Outline

- Allocation
- Free space management
- Memory mapped files
- Buffer caches

Extent-Based Systems

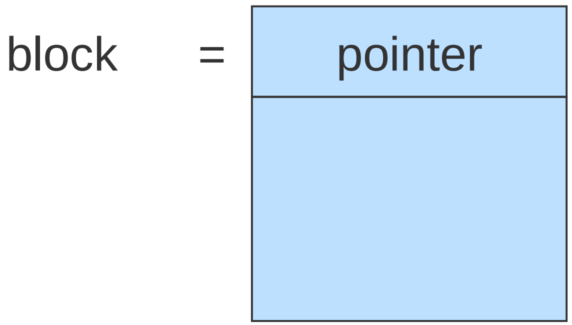
- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme

- Extent-based file systems allocate disk blocks in **extents**

- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents.

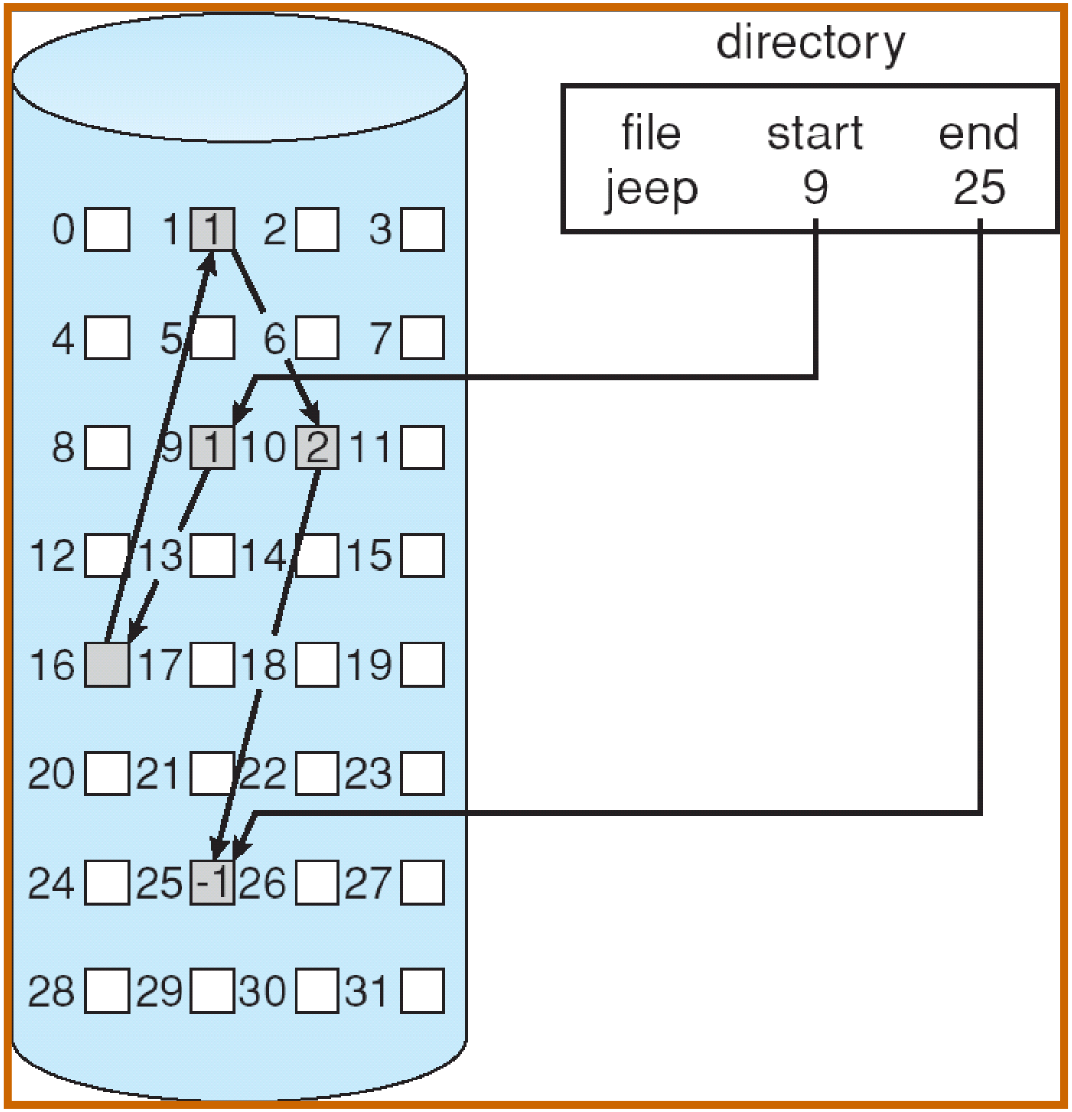
Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.

