BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

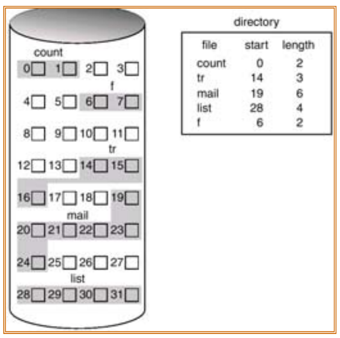
File Systems Implementation
-- Part 3

Notice: The slides for this lecture have been largely based on Professor Perrone's notes. Revised by Xiannong Meng.

Contiguous Allocation

- Each file occupies a set of **contiguous blocks** on the disk.
- Simple: only starting location (block #) and length (number of blocks) are required.
- Suitable for **sequential** and **random** access.
- Wasteful of space: dynamic storage-allocation problem; external fragmentation.
- Files cannot grow unless more space than necessary is allocated when file is created (clearly this strategy can lead to **internal fragmentation**).

Contiguous Allocation of Disk Space



To deal with the dynamic allocation problem (external fragmentation), the system should periodically **compact** the disk.

Compaction may take a long time, during which the system is effectively **down**.

To deal with possibly growing files, one needs to pre-allocate space larger than required at the initial time => this leads to **internal fragmentation**.

Extent-Based Systems

- Many newer file systems (i.e. Linux File System) use a modified contiguous allocation scheme.
- Extent-based file systems allocate disk blocks in **extents**.
- An **extent** is a contiguous set of blocks. Extents are allocated for each file. A file consists of one or more extents.
- Extents can be added to an existing file that needs space to grow. A block can be found given by the location of the first block in the file and the block count, plus a link to the first extent.

Linked Allocation

Each file is a linked list of disk blocks.

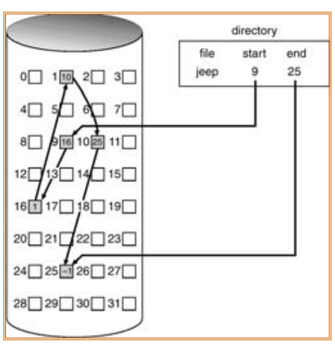
Simple: need only starting address.

Overhead: each block links to the next.

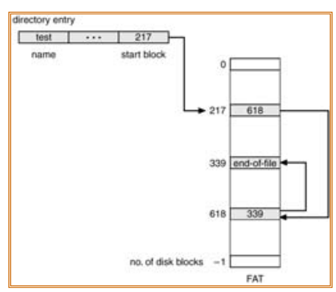
Space cost to store pointer.

Time cost to read one block to find the next.

Internal fragmentation, but not external.
Sequential access comes naturally, random does not.



File-Allocation Table (FAT)



Simple and efficient: One entry for each block; indexed by block number. The table is implements the list linking the blocks in a file.

Growing a file is easy: find a free block and link it in.

Random access is easy.

If the FAT is not cached in memory, a considerable number of disk seeks happens.

Used by MS-DOS and OS/2.

Indexed Allocation

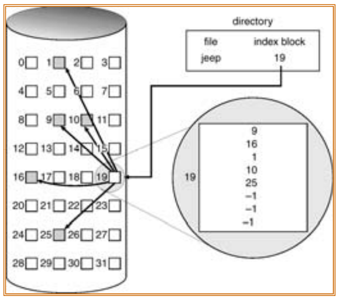
Brings all pointers together into an *index block.*

One index block ***per file***.

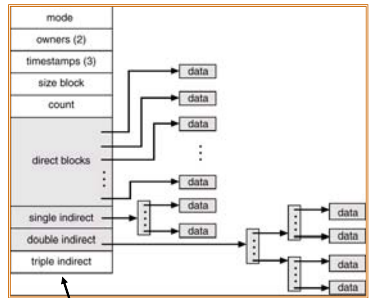
Random access comes easy.

Dynamic access without external fragmentation, but have overhead of index block.

Wasted space: how large should an index block be to minimize the overhead?
- linked index blocks
- multilevel index
- combined scheme



Combined Scheme: UNIX



inode

If file is small enough, use only **direct blocks** pointers.

If number of blocks in file is greater than the number of direct block pointers, use **single**, **double**, or **triple indirect**.

Additional levels of indirection increase the number of blocks that can be associated with a file.

Index blocks can be cached in memory, like FAT. **Access to data blocks, however, may require many disk seeks**.

For Llinux inode, see http://lxr.free-electrons.com/source/fs/ext3/ext3.h

Free-Space Management

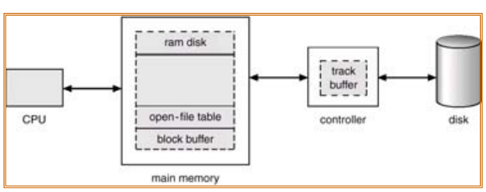
- **Bit map** (1 bit per disk block, fixed size)
  - internal fragmentation
- **Linked list** (free list)
  - external fragmentation
- **Grouping**
  - first free block has address of n free blocks (the last of which has the address of the next n free blocks and so on)
- **Counting**
  - like linked list, but each node points to a cluster of contiguous, free blocks

The OS can cache in memory the free-space management structures for increased performance. Depending on disk size, this may not be easy.

Efficiency and Performance

- **Efficiency dependent on:**
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- **Performance**
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.

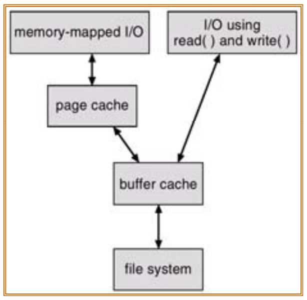
Various Disk-Caching Locations



Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques.
- 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.

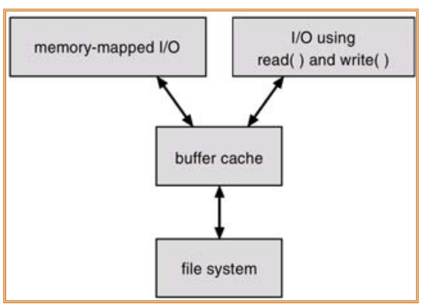
I/O Without a Unified Buffer Cache



Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.

I/O Using a Unified Buffer Cache



Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
- Use system programs to *back up* data from disk to another storage device (floppy disk, magnetic tape).
- Recover lost file or disk by *restoring* data from backup.

Log Structured File Systems

- **Log structured** (or journaling) file systems record each 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. However, the file system may not yet be updated.
- The transactions in the log are asynchronously written to the file system. When the file system is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed.

NFS Protocol

- Provides a set of remote procedure calls (RPC) for remote file operations.
- Early versions of NFS servers are **stateless**; each request has to provide a full set of arguments; NFS V4 is very different, stateful, supporting many more operations
  - http://www.centos.org/docs/4/html/rhel-rg-en-4/ch-nfs.html
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

Three Major Layers of File System Architecture

- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

Schematic View of NFS Architecture