Lecture 14 objectives are to:

- Define general terms associated with file systems;
- Present a simple file system,
- Discuss performance measures such as
  - Read/write data access time
  - Vulnerability to disk block errors
  - Internal and external fragmentation

1. Introduction

Solid-state disks can be made from battery-backed RAM or flash EEPROM. A personal computer uses disks made with magnetic storage media and moving parts. While this hard disk technology is acceptable for the personal computer because of its large size (>100s of gibibytes) and low cost (<$100 OEM), it is not appropriate for an embedded system, because of its physical dimensions, electrical power requirements, noise, sensitivity to motion (maximum acceleration), and weight. Embedded applications that might require disk storage include data acquisition, data base systems, and signal generation. You can also use a solid state disk in an embedded system to log debugging information. The personal desk accessory (PDA) devices currently employ solid-state disks because of their small physical size, and low power requirements. Unfortunately, solid-state disks have smaller sizes and higher cost/bit than the traditional magnetic storage disk. In Lab 5, you will implement a solid state disk using an SD card. A typical 2-Gibibyte SD card costs about $10. The cost/bit is therefore about $5/Gibibyte. Compare this cost to a 1-Tibibyte hard drive that costs about $100. This cost/bit is 50 times cheaper at $0.1/Gibibyte.

2. File system allocation

There are three components of the file system: directory, allocation, and free-space management. This lecture introduces fundamental concepts and describes a simple file system. When designing a file system, it is important to know how it will be used. For example, when recording and playing back sound, the data will be written and read in a sequential manner. Conversely, an editor produces more of a random access pattern for data reading and writing. Furthermore, an editor requires data insertion and removal anywhere from the beginning to the end of a file. The reliability of the storage medium and the cost of lost information will also affect the design of a file system.

Contiguous allocation places the data for each file at consecutive blocks on the disk as shown in Figure 14.1. Each directory entry contains the file name, the block number of the first block, the length in blocks. This method has similar problems as a memory manager. You could choose first fit, best fit, or worst fit algorithms to manage storage. If the file can increase in size, either you can leave no extra space, and copy the file elsewhere if it expands, or you can leave extra space when creating a new file. Assuming the directory is in memory, it takes only one disk block read to access any data in the file. A disadvantage of this method is you need to know the maximum file size when a file is created, and it will be difficult to grow the file size beyond its initial allocation.

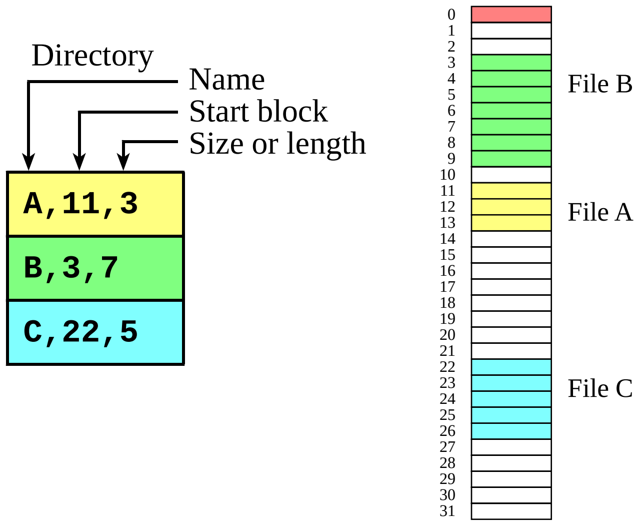


Figure 14.1. A simple file system with contiguous allocation.

Checkpoint 14.1: *This disk in Figure 14.1 has 32 blocks with the directory occupying block 0. The disk block size is 512 bytes. What is the largest new file that can be created?*

Checkpoint 14.2: *You wished to allocate a new file requiring 1 block on the disk in Figure 14.1. Using first fit allocation, where would you put the file? Using best fit allocation, where would you put the file? Using worst fit allocation, where would you put the file?*

Linked allocation places a block pointer in each data block containing the block address of the next block in the file. Each directory entry contains: a file name, and the block number of the first block. There needs to be a way to tell the end of a file. The directory could contain the file size, each block could have a counter, or there could be an end-of-file marker in the data itself. Sometimes, there is also a pointer to the last block, making it faster to add to the end of the file. Assuming the directory is in memory, and the file is stored in N blocks, it takes on average N/2 disk block reads to access any random piece of data on the disk. Sequential reading and writing are efficient, and it will also be easy to append data at the end of the file.