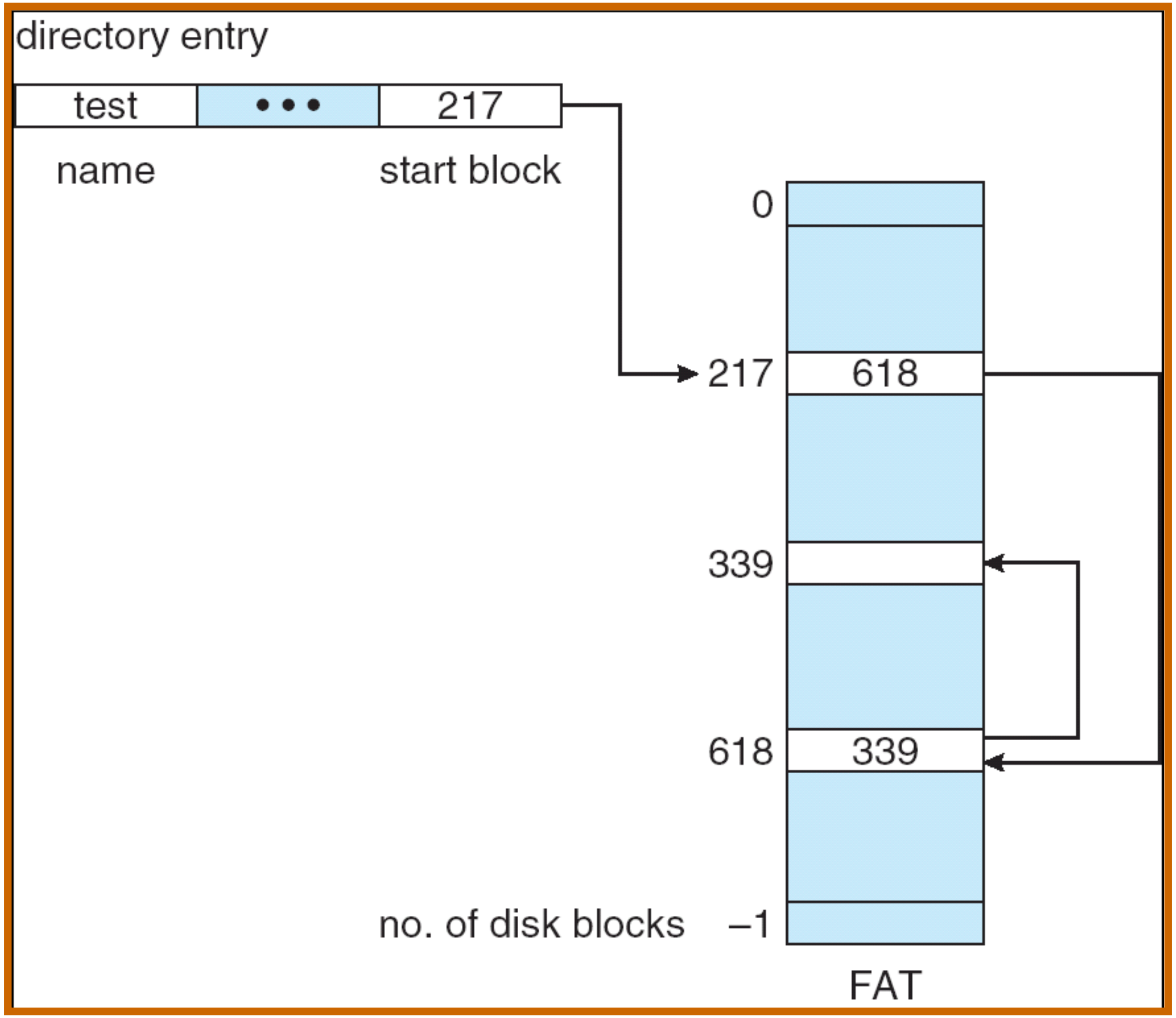


- Simple – need only starting address
- Free-space management system – no waste of space
- No random access

Linked Allocation

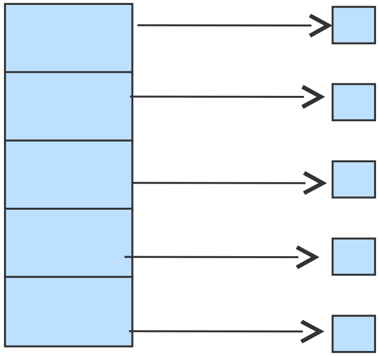


File-Allocation Table (DOS FAT)



