struct ext2_inode {  
...  
union {  
struct {  
__u8 l_i_frag; /* Fragment number */ __u8 l_i_fsize; /* Fragment size */  
__u16 l_i_pad1; __le16 l_i_uid_high; /* these 2 fields */  
__le16 l_i_gid_high; /* were reserved2[0] */  
__u32 l_i_reserved2;  
} linux2;  
struct {  
__u8 h_i_frag; /* Fragment number */ __u8 h_i_fsize; /* Fragment size */  
__le16 h_i_mode_high; __le16 h_i_uid_high;  
__le16 h_i_gid_high;  
__le32 h_i_author;  
} hurd2;  
struct {  
__u8 m_i_frag; /* Fragment number */ __u8 m_i_fsize; /* Fragment size */  
__u16 m_pad1; __u32 m_i_reserved2[2];  
} masix2;  
} osd2; /* OS dependent 2 */
```

# Ext2 FS Layout: Directory entry

	inode	rec_len	file_type	name_len	name						
0	21	12	1	2	.	\0	\0	\0			
12	22	12	2	2	.	.	\0	\0			
24	53	16	5	2	h	o	m	e	1	\0	\0
40	67	28	3	2	u	s	r	\0			
52	0	16	7	1	o	l	d	f	i	l	e
68	34	12	4	2	s	b	i	n			

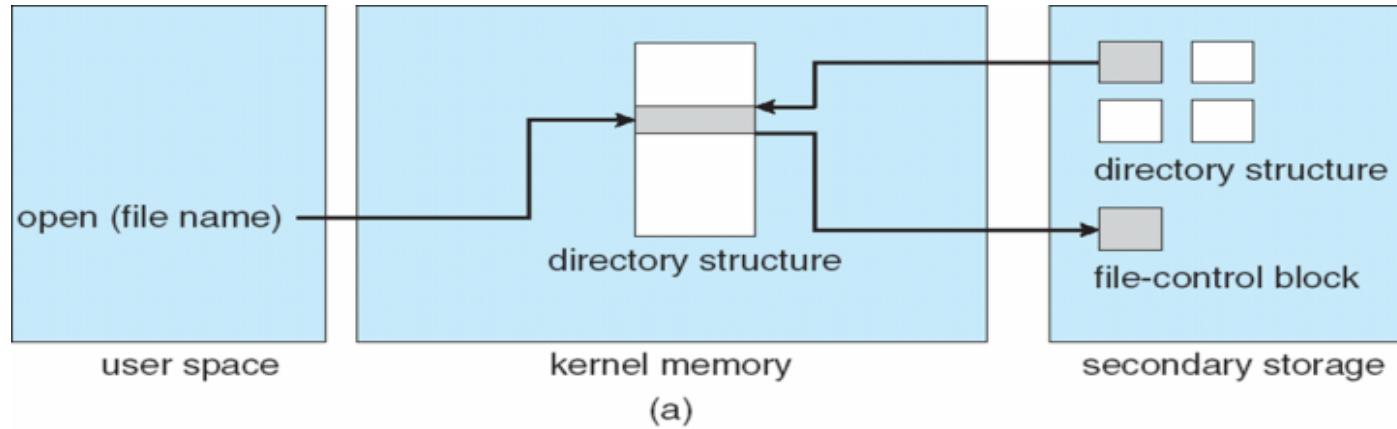
Let's see a program to read superblock of an ext2  
file system.

**Efficiency and Performance  
(and the risks created  
while trying to achieve it!)**

# In memory data structures

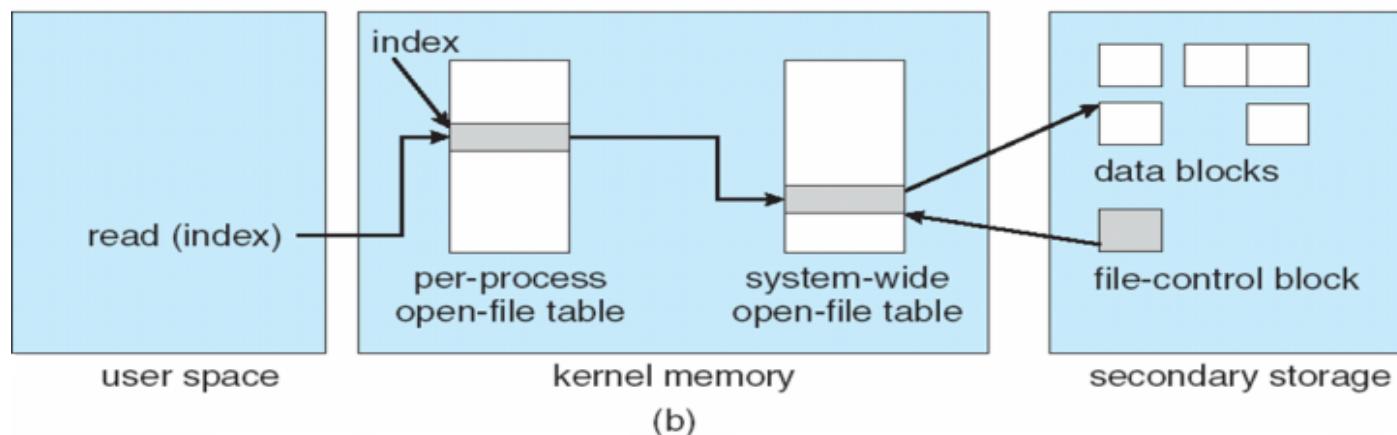
- **Mount table**
  - storing file system mounts, mount points, file system types
- **See next slide for “file” realated data structures**
- **Buffers**
  - hold data blocks from secondary storage

# In memory data structures: for open,read,write, ...



Open returns a file handle for

Data from read eventually co



# Efficiency

- **Efficiency dependent on:**
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

# Performance

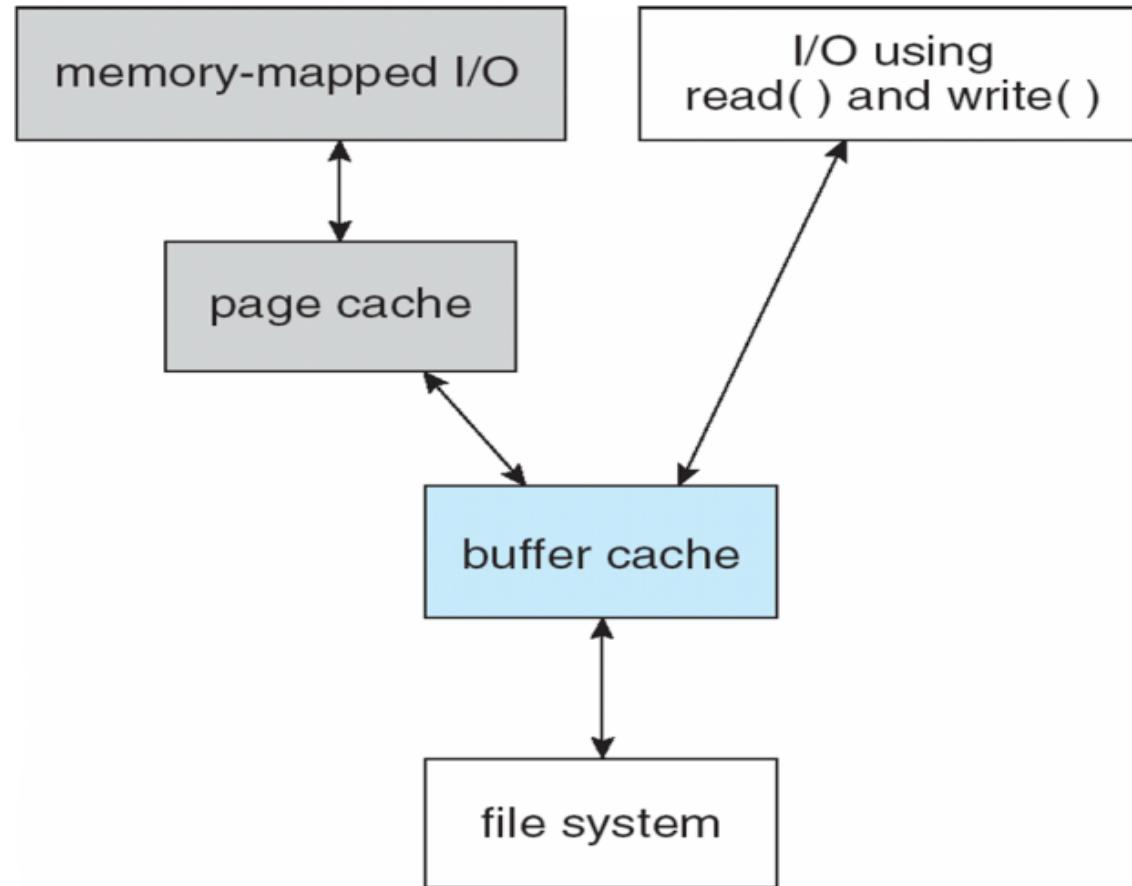
- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement

Asynchronous writes more common, buffer

# Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

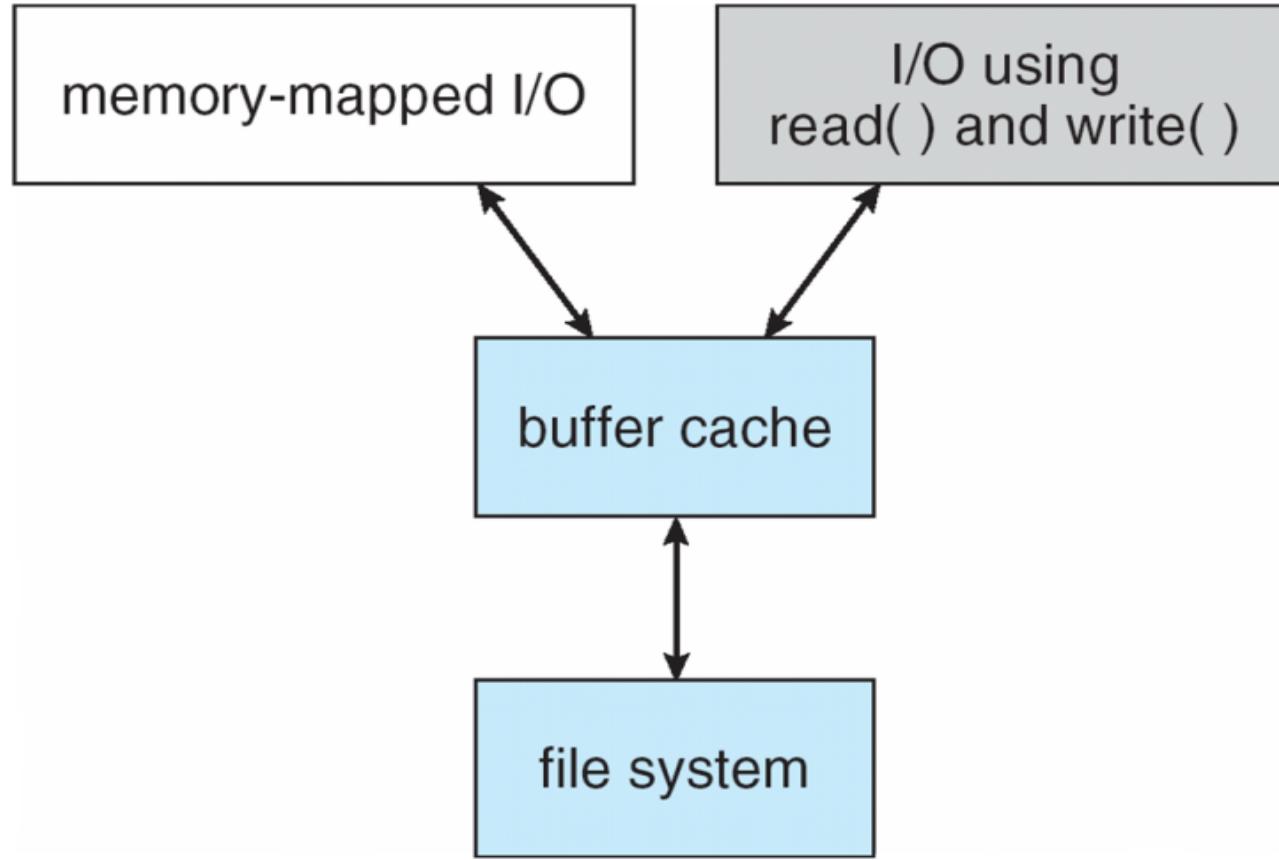
# I/O Without a Unified Buffer Cache



# Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

# I/O Using a Unified Buffer Cache



# Recovery

- **Problem. Consider creating a file on ext2 file system.**
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS

# Recovery

- **Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
  - Can be slow and sometimes fails
- **Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)**
- **Recover lost file or disk by restoring data**

# Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated

# Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!















# **File System Code**

**open, read, write, close, pipe, fstat, chdir, dup,  
mknod, link, unlink, mkdir,**

**Files, Inodes, Buffers**

# What we already know

- **File system related system calls**
  - deal with ‘**fd**’ arrays (**ofile** in xv6). **open()** returns first empty index. **open** should ideally locate the inode on disk and initialize some data structures
  - maintain ‘**offsets**’ within a ‘file’ to support sequential read/write
  - **dup()** like system calls duplicate pointers in fd-array
  - read/write like system calls going through ‘**ofile**’

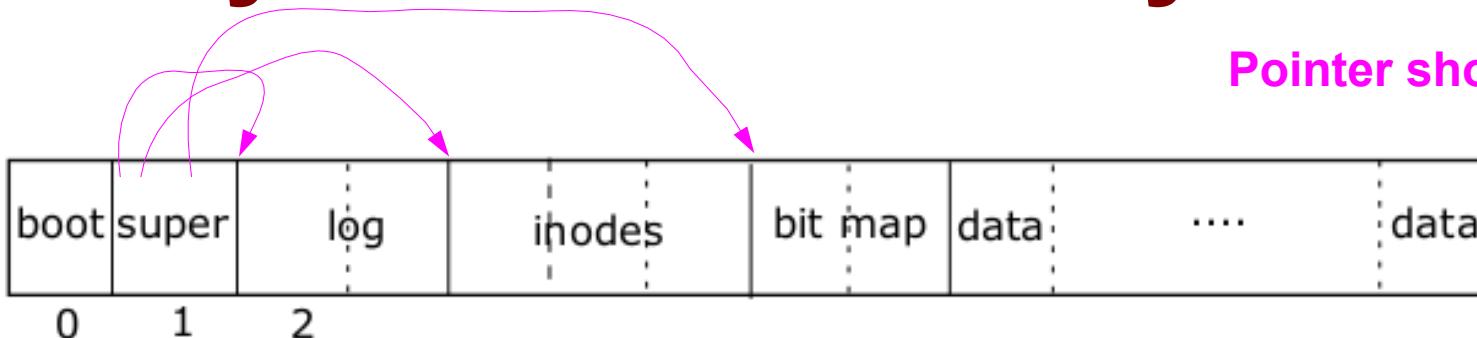
# xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartmentalized

# Layers of xv6 file system code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipellem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iiinit, ialloc, iupdate, igeet, idup, ilock, unlock, iinput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> ,
Logging	Block allocation on disk: <b>balloc</b> , <b>bfree</b> <b>log.c</b> : begin_op, end_op, initlog, commit,
Buffer cache	<b>bio.c</b> binit, bget, bread, bwrite, brelse

# Layout of xv6 file system

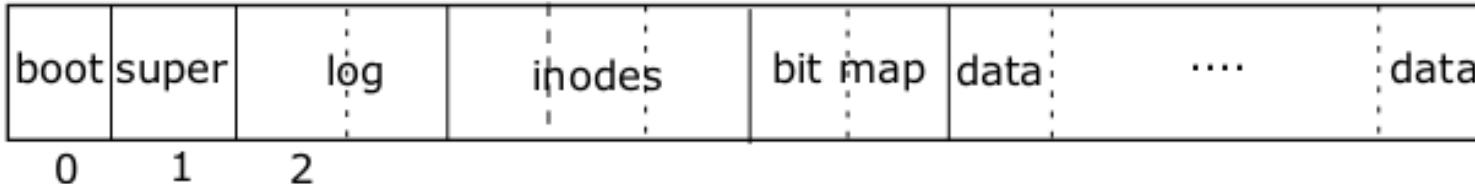


Pointer shown are concept

May see the code of `mkfs.c` to get insight into the layout

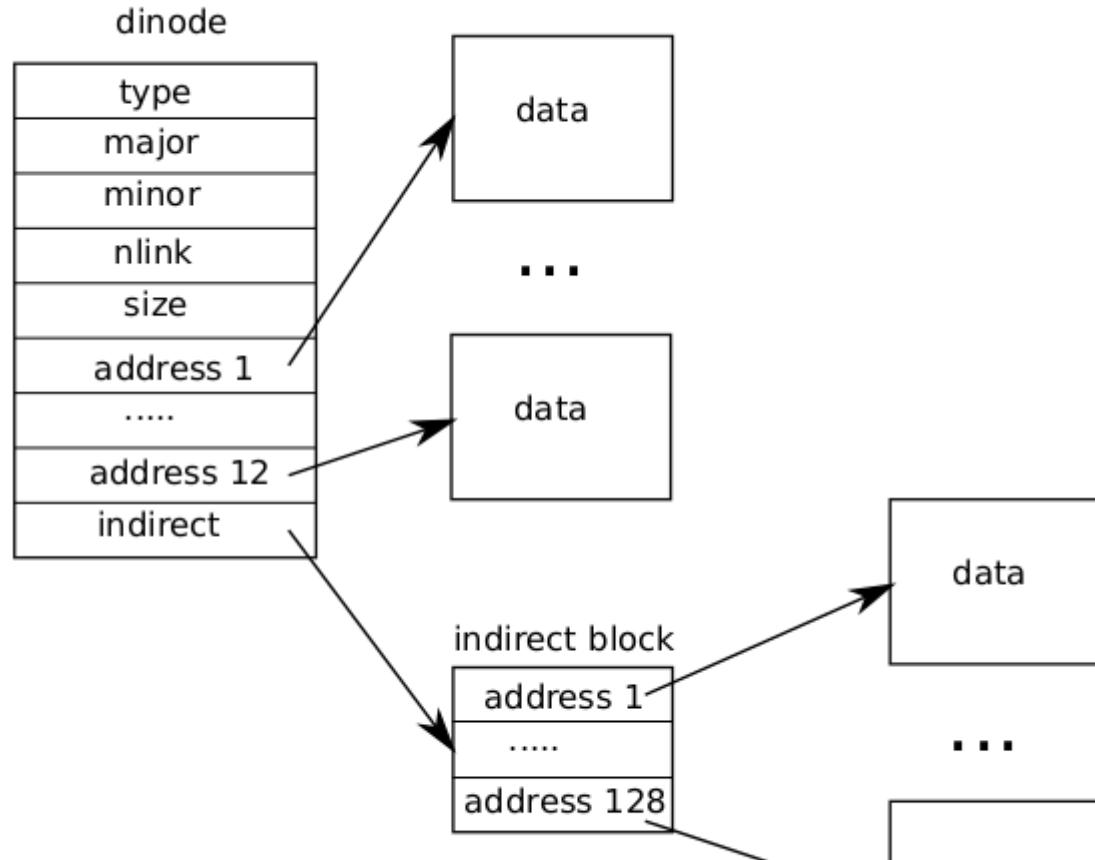
```
struct superblock {  
    uint size; // Size of file system image (blocks)  
    uint nblocks; // Number of data blocks  
    uint ninodes; // Number of inodes.  
    uint nlog; // Number of log blocks  
    uint logstart; // Block number of first log block  
    uint inodestart; // Block number of first inode block  
    uint bmapstart; // Block number of first free map block
```

# Layout of xv6 file system



```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
    short type; // File type
    short major; // Major device number (T_DEV only)
    short minor; // Minor device number (T_DEV only)
    short nlink; // Number of links to inode in file system
    uint size; // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
};
```

# File on disk



# Let's discuss lowest layer first

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipellem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> ,
Logging	Block allocation on disk: <b>balloc</b> , <b>bfree</b> <b>log.c</b> : begin_op, end_op, initlog, commit,
Buffer cache	<b>bio.c</b> binit, bget, bread, bwrite, brelse

# **ide.c: idewait, ideinit, idestart, ideintr, iderw**

**static struct spinlock idelock;**

**static struct buf \*idequeue;**

**static int havedisk1;**

- **ideinit**

- was called from **main.c: main()**
  - Initialized IDE controller by writing to certain ports

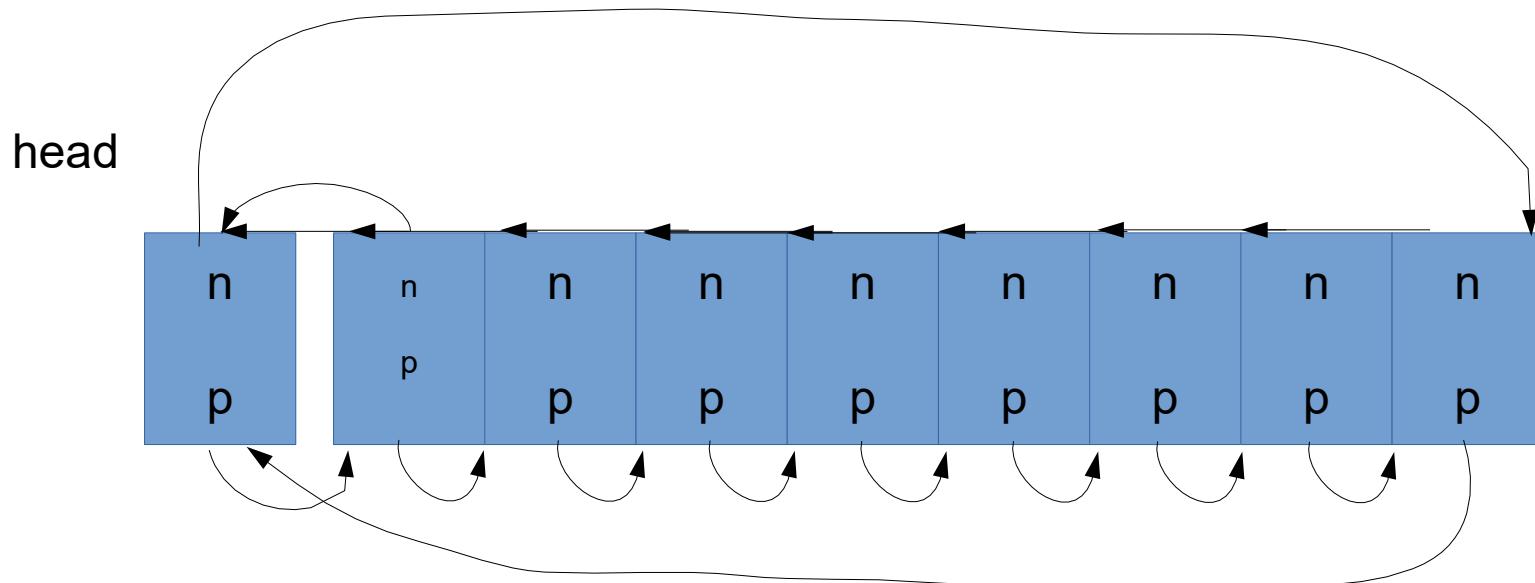
# ide.c: idewait, ideinit, idestart, ideintr, iderw

- **void idestart(buf \*b)**
  - static void **idestart(struct buf \*b)**
  - Calculate sector number on disk using b->blockno
  - Issue a read/write command to IDE controller.
  - (This is the first buf on **idequeue**)
- **ideintr**
  - Take **idelock**. Called on IDE interrupt (through alltraps

# Let's see buffer cache layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipellem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> ,
Logging	Block allocation on disk: <b>balloc</b> , <b>bfree</b> <b>log.c</b> : begin_op, end_op, initlog, commit,
Buffer cache	<b>bio.c</b> binit, bget, bread,

# Reminder: After main() -> binit()



Conceptually Li

Buffers keep me

# struct buf