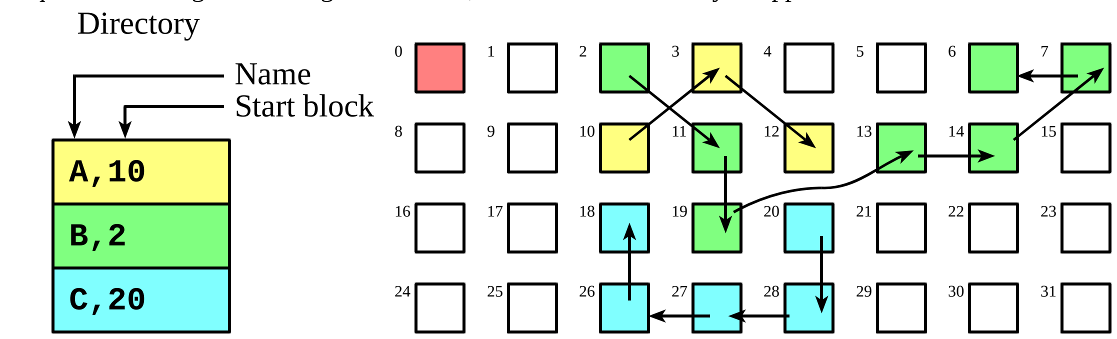


Figure 14.2. A simple file system with linked allocation.

Checkpoint 14.3: *If the disk holds 2 Gibibytes of data broken into 512-byte blocks, how many bits would it take to store the block address?*

Checkpoint 14.4: *If the disk holds 2 Gibibytes of data broken into 32k-byte blocks, how many bits would it take to store the block address?*

Checkpoint 14.5: *This disk in Figure 14.2 has 32 blocks with the directory occupying block 0. The disk block size is 512 bytes. What is the largest new file that can be created?*

Checkpoint 14.6: *How would you handle the situation where the number of bytes stored in a file is not an integer multiple of the number of data bytes that can be stored in each block?*

We can also use the linked to manage the free space as shown in Figure 14.3. If the directory were lost, then all file information except the filenames could be recovered. Putting number of the last block into the directory with double-linked pointers improves recoverability. If one data block were damaged, then remaining data blocks could be rechained loosing the information in the one damaged block.

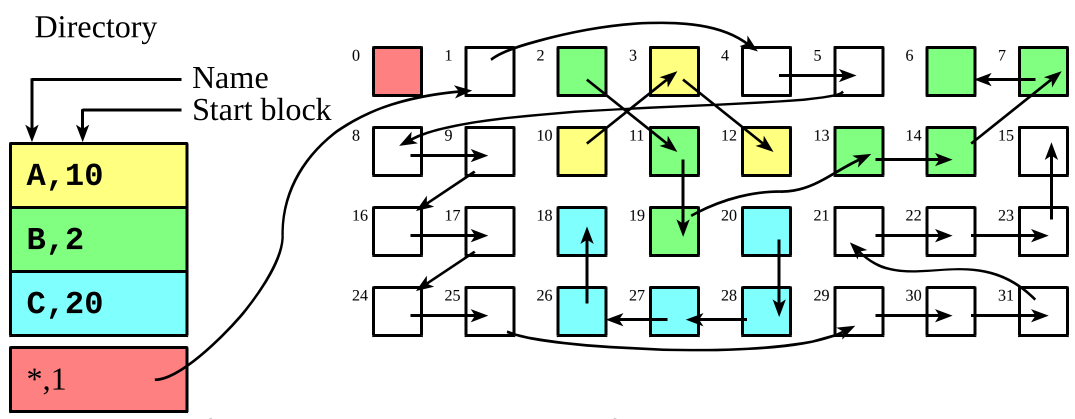


Figure 14.3. A simple file system with linked allocation and free space management.

Indexed allocation uses an index table to keep track of which blocks are assigned to which files. Each directory entry contains: a file name, an index for the first block, and the total number of blocks as shown in Figure 14.4. One implementation of indexed allocation places all pointers for all files on the disk together in one index table. Another implementation allocates a separate index table for each file. Often, this table is so large it is stored in several disk blocks. For example, if the block number is a 16-bit number, and the disk block size is 512 bytes, then only 256 index values can be stored in one block. Also for reliability, we can store multiple copies of the index on the disk. Typically the entire index table is loaded into memory while the disk is in use. The RAM version of the table is stored onto the disk periodically, and when the system is shut down. Indexed allocation is faster than linked allocation if we employ random access. If the index table is in RAM, then any data within the file can be found with just one block read. One way to improve reliability is to employ both indexed and linked allocation. The indexed scheme is used for fast access, and the links can be used to rebuild the file structure after a disk failure.