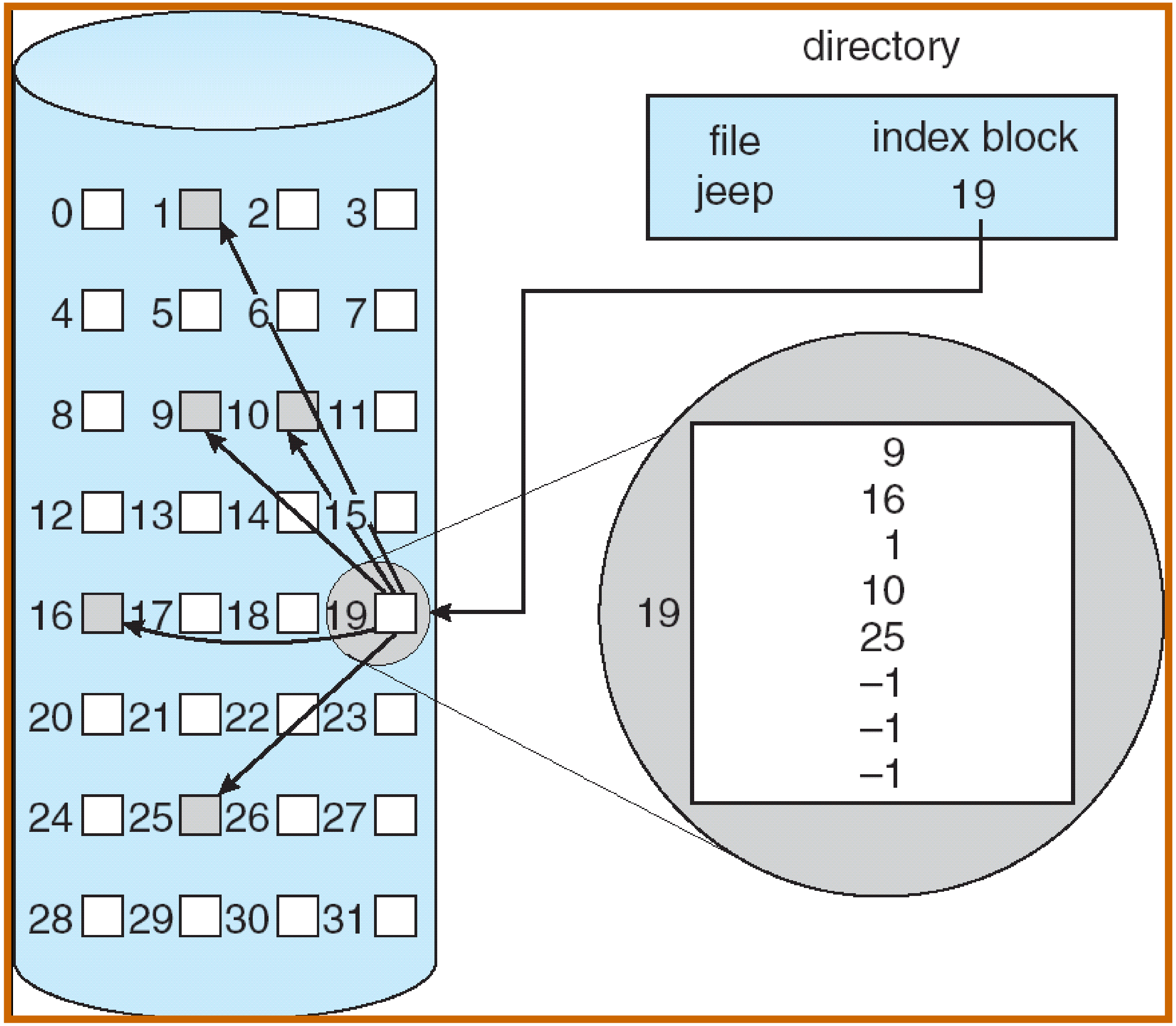
Indexed Allocation

- Brings all pointers together into the *index block.*
- Logical view.