```
struct buf {  
    int flags; // 0 or B_VALID or B_DIRTY  
    uint dev; // device number  
    uint blockno; // seq block number on device  
    struct sleeplock lock; // Lock to be held by process using it  
    uint refcnt; // Number of live accesses to the buf  
    struct buf *prev; // cache list  
    struct buf *next; // cache list  
    struct buf *qnext; // disk queue  
    uchar data[BSIZE]; // data 512 bytes  
};
```

## buffer cache:

**static struct buf\* bget(uint dev, uint blockno)**

- The bcache.head list is maintained on Most Recently Used (MRU) basis
  - head.next is the Most Recently Used (MRU) buffer
  - hence head.prev is the Least Recently Used (LRU)
- Look for a buffer with **b->blockno = blockno** and **b->dev = dev**
  - Search the head.next list for existing buffer (MRU)

**buffer cache:**

**struct buf\* bread(uint dev, uint blockno)**

**struct buf\***

**bread(uint dev, uint  
blockno)**

**{**

**struct buf \*b;**

**b = bget(dev, blockno); panic("bwrite");**

**void**

**bwrite(struct buf \*b)**

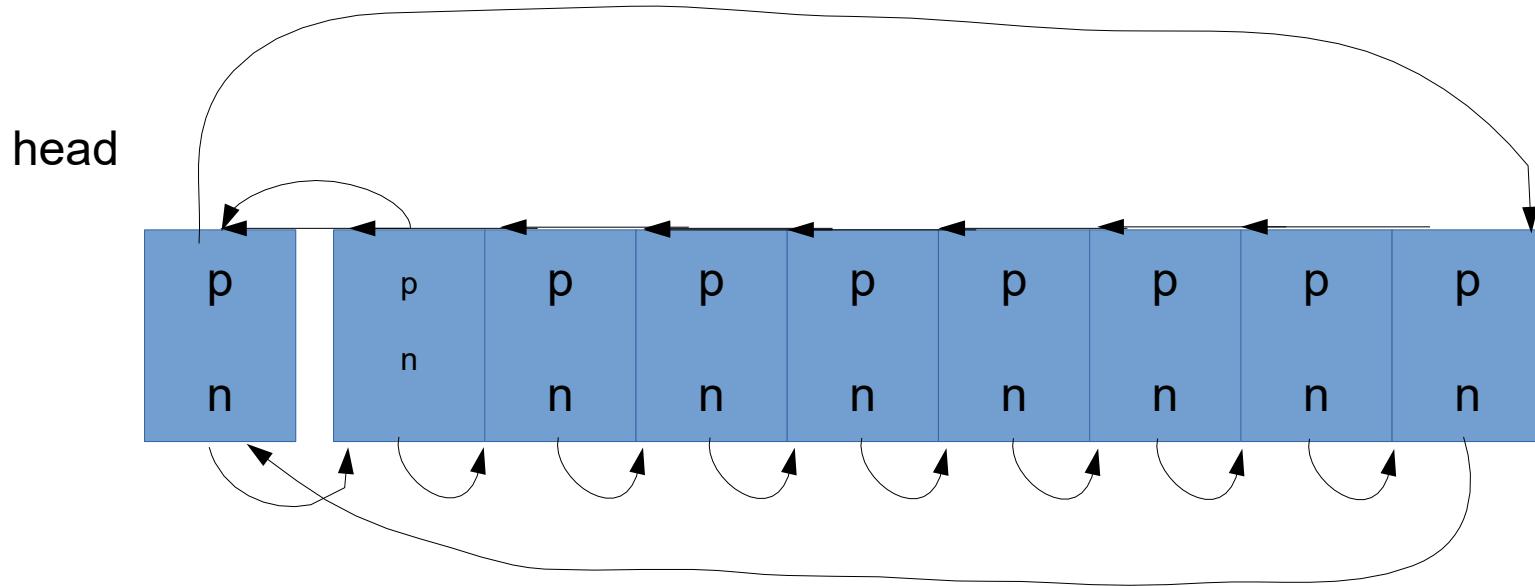
**{**

**if(!holdingsleep(&b->lock))**

## buffer cache: **void brelse(struct buf \*b)**

- **release lock on buffer**
- **b->refcnt = 0**
- **If b->refcnt = 0**
  - Means buffer will no longer be used
  - Move it to **front** of the front of **bcache.head**

# Overall in this diagram



Buffers keep moving to the front of the list and around  
The list always contains **NBLIF=30** buffers

# Let's see logging layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipellem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> ,
Logging	Block allocation on disk: <b>balloc</b> , <b>bfree</b> <b>log.c</b> : <b>begin_op</b> , <b>end_op</b> , <b>initlog</b> , <b>commit</b> ,
Buffer cache	

# log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
  - some writes succeed and some don’t
  - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an

# log in xv6

- **xv6 system call does not directly write the on-disk file system data structures.**
- **A system call calls begin\_op() at beginning and end\_op() at end**
  - begin\_op() increments log.outstanding
  - end\_op() decrements log.outstanding, and if it's 0, then calls commit()

# log

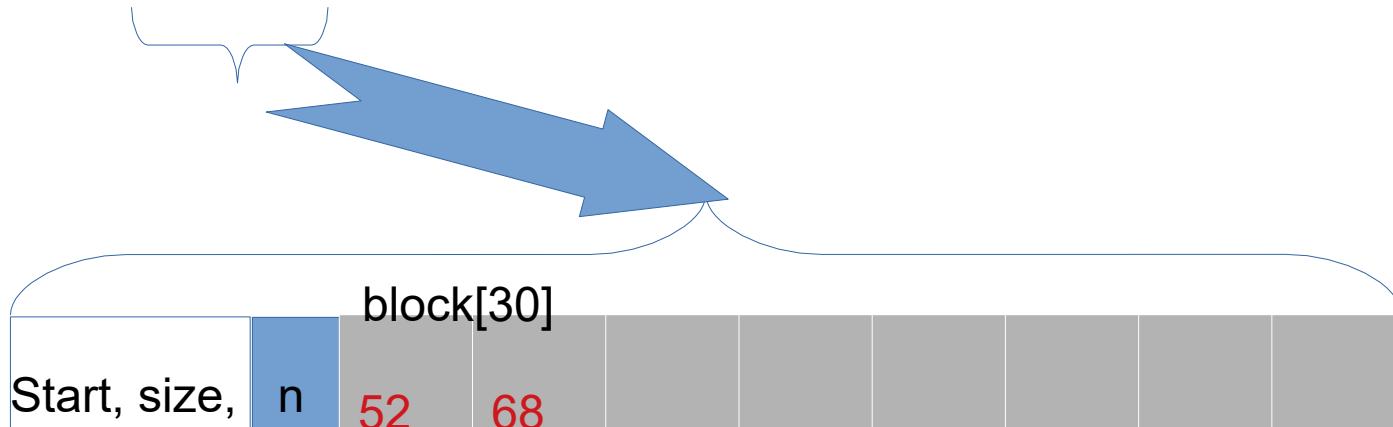
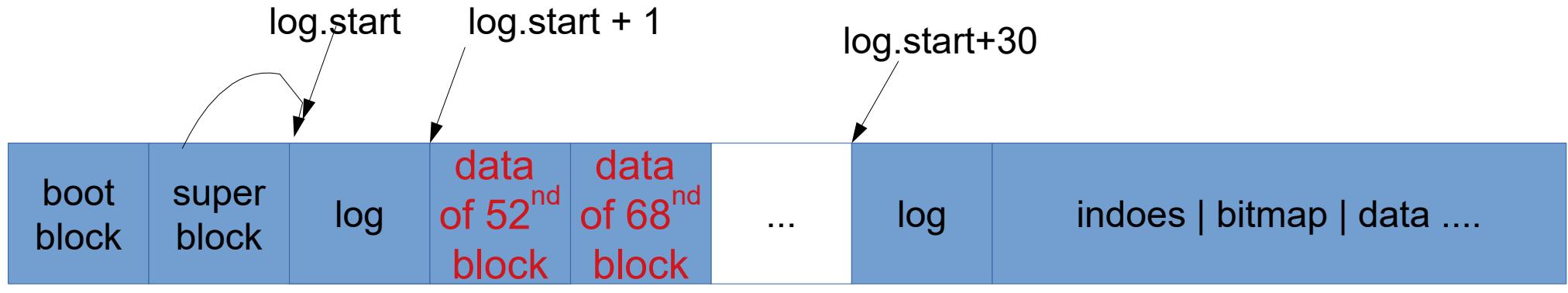
```
struct logheader { // ON DISK
    int n; // number of entries in use in block[]
below

    int block[LOGSIZE]; // List of block numbers
stored

};

struct log { // only in memory
```

# log on disk



# Typical use case of logging

```
/* In a system call  
code */  
begin_op();  
...  
bp = bread(...);  
bp->data[...] = ...;
```

prepare for logging.  
Wait if logging system  
is not ready or  
'committing'.  
**++outstanding**

read and get access  
to a data block – as a  
buffer

# Example of calls to logging

```
//file_write() code  
begin_op();  
ilock(f->ip);  
/*loop */ r = writei(f->ip, ...);  
iunlock(f->ip);
```

- each writei() in turn calls bread(), log\_write() and brelse()
- also calls iupdate(ip) which also calls bread, log\_write and brelse

# Logging functions

- **Initlog()**
  - Set fields in global **log.xyz** variables, using FS superblock
  - Recovery if needed
  - Called from first forkret ()
- **write\_log(void)**
  - Called only from commit ()
  - Use block numbers specified in **log.lh.block** and copy those blocks from memory to log-blocks

Following three

# Let's see block allocation layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipellem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> ,
Logging	<b>Block allocation on disk: balloc, bfree</b> <b>log.c</b> : begin_op, end_op, initlog, commit,
Buffer cache	<b>bio.c</b> binit, bget, bread, bwrite, brelse

# allocating & deallocating blocks on DISK

- **balloc(devno)**
  - looks for a block whose bitmap bit is zero, indicating that it is free.
  - On finding updates the bitmap and returns the block.
- **bfree(devno,  
blockno)**
  - finds the right bitmap block and clears the right bit.
  - Also calls log\_write()

# Let's see Inode Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipellem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	
Buffer cache	Block allocation on disk: balloc, bfree <b>log.c</b> : begin op, end op, initlog, commit,

# On disk & in memory inodes

```
struct {  
    struct spinlock  
    lock;  
  
    struct inode  
    inode[NINODE];
```

// in-memory copy of  
an inode

```
struct inode {  
    uint dev; // Device  
    number
```

```
    uint inum; // Inode  
    number
```

# In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
  - as long as ‘ref’ is >0
- The **iget** and **iput** functions acquire and release pointers
- See the caller graph of **iget()**
  - all those who call **iget()**
- Sleep lock in ‘inode’ protects
  - fields in inode

# iget and iupdate

- **iget**

- searches for an existing/free inode in icache and returns pointer to one
- if found, increments ref and returns pointer to inode

- **iupdate(inode \*ip)**

- read on disk block of inode
- get on disk inode
- modify it as specified in ‘ip’
- modify disk block of inode

# itrunc , iput

- **iput(ip)**
  - if ref is 1
    - **itrunc(ip)**
    - type = 0
    - **iupdate(ip)**
    - **i->valid = 0 // free in memory**
  - else
- **itrunc(ip)**
  - write all data blocks of inode to disk
    - **using bfree()**
  - **ip->size = 0**
    - **Inode is freed from use**
  - **iupdate(ip)**
    - **called from iput() only**

# race in iput ?

- A concurrent thread might be waiting in ilock to use this inode

- and won't be prepared to find the inode is not longer allocated

```
void  
iput(struct inode *ip)  
{  
    acquireSleep(&ip->lock);  
    if(ip->valid && ip->nlink == 0)  
        ip->valid = 0;  
    else if(ip->nlink > 0)  
        ip->nlink--;  
    else  
        ip->valid = 0;  
    if(ip->valid == 0)  
        ip->list.next = ip->list.prev = ip;  
    else  
        ip->list.next = ip->list.prev = ip->parent;  
    ip->parent->list.next = ip->parent->list.prev = ip;  
}
```

- This is not possible.

# buffer and inode cache

- to read an **inode**, its block must be read in a buffer
- So the buffer always contains a copy of the on-disk **dinode**
  - duplicate copy in in-memory **inode**
- The **inode cache** is write-through,
  - code that modifies a cached inode must immediately write it to disk with **iupdate**
- Inode may still exist in the buffer cache

# allocating inode

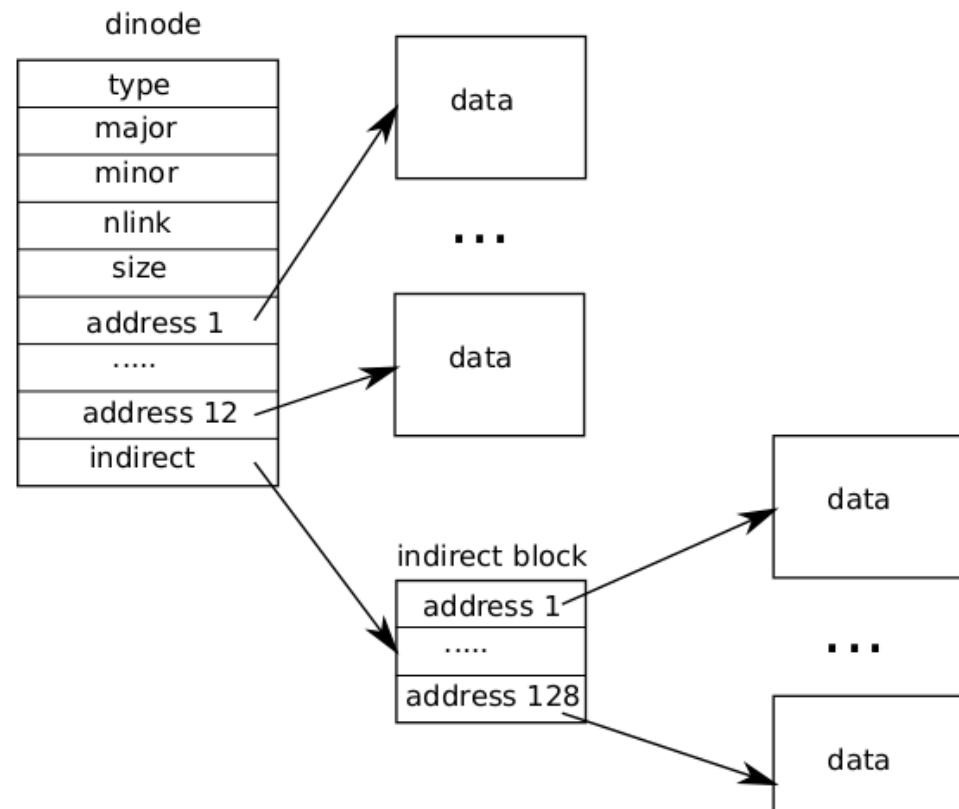
- **ialloc(dev, type)**
  - Loop over all disk inodes
  - read inode (from its block)
  - if it's free (note inum)
  - zero on disk inode
- **ilock**
  - code must acquire ilock before using inode's data/fields
  - **Ilock reads inode if it's already not in memory**

# Trouble with iput() and crashes

- **iput() doesn't truncate a file immediately when the link count for the file drops to zero, because**
  - some process might still hold a reference
- **if a crash happens before the last process closes the file descriptor for the file,**
  - then the file will be marked allocated on disk but no directory

# Get Inode data: bmap(ip, bn)

- **Allocate ‘bn’th block for the file given by inode ‘ip’**
- **Allocate block on disk and store it in either direct entries or block of indirect entries**



# writing/reading data at a given offset in file

```
readi(struct inode *ip,  
char *dst, uint off, uint  
n)
```

```
writei(struct inode *ip,  
char *src, uint off, uint  
n)
```

- Calculate the block number in file where ‘off’ belongs
- Read sufficient blocks to read ‘n’ bytes
- using bread(), brelse()

# Reading Directory Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipelem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> ,
Logging	Block allocation on disk: <b>balloc</b> , <b>bfree</b>
Buffer cache	<b>log.c</b> : begin_op, end_op, initlog, commit, <b>bio.c</b> binit, bget, bread, bwrite, brelse

# directory entry

```
#define DIRSIZ 14
```

```
struct dirent {  
    ushort inum;  
    char name[DIRSIZ];  
};
```

**struct inode\***  
**dirlookup(struct inode \*dp, char \*name, uint \*poff)**

- **Given a pointer to directory inode (dp), name of file to be searched**
  - return the pointer to inode of that file (NULL if not found)
  - set the ‘offset’ of the entry found, inside directories data blocks, in poff
- **How was ‘dp’ obtained? Who should be calling dirlookup? Why is poff returned?**

**int**

**dirlink(struct inode \*dp, char \*name, uint inum)**

- **Create a new entry for ‘name’\_’inum’ in directory given by ‘dp’**
  - inode number must have been obtained before calling this. How to do that?
- **Use dirlookup() to verify entry does not exist!**
- **Get empty slot in directory’s data block**

# namex

- **Called by namei(), or nameiparent()**
- **Just iteratively split a path using “/” separator and get inode for last component**
- **iget() root inode, then**
- **Repeatedly calls**
  - split on “/”, dirlookup() for next component

# races in namex()

- **Crucial. Called so many times!**
- **one kernel thread is looking up a pathname  
another kernel thread may be changing the  
directory by calling unlink**
  - when executing dirlookup in namex, the lookup  
thread holds the lock on the directory and  
dirlookup() returns an inode that was obtained using  
iget.

# File descriptor layer code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	<b>file.c</b> fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	<b>fs.c</b> namex, namei, nameiparent, skipellem
Directory	<b>fs.c</b> dirlookup, dirlink
Inode	<b>fs.c</b> iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, <b>bmap</b> ,
Logging	Block allocation on disk: <b>ballo</b> , <b>bfree</b>
Buffer cache	<b>log.c</b> : begin_op, end_op, initlog, commit, <b>bio.c</b> binit, bget, bread, bwrite, brelse

# data structures related to “file” layer

```
struct file {  
    enum { FD_NONE,  
          FD_PIPE, FD_INODE }  
    type;  
    int ref; // reference  
    count  
    char readable;
```

```
struct proc {  
    ...  
    struct file  
    *ofile[NOFILE]; //Open  
    files per process  
    ...  
    }
```

# Multiple processes accessing same file.

- **Each will get a different ‘struct file’**
  - but share the inode !
  - different offset in struct file, for each process
  - Also true, if same process opens file many times
- **File can be a PIPE (more later)**
  - what about STDIN, STDOUT, STDERR files ?
  - Figure out!

# file layer functions

- **filealloc**

- find an empty struct file in ‘ftable’ and return it
- set ref = 1

- **filedup(file \*)**

- simply ref++

- **fileclose**

- --ref
- if ref = 0
  - free struct file
  - input() / pipeclose()
  - note – transaction if input() called

- **filestat**

# file layer functions

- **fileread**
  - call readi() or piperead()
  - readi() later calls device-read or inode read (using bread())
- **filewrite**
  - call pipewrite() or writei()
- **Why does readi() call read on the device , why not fileread() itself call device read ?**

# pipes

```
struct pipe {  
    struct spinlock lock;  
    char data[PIPE_SIZE];  
    uint nread;  
    // number of bytes  
    read
```

- functions

- pipealloc
- pipeclose
- piperead
- pipewrite

# pipes

- **pipealloc**

- allocate two struct file
- allocate pipe itself using kalloc (it's a big structure with array)
- init lock
- initialize both struct file as 2 ends (r/w)

- **pipewrite**

- wait if pipe full
- write to pipe
- wakeup processes waiting to read

- **piperead**

- wait if no data

# **Further to reading system call code now**

- **Now we are ready to read the code of system calls on file system**
  - sys\_open, sys\_write, sys\_read , etc.
- **Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.**

# Synchronization

# My formulation

- **OS = data structures + synchronization**
- **Synchronization problems make writing OS code challenging**
- **Demand exceptional coding skills**

# Race problem

```
long c = 0, c1 = 0, c2 =    int main() {  
0, run = 1;  
  
void *thread1(void  
*arg) {  
  
while(run == 1) {  
c++;  
  
c1++;  
  
pthread_t th1, th2;  
pthread_create(&th1,  
NULL, thread1, NULL);  
pthread_create(&th2,  
NULL, thread2, NULL);  
  
//fprintf(stderr,
```

# Race problem

- **On earlier slide**
  - Value of c should be equal to  $c_1 + c_2$ , but it is not!
  - Why?
- **There is a “race” between thread1 and thread2 for updating the variable c**
- **thread1 and thread2 may get scheduled in any order and *interrupted* any point in time**

# Race problem

- C++, when converted to assembly code, could be
  - Now following sequence of instructions is possible among thread1 and thread2
  - What will be value in c, if initially c was, say 5?
    - It will be 6, when it is expected to be 7. Other

# Races: reasons

- **Interruptible kernel**

- If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete
- This introduces concurrency

- **Multiprocessor systems**

- On SMP systems: memory is shared, kernel and process code run on all processors

# Critical Section Problem

- Consider system of n processes {p<sub>0</sub>, p<sub>1</sub>, ..., p<sub>n-1</sub>}
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section

# Critical Section problem

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

# Expected solution characteristics

- **1. Mutual Exclusion**

- If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections

- **2. Progress**

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes

# suggested solution - 1

```
int flag = 1;
```

```
void *thread1(void  
*arg) {
```

```
while(run == 1) {
```

```
    while(flag == 0)
```

```
;
```

- What's wrong here?
- Assumes that

# suggested solution - 2

```
int flag = 0;
```

```
void *thread1(void  
*arg) {
```

```
while(run == 1) {
```

```
if(flag)
```

```
c++;
```

```
void *thread2(void  
*arg) {
```

```
while(run == 1) {
```

```
if(!flag)
```

```
c++;
```

```
else
```

# Peterson's solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
- The variable turn indicates whose turn it is to enter the critical section

# Peterson's solution

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j)  
        ;  
    critical section  
    flag[i] = FALSE;  
    remainder section
```

# Hardware solution – the one actually implemented

- **Many systems provide hardware support for critical section code**
- **Uniprocessors – could disable interrupts**
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable

# Solution using test-and-set

```
lock = false; //global  
  
do {  
    while ( TestAndSet (&lock ) )  
        ; // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Definition:

```
boolean TestAndSet (boolean  
*target)  
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;
```

# Solution using swap

```
lock = false; //global  
  
do {  
    key = true  
    while ( key == true))  
        swap(&lock, &key)  
        // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

# Spinlock

- A lock implemented to do ‘busy-wait’
- Using instructions like T&S or Swap
- As shown on earlier slides
- `spinlock(int *lock){`
- `}`
- `spinunlock(lock *lock) {`

# Bounded wait M.E. with T&S

```
do {  
    waiting[i] = TRUE;  
    key = TRUE;  
    while (waiting[i] && key)  
        key = TestAndSet(&lock);  
    waiting[i] = FALSE;  
    // critical section  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = FALSE;
```

# Some thumb-rules of spinlocks

- **Never block a process holding a spinlock !**
- **Typical code:**
- **Hold a spin lock for only a short duration of time**
  - Spinlocks are preferable on multiprocessor systems
  - Cost of context switch is a concern in case of sleep-wait locks