Checkpoint 14.7: *If the block number is a 16-bit number, and the block size is 512 bytes, what is the maximum disk size?*

Checkpoint 14.8: *A disk with indexed allocation has 2 gibibytes storage. Each file has a separate index table, and that index occupies just one block. The disk block size is 1024 bytes. What is the largest file that can be created? Give two ways to change the file system to support larger files.*

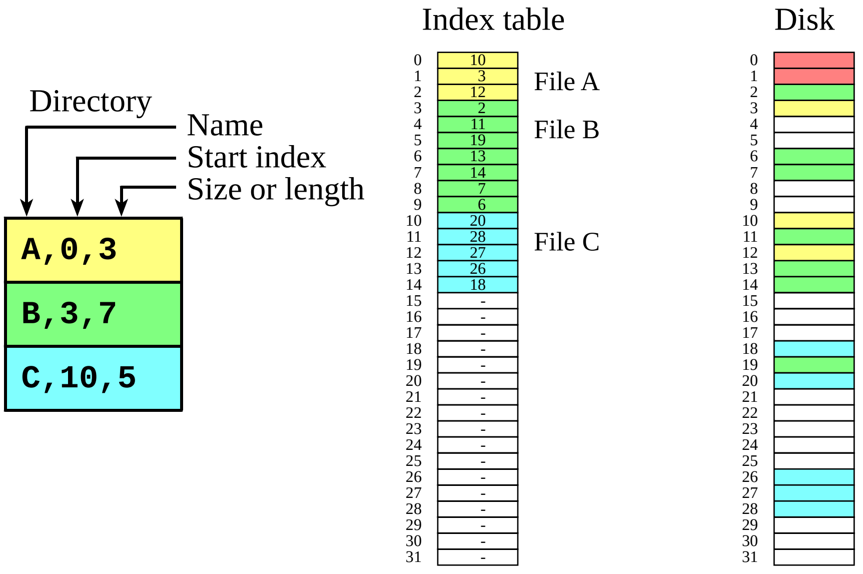


Figure 14.4. A simple file system with indexed allocation.

> ***Checkpoint 14.9:*** *This disk in Figure 14.4 has 32 blocks with the directory occupying block 0 and the index table in block 1. The disk block size is 512 bytes. What is the largest new file that can be created?*

3. Simple file system

3.1. Directory

The first component is the **directory**, as shown in Figure 14.5. The block size is 512 bytes. In order to support disks larger than 32 Mebibytes, 32-bit block pointers will be used. The directory contains a mapping between the symbolic filename and the physical address of the data. Specific information contained in the directory might include the file name, the number of the first block containing data, and the total number of bytes stored in the file. One possible implementation places the directory in block 0. In this simple system, all files are listed in this one directory (there are no subdirectories). There is one fixed-size directory entry for each file. A filename is stored as an ASCII string in a fixed-size array. A null-string (first byte 0) means no file. Since the directory itself is located in block 0, 0 can be used as a null-block pointer. In this simple scheme the entire directory must fit into the block 0, the maximum number of files can be calculated by dividing the block size by the number of bytes used for each directory entry. In Figure 14.5 each directory entry is 16 bytes, so there can be up to 512/16 = 32 files. We will need one directory entry to manage the free space on the disk, so this disk format can have up to 31 files.