index table

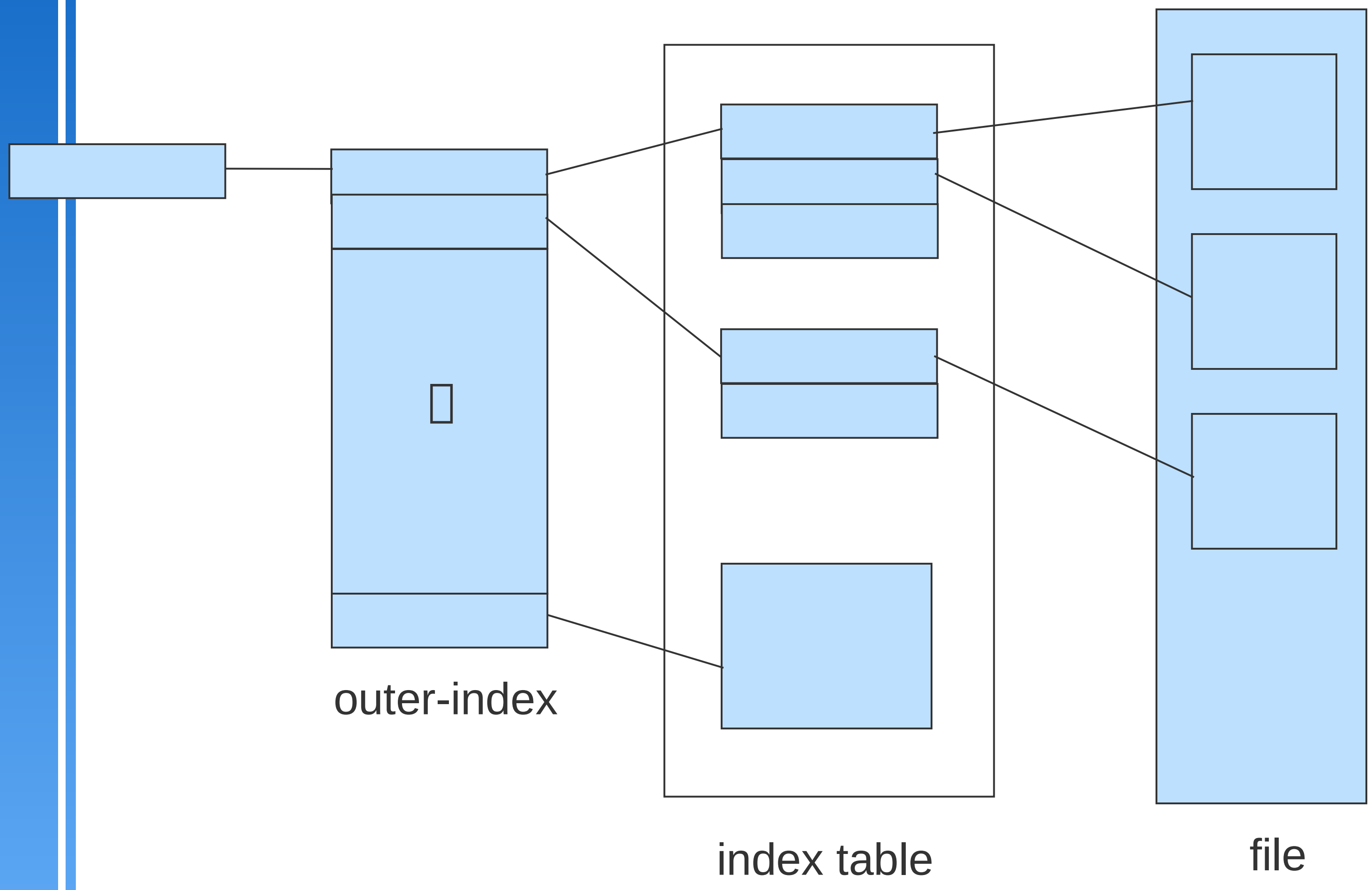
Example of Indexed Allocation



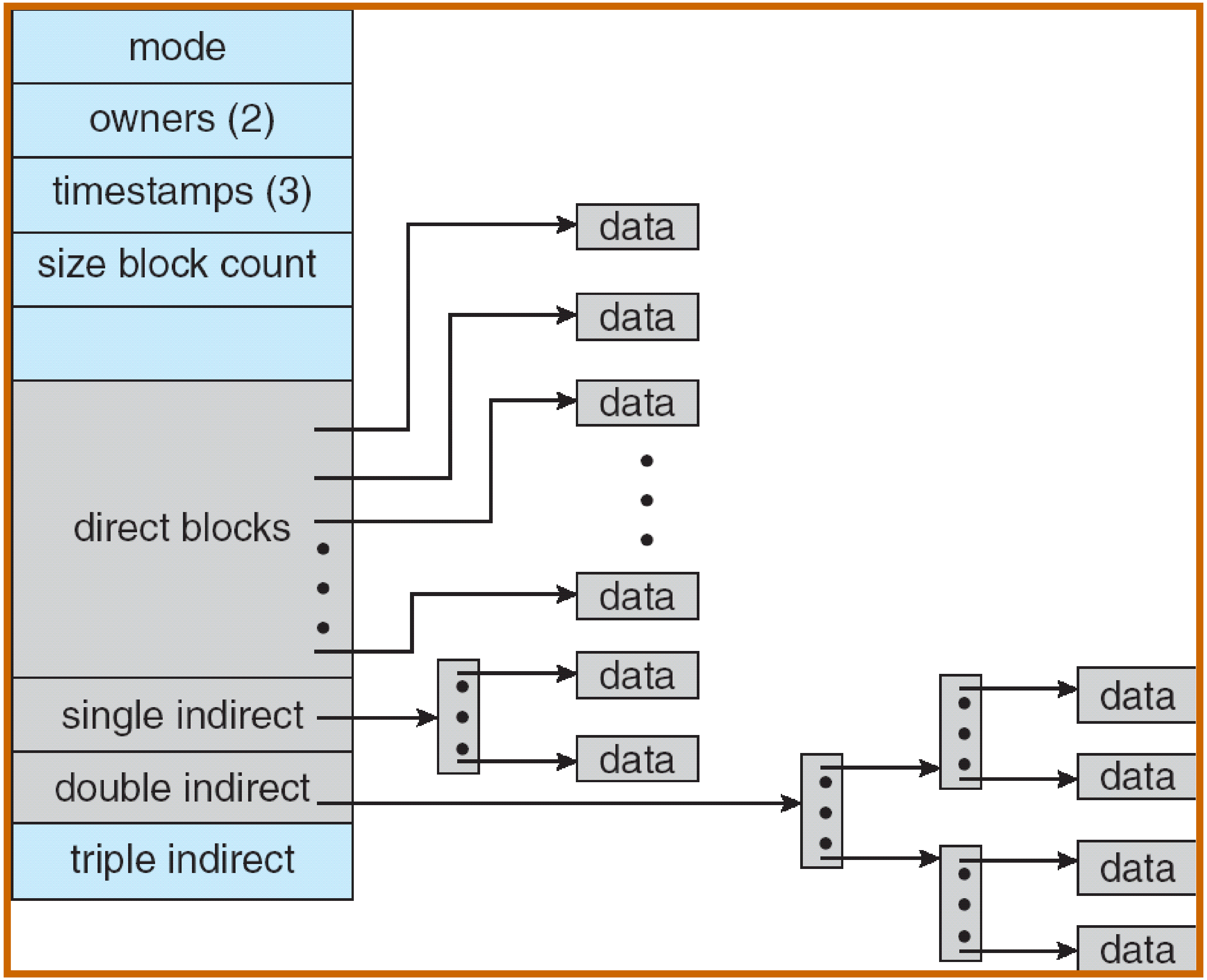
Indexed Allocation (Cont.)

- Need index table to store pointers
- Allows random access by using the indexes
- Dynamic access without external fragmentation, but have overhead of index block.
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.

Indexed Allocation – Mapping (Cont.)