# **sleep-locks**

- **Spin locks result in busy-wait**
- **CPU cycles wasted by waiting processes/threads**
- **Solution – threads keep waiting for the lock to be available**
  - Move thread to wait queue
  - The thread holding the lock will wake up one of

# Sleep locks/mutexes

```
//ignore syntactical  
issues  
typedef struct mutex  
{  
wait(mutex *m) {  
  
}  
  
Block(mutex *m,  
spinlock *sl) {  
  
}  
  
release(mutex *m) {
```

# Locks in xv6 code

# **struct spinlock**

**// Mutual exclusion lock.**

**struct spinlock {**

**uint locked; // Is the lock held?**

**// For debugging:**

**char \*name; // Name of lock.**

# spinlocks in xv6 code

```
struct {  
    struct spinlock lock;  
    struct buf buf[NBUF];  
    struct buf head;  
} bcache;  
  
struct {  
    struct spinlock lock;
```

```
    static struct spinlock  
    idelock;  
  
    struct {  
        struct spinlock lock;  
        int use_lock;  
        struct run *freelist;  
    } kmem;  
  
    struct log {
```

```
static inline uint
xchg(volatile uint
*addr, uint newval)
{
    uint result;
    // The + in "+m"
    // denotes a read-
    // modify-write operand.
    asm volatile("lock;
        xchgl %0,%1"
        : "=r" (result)
        : "r" (newval));
}
```

## Spinlock on xv6

```
void acquire(struct
spinlock *lk)
{
    pushcli(); // disable
    // interrupts to avoid
    // deadlock.
    // The xchg is atomic.
```

```
Void acquire(struct  
spinlock *lk)  
{  
    pushcli(); // disable  
interrupts to avoid  
deadlock.  
  
if(holding(lk))  
panic("acquire");  
.....
```

## spinlocks

- Pushcli() - disable interrupts on that processor
- One after another many acquire() can be called on different spinlocks

Keep a count of them

```
void  
release(struct spinlock  
*lk)  
{  
...  
asm volatile("movl $0,  
%0" : "+m" (lk-  
>locked) : );  
popcli();
```

## spinlocks

- **Popcli()**

- Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called

# spinlocks

- **Always disable interrupts while acquiring spinlock**
  - Suppose **iderw** held the idelock and then got interrupted to run **ideintr**.
  - **Ideintr** would try to lock **idelock**, see it was held, and wait for it to be released.
  - In this situation, idelock will never be released

# sleeplocks

- **Sleeplocks don't spin. They move a process to a wait-queue if the lock can't be acquired**
- **XV6 approach to “wait-queues”**
  - Any memory address serves as a “wait channel”
  - The sleep() and wakeup() functions just use that address as a ‘condition’
  - There are no per condition process queues! Just shared ones

**void** **sleep(void \*chan,**

**sleep()**

**struct spinlock \*lk)**

**{**

**struct proc \*p =**  
**myproc();**

**....**

**if(lk != &ptable.lock){**

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched() hold

# Calls to sleep() : examples of “chan” (output from cscope)

0 console.c

consoleread 251

sleep(&input.r, &cons.lock);

2 ide.c iderw

169 sleep(b, &idelock);

3 log.c begin\_op

131 sleep(&log, &log.lock);

6 pipe.c piperead

111 sleep(&p->nread, &p-

7 proc.c wait 317

sleep(curproc,  
&phtable.lock);

8 sleeplock.c

acquiresleep 28

sleep(lk, &lk->lk);

9 sysproc.c sys\_sleep

```
void wakeup(void  
*chan)  
{  
    acquire(&ptable.lock);  
    wakeup1(chan);  
    release(&ptable.lock);  
}  
  
static void  
wakeup1(void *chan)
```

## Wakeup()

- Acquire ptable.lock since you are going to change ptable and p-> values
- just linear search in process table for a process where p->chan is given

# sleeplock

// Long-term locks for processes

```
struct sleeplock {
```

```
    uint locked; // Is the lock held?
```

```
    struct spinlock lk; // spinlock protecting this  
    sleep lock
```

# Sleeplock acquire and release

```
void  
acquiresleep(struct  
sleeplock *lk)  
{  
    acquire(&lk->lk);  
    while (lk->locked) {
```

```
void  
releasesleep(struct  
sleeplock *lk)  
{  
    acquire(&lk->lk);  
    lk->locked = 0;
```

# Where are sleeplocks used?

- **struct buf**
  - waiting for I/O on this buffer
- **struct inode**
  - waiting for I/o to this inode
- Just two !

# Sleeplocks issues

- **sleep-locks support yielding the processor during their critical sections.**
- **This property poses a design challenge:**
  - if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),
  - and thread T2 wishes to acquire L1,
  - we have to ensure that T1 can execute

# More needs of synchronization

- Not only critical section problems
- Run processes in a particular order
- Allow multiple processes read access, but  
only one process write access
- Etc.

# Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S
  - Can only be accessed via two indivisible (atomic) operations
  - **wait (S) {**
  - **while S <= 0**
  - **; // no-op**

# Semaphore for synchronization

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion
- **Semaphore mutex;** // initialized to 1
- **do {**

# Semaphore implementation

```
Wait(sem *s) {  
    while(s <=0)  
        block(); // could  
be ";"  
    s--;  
}
```

- Left side – expected behaviour
- Both the wait and signal should be atomic.
- This is the semantics of the semaphore.

# Semaphore implementation? - 1

```
struct semaphore {          signal(semaphore *s) {  
    int val;  
    spinlock lk;  
};  
  
sem_init(semaphore  
*s, int initval) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}  
- suppose 2 processes
```

# Semaphore implementation? - 2

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
  
sem_init(semaphore  
*s, int initval) {
```

```
    wait(semaphore *s) {  
        spinlock(&(s->sl));  
        while(s->val <=0) {  
            spinunlock(&(s-  
                >sl));  
            spinlock(&(s->sl));  
        }  
        s->val++;  
    }  
}
```

# Semaphore implementation? - 3, idea

```
struct semaphore {          wait(semaphore *s) {  
    int val;  
    spinlock lk;  
};  
  
sem_init(semaphore  
*s, int initval) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        Block();  
    }  
    (s->val)--;
```

# Semaphore implementation? - 3a

```
struct semaphore {          wait(semaphore *s) {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore
```

```
        spinlock(&(s->sl));  
        while(s->val <=0) {  
            spinunlock(&(s->sl));  
            block(s);
```

# Semaphore implementation? - 3b

```
struct semaphore {          wait(semaphore *s) {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore           spinlock(&(s->sl));  
         *s) {  
    s->val = 0;  
    s->lk = spinlock_create();  
    s->l = list_create();  
}  
int sem_wait(semaphore *s) {  
    spinlock lk;  
    list l;  
    if (s->val <= 0) {  
        block(s);  
    } else {  
        s->val--;  
    }  
}
```

# Semaphore implementation? - 3c

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl)); // A  
    while(s->val <=0) {  
        block(s);  
        spinlock(&(s->sl)); // B
```

# Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have busy waiting in critical section implementation

# Semaphore in Linux

```
struct semaphore {  
    raw_spinlock_t lock;  
    unsigned int count;  
    struct list_head wait_list;  
};  
  
static noinline void __sched  
__down(struct semaphore  
*sem)
```

```
void down(struct semaphore  
*sem)  
{  
    unsigned long flags;  
  
    raw_spin_lock_irqsave(&sem->lock, flags);  
    if (likely(sem->count > 0))
```

# Semaphore in Linux

```
static inline int __sched
__down_common(struct
semaphore *sem, long state,
long timeout)
{
    struct task_struct *task =
current;
    struct semaphore_waiter
waiter;
```

```
for (;;) {
    if
(signal_pending_state(state,
task))
        goto interrupted;
    if (unlikely(timeout <= 0))
        goto timed_out;
    __set_task_state(task, state);
row_spin_unlock_irq(8.com
```

# **Different uses of semaphores**

# For mutual exclusion

**/\*During initialization\*/**

**semaphore sem;**

**initsem (&sem, 1);**

**/\* On each use\*/**

**P (&sem);**

# Event-wait

```
/* During initialization */  
semaphore event;  
initsem (&event, 0); /* probably at boot time */  
  
/* Code executed by thread that must wait on  
event */
```

# Control countable resources

```
/* During initialization */  
semaphore counter;  
initsem (&counter, resourceCount);  
  
/* Code executed to use the resource */  
P (&counter); /* Blocks until resource is  
available */
```

# Drawbacks of semaphores

- Need to be implemented using lower level primitives like spinlocks
- Context-switch is involved in blocking and signaling – time consuming
- Can not be used for a short critical section

# **Deadlocks**

# Deadlock

- two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

□

P0

□

wait (S);

P1

wait (Q);

# Example of deadlock

- Let's see the pthreads program : `deadlock.c`
- Same program as on earlier slide, but with `pthread_mutex_lock();`

# Non-deadlock, but similar situations

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion**
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the

# Livelock

- **Similar to deadlock, but processes keep doing ‘useless work’**
- **E.g. two people meet in a corridor opposite each other**
  - Both move to left at same time
  - Then both move to right at same time
  - Keep Repeating!

# Livelock example

```
#include <stdio.h>
#include <pthread.h>

struct person {
    int otherid;
    int otherHungry;
    int myid;
    /* thread two runs in
       this function */
    int spoonWith = 1;
    void *eat(void *param)
    {
        int eaten = 0;
```

# More on deadlocks

- Under which conditions they can occur?
- How can deadlocks be avoided/prevented?
- How can a system recover if there is a deadlock ?

# System model for understanding deadlocks

- **System consists of resources**
- **Resource types R<sub>1</sub>, R<sub>2</sub>, . . . , R<sub>m</sub>**
  - CPU cycles, memory space, I/O devices
  - Resource: Most typically a lock, synchronization primitive
- **Each resource type R<sub>i</sub> has W<sub>i</sub> instances.**
- **Each process utilizes a resource as follows:**

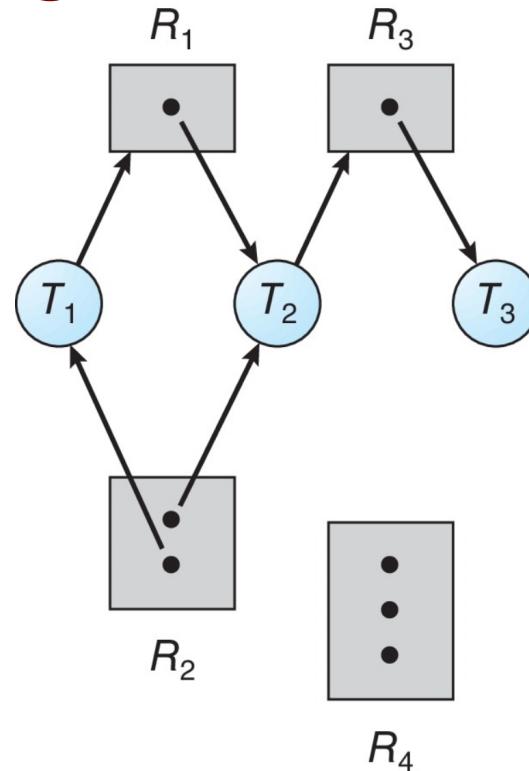
# Deadlock characterisation

- **Deadlock is possible only if ALL of these conditions are TRUE at the same time**
  - Mutual exclusion:only one process at a time can use a resource
  - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption:a resource can be released only

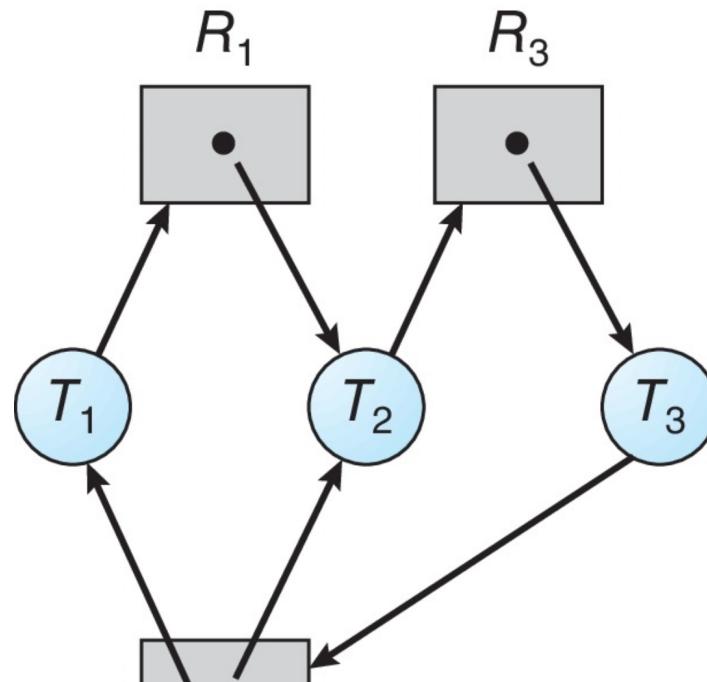
# Resource Allocation Graph

## Example

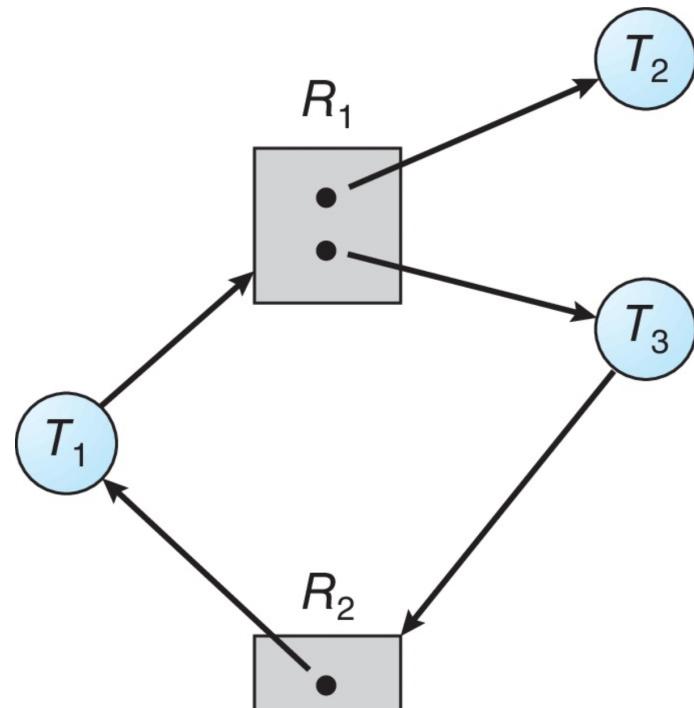
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an



# Resource Allocation Graph with a Deadlock



# Graph with a Cycle But no Deadlock



# Basic Facts

- **If graph contains no cycles -> no deadlock**
- **If graph contains a cycle :**
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

# Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
- 3) Allow the system to enter a deadlock state and then recover
- 4) Ignore the problem and pretend that deadlocks never occur in the system.

# (1) Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a process requests a resource, it

# (1) Deadlock Prevention (Cont.)

- **No Preemption:**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its

# (1) Deadlock prevention: Circular Wait

- **In invalidating the circular wait condition is most common.**
- **Simply assign each resource (i.e., mutex locks) a unique number**

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
```

# (1) Preventing deadlock: cyclic wait

- **Locking hierarchy : Highly preferred technique in kernels**
  - Decide an ordering among all ‘locks’
  - Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!
  - Poses coding challenges!
  - A key differentiating factor in kernels

# (1) Prevention in Xv6: Lock Ordering

- lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, `idelock`, and `ptable.lock`.

## (2) Deadlock avoidance

- **Requires that the system has some additional a priori information available**
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

## (2) Deadlock avoidance

- Please see: concept of safe states, unsafe states, Banker's algorithm

# (3) Deadlock detection and recovery

- How to detect a deadlock in the system?
- The Resource-Allocation Graph is a graph.  
Need an algorithm to detect cycle in a graph.
- How to recover?
  - Abort all processes or abort one by one?
  - Which processes to abort?
    - Priority ?

# **“Condition” Synchronization Tool**

# What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

**Struct condition {**

**Proc \*next**

**Proc \*prev**

# Code for condition variables

```
//Spinlock s is held  
before calling wait
```

```
void wait (condition  
*c, spinlock_t *s)
```

```
(
```

```
spin_lock (&c-  
>listLock);
```

```
void do_signal  
(condition *c)
```

```
/*Wakeup one thread  
waiting on  
the condition*/
```

```
{
```

```
spin_lock (&c-
```

# Semaphore implementation using condition variables?

- Is this possible?
- Can we try it?

```
typedef struct semaphore {
```

```
//something
```

```
condition c;
```

```
}semaphore;
```

# **Classical Synchronization Problems**

# Bounded-Buffer Problem

- **Producer and consumer processes**
  - N buffers, each can hold one item
- **Producer produces ‘items’ to be consumed by consumer , in the bounded buffer**
- **Consumer should wait if there are no items**
- **Producer should wait if the ‘bounded buffer’ is full**

# Bounded-Buffer Problem: solution with semaphores

- **Semaphore mutex initialized to the value 1**
- **Semaphore full initialized to the value 0**
- **Semaphore empty initialized to the value N**

# Bounded-buffer problem

The structure of the producer process

```
do {  
    // produce an item in nextp  
    wait (empty);  
    wait (mutex);  
    ...  
    signal (mutex);  
    signal (full);  
}
```

The structure of the Consumer process

```
do {  
    wait (full);  
    wait (mutex);  
    ...  
    signal (mutex);  
    signal (empty);  
}
```

# Bounded buffer problem

- Example : pipe()
- Let's see code of pipe in xv6 – a solution using sleeplocks

# Readers-Writers problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time

## The structure of a writer process

```
do {  
    wait (wrt) ;  
    // writing is performed  
    signal (wrt) ;  
} while (TRUE);
```

## The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1)  
        wait (wrt) ;  
    signal (mutex)  
    // reading is performed  
    wait (mutex) ;  
    readcount - - ;  
    if (readcount == 0)
```

# **Readers-Writers Problem Variations**

- **First variation – no reader kept waiting unless writer has permission to use shared object**
- **Second variation – once writer is ready, it performs write asap**
- **Both may have starvation leading to even more variations**

# Reader-write lock

- A lock with following operations on it
  - Lockshared()
  - Unlockshared()
  - LockExcl()
  - UnlockExcl()
- Possible additions
  - Downgrade() -> from excl to shared

# Code for reader-writer locks

```
struct rwlock {  
    int nActive; /* num of  
active readers, or -1 if a  
writer is active */  
  
    int nPendingReads;  
  
    int nPendingWrites;  
  
    spinlock_t sl;
```

```
void lockShared  
(struct rwlock *r)  
{  
    spin_lock(&r->sl);  
  
    r->nPendingReads++;  
  
    if (r->nPendingWrites  
        > 0)
```

# Code for reader-writer locks

```
void unlockShared  
(struct rwlock *r)  
  
{  
    spin_lock (&r->sl);  
  
    r->nActive--;  
  
    if (r->nActive == 0) {
```

```
void lockExclusive  
(struct rwlock *r)  
  
(  
    spin_lock (&r->sl);  
  
    r->nPendingWrltes++;  
  
    while (r->nActive)
```

# Code for reader-writer locks

```
void unlockExclusive  
(struct rwlock *r){  
  
    boolean t  
    wakeReaders;  
  
    spin_lock (&r->sl);  
  
    r->nActive = 0;  
  
    wakeReaders = (r-
```

Try writing code for  
downgrade and  
upgrade

Try writing a reader-  
writer lock using  
semaphores!

# Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat



# Dining philosophers: One solution

The structure of Philosopher i:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
    // eat  
    signal ( chopstick[i] );
```

# Dining philosophers: Possible approaches

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - to do this, she must pick them up in a critical section

# Other solutions to dining philosopher's problem

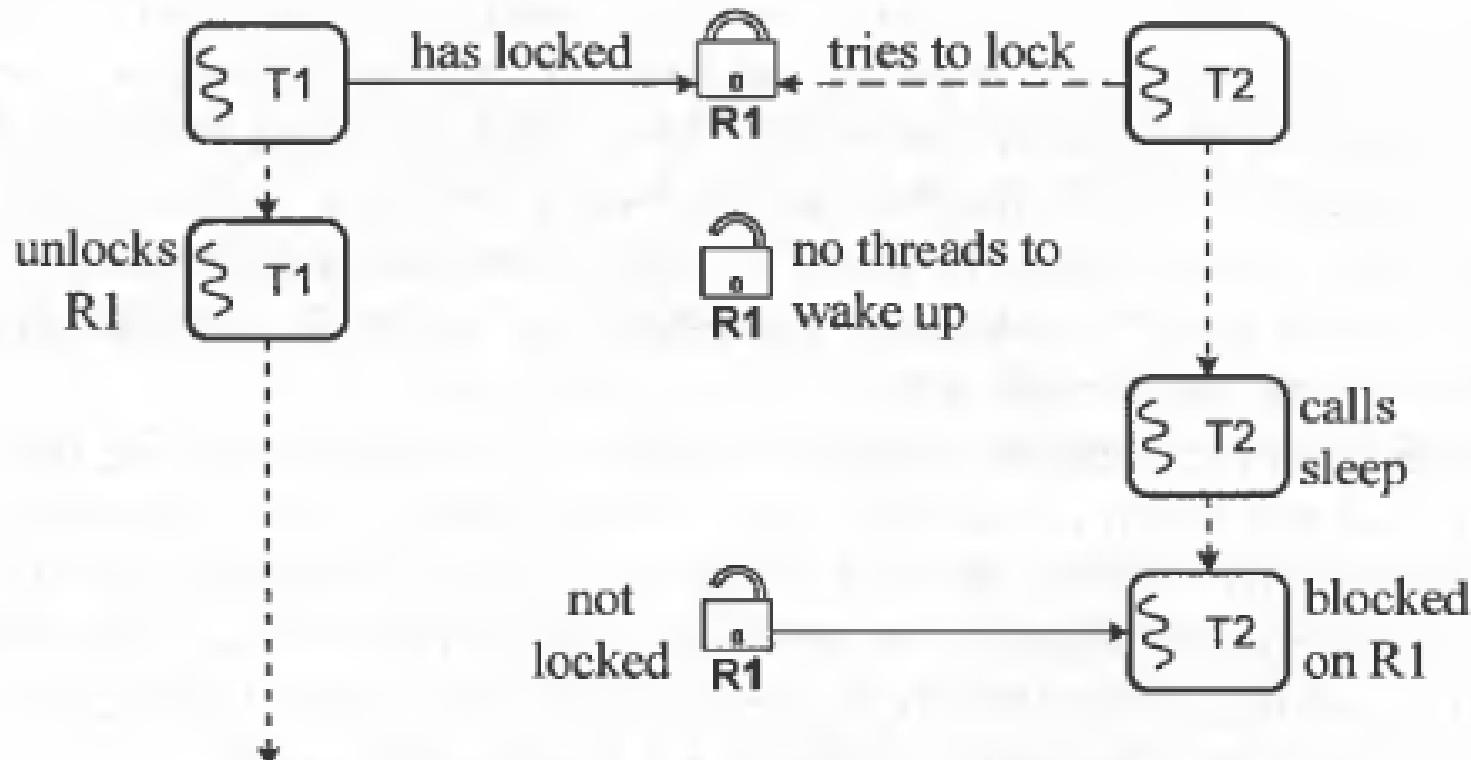
- Using higher level synchronization primitives like 'monitors'

# **Practical Problems**

# Lost Wakeup problem

- **The sleep/wakeup mechanism does not function correctly on a multiprocessor.**
- **Consider a potential race:**
  - Thread T1 has locked a resource R1.
  - Thread T2, running on another processor, tries to acquire the resource, and finds it locked.
  - T2 calls sleep() to wait for the resource.

# Lost Wakeup problem



# Thundering herd problem

- **Thundering Herd problem**
  - On a multiprocessor, if several threads were locked the resource
  - Waking them all may cause them to be simultaneously scheduled on different processors
  - and they would all fight for the same resource again.

## Starvation

# **Case Studies**

# Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks

# Linux Synchronization

- **Atomic variables**
- **Consider the variables**

# Pthreads synchronization

- **Pthreads API is OS-independent**
- **It provides:**
  - mutex locks
  - condition variables
- **Non-portable extensions include:**
  - read-write locks
  - spinlocks

# **Synchronization issues in xv6 kernel**

# Difference approaches

- **Pros and Cons of locks**
  - Locks ensure serialization
  - Locks consume time !
- **Solution – 1**
  - One big kernel lock
  - Too inefficient
- **Solution – 2**

# Three types of code

- **System calls code**
  - Can it be interruptible?
  - If yes, when?
- **Interrupt handler code**
  - Disable interrupts during interrupt handling or not?
  - Deadlock with iderw ! - already seen
- **Process's user code**

# Interrupts enabling/disabling in xv6

- **Holding every spinlock disables interrupts!**
- **System call code or Interrupt handler code won't be interrupted if**
  - The code path followed took at least once spinlock !
  - Interrupts disabled only on that processor!
- **Acquire calls pushcli() before xchg()**
- **Release calls popcli() after xchg()**

# Memory ordering

- Compiler may generate machine code for out-of-order execution !
- Processor pipelines can also do the same!
- Consider this
  - 1)l = malloc(sizeof \*l);
  - 2)l->data = data;
  - 3)acquire(&listlock);
  - 4)l->next = list;
  - 5)list = l;
  - 6)release(&listlock);

# Lost Wakeup?

- **Do we have this problem in xv6?**
- **Let's analyze again!**
  - The race in acquiresleep()'s call to sleep() and releasesleep()
- **T1 holding lock, T2 willing to acquire lock**
  - Both running on different processor
  - Or both running on same processor

# Code of sleep()

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

- Why this check?
- Deadlock otherwise!

# **Exercise question : 1**

**Sleep has to check lk != &ptable.lock to avoid a deadlock**

**Suppose the special case were eliminated by replacing**

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);
```

# bget() problem

- **bget() panics if no free buffers!**
- **Quite bad**
- **Should sleep !**
- **But that will introduce many deadlock problems. Which ones ?**

# **iget() and ilock()**

- **iget() does no hold lock on inode**
- **ilock() does**
- **Why this separation?**
  - Performance? If you want only “read” the inode, then why lock it?
- **What if iget() returned the inode locked?**

# Interesting cases in namex()

Xv6

Interesting case of holding and releasing  
ptable.lock in scheduling

**One process acquires, another releases!**

# Giving up CPU

- **A process that wants to give up the CPU**
  - must acquire the process table lock ptable.lock
  - release any other locks it is holding
  - update its own state (proc->state),
  - and then call sched()
- **Yield follows this convention, as do sleep and exit**

# Interesting race if ptable.lock is not held

- Suppose P1 calls yield()
- Suppose yield() does not take ptable.lock
  - Remember yield() is for a process to give up CPU
- Yield sets process state of P1 to RUNNABLE
- Before yield's sched() calls swtch()
- Another processor runs scheduler() and sets P1 to that processor

# Homework

- **Read the version-11 textbook of xv6**
- **Solve the exercises!**

# Scheduling

Abhijit A.M.