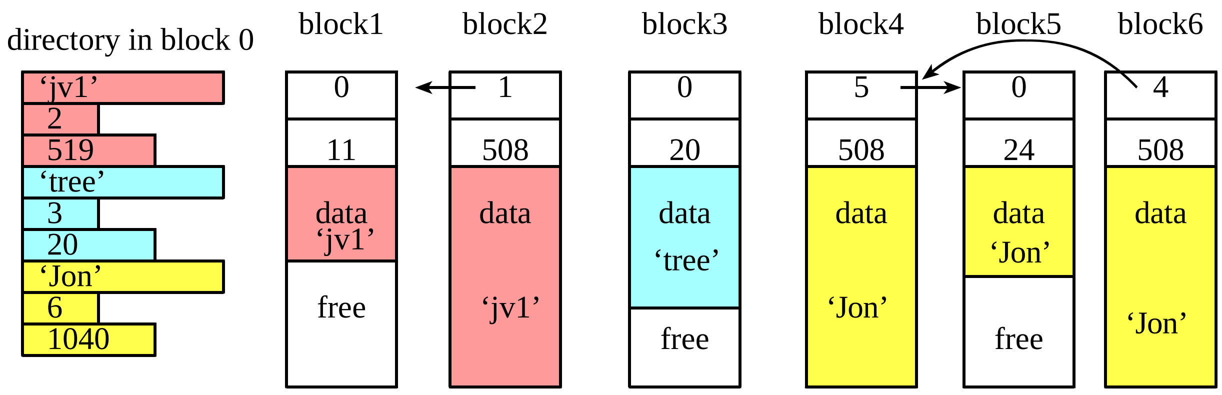


Figure 14.5. Linked file allocation with 512-byte blocks.

Other information that one often finds in a directory entry is a pointer to the last block of the file, access rights, date of creation, date of last modification, and file type.

3.2. Allocation

The second component of the file system is the **logical to physical address translation**. Logically, the data in the file are addressed in a simple linear fashion. The logical address ranges from the first to the last. There are many algorithms one could use to keep track of where all the data for a file belongs. This simple file system uses **linked allocation** as illustrated in Figure 14.5. Recall that the directory contains the block number of the first block containing data for the file. The start of every block contains a link (the block number) of the next block, and a byte count (the number of data bytes in this block). If the link is zero, this is last block of the file. If the byte count is zero, this block is empty (contains no data). Once the block is full, the file must request a free block (empty and not used by another file) to store more data. Linked allocation is effective for systems that employ sequential access. Sequential read access involves two functions similar to a magnetic tape: rewind (start at beginning), and read the next data. Sequential write access simply involves appending data to the end of the file. Figure 14.5 assumes the block size is 512 bytes and the filename has up to 9 characters. The null-terminated ASCII string is allocated 10 bytes regardless of the size of the string. The size entry in the directory (e.g., file 'jv1' has 519 bytes) is 32 bits but the block pointers have only 16-bit precision. Since each data block has a 2-byte link and a 2-byte counter, each block can store up to 508 bytes of data.

3.3. Free-space management

The third component of the file system is **free-space management**. Initially all blocks except the one used for the directory are free, available for files to store data. To store data into a file, blocks must be allocated to the file. When a file is deleted, its blocks must be made available again. One simple free-space management technique uses **linked allocation**, similar to the way data is stored. Assume there are N blocks numbered from 0 to N-1. An empty file system is shown in Figure 14.6. Block 0 contains the directory and blocks 1 to N-1 are free. You could assign the last directory entry for free space management. This entry is hidden from the user. E.g., this free space file can't be opened, printed, or deleted. It doesn't use any of the byte count fields, but does use the links to access all the free blocks. Initially, all the blocks (except the directory itself) are linked together, with the special directory entry pointing to the first one and the last one having a null pointer.