Combined Scheme: UNIX (4K bytes per block)



Free-Space Management

- Bit vector (n blocks)

0	1	2					n-1
						…	

$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

- Block number calculation = (number of bits per word) * (number of 0-value words) + offset of first 1 bit

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

    block size = 2^{12} bytes

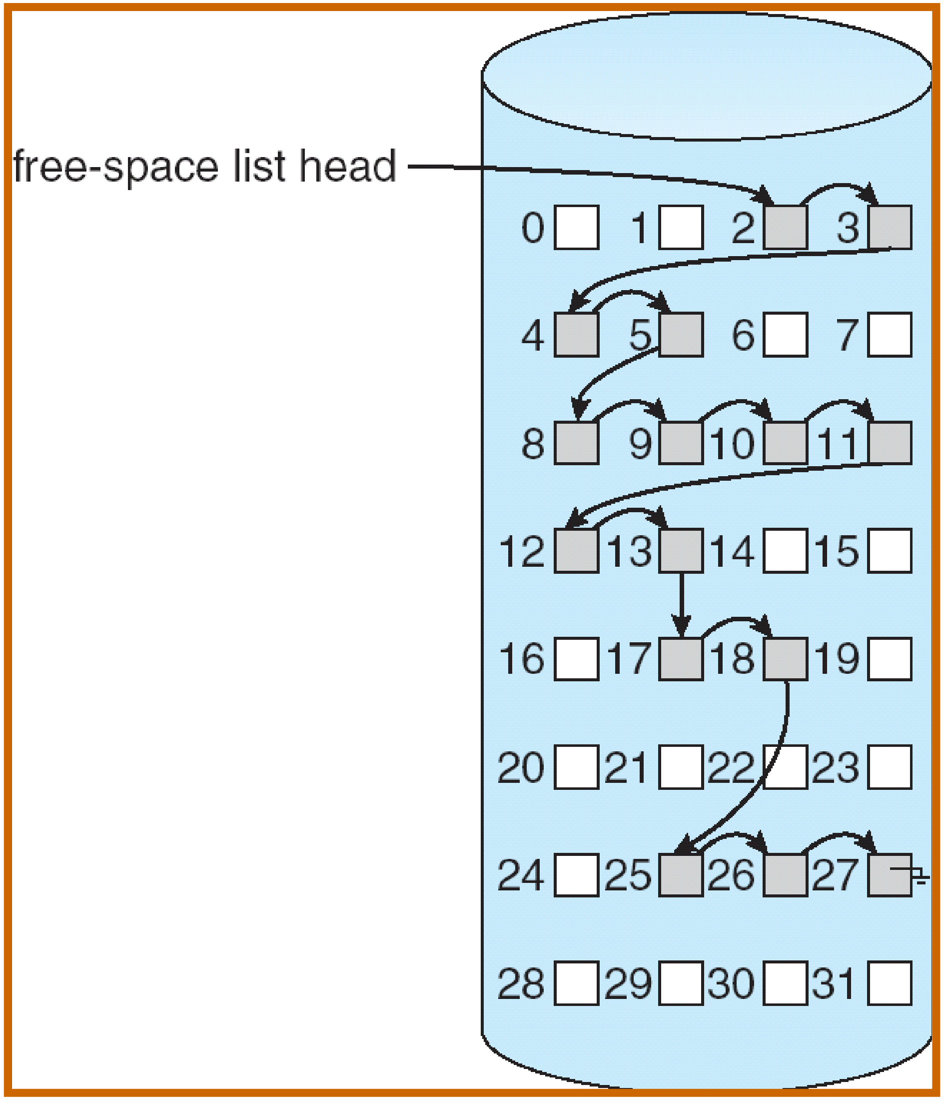
    disk size = 2^{38} bytes (256 Gigabyte)