[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

Credits: Slides from os-book.com

# Necessity of scheduling

## Multiprogramming

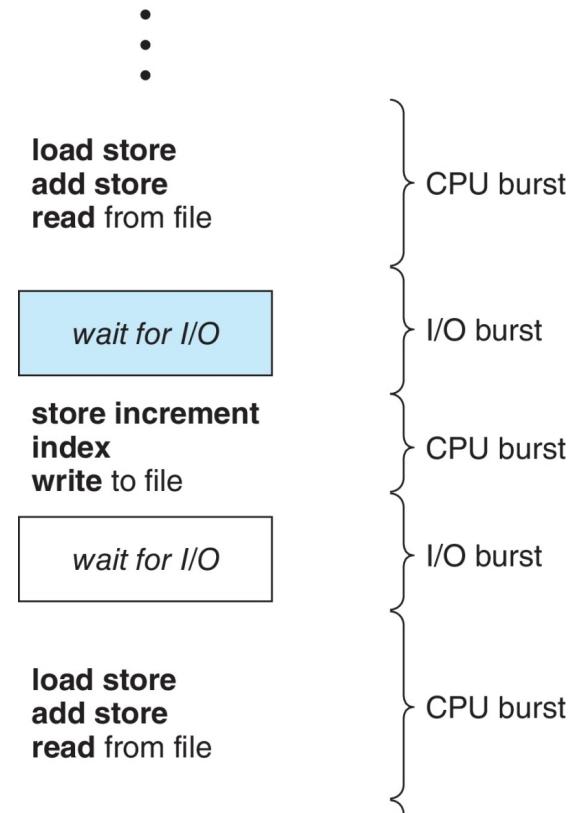
- Increase use of CPU
  - CPU utilisation

# CPU Scheduling

- The task of selecting ‘next’ process/thread to execute on CPU and doing a context switch
- Scheduling algorithm
  - Criteria for selecting the ‘next’ process/thread and it’s implementation
- Why is it important?

# Observation: CPU, I/O Bursts

- Process can ‘wait’ for an event (disk I/O, read from keyboard, etc. )
- During this period another process can be scheduled



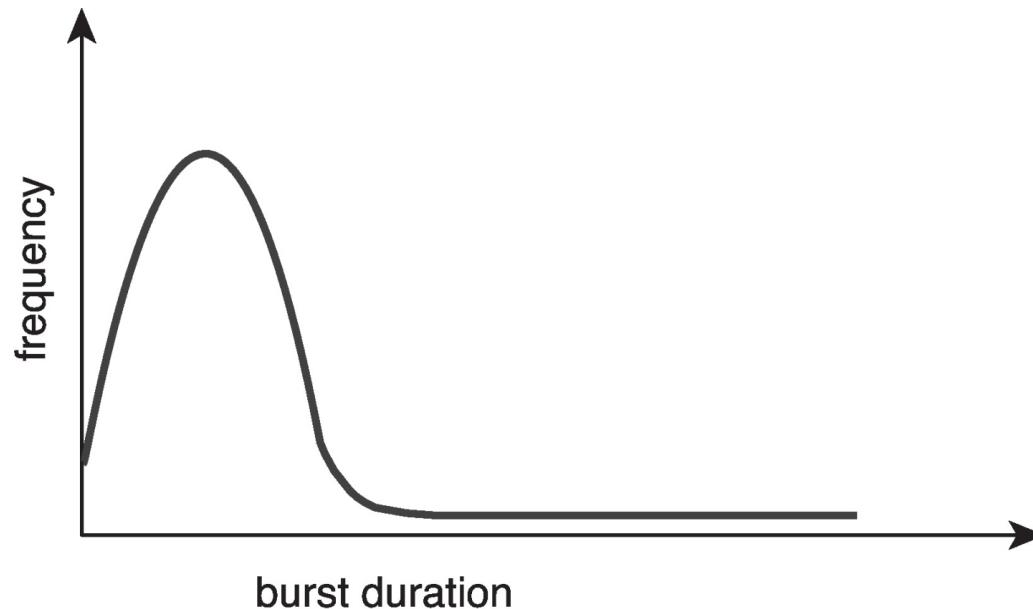
# Let's understand the problem

- Programs have alternate CPU and I/O bursts
  - Some are CPU intensive
  - Some are I/O intensive
  - Some are mix of both

- A C program example:

```
f(int i, int j, int k) {  
    j = k * i; // CPU burst  
    scanf("%d", &i); // I/O burst  
    k = i * j; // CPU burst
```

# CPU bursts: observation



# Scheduler, what does it do?

- **From a list of processes, ready to run**
  - Selects a process for execution
  - Allocates a CPU to the process for execution
  - Does “context switch”
    - Context: Set of registers
    - Switch from context of one process to another process
    - May be done like this: P1 context -> scheduler context -> P2 context

# When is scheduler invoked?

## 1) Process Switches from running to waiting state

- Waiting for I/O, etc.

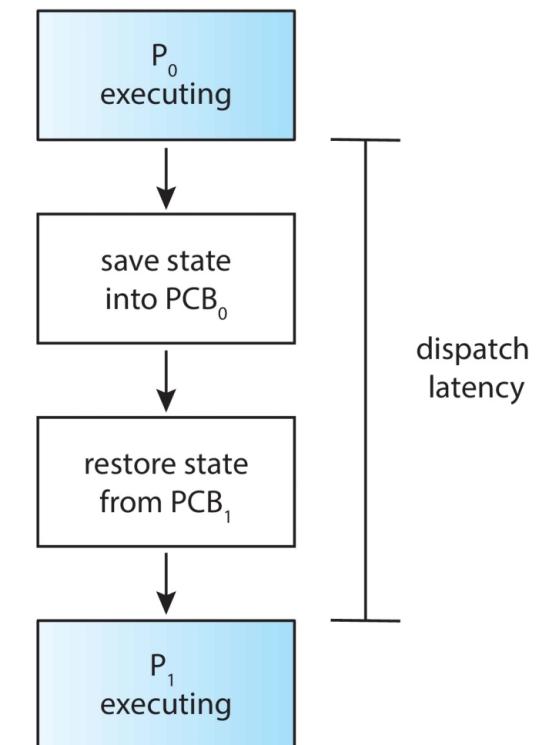
## 2) Switches from running to ready state

- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data

# Dispatcher: A part of scheduler

- Gives control of the CPU to the process selected by the short-term scheduler; this involves:

- switching context
- switching to user mode
- jumping to the proper location in the user program to restart



# Dispatcher in action on Linux

- **Run “vmstat 1 3”**
  - Means run vmstat 3 times at 1 second delay
- **In output, look at CPU:cs**
  - Context switches every second
- **Also for a process with pid 3323**
  - **Run**
  - **“*cat /proc/3323/status*”**
    - See
      - voluntary\_ctxt\_switches
      - --> Process left CPU
      - nonvoluntary\_ctxt\_switches

# Scheduling criteria

- **CPU utilization: Maximise**
  - keep the CPU as busy as possible. Linux: idle task is scheduled when no process to be scheduled.
- **Throughput : Maximise**
  - # of processes that complete their execution per time unit
- **Turnaround time : Minimise**

# Calculations of different criteria

- **If you want to evaluate an algorithm practically, you need a proper workload !**
  - Processes with CPU and I/O bursts
  - Different durations of CPU bursts
  - Different durations of I/O bursts
    - How to do this programmatically?
    - How to ensure that after 2 seconds an I/O takes place?

# Calculations of different criteria

## □ CPU Utilization

- % time spent in doing ‘useful’ work
- What is useful work?
  - On linux
    - there is an “idle” thread, scheduled when no other task is RUNNABLE
    - Not running idle thread is productive work
    - Includes process + scheduling time + interrupts
  - On other systems?

# Calculations of different criteria

## □ Throughput

- # processes that complete execution per unit time
- Formula: total # processes completed / total time
- Simply divide by your total workload that completed by the time taken
- Depends on the workload as well. ‘long’ or ‘short’ processes.

# Calculations of different criteria

## □ Turnaround time

- Amount of time required for one process to complete
- For every process, note down the starting and ending time, difference is TA-time
- For process P1 -> Time when process ended – time when process started
- Do the average TA-time

# Calculations of different criteria

- Waiting time
  - amount of time a process has been waiting in the *ready queue*.
  - To be minimised.

# Scheduling Criteria

- **Response time**
  - **amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).**
  - To be minimised.
  - E.g. time between your press of a key , and that key being shown in screen

# **Challenges in implementing the scheduling algorithms**

- **Not possible to know number of CPU and I/O bursts and the duration of each before the process runs !**
  - Although when we do numerical problems around each algorithm, we assume some values for CPU and I/O bursts !

# GANTT chart

- A timeline chart showing the sequence in which processes get scheduled
- Used for analysing a scheduling algorithm

# Scheduling Criteria: Differing requirements

- Different uses need different treatment of the scheduling criteria
- Knowing the workload is a challenge
- E.g. a desktop system
  - Response time is important
    - Minimize average response time Vs minimize variance in response time?

# **Our discussion on scheduling algorithms**

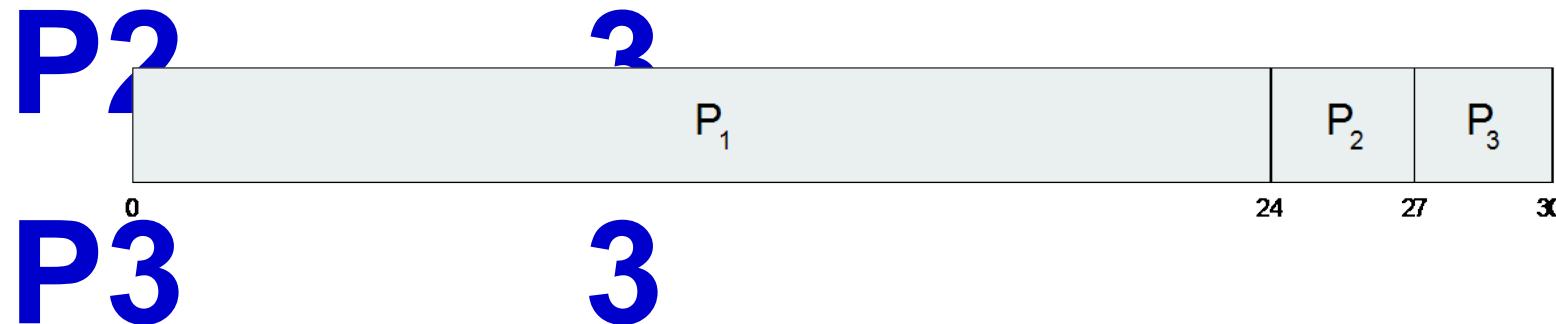
- **Analysis with only one CPU burst per process**
  - Ideally we should do for hundreds of CPU bursts
- **Only waiting time considered as criteria**

# **Scheduling Algorithms**

# First- Come, First-Served (FCFS) Scheduling

## Process Burst Time

P1            24



# FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:



The Gantt chart for the schedule is:

# FCFS: Convoy effect

- Consider one CPU-bound and many I/O-bound processes
- CPU bound process over, goes for I/O. I/O bound processes run quickly, move

# FCFS: further evaluation

- **Troublesome for interactive processes**
  - CPU bound process may hog CPU
  - Interactive process may not get a chance to run early and response time may be quite bad

# **Shortest-Job-First (SJF) Scheduling**

- **Associate with each process the length of its next CPU burst**
  - Use these lengths to schedule the process with the shortest time. Better name – **Shortest Next CPU Burst Scheduler**
- **SJF is optimal – gives minimum average waiting time for a given set of processes**

The difficulty is knowing the length of the next CPU

# Example of SJF

**Process**      **Burst Time**

**P1**    6

**P2**                8



0            3            9            16            24  
**P3**                7

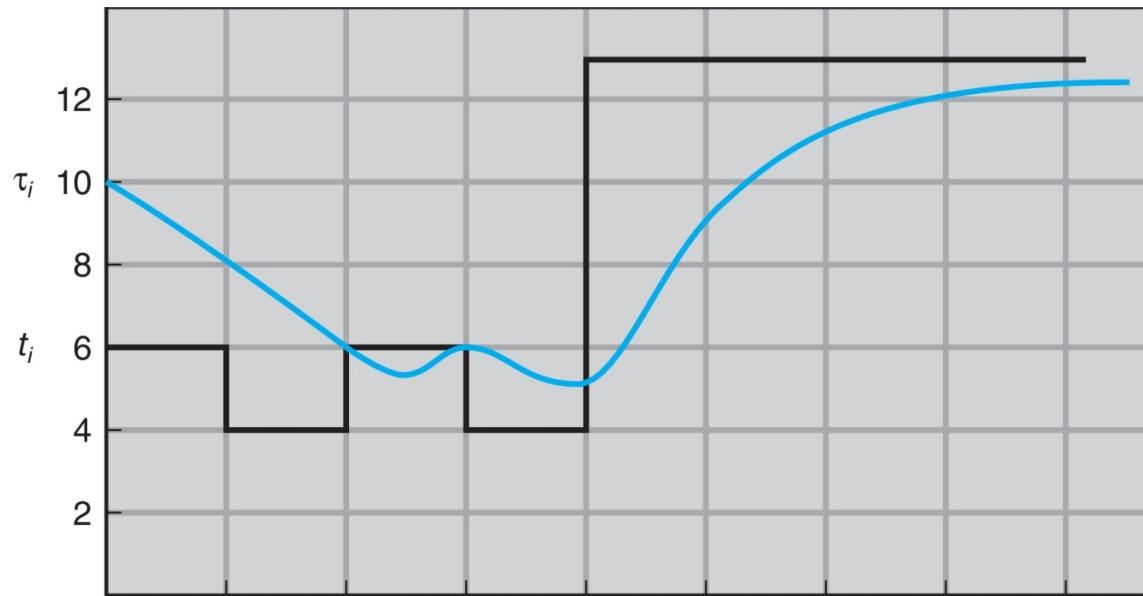
# Determining Length of Next CPU Burst

- Not possible to implement SJF as can't know "next" CPU burst. Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

Can be done by using the length of

# Prediction of the Length of the Next CPU Burst

- $A = 1/2$  ,  $\tau_0 = 10$

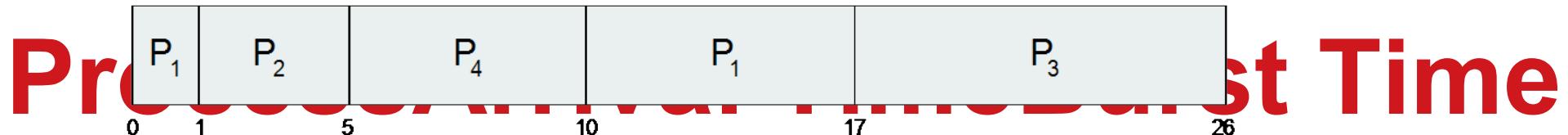


# Examples of Exponential Averaging

- ✓ = 0
  - $\Delta_{n+1} = \Delta_n$
  - Recent history does not count
  
- ✓ = 1
  - $\Delta_{n+1} = \checkmark t_n$
  - Only the actual last CPU burst counts

# Example of Shortest-remaining-time-first

**Preemptive SJF = SRTF. Now we add the concepts of varying arrival times and preemption to the analysis**



# Round Robin (RR) Scheduling

- **Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds.**
  - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- **If there are n processes in the ready queue and the time quantum is q, then each**

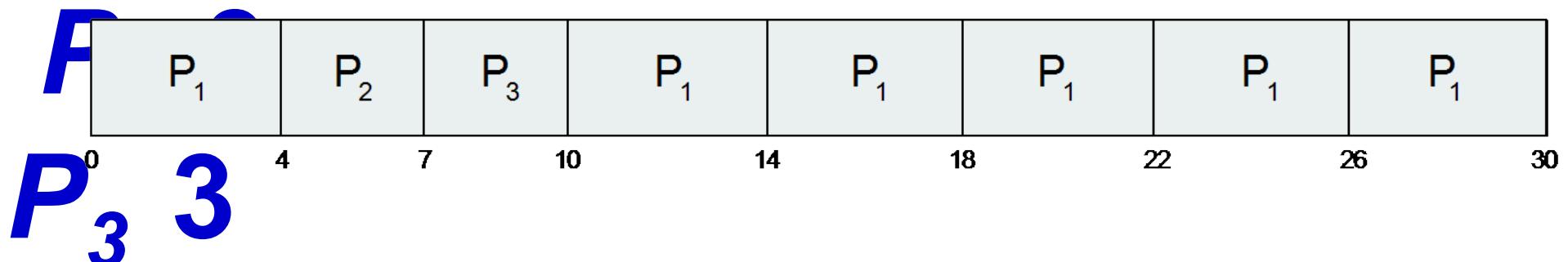
# Round Robin (RR) Scheduling

- Timer interrupts every quantum to schedule next process
- Performance
  - $q$  large □ FIFO
  - $q$  small □  $q$  must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

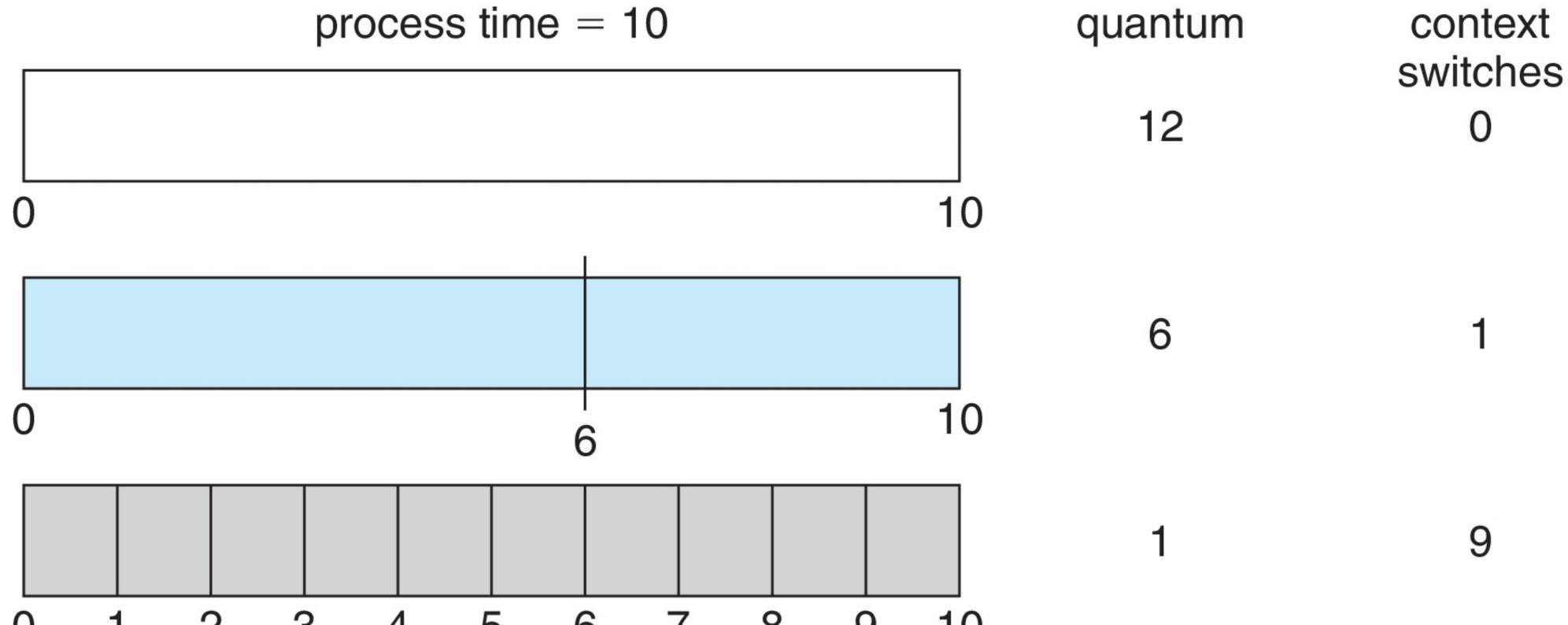
Process  
Burst Time

$P_1$  24

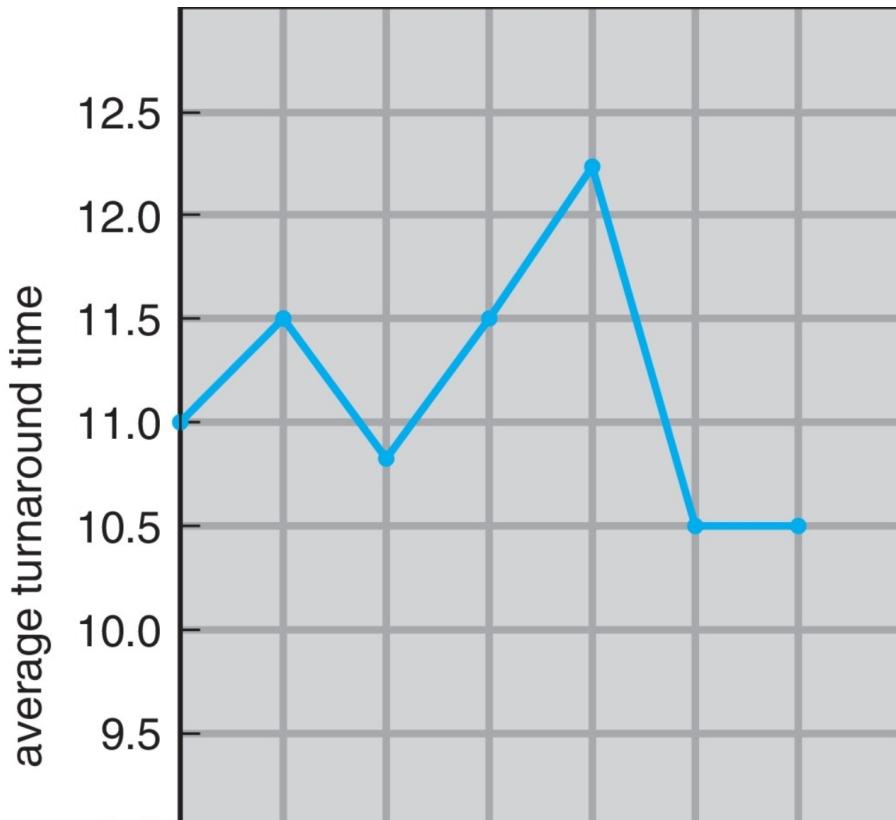


The Gantt chart is:

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

80% of CPU bursts should

# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer □ highest priority)

Preemptive (timer interrupt more time)

# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

- Priority scheduling Gantt Chart



# Priority Scheduling with Round-Robin

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$ 43		
$P_2$ 52		
$P_3$ 8 2		
$P_4$ 7 1		
$P_5$ 3 3		

Run the process with the highest priority. Processes with the same priority run round-robin

Gantt Chart with 2 ms time quantum

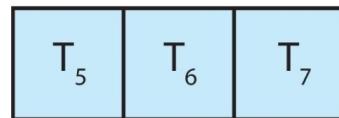
# Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!