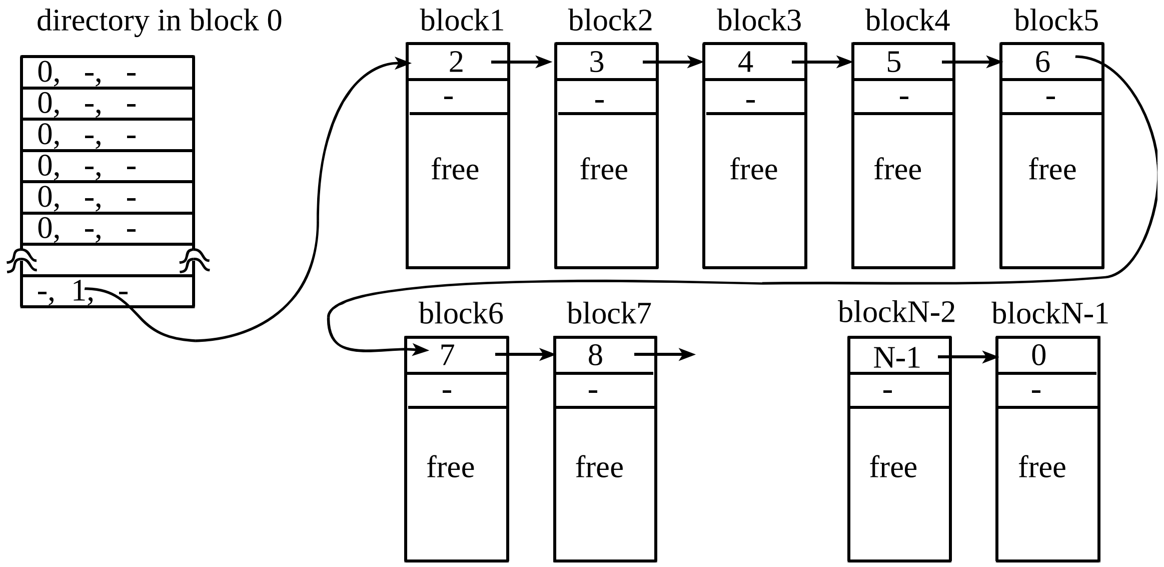


Figure 14.6. Free-space management.

When a file requests a block, it is unlinked from the free space and linked to the file. When a file is deleted, all its blocks are linked to the free space again. Figure 14.7 shows the free space after File ‘i’ is created and has 26 bytes of data stored, and File ‘f’ is created and has 555 bytes of data stored. Blocks 1 to 3 have file data and blocks 4 to N-1 are free.

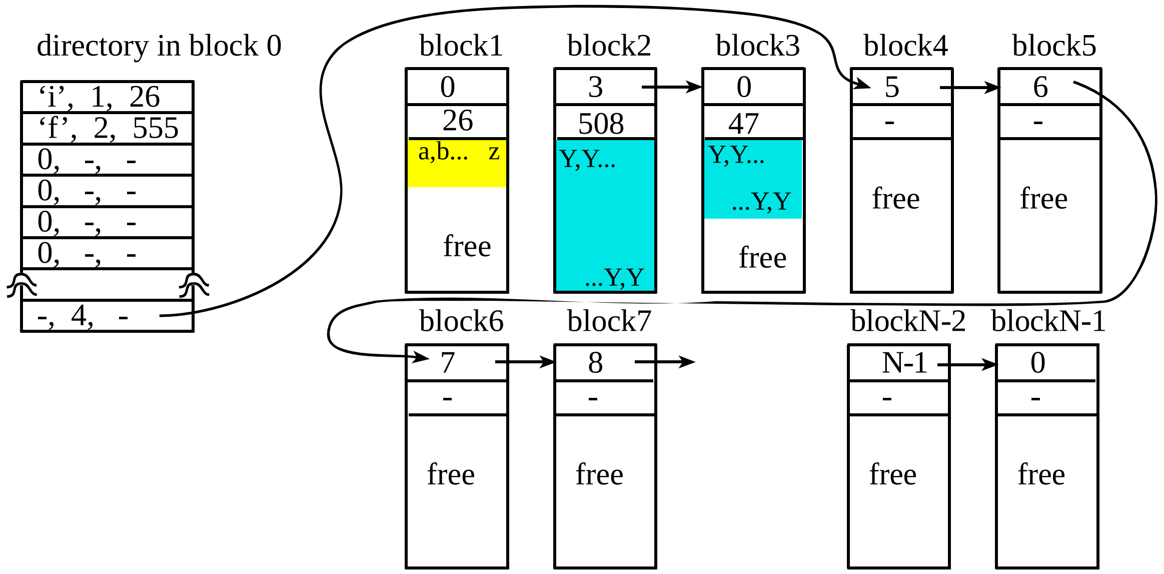


Figure 14.7. Free-space management after two files are created.

Now, if file ‘i’ is deleted its block is returned to the free space list. We could add it to the beginning or the end of the list. Figure 14.8 shows the free block was added to the beginning of the free space list.