    $n = 2^{38}/2^{12} = 2^{26}$ bits (or 8 Mbytes)
- Easy to get contiguous files
- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping
- Counting

Free-Space Management (Cont.)

- Need to protect against inconsistency:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ
    - Cannot allow for block[i] to have a situation where bit[i] = 1 in memory and bit[i] = 0 on disk
  - Solution:
    - Set bit[i] = 1 in disk
    - Allocate block[i]
    - Set bit[i] = 1 in memory

Linked Free Space List on Disk



Efficiency and Performance

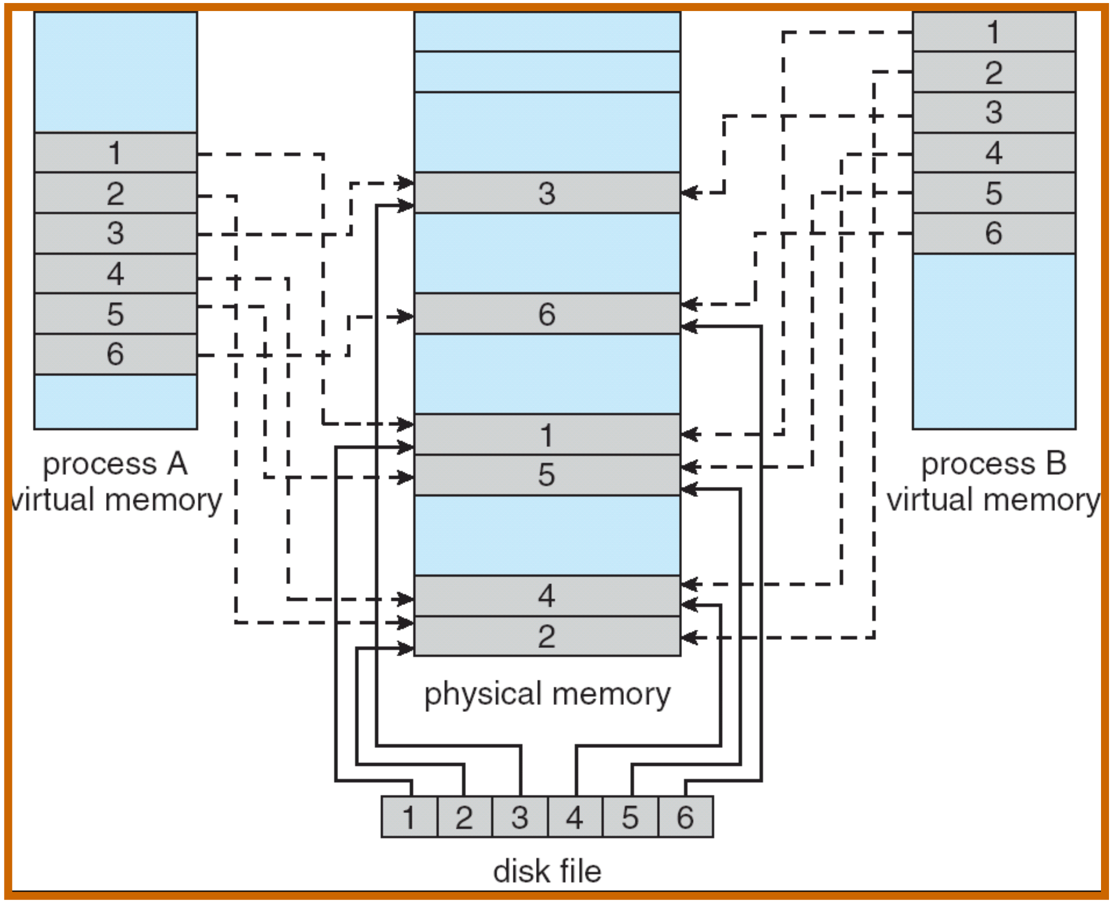
- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry

- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
    - Compare these to LRU
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk
    - It was observed that temporary files were accessed frequently - hence make tmpfs using RAM memory

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than read() write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Sample code using mmap

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

main(int argc, char *argv[], char *envp[]) {
  int fd;
  char *ptr, *path = (argc == 2) ? argv[1] : "file";

  /* Open a file and write some contents. If file already exists,
     delete old contents */
  fd = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0660);
  write(fd, "hello", strlen("hello"));
  write(fd, " world", strlen(" world"));
  close(fd);
```

(continued)

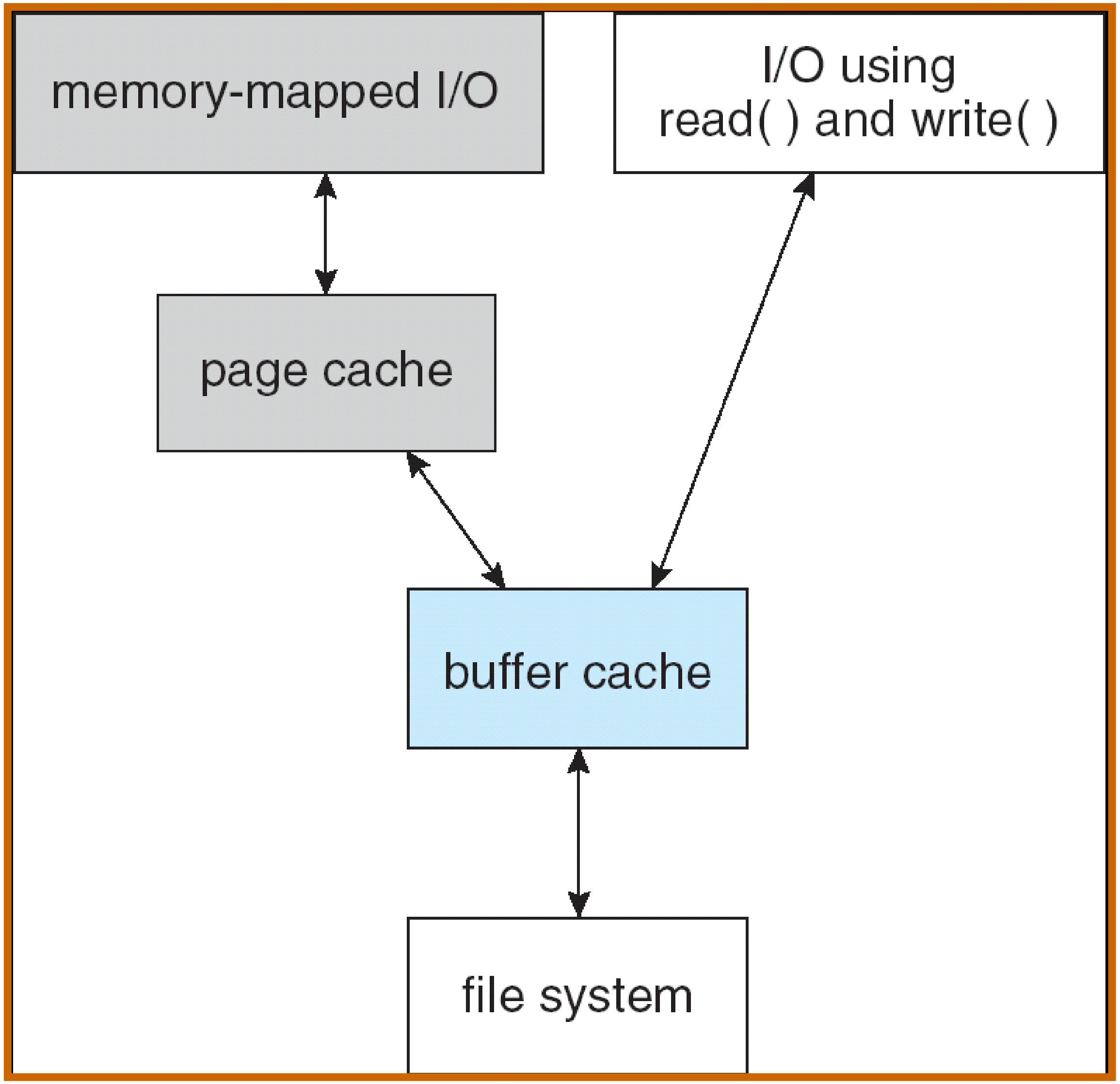
```
fd = open(path, O_RDWR);