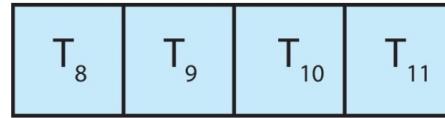
priority = 0



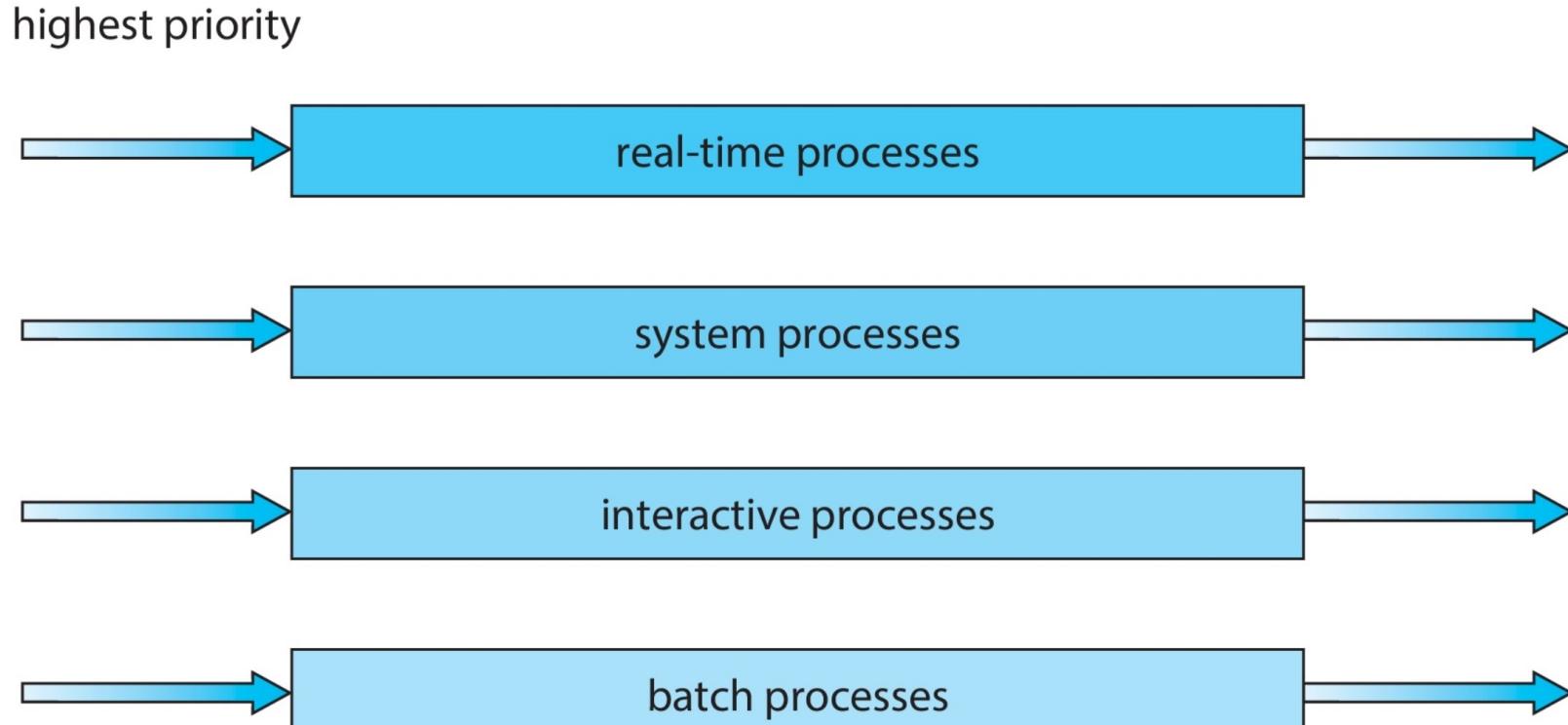
priority = 1



priority = 2



# Multilevel Queue



# Implementing multilevel queue

- **Processes need to have a priority**
  - Either modify fork()/exec() to have a priority
  - Or add a nice() system call to set priority
- **How to know the priority?**
  - The end user of the computer system needs to know this from needs of real life
  - E.g. on a database system, the database process

# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a

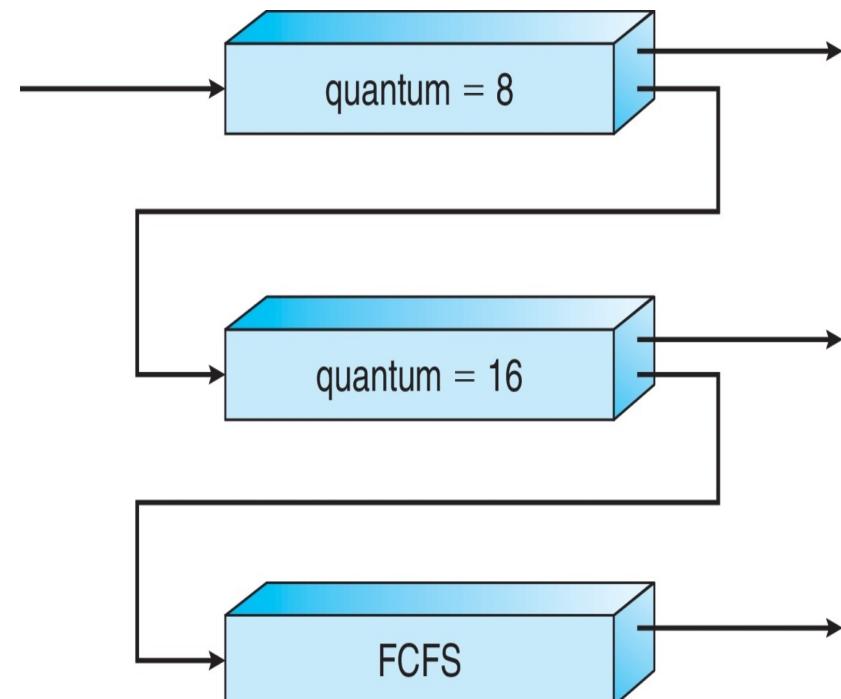
# Example of Multilevel Feedback Queue

- **Three queues:**

- $Q_0$  – RR with time quantum 8 milliseconds
- $Q_1$  – RR time quantum 16 milliseconds
- $Q_2$  – FCFS

- **Scheduling rules**

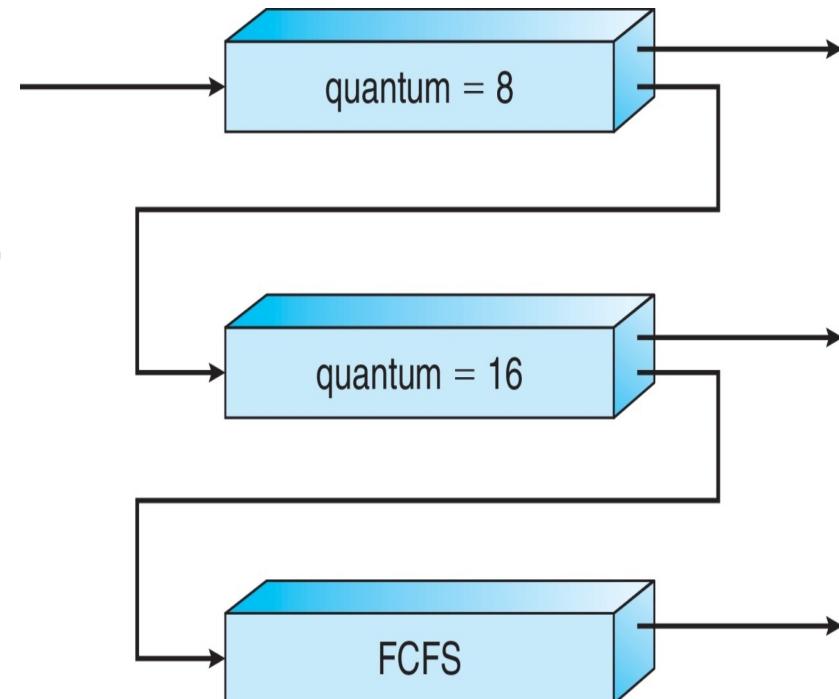
- Serve all processes in  $Q_0$  first
- Only when  $Q_0$  is empty, serve



# Example of Multilevel Feedback Queue

## Scheduling

- A new job enters queue  $Q_0$ 
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$
- At  $Q_1$ , job receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue  $Q_2$
- To prevent starvation, move a job from  $Q_2$  back to  $Q_0$  after every 16 milliseconds



# Thread Scheduling

- **Distinction between user-level and kernel-level threads**
- **When threads supported, threads scheduled, not processes**
- **Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP**

# Pthread Scheduling

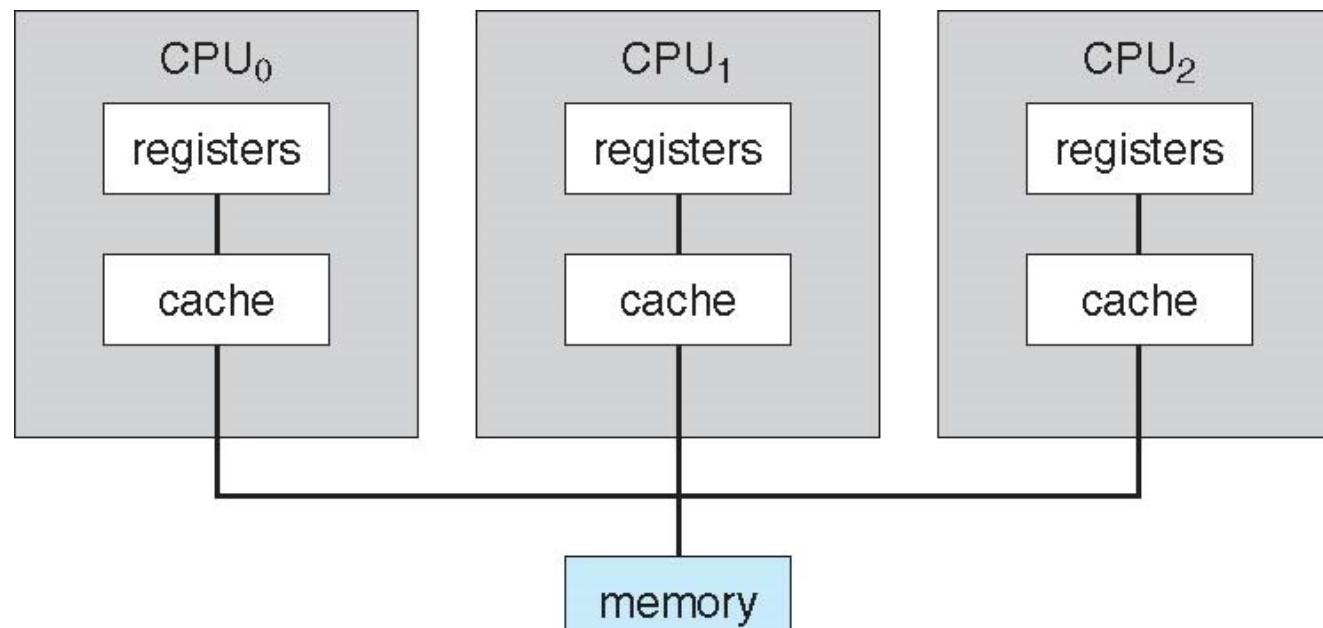
- ❑ **PTHREAD\_SCOPE\_PROCESS** schedules threads using PCS scheduling
- ❑ **PTHREAD\_SCOPE\_SYSTEM** schedules threads using SCS scheduling
- ❑ Linux and macOS only allow **PTHREAD\_SCOPE\_SYSTEM**
- ❑ Let's see a Demo using a program

# **Multi Processor Scheduling**

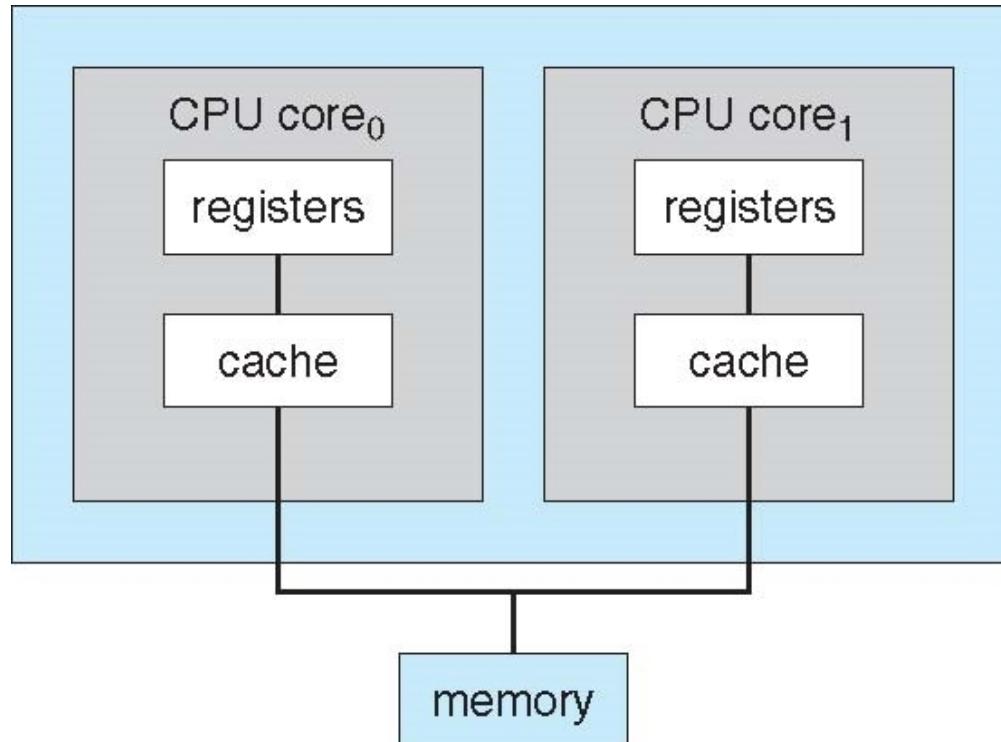
# Multiprocessor systems

- **Each processor has separate set of registers**
  - All: eip, esp, cr3, eax, ebx, etc.
- **Each processor runs independently of others**
- **Main difference is in how do they access memory**

# Symmetric multiprocessing (SMP)



# Multicore systems (also SMP)



# Booting multi-processor systems

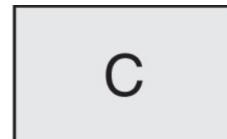
## □ X86 + xv6

- One processor starts, runs BIOS, loads kernel, initializes kernel data structures
- **Mpinit()** : first processor scans certain memory addresses for information about other processors and obtains configuration and configures them

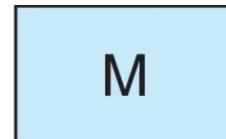
**Startothers()** : First processor initializes kernel

# Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



compute cycle



memory stall cycle

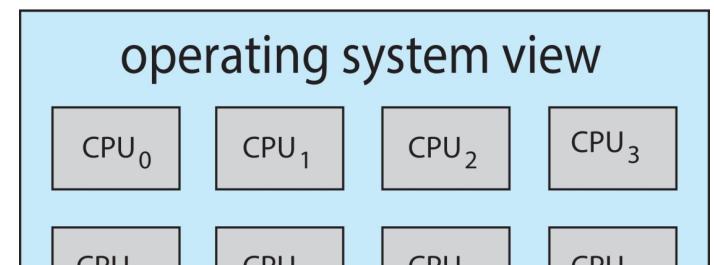
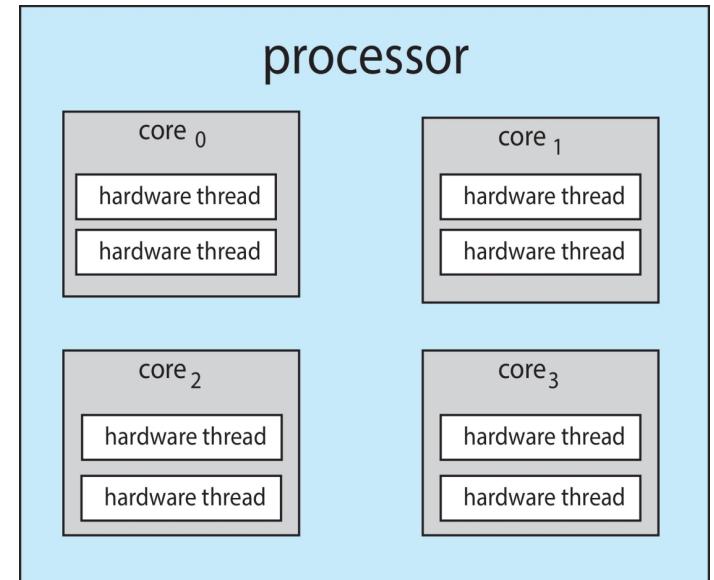
# Multithreaded Multicore System

Each core has > 1 hardware threads.

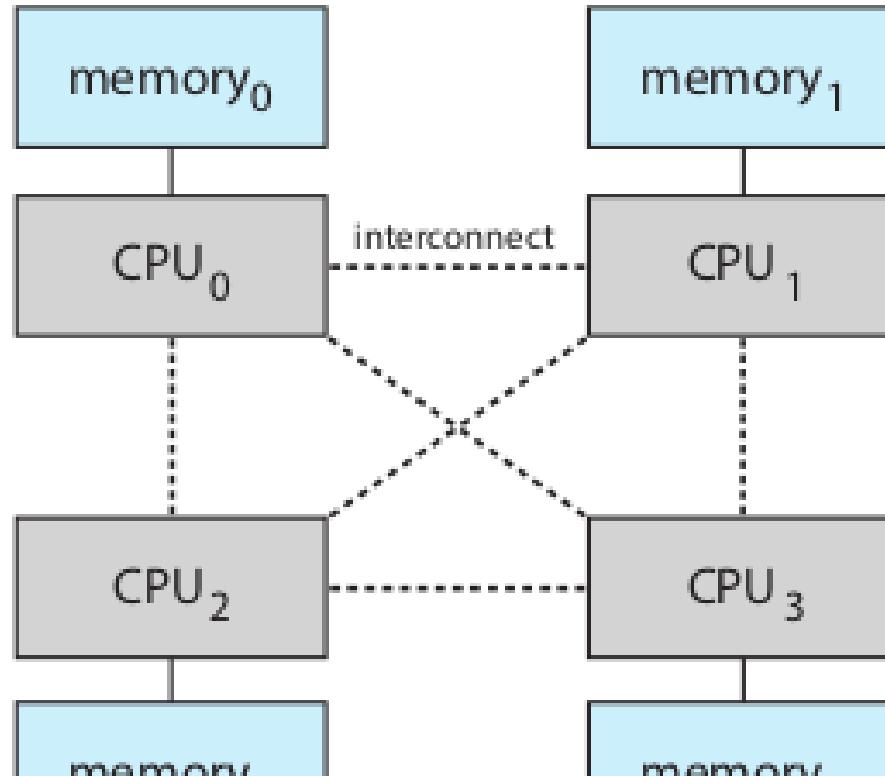
**Chip-multithreading (CMT)** assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)

On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

If one thread has a memory stall, switch to another thread!



# Non-Uniform Memory Architecture (NUMA)



# More on SMP systems

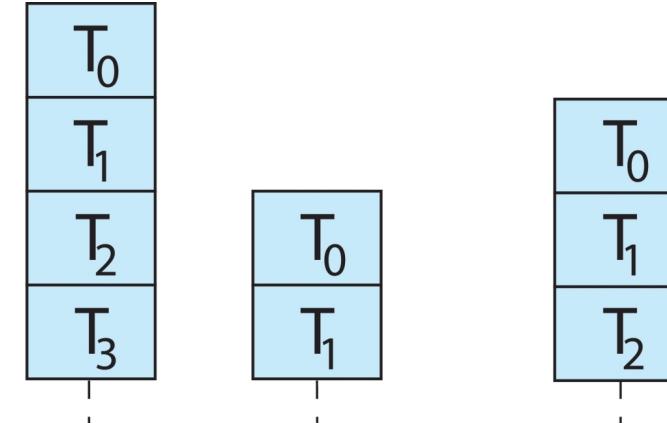
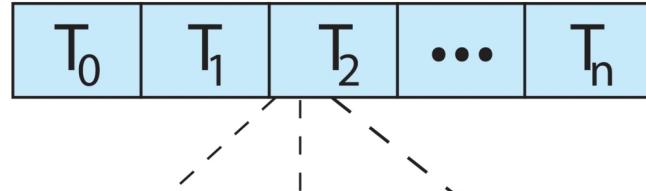
- During booting – each CPU needs to be turned on
  - Special I/O instructions writing to particular ports
  - See lpicstartap() in xv6
  - Need to setup CR3 on each processor
  - Segmentation, Page tables are shared (same memory for all CPUs)

# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems

# Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



# Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- Pull migration – idle processors pulls waiting task from busy processor

# Multiple-Processor Scheduling – Load Balancing

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a

# Multiple-Processor Scheduling – Load Balancing

- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

# SMP in xv6

- Only one process queue
- No load balancing, no affinity
- A process may run any CPU burst /allotted-time-quantum on any processor randomly
- See the code of:
  - Startothers(), mpenter(), mpmain()

Different scheduling mechanisms approach

**End**

# Pointers in C

Abhijit A.M.