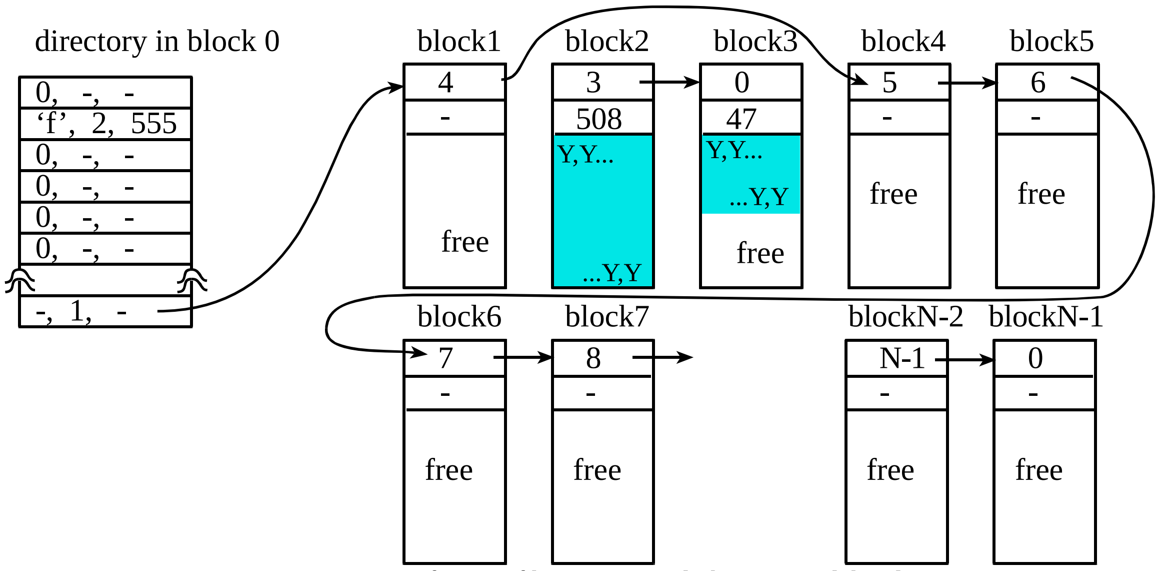


Figure 14.8. Free-space management after two files are created, then one is deleted.

> ***Checkpoint 14.10:*** *If the directory shown in Figure 14.5 allocated 6 bytes for the filename instead of 10, how many files could it support?*

4. File allocation table

The file allocation table (FAT) is a mixture of indexed and linked allocation as shown in Figure 14.9. Each directory entry contains: a file name, and the block number of the first block. The FAT is just a table containing a linked list of blocks for each file. Figure 14.9 shows file A in blocks 10, 3, and 12. The directory has block 10, the initial block. The FAT contents at index 10 is a 3, so 3 is the second block. The FAT contents at index 3 is a 12, so 12 is the third block. The FAT contents at index 12 is a NULL, so there are no more blocks in file A.

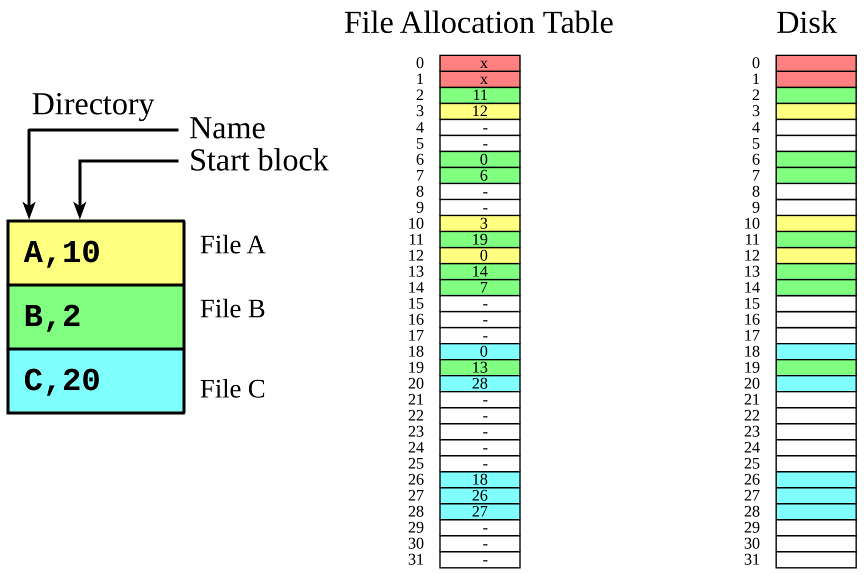


Figure 14.9. A simple file system with a file allocation table.

Silberschatz, Galvin and Gagne call the FAT a “linked” scheme, because it has links. However, some scientists call it an “indexed” scheme, because it has the speed advantage of an “indexed” scheme when the table for the entire disk is kept in main memory. If the directory and FAT are in memory, it takes just one disk read to access any data in a file. If the disk is very large, the FAT may be too large to fit in main memory. If the FAT is stored on the disk, then it will take 2 or 3 disk accesses to find an element within the file. The - in Figure 14.9 represent free blocks. In Figure 14.10 we can chain them together in the FAT to manage free space.