// mmap(addr, len, prot, flags, fildes, off);
ptr = mmap(0, 4, PROT_READ|PROT_WRITE,
  MAP_SHARED, fd, 0);
ptr+=2;
memcpy(ptr, "lp ", 3);
munmap(ptr, 4);
close(fd);
}
```

- Transform “hello world” into “help world”

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

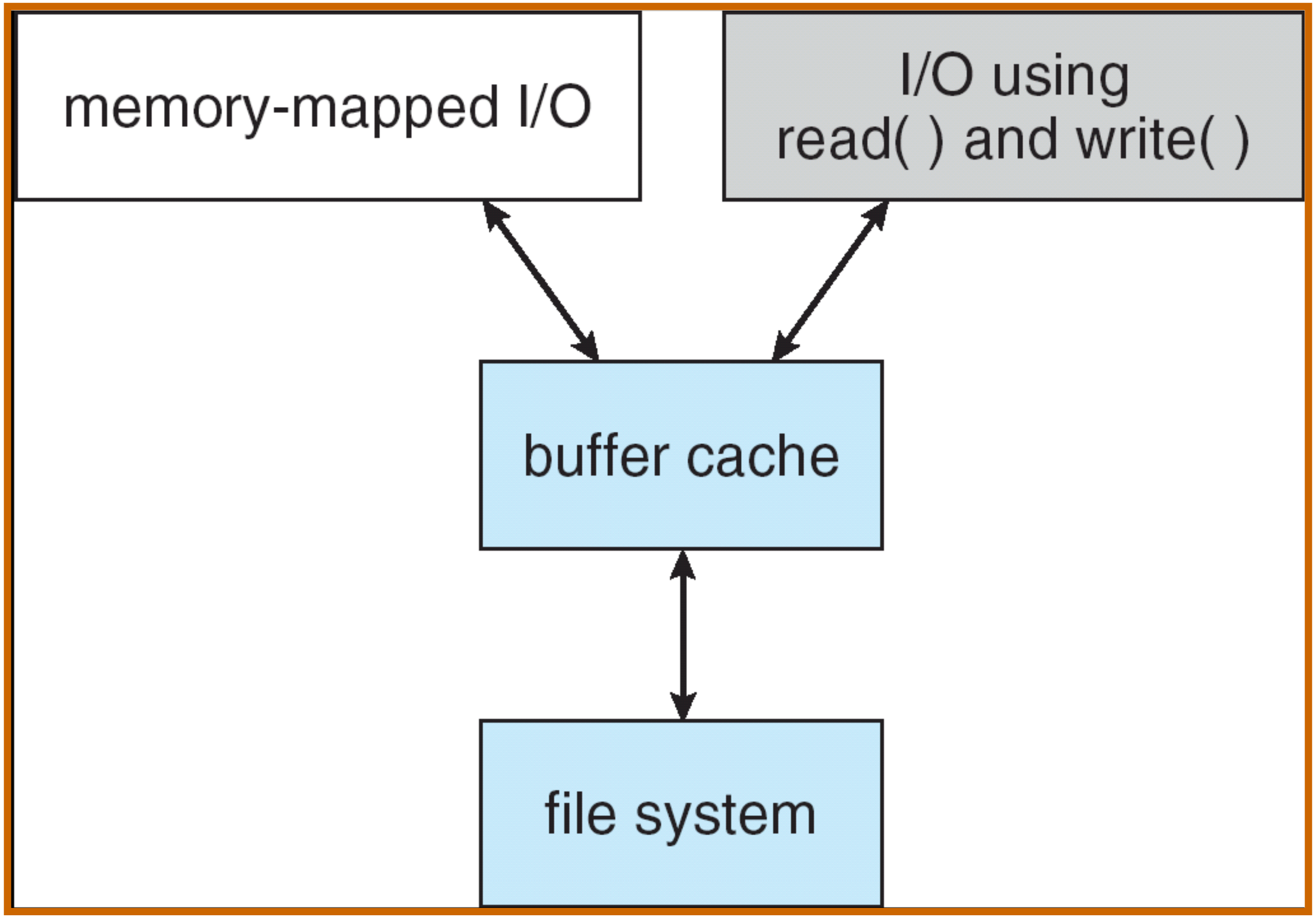
I/O Without a Unified Buffer Cache



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
  - scandisk in DOS, fsck in unix
- Use system programs to **back up** data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)
- 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