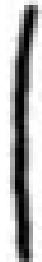
[abhijit13@gmail.com](mailto:abhijit13@gmail.com)

(C) Abhijit A.M.

Shared under Creative Commons Attribution

Sharealike International License V3.0

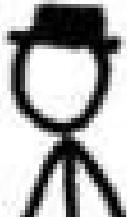
MAN, I SUCK AT THIS GAME.  
CAN YOU GIVE ME  
A FEW POINTERS?



0x3A28213A  
0x6339392C,  
0x7363682E.



I HATE YOU.



# Pointer

- **Pointer is a variable which can store addresses**
- **Pointer has a “type” (except void pointer)**
  - e.g. int \*p; char \*cp; double \*dp;
  - Here p, cp, dp are respectively pointers to integer, character, and double
- **Size of pointer is decided by compiler**

# Operations on (and related to) Pointers

&

\*

=

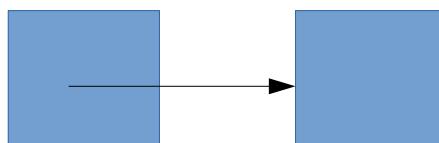
+int -int

-

[ ]

# Operations related to pointers: &

```
int *p, i;  
p = &i;
```



- **& fetches the address of variable**
  - Called Referencing operator
  - Here, &i is address of i
  - RHS is address of integer, LHS is variable which can store

# Operations on pointers: \*

```
int *p, i, j;
```

```
i = 10;
```

```
p = &i;
```

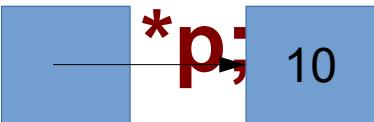
```
j = *p;
```



- \* fetches the value stored at a given address
  - Called dereferencing operator
- \*p : here “value of p” is itself an address (of i), so \*p is value

# Operations on pointers: \*

```
int *p, i,      char *cp,  
j;                c, d;  
  
i = 10;          c = 'x';  
  
p = &i;           cp = &c;  
  
*p;              d = *cp;  
  
p               i
```



- What is the difference between \* in the two codes?
- The \*p fetches the value at given address in sizeof(int) bytes, while \*cp fetches the value at

# Operations on pointers: =

```
int *p, i,  
*q;
```

```
i = 10;
```

```
p = &i;
```



p

i



q

- **Pointers can be copied**
  - Can arrays be copied?
- **Thumb rule:**
  - After pointer copy, both pointers point to same location

# Operations on pointers: +- int

```
int *p, a[4],
```

```
*q;
```

```
p = &a[1];
```

```
q = p + 2;
```

```
*q = 40;
```



- C allows adding or subtracting an int from a pointer!
  - Weird, but true!
  - e.g. `int *p, n; p + n;`
- The result is a pointer of the same type.

# Problems: Draw diagrams for the code

```
int main() {  
    int *p, *q, a[3], b;  
    a[0] = 10; b = 2;  
    p = &a[1];  
    q = p + 1;  
    p = q - b;  
    *p = 30;
```

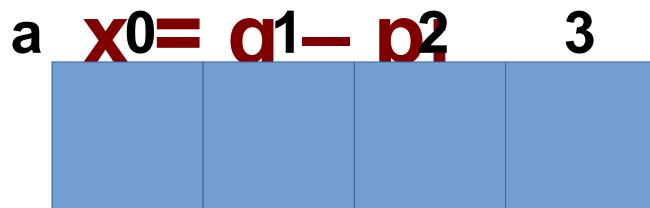
```
int main() {  
    int *p, *q, a[3], b;  
    a[0] = 10; b = 1;  
    p = &a[3];  
    q = p - 3;  
    p = q + 1;  
    *(q + 1) = 30;
```

# Operations on pointers: subtracting two pointers

```
int *p, a[4], *q,  
x;
```

```
p = &a[1];
```

```
q = p + 2;
```



- Two pointers of the *same type* can be subtracted
  - Result type is int
  - Result value = no. Of elements of `sizeof(type)` between two pointers

# Operations on pointers: [ ]

```
int *p, a[4],  
*q;
```

```
p = &a[1];
```

```
p[2] = 20;
```

```
p[-1] = 10;
```

a	0	1	2	3
	10			20

- Interestingly, C allows [ ] notation to be applied to all pointers!
  - You must be knowing that [ ] is normally used for arrays
  - $p[i]$  means  $* (p + i)$

# A peculiar thing about arrays

- Name of an array is equivalent to the address of (the zeroeth element) the array
- Because it's an address, it can be stored in a pointer
- What do the following mean?
  - Exception: when passed to `sizeof()` operator, the name is not the address

# Pointer as *if it was an array*

- Combine the concepts of
  - Pointer arithmetic (+- int)
  - [ ] notation for pointers
  - Array name as address of array
- ```
int a[3], *p;  
p = a;  
p[2] = 10;
```

  - Here we are using p as if it was an array name

# Pointers != Arrays

- **Array is a continuous collection of elements of one type.**
- **Array has name, the name also stands for the address of**
- **Pointer is a variable that can store an address**
- **Pointers can be of various types**
- **[ ], = , +- int, subtraction are**

# Arrays as arguments to functions

```
char f(char *c) {  
    return c[0];  
}  
  
int main() {  
    char arr[16], x;  
    x = f(arr);
```

- Here actual argument is ‘arr’
- Name of array is address of array, that is &arr[0]
  - Address of char
- So formal argument

# 2-d array as argument to function

```
char f(char **c) {  
    return c[0][2];  
}  
  
int main() {  
    char arr[16][6], x;  
    x = f(arr);
```

```
char f(char c[][6]) {  
    return c[0][2];  
}  
  
int main() {  
    char arr[16][6], x;  
    x = f(arr);
```

# 2-d array as argument to function

```
char f(char (*c)[6]) {  
    return c[0][2];  
}
```

```
int main() {  
    char arr[16][6], x;  
    x = f(arr);
```

```
char f(char c[][6]) {  
    return c[0][2];  
}
```

```
int main() {  
    char arr[16][6], x;  
    x = f(arr);
```

# **Dynamic Memory Allocation**

# Concept of (Binding) “Time”

- **Compile Time**

- When you are running commands like
- cc program.c -o program

- **Load time**

- After you type commands like
- ./program
- Before the main() starts running

# Lifetime of variables and Memory Allocation

- **Global Variables, Static Variables (g, t, and s here)**

- Allocated Memory at *load time*
- They are alive (available) till the program exits

**int g;**

**static int t =  
20;**

**int f(int a, int  
b) {**

## Local Variables, Formal

# Man pages

Understanding library functions:

Read the manual pages, using ‘man’ command

> man sqrt

> man 3 printf

# malloc()

- **malloc() is a standard C library function for allocating memory dynamically (at run time)**
- **#include <stdlib.h> for malloc()**
- **Function prototype**
  - Run “man malloc” to see it
  - **void \*malloc(size\_t size);**
  - **size\_t** is a typedef in stdlib.h

# **void \***

- **A void pointer is a typeless pointer;**
  - Pure address
  - No type --> No “size” information about the type
- **You can declare a void pointer**
- **Void pointers can be copied**
  - q also stores address of a now
- **The Dereferencing operator has no meaning when applied to a void pointer**

# malloc()

- **malloc() returns a “void \*”**
  - Returns a pure address
  - This address can be stored in a “void \*” variable
  - This address can also be stored in any pointer variable
- **Suppose we do**
  - `void *p; p = malloc(8);`

# malloc()

```
int *p;  
p = malloc(8)
```

- This code allocates 8 bytes and then pointer p will point to the malloced memory
- This code can result in a “warning” because we are

# malloc()

```
int *p;
```

```
p = (int *) malloc(8);
```

- This code does away with the warning as we are converting the “void \*\*” into “int \*\*” using explicit type casting
- Suppose size of integer was 4 bytes

# malloc()

```
int *p;
```

```
p = (int *) malloc(8);
```

- p points to 8 byte location
- However now, \*p means referencing in “4 bytes”
  - as size of int is 4 bytes

- p[0] means \*(p + 0) that is \*p
- p[1] means \*(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !

# malloc()

```
int *p;
```

```
p = (int *) malloc(8);
```

- p points to 8 byte location
- However now, \*p means referencing in “4 bytes”
  - as size of int is 4 bytes

- p[0] means \*(p + 0) that is \*p
- p[1] means \*(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !

# Malloc(): Allocating arrays dynamically

```
int *p;
```

```
p = (int *) malloc(  
    sizeof(int) * 4);
```

- We can allocate arrays of any type dynamically using malloc()

- Use of `sizeof(int)` here makes sure that the code is *portable*, appropriately sized array will be allocated irrespective of size

# malloc(): Allocating array of structures

```
typedef struct test {  
    int a, b;  
    double g;  
}test;  
  
test *p;  
  
p = (test *) malloc(
```

- This code allocates an array of 4 structures
- p points to the array of structures
- p[0] is the 0<sup>th</sup> structure, p[1] is the

# Homework

- **Write code using malloc() to**
  - Allocate an array of 10 doubles
  - Read n from user and allocate an array of n shorts
- **What does the following code mean?**
- **What does the following code mean?**

# **free()**

- **free() will give the malloced memory back**
- **malloc() and free() work together to manage what is called as “heap memory” which the memory management library has obtained from the OS**
- **Usage**
- **free() must be given an address which was**

# Pointer to Pointer

```
int **pp, *p, p;
```

```
pp = (int **) malloc(sizeof(int *) * 4);
```

```
pp[1] = (int *)malloc(sizeof(int) * 3);
```

- A pointer to pointer, is essentially a pointer!
  - Can store an address, has a type
  - The type is “pointer to pointer” so all \* or [ ] operations work with sizeof(pointer) which is

# 2-d array with pointer to pointer

```
#include <stdlib.h>
```

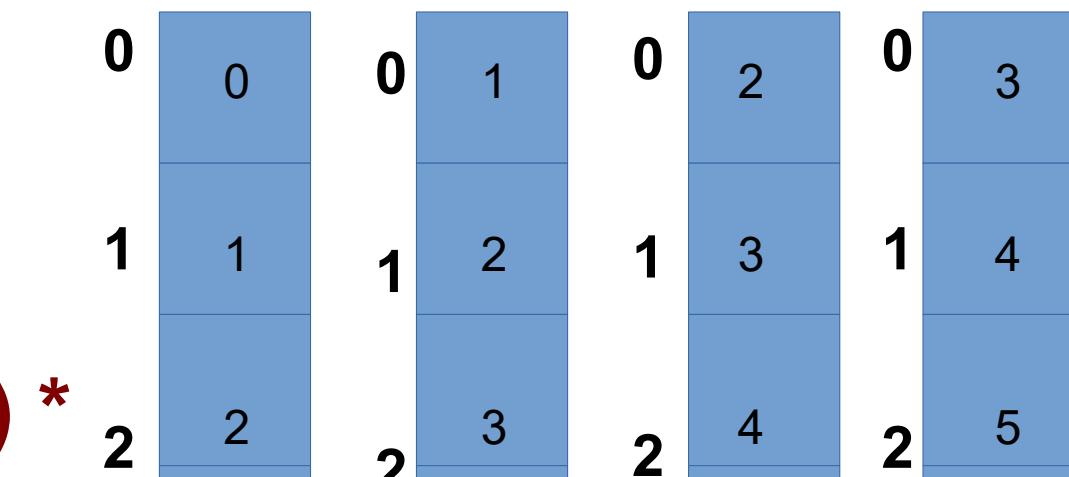
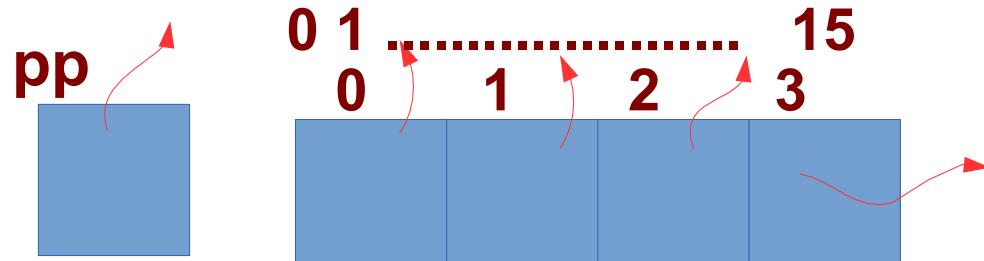
```
#define N 4
```

```
int main() {
```

```
    int **pp, i, j;
```

```
    pp = (int **) malloc
```

```
        (sizeof(int *) *
```



# 2-d array with pointer to pointer

```
#include <stdlib.h>

#define N 4

int main() {
    int **pp, i, j;
    pp = (int **) malloc
        (sizeof(int *) *
```

- How does **pp[i][j]** work here?
- **pp[i]** is **\*(pp + i)** which is the pointer at the i'th location in the array of pointers allocated

↳ **\*** works with size

# **Self Referential Pointers in Structures**

# **Self Referential Pointers**

- “**Self Referential Pointer**” is a kind of a misnomer
- C allows structures like this

```
struct test {
```

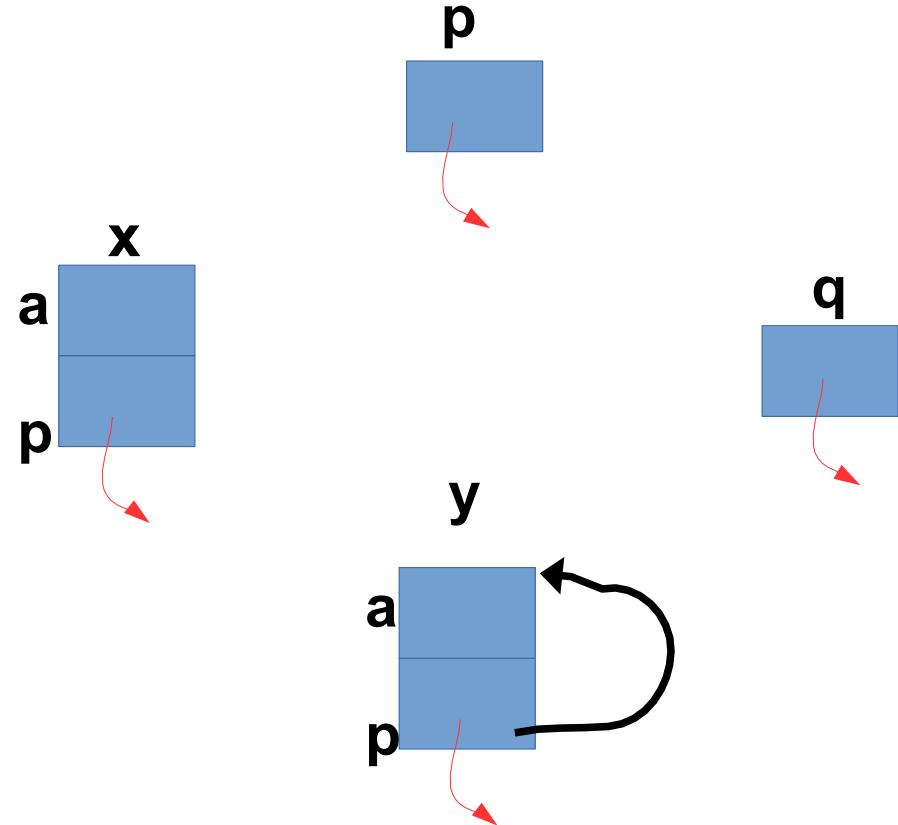
```
    int a;
```

```
    struct test *p;
```

# Self Referential Pointers

- Consider following code

```
typedef struct test {  
    int a;  
    struct test *p;  
}test;
```



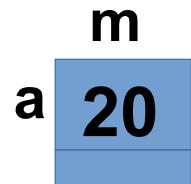
## -> notation

```
typedef struct test {  
    int a;  
    struct test *p  
}test;  
  
test m, n, *x;  
  
m.a = 20;  
  
x = &n;
```

□ (\*x) is the entire structure to which x points

□ (\*x).a is the variable 'a' in that structure

□ x->a is another notation for  
 
$$\begin{matrix} & x \\ & \downarrow \\ (*x).a \end{matrix}$$



n

**2 Common Problems involving Pointers:**

**(Dereferencing) Dangling Pointers  
And  
Garbage Memory**

# NULL

- **NULL is a symbolic constant defined in stdio.h**
  - `#define NULL 0`
- **This is a special pointer value, defined by C language**
  - The number 0 is not the same as the address 0 !  
They are different types !

# A side note on **NULL**

- **NULL is not the number 0**
- **NULL is not necessarily the address 0**
- **NULL is just a special value for pointers told to us by C language.**
  - Very often we need special values for a certain type
  - E.g. the value '\0' for a character is universally taken to be a special value indicating end of a character

# Dangling Pointers

- Total 3 Possibilities for a pointer value
  - Points to memory owned by program
    - Local variables, Global Variables, Malloced Memory
  - = NULL
  - Dangling
    - Some texts differentiate between ‘wild’ (uninitialized) and ‘dangling’ pointers

Dereferencing (that is `*`) a dangling (or NULL) pointer is undefined behavior.

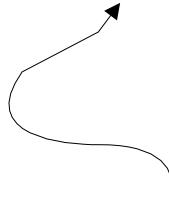
# Dangling Pointers

- A pointer is dangling when declared & not initialized
- A pointer becomes non dangling, when assigned to a “good” memory location like local variables, global variables, malloc-ed memory
- A pointer can become dangling again due to
  - Mistakes in pointer arithmetic

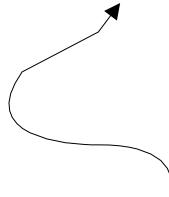
```
int main() {  
    int *p, i, j, *q;// p,q are dangling  
}  
  

```

```
int main() {  
    int *p, i, j, *q;// p,q are dangling  
    p = &i; // p not dangling  
  
}
```

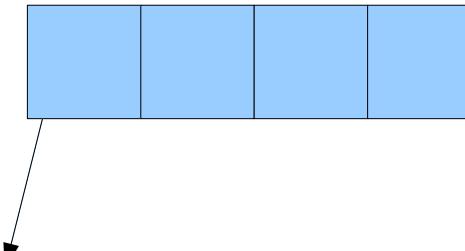


```
int main() {  
    int *p, i, j, *q;// p,q are dangling  
    p = &i; // p not dangling  
    i = 10; // *p = 10  
  
}
```

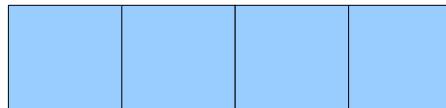


```
int main() {  
    int *p, i, j, *q;// p,q are dangling  
    p = &i; // p not dangling  
    i = 10; // *p = 10  
    q = &j; // q not dangling  
  
}
```

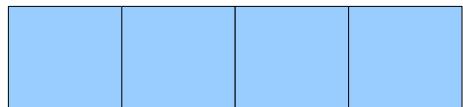
```
int main() {  
    int *p, i, j, *q;// p,q are dangling  
    p = &i; // p not dangling  
    i = 10; // *p = 10  
    q = &j; // q not dangling  
    q = (int *)malloc(sizeof(int) * 4);  
// q not dangling  
  
}
```



```
int main() {
    int *p, i, j, *q;// p,q are dangling
    p = &i; // p not dangling
    i = 10; // *p = 10
    q = &j; // q not dangling
    q = (int *)malloc(sizeof(int) * 4);
// q not dangling
    q = q + 3; // q not dangling
}
```



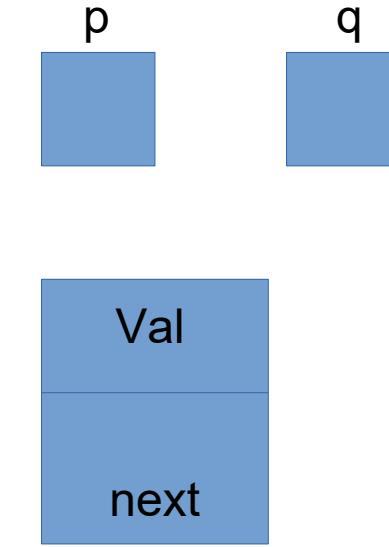
```
int main() {
    int *p, i, j, *q;// p,q are dangling
    p = &i; // p not dangling
    i = 10; // *p = 10
    q = &j; // q not dangling
    q = (int *)malloc(sizeof(int) * 4);
    // q not dangling
    q = q + 3; // q not dangling
    q = q + 1; // q IS dangling now
}
```



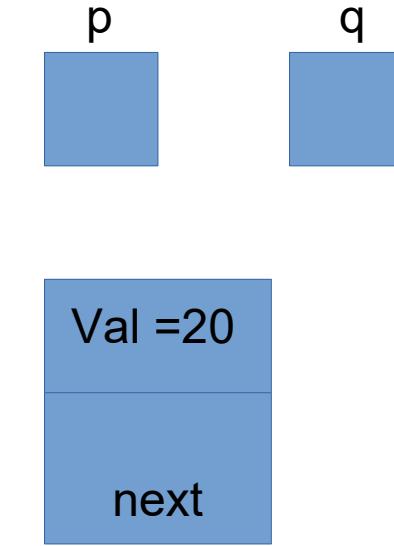
```
int main() {
    int *p, i, j, *q;// p,q are dangling
    p = &i; // p not dangling
    i = 10; // *p = 10
    q = &j; // q not dangling
    q = (int *)malloc(sizeof(int) * 4);
    // q not dangling
    q = q + 3; // q not dangling
    q = q + 1; // q IS dangling now
    p = q - 6; // p IS also dangling
}
```



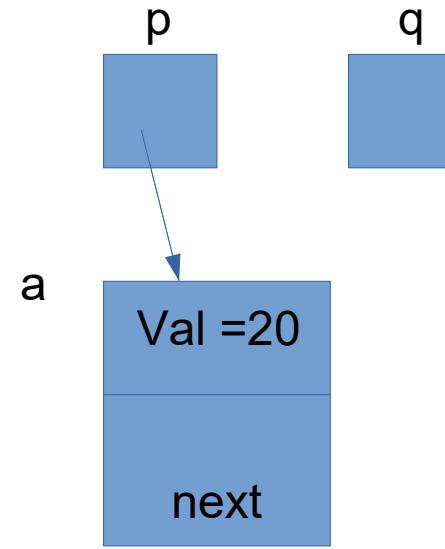
```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
}
```



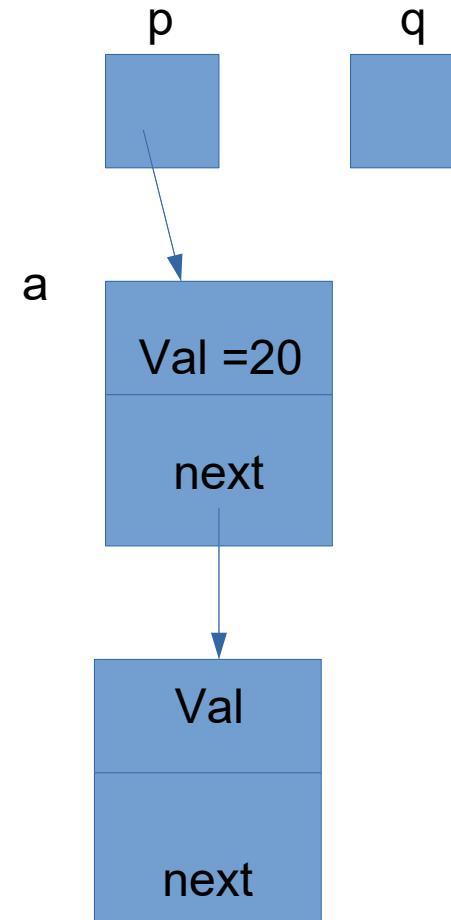
```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
    a.val = 20;  
}
```



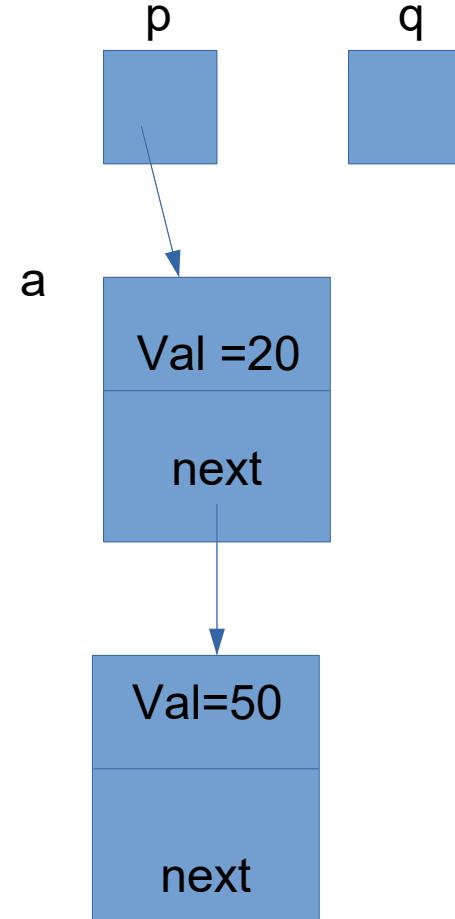
```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
    a.val = 20;  
    p = &a;  
  
}
```



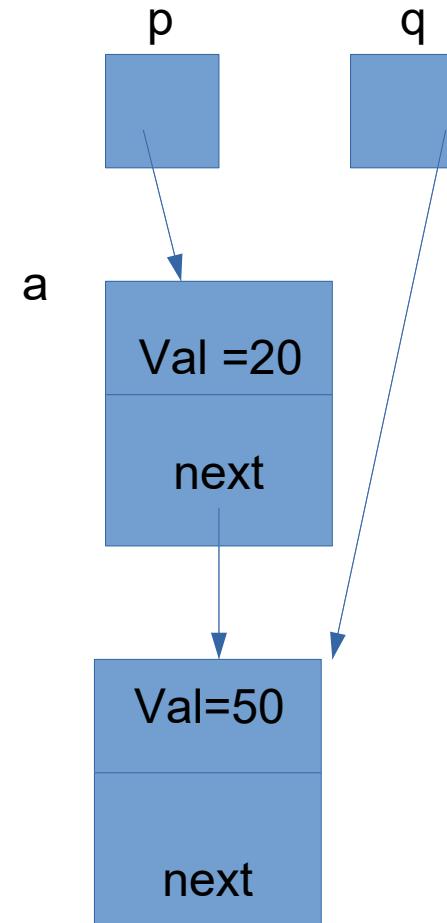
```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
    a.val = 20;  
    p = &a;  
    a.next = (node *)malloc(sizeof(node));  
}
```



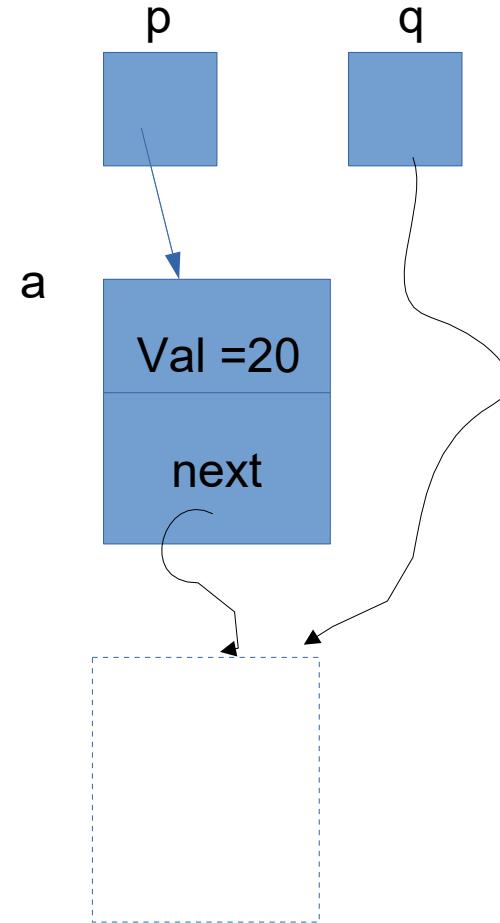
```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
    a.val = 20;  
    p = &a;  
    a.next = (node *)malloc(sizeof(node));  
    a.next->val = 50;  
}
```



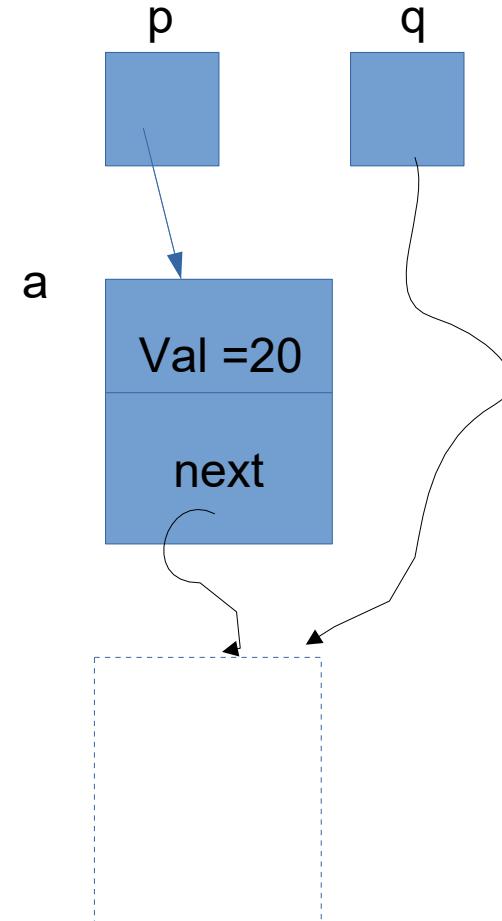
```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
    a.val = 20;  
    p = &a;  
    a.next = (node *)malloc(sizeof(node));  
    a.next->val = 50;  
    q = a.next;  
}
```



```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
    a.val = 20;  
    p = &a;  
    a.next = (node *)malloc(sizeof(node));  
    a.next->val = 50;  
    q = a.next;  
    free(a.next);  
}  
}
```



```
typedef struct node {  
    int val;  
    struct node *next;  
}node;  
int main() {  
    node *p, *q;  
    node a;  
    a.val = 20;  
    p = &a;  
    a.next = (node *)malloc(sizeof(node));  
    a.next->val = 50;  
    q = a.next;  
    free(a.next);  
    q->val = 100; // segfault  
}
```



# Remember

- Dereferencing a dangling pointer is a cause of segmentation fault
- Dereferencing can occur using \* or [ ]
- Having a dangling pointer does not cause segmentation fault
  - Dereferencing it causes
  - Having dangling pointers is not wrong, but why

# Garbage Memory

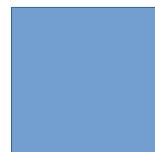
```
int *p, i;
```

```
p = malloc
```

```
(sizeof(int) * 4);
```

```
p = &i;
```

i



- p was pointing to malloced memory
- In the end p points to i
- How to access the malloced memory now?
  - No way! It's Lost!
  - It became “garbage memory”
- Problems like this

# Segmentation Fault

# What is segmentation fault?

- A program accessing an “illegal” memory location
  - Is punished by the operating system for doing so
  - The program gets “terminated” (killed)
  - “segmentation” comes from “memory management” concepts in operating systems
- It is **ALWAYS**a fault of the programmer!

# Some Typical Reasons for Seg-fault

Deferencing Dangling Pointers

Array Index Violation

Incorrect Function Arguments

# Dangling Pointer Dereference: Some examples

```
int *p, i;
```

```
*p = 20;
```

```
int *p, i;
```

```
p = malloc(
```

```
size_t size);  
int *p, i;  
p = &i;
```

```
int *f(int *p) {
```

```
int x = *p + 2;
```

```
return &x;
```

```
}
```

```
int main() {
```

```
int i, *q;
```

```
scanf("%d", &i);
```

# Array Index Violation

Valid indices for an array of size n: 0 .. n-1

Accessing any index  $\leq -1$  or  $\geq n$  may result in seg-fault (it may not result in some cases, but it is STILL WRONG to do it!)

Try this code:

At what value of i does the code seg-fault? Try 2-3 times.

```
#include <stdio.h>
int main() {
    int a[16], i = 0;
    while(1) {
        a[i] = i;
        printf("%d\n", a[i]);
        i++;
    }
}
```

# Functions: Pointer Arguments

- Rule: When you want a function to change ACTUAL arguments
  - Function takes pointer arguments
  - Call passes address of “actual argument”
- Example: Swap function (correct code)

```
void swap(int *a, int *b ) {  
    int temp;  
    temp = *a;  
    *a = * b;  
    *b = temp;
```

# Incorrect Pointer Arguments - Segfault

```
int i;  
scanf("%d", i);
```

# Guidelines to avoid segfaults

- Always initialize pointers to NULL
  - This will not avoid segfaults, but may lead to early detection of a problem
- While accessing arrays, make sure that array index is valid
- Never return address of local variable of a function

**Let's draw diagrams for some programs using  
pointers, malloc, free, ...**

# **Introduction to Linux Desktop and Command Line**

**11 Jan 2022**

Abhijit A. M.  
[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

# **Why GNU/Linux ?**

## Why GNU/Linux ?

1. Programmer's Paradise : most versatile, vast, all pervasive programming environment
2. Free Software ( or Open Source?) : Free as in freedom. *Freely* Use, copy, modify, redistribute.
3. Highly Productive : Do more in less time
4. Better quality, more secure, very few crashes

# Why Command Line ?

1. Not for everyone ! Absolutely !
2. Those who do it are way more productive than others
3. Makes you think !
4. Portable. Learn once, use everywhere on all Unixes, Linuxes, etc

# Few Key Concepts

- ***Files don't open themselves***
  - Always some application/program open()s a file.
- ***Files don't display themselves***
  - A file is displayed by the program which opens it.  
Each program has it's own way of handling ifles

# Few Key Concepts

- **Programs don't run themselves**
  - You click on a program, or run a command --> equivalent to request to Operating System to run it. The OS runs your program
- **Users (humans) request OS to run programs, using Graphical or Command line interface**
  - and programs open files

# Path names

- **Tree like directory structure**
- **Root directory called /**
- **A complete path name for a file**
  - /home/student/a.c
- **Relative path names**

# A command

- **Name of an executable file**
  - For example: 'ls' is actually "/bin/ls"
- **Command takes arguments**
  - E.g. ls /tmp/
- **Command takes options**
  - E.g. ls -a

# A command

- **Command can take both arguments and options**
  - E.g. ls -a /tmp/
- **Options and arguments are basically argv[] of the main() of that program**

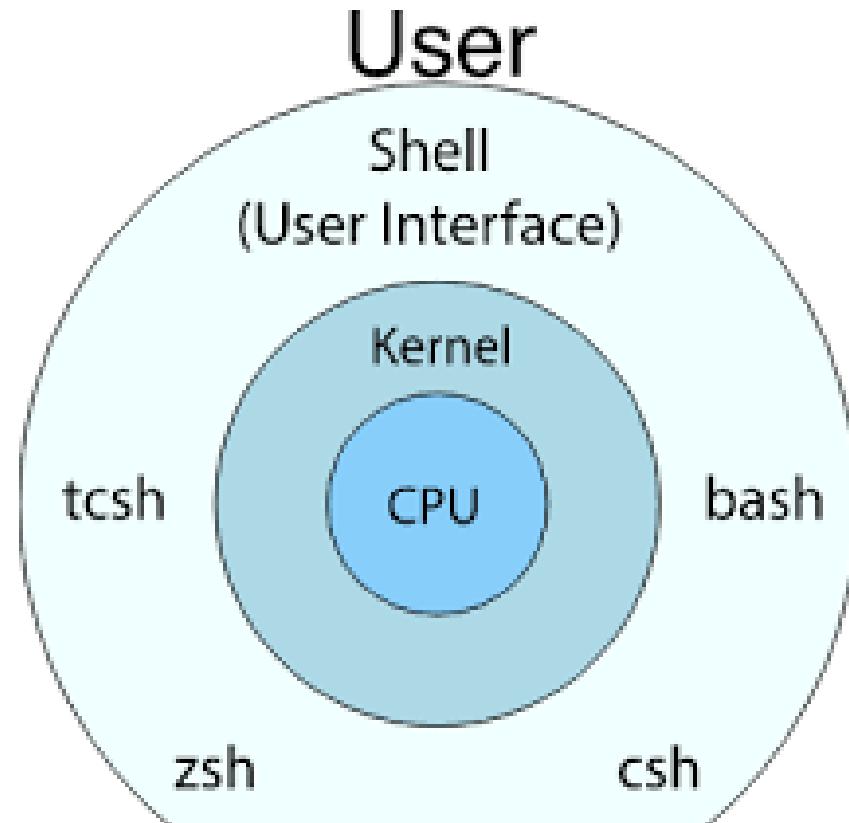
# Basic Navigation Commands

- **pwd**
- **ls**
  - ls -l
  - ls -l /tmp/
  - ls -l /home/student/Desktop
  - ls -l ./Desktop
  - ls -a
  - \ls -F
- **cd**
  - cd /tmp/

**Map these  
commands to  
navigation using a  
graphical file  
browser**

# The Shell

- **Shell = Cover**
- **Covers some of the Operating System's “System Calls” (mainly fork+exec) for the *Applications***
- **Talks with Users**



# The Shell

Shell waits for user's input

Requests the OS to run a program which the user  
has asked to run

Again waits for user's input

CLI is a Shell !

# Let's Understand fork() and exec()

```
#include <unistd.h>

int main() {
    fork();
    printf("hi\n");
    return 0;
}
```

```
#include <unistd.h>

int main() {
    printf("hi\n");
    execl("/bin/ls", "ls",
          NULL);
    printf("bye\n");
    return 0;
}
```

# A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char string[128];
    int pid;
    while(1) {
        printf("prompt>");
        scanf("%s", string);
        pid = fork();
        if(pid == 0) {
            execl(string, string, NULL);
        } else {
```

# File Permissions on Linux

- **Two types of users**
  - root and non-root
  - Users can grouped into 'groups'
- **3 sets of 3 permission**
  - Octal notation
  - Read = 4, Write = 2, Execute = 1
  - 644 means

# File Permissions on Linux

-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01\_linux\_basics.ppt

-rw-r--r-- 1 abhijit abhijit 341736 May 17 10:39 Debian Family Tree.svg

drwxr-xr-x 2 abhijit abhijit 4096 May 17 11:16 fork-exec

-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp

3 sets of 3 permissions

Owner

size

name

# File Permissions on Linux

- **r on a file : can read the file**
  - open(... O\_RDONLY) works
- **w on a file: can modify the file**
- **x on a file: can ask the os to run the file as an executable program**
- **r on a directory: can do 'ls'**

# Access rights examples

- **-rw-r--r--**  
**Readable and writable for file owner, only readable for others**
- **-rw-r-----**  
**Readable and writable for file owner, only readable for users belonging to the file group.**
- **drwx-----**  
**Directory only accessible by its owner**
- **-----r-x**  
**File executable by others but neither by your friends nor by yourself. Nice protections for a trap...**

# Man Pages

## ● Manpage

- \$ man ls
- \$ man 2 mkdir
- \$ man man
- \$ man -k mkdir

## ● Manpage sections

- 1 User-level cmds and apps
  - /bin/mkdir

- 4 Device drivers and network protocols
  - /dev/tty
- 5 Standard file formats
  - /etc/hosts
- 6 Games and demos
  - /usr/games/fortune
- 7 Misc. files and docs
  - man 7 locale
- 8 System admin. Cmds

# GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/Root directory

/bin/Basic, essential system commands

/boot/Kernel images, initrd and configuration files

/dev/Files representing devices

/dev/hda: first IDE hard disk

/etc/System and application configuration files

/home/User directories

/lib/Basic system shared libraries

# GNU / Linux filesystem structure

**/lost+found** Corrupt files the system tried to recover

**/media** Mount points for removable media:

**/media/usbdisk, /media/cdrom**

**/mnt** Mount points for temporarily mounted filesystems

**/opt** Specific tools installed by the sysadmin

**/usr/local/** often used instead

**/proc** Access to system information

**/proc/cpuinfo, /proc/version ...**

**/root** root user home directory

**/sbin** Administrator-only commands

**/sys** System and device controls

(cpu frequency, device power, etc.)

# GNU / Linux filesystem structure

**/tmp/Temporary files**

**/usr/Regular user tools (not essential to the system)**

**/usr/bin/, /usr/lib/, /usr/sbin...**

**/usr/local/Specific software installed by the sysadmin  
(often preferred to /opt/)**

**/var/Data used by the system or system servers**

**/var/log/, /var/spool/mail (incoming mail), /var/spool/lpd (print  
jobs)...**

# Files: cut, copy, paste, remove,

- **cat <filenames>**
  - cat /etc/passwd
  - cat fork.c
  - cat <filename1> <filename2>
- **cp <source> <target>**
- **mv <source> <target>**
  - mv a.c b.c
  - mv a.c /tmp/
  - mv a.c /tmp/b.c
- **rm <filename>**
  - rm a.c
  - rm a.c b.c c.c
  - rm -r /tmp/a

# Useful Commands

- **echo**

- echo hi
- echo hi there
- echo “hi there”
- j=5; echo \$j

- **sort**

- sort

- **grep**

- grep bash /etc/passwd
- grep -i display  
/etc/passwd
- egrep -i 'a|b'  
/etc/passwd

- **less <filename>**

- **head <filename>**

# Useful Commands

- **alias**
- **tar**
- **gzip**
- **touch**
- **strings**
- **adduser**
- **su**

# Useful Commands

- **df**
- **du**
- **bc**
- **time**
- **date**
- **diff**
- **wc**

# Network Related Commands

- **ifconfig**
- **ssh**
- **scp**
- **telnet**
- **ping**
- **w**
- **last**
- **whoami**

# Unix job control

- **Start a background process:**

- gedit a.c &
  - gedit

*hit ctrl-z*

**bg**

- **Where did it go?**

- jobs
  - ps

- **Terminate the job: kill it**

# Shell Wildcards

- ? (question mark)
  - Any one character
  - ls a?c
  - ls ??c
- \*
- [ ]
  - Matches a range
  - ls a[1-3].c
- {}
  - ls pic[1-3].{txt,jpg}

any number of

# Configuration Files

- Most applications have configuration files in TEXT format
- Most of them are in /etc
  - /etc/passwd and /etc/shadow
    - Text files containing user accounts
  - /etc/resolv.conf
    - DNS configuration
  - /etc/network/interfaces
    - Network configuration
- /etc/hosts

# **~/.bashrc file**

- **~/.bashrc**

**Shell script read each time a bash shell is started**

- **You can use this file to define**

- Your default environment variables ([PATH](#), [EDITOR](#)...).
  - Your aliases.
  - Your prompt (see the [bash](#) manual for details).
  - A greeting message.

[Also `~/.bash\_history`](#)

# Special devices (1)

## Device files with a special behavior or contents

- **/dev/null**

**The data sink! Discards all data written to this file.**

**Useful to get rid of unwanted output, typically log information:**

```
mplayer black_adder_4th.avi &> /dev/null
```

- **/dev/zero**

**Reads from this file always return \0 characters**

**Useful to create a file filled with zeros:**

```
dd if=/dev/zero of=disk.img bs=1k count=2048
```

# Special devices (2)

- **/dev/random**  
**Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness (“entropy”).**  
**Reads can be blocked until enough entropy is gathered.**
- **/dev/urandom**  
**For programs for which pseudo random numbers are fine.**  
**Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).**

# Special devices (3)

- **/dev/full**
    - Mimics a full device.**
    - Useful to check that your application properly handles this kind of situation.**
- See man full for details.**

# Files names and inodes

Hard Links Vs Soft Links

**EMBEDDED OLE  
OBJECT**

# Shell Programming

# Shell Programming

- **is “programming”**
- **Any programming: Use existing tool to create new utilities**
- **Shell Programming: Use shell facilities to create better utilities**
  - Know “commands” --> More tools
  - Know how to combine them

# Shell Variables

- **Shell supports variables**

- Try:
  - `j=5; echo $j`
  - No space in `j=5`
- Try
  - `set`

# Shell's predefined variables

- **USER**
  - Name of current user
- **HOME**
  - Home directory of current \$USER
- **PS1**
  - The prompt
- **HOSTNAME**
  - Name of the computer
- **OLDPWD**
  - Previous working directory
- **PATH**
  - List of locations to

# Redirection

- **cmd > filename**
  - Redirects the output to a file
  - Try:
    - `ls > /tmp/xyz`
    - `cat /tmp/xyz`
    - `echo hi > /tmp/abc`
    - `cat /tmp/abc`

# Pipes

- **Try**
  - last | less
  - grep bash /etc/passwd | head - 1
  - grep bash /etc/passwd | head - 2 | tail -1
- **Connect the output of LHS command as input of RHS command**
- **Concept of *filters* – programs which read**

# The *test* command

- **test**

- test 10 -eq 10
- test "10" == "10"
- test 10 -eq 9
- test 10 -gt 9
- test "10" >= "9"
- test -f /etc/passwd

**Shortcut notation  
for calling test**

[ ]

[ 10 -eq 10 ]

**Note the space after  
'[' and before ']'**

# The *expr* command and backticks

- **expr**
  - `expr 1 + 2`
  - `a=2; expr $a + 2`
  - `a=2; b=3; expr $a + $b`
  - `a=2;b=3; expr $a \* $b`
  - `a=2;b=3; expr $a | $b`
- **Used for mathematical calculations**
- **backticks ``**
  - `j=`ls`; echo $j`
  - `j=`expr 1 + 2`; echo $j`

# **if then else**

**if [ \$a -lt \$b ]**

**then**

**echo \$a**

**else**

**echo \$b**

**fi**

**if [ \$a -lt \$b ];then  
echo \$a; else echo  
\$b; fi**

**0            TRUE**

**Nonzero FALSE**

# while do done

```
while [ $a -lt $b ]
```

```
do
```

```
echo $a
```

```
a=`expr $a + 1`
```

```
done
```

```
while [ $a -lt $b ]
```

```
do
```

```
while [ $a -lt $b ]; do  
echo $a; a=`expr $a  
+ 1`; done
```

# **for x in ... do done**

```
for i in {1..10}
```

```
do
```

```
echo $i
```

```
done
```

```
for i in *; do echo $i;  
done
```

```
for i in *
```

**read space**

**case \$space in**

**[1-6]\*)**

**Message="one to 6"**

**;;**

**[7-8]\*)**

**Message="7 or 8"**

**;;**

**or [1-8])**

**case ... esac**

**Syntax**

# Try these things

- Print 3<sup>rd</sup> line from /etc/passwd, which contains the word bash
- Print numbers from 1 to 1000
- Create files named like this: file1, file2, file3, ... filen where n is read from user
  - Read i%5<sup>th</sup> file from /etc/passwd and store it in filei
- Find all files ending in .c or .h and create a

# The Golden Mantra

Everything can be done from command line !

Command line is far more powerful than graphical interface

Command line makes you a better programmer

# Mounting

# Partitions

The screenshot shows the Windows Disk Management tool window. At the top, there's a menu bar with File, Action, View, and Help. Below the menu is a toolbar with icons for creating, deleting, and managing partitions. The main area is a table displaying disk information. The columns are Volume, Layout, Type, File System, Status, Capacity, Free Space, and % Fr.

| Volume             | Layout    | Type  | File System | Status         | Capacity | Free Space | % Fr |
|--------------------|-----------|-------|-------------|----------------|----------|------------|------|
| (E:)               | Partition | Basic | FAT32       | Healthy        | 9.76 GB  | 8.37 GB    | 85 % |
| (F:)               | Partition | Basic | FAT32       | Healthy        | 9.76 GB  | 7.24 GB    | 74 % |
| OLDDRIVE (H:)      | Partition | Basic | FAT32       | Healthy (A...) | 4.99 GB  | 586 MB     | 11 % |
| WINDOWS XP (C:)    | Partition | Basic | FAT32       | Healthy (S...) | 9.76 GB  | 2.61 GB    | 26 % |
| Windows Vista (G:) | Partition | Basic | NTFS        | Healthy        | 8.00 GB  | 1.61 GB    | 20 % |
| XPBACKUP (I:)      | Partition | Basic | FAT32       | Healthy        | 4.99 GB  | 4.33 GB    | 86 % |
| XXCOPY (J:)        | Partition | Basic | FAT32       | Healthy        | 9.00 GB  | 4.32 GB    | 47 % |

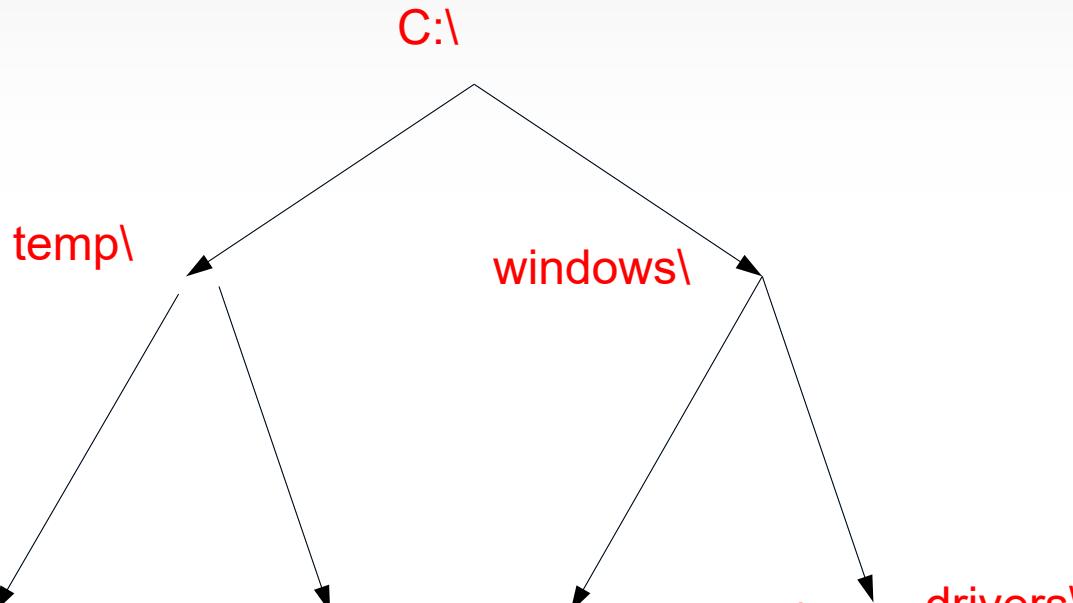
At the bottom left, there's a summary for Disk 0: Basic, 37.30 GB, Online. The partitions (C:, E:, F:, G:) are highlighted with a green border.

|        |                                                      |                                  |                                  |                                               |
|--------|------------------------------------------------------|----------------------------------|----------------------------------|-----------------------------------------------|
| Disk 0 | WINDOWS XP (C:)<br>9.77 GB FAT32<br>Healthy (System) | (E:)<br>9.77 GB FAT32<br>Healthy | (F:)<br>9.77 GB FAT32<br>Healthy | Windows Vista (G:)<br>8.00 GB NTFS<br>Healthy |
|--------|------------------------------------------------------|----------------------------------|----------------------------------|-----------------------------------------------|

# Windows Namespace

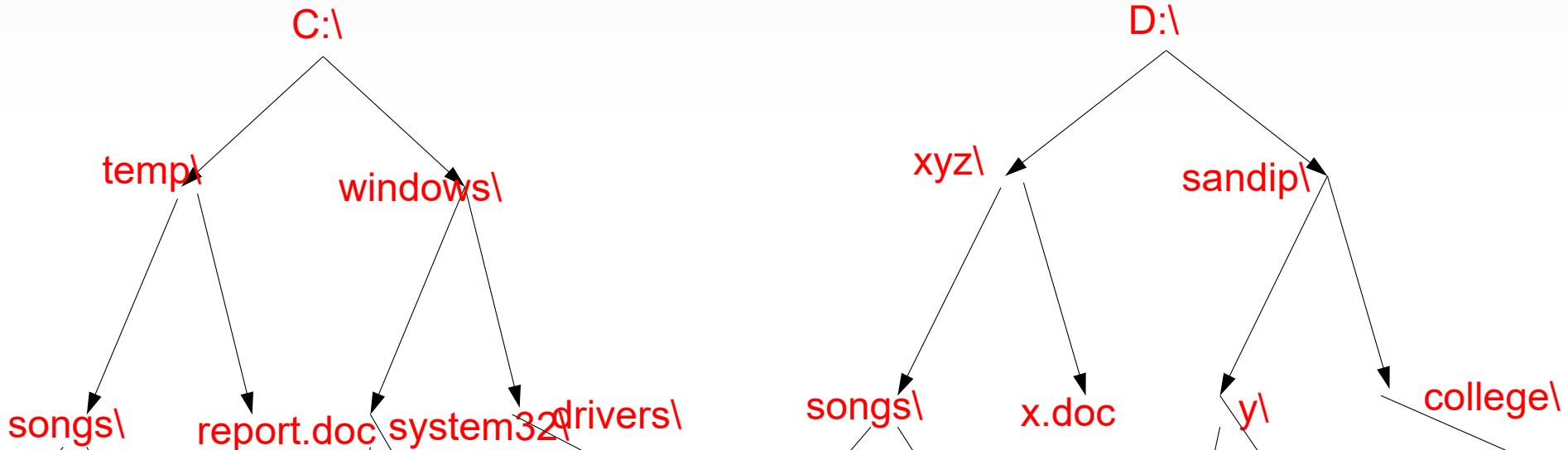
c:\temp\songs\xyz.mp3

- Root is C:\ or D:\ etc
- Separator is also “\”



# Windows Namespace

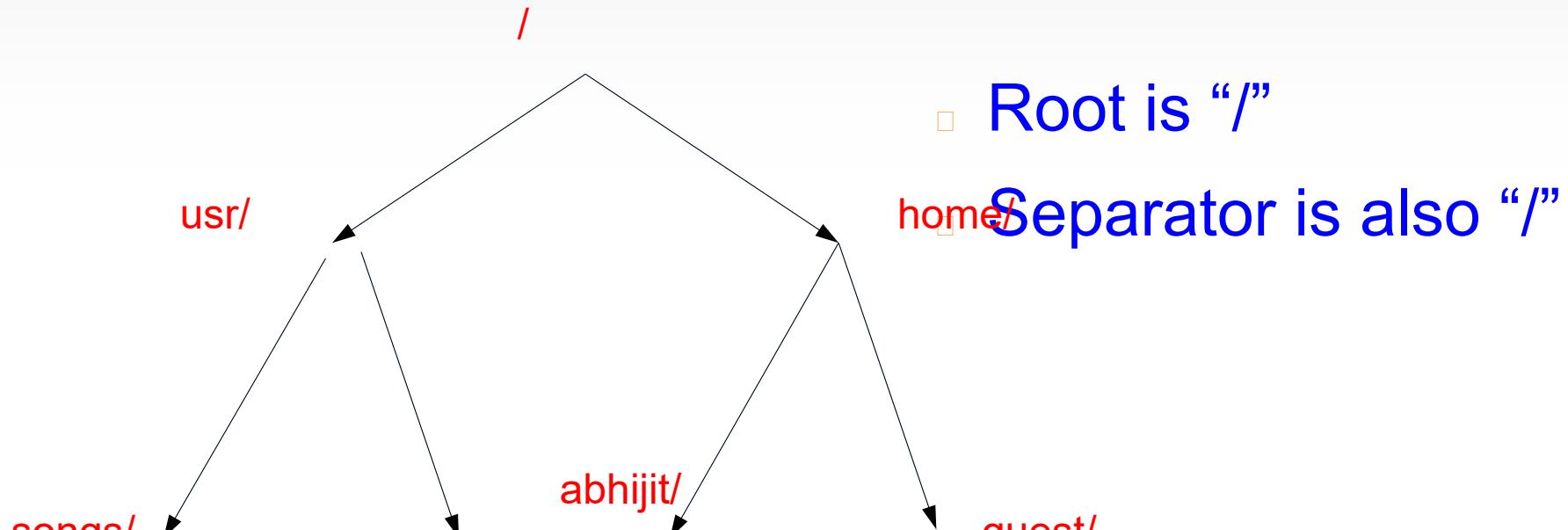
- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D. contains data
  - One “tree” per partition
  - Together they make a “forest”



# Linux Namespace: On a partition

/usr/songs/xyz.mp3

- On every partition:

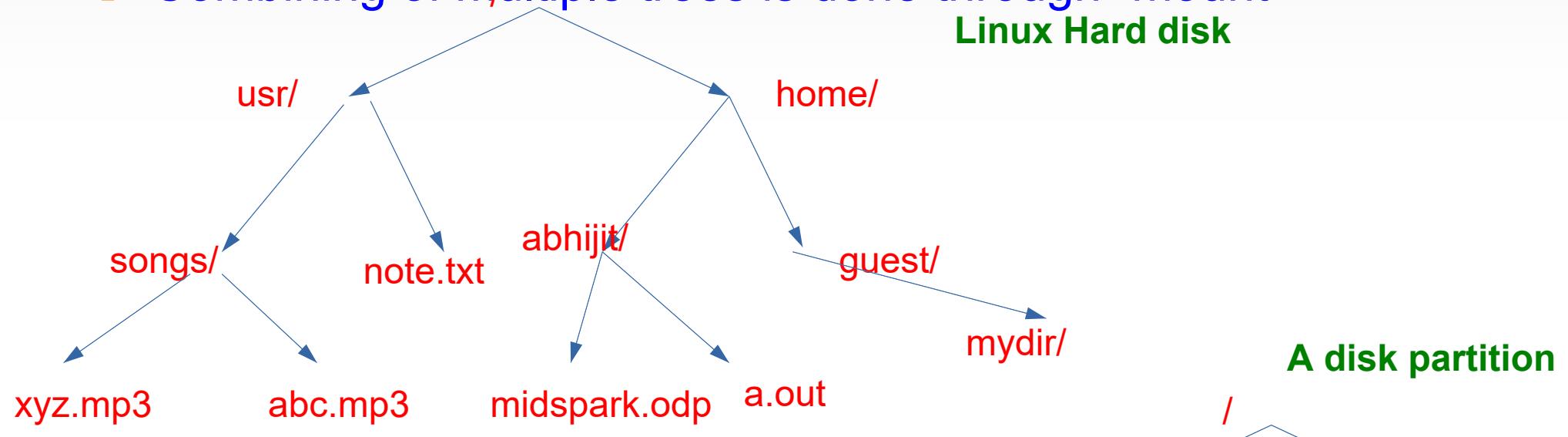


- Root is “/”

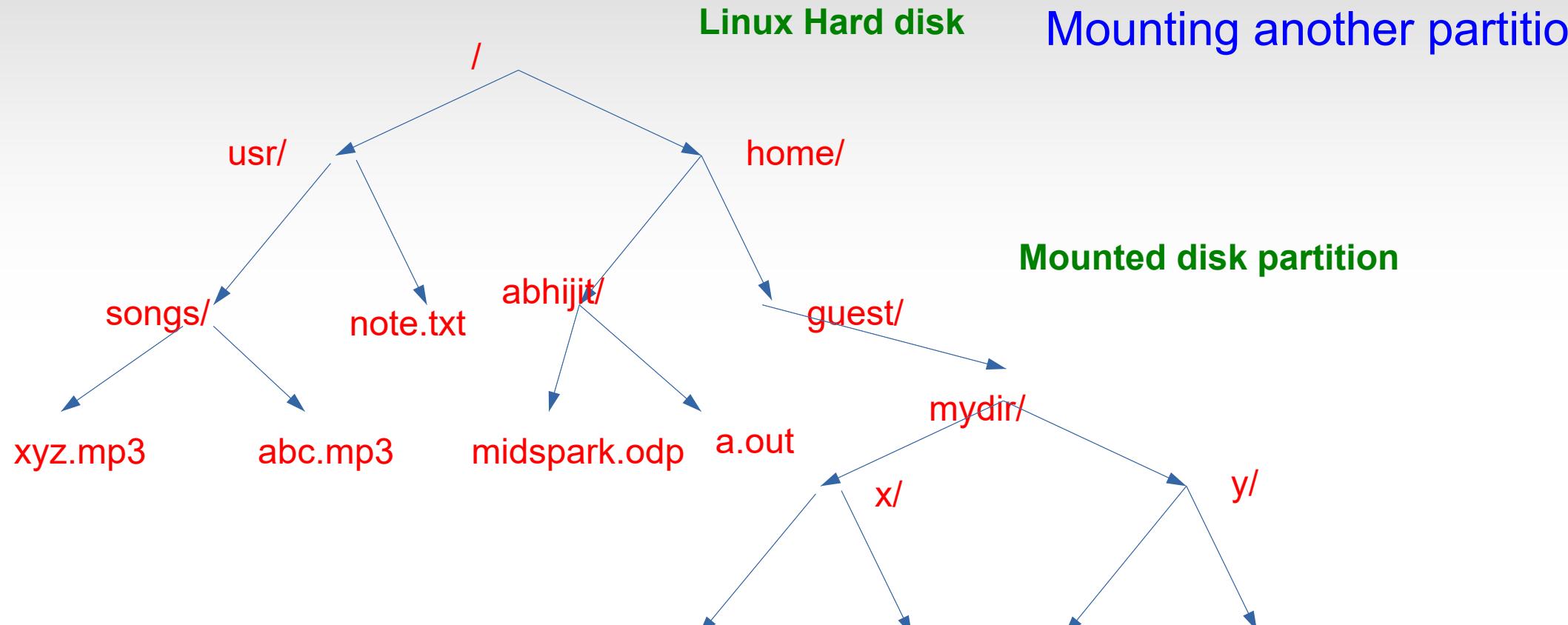
- Separator is also “/”

# Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of **multiple trees** is done through “mount”



# Linux namespace Mounting a partition



**Let's go for a live installation Demo !**

# **Some Shell Gimmicks**

# Terminal Tricks

**Ctrl + n : same as Down arrow.**

**Ctrl + p : same as Up arrow.**

**Ctrl + r : begins a backward search through command history.(keep pressing Ctrl + r to move backward)**

**Ctrl + s : to stop output to terminal.**

# Terminal Tricks

**Ctrl + a : move to the beginning of line.**

**Ctrl + e : move to the end of line.**

**Ctrl + d : if you've type something, Ctrl + d deletes the character under the cursor, else, it escapes the current shell.**

**Ctrl + k : delete all text from the cursor to the end of line**

# Terminal Tricks

**Ctrl + w : cut the word before the cursor; then  
Ctrl + y paste it**

**Ctrl + u : cut the line before the cursor; then  
Ctrl + y paste it**

**Ctrl + \_ : undo typing.**

**Ctrl + I : equivalent to clear.**

# Run from history

First: What's history?

Ans: Run 'history

\$ history

\$ !53

\$ !!

\$ ! - 1

# Math

```
$ echo $(( 10 + 5 )) #15
```

```
$ x=1
```

```
$ echo $(( x++ )) #1 , notice that it is still 1,  
since it's post-incremen
```

```
$ echo $(( x++ )) #2
```

# More Math

```
$ seq 10|paste -sd+|bc
```

```
# How does that work ?
```

```
Using expr
```

```
$ expr 10+20 #30
```

```
$ echo "10*20" |bc
```

# More Math

Using bc

\$ bc

obase=16

ibase=16

AA+1

# More Math

Using bc

\$ bc

ibase=16

obase=16

AA+1

25 / 15

# Fun with grep

```
$ grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
```

# above will only search for IP addresses!

```
$ grep -v bbo filename
```

```
$ grep -w 'abhijit' /etc/passwd
```

```
$ grep -v '^#' file.txt
```

# **xargs: convert stdin to args**

```
$ find . | grep something | xargs rm
```

**xargs is highly powerful**

# rsync

The magic tool to sync your folders!

```
$ rsync -rvupt ~/myfiles /media/abhijit/PD
```

```
$ rsync -rvupt --delete ~/myfiles  
/media/abhijit/PD
```

# find

```
$ find .
```

```
$ find . -type f
```

```
$ find . -type d
```

```
$ find . -name '*.php'
```

```
$ find / -type f -size +4G
```

```
$ find / -type f -size +1M -atime 1
```

# Download : wget, curl

\$ wget foss.coep.org.in

\$ wget -r foss.coep.org.in

\$ wget -r --convert-links foss.coep.org.in

\$ wget -r --convert-links --no-parent  
foss.coep.org.in/fossmeeet/

\$ curl

# Random data

# shuffle numbers from 0-100, then pick 15 of them randomly

```
$ shuf -i 0-100 -n 15
```

# Random pick 100 lines from a file

```
$ shuf -n 100 filename
```

# Run commands remotely

```
$ ssh administrator@foss.coep.org.in
```

```
$ ssh -X administrator@foss.coep.org.in
```

```
$ ssh -X administrator@foss.coep.org.in
```

```
firefox
```

# System Information

```
# Show memory usage,. # print 10 times, at 1  
second interval
```

```
$ free -c 10 -mhs 1
```

```
# Display CPU and IO statistics for devices  
and partitions. # refresh every second
```

```
$ iostat -x -t 1
```

# Surf the web

\$ w3m

\$ links

# Add a user without commands

Know how to edit the */etc/passwd* and  
*/etc/shadow files*

# More tricks

# Show 10 Largest Open Files

```
$ lsof / | awk '{ if($7 > 1048576) print  
$7/1048576 "MB" " " $9 " " $1 }' | sort -n -u | tail
```

# Generate a sequence of numbers

```
$ echo {01..10}
```

# More tricks

# Rename all items in a directory to lower case

```
$ for i in *; do mv "$i" "${i,,}"; done
```

# List IP addresses connected to your server  
on port 80

```
$ netstat -tn 2>/dev/null | grep :80 | awk '{print
```

Credits: <https://onceupon.github.io/Bash-Oneliner/>

<http://www.bashoneliners.com/>

# User Administration

# Users and Groups

- **There is a privileged user called “root”**
  - Can do anything, like “administrator” on Windows
  - Can't login in graphical mode !
- **Other users are normal users**
- **Some users are given “sudo” privileges: called *sudoers***
  - Sudo means “**do as a superuser**”

# Adding/Deleting/Changing users

- **System → Administration → Users and Groups**
- **Click on “Add” to add a user**
  - Asks for password !
  - Provide the details asked for
  - Verify the user was created, by doing 'switch user'
- **Try 'deleting' the user created**

# Software installation

# Some terms

- **.deb**
  - The “setup” file. The installer package. Similar to Setup.exe on windows.
  - Contains all *binary* files and some *shell scripts*
- **repository**
  - A collection of .deb files, categorized according to type (security, main, etc.)

# Software Installation Concept

- **Online installation**

- Use the “Ubuntu Software Center” to select the software, click and install !
- Software is fetched automatically and installed !
- Much easier than Windows !

- **Offline installation**

- Collect ALL .deb files for your application

# Software Installation

- **When we install using Software Center**
  - .deb files are stored in /var/cache/apt/archives folder
- **One needs to be a *sudoer* to install software**
- **Try installing some software on your own and try them out !**

# Network configuration

# Setting up network for a desktop

- **DHCP**
  - Nothing needs to be done !
  - Default during installing Linux
- **Static I/P**
  - System → Preferences → Network connections
  - System → Administration → Network
  - System → Administration → Network tools

# **Network setup for wireless**

- **Just plug and Play !**
- **Network icon shows available wireless network, just click and connect.**

# Reliance/Tata/Idea USB devices

- **Each has a different procedure**
- **One needs to search the web for setting it up**
  - Most of the devices work plug and play on Ubuntu 12.04
  - Some do not work, as fault of the providers, they have not given an installer CD for Linux!

Still Linux community have found ways to work around it !

# Disk Management

# Partition

- **What is C:\ , D:\, E:\ etc on your computer ?**
  - “Drive” is the popular term
  - Typically one of them represents a CD/DVD RW
- **What do the others represent ?**

# Partition

- Your hard disk is one contiguous chunk of storage
  - Lot of times we need to “logically separate” our storage
  - Partition is a “logical division” of the storage
  - Every “drive” is a partition
- A logical chunk of storage is partition

# Partitions

The screenshot shows the Windows Disk Management tool window. At the top, there's a menu bar with File, Action, View, and Help. Below the menu is a toolbar with icons for creating, deleting, and managing partitions. The main area is a table displaying disk information. The columns are Volume, Layout, Type, File System, Status, Capacity, Free Space, and % Fr.

| Volume             | Layout    | Type  | File System | Status         | Capacity | Free Space | % Fr |
|--------------------|-----------|-------|-------------|----------------|----------|------------|------|
| USB (E:)           | Partition | Basic | FAT32       | Healthy        | 9.76 GB  | 8.37 GB    | 85 % |
| USB (F:)           | Partition | Basic | FAT32       | Healthy        | 9.76 GB  | 7.24 GB    | 74 % |
| OLDDRIVE (H:)      | Partition | Basic | FAT32       | Healthy (A...) | 4.99 GB  | 586 MB     | 11 % |
| WINDOWS XP (C:)    | Partition | Basic | FAT32       | Healthy (S...) | 9.76 GB  | 2.61 GB    | 26 % |
| Windows Vista (G:) | Partition | Basic | NTFS        | Healthy        | 8.00 GB  | 1.61 GB    | 20 % |
| XPBACKUP (I:)      | Partition | Basic | FAT32       | Healthy        | 4.99 GB  | 4.33 GB    | 86 % |
| XXCOPY (J:)        | Partition | Basic | FAT32       | Healthy        | 9.00 GB  | 4.32 GB    | 47 % |

At the bottom left, there's a summary for Disk 0: Basic, 37.30 GB, Online. The summary table has four rows corresponding to the partitions above, with the second row highlighted by a green border.

|        |                                                      |                                  |                                  |                                               |
|--------|------------------------------------------------------|----------------------------------|----------------------------------|-----------------------------------------------|
| Disk 0 | WINDOWS XP (C:)<br>9.77 GB FAT32<br>Healthy (System) | (E:)<br>9.77 GB FAT32<br>Healthy | (F:)<br>9.77 GB FAT32<br>Healthy | Windows Vista (G:)<br>8.00 GB NTFS<br>Healthy |
|--------|------------------------------------------------------|----------------------------------|----------------------------------|-----------------------------------------------|

# Managing partitions and hard drives

- **System → Administration → Disk Utility**
- **Hard drive partition names on Linux**
  - /dev/sda → Entire hard drive
  - /dev/sda1, /dev/sda2, /dev/sda3, .... Different partitions of the hard drive
  - Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.