Checkpoint 14.11: *This disk in Figure 14.9 has 32 blocks with the directory occupying block 0 and the FAT in block 1. The disk block size is 512 bytes. What is the largest new file that can be created?*

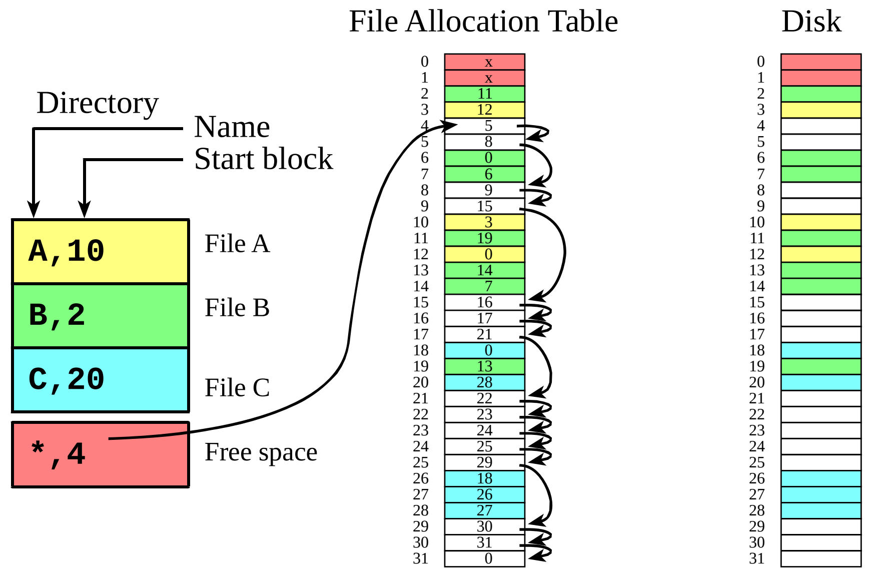


Figure 14.10. The simple file system with a file allocation table showing the free-space management.

5. Internal fragmentation

Internal fragmentation is storage that is allocated for the convenient of the operating system but contains no information. This space is wasted. Often this space is wasted in order to improve speed or provide for a simpler implementation. The fragmentation is called "internal" because the wasted storage is inside the allocated region. In most file systems, whole blocks (or even clusters of blocks) are allocated to individual files, because this simplifies organization and makes it easier to grow files. Any space left over between the last byte of the file and the first byte of the next block is a form of internal fragmentation called *file slack* or *slack space*. File ‘i’ in Figure 14.7 was allocated an entire block capable of storing 508 bytes of data. However, only 26 of those locations contained data, so the remaining 482 bytes can be considered as internal fragmentation. The pointers and counters used by the OS to manage the file are not considered internal fragmentation because even though the locations do not contain data, the space is not wasted. Whether or not to count the OS pointers and counters as internal fragmentation is a matter of debate. As is the case with most definitions, it is appropriate to document your working definition of internal fragmentation whenever presenting performance specifications to your customers.

Many compilers will align variables on a 32-bit boundary. If the size of a data structure is not divisible by 32 bits, it will skip memory bytes so the next variable is aligned onto a 32-bit boundary. This wasted space is also internal fragmentation.

Standard ASCII requires just 7 bits per character. Most computer systems assign 8 or 16 bits for each character. Similarly, if you store 12-bit ADC data into 2 bytes per sample, there will be 4 bits of wasted space for each sample. The unused bits are a form of internal fragmentation.

6. External fragmentation

External fragmentation exists when the largest file that can be allocated is less than the total amount of free space on the disk. External fragmentation occurs in systems that require contiguous allocation, like a memory manager. External fragmentation would occur within a file system that allocated disk space in contiguous blocks. Over time free storage becomes divided into many small pieces. It is a particular problem when an application allocates and deallocates regions of storage of varying sizes. The result is that although free storage is available, it is effectively unusable because it is divided into pieces that are too small to satisfy the demands of the application. The term "external" refers to the fact that the unusable storage is outside the allocated regions.

For example, assume we have a file system employing contiguous allocation. A new file with five blocks might be requested, but the largest contiguous chunk of free disk space is only three blocks long. Even if there are ten free blocks, those free blocks may be separated by allocated files, one still cannot allocate the requested file with five blocks, and the allocation request will fail. This is external fragmentation because there are ten free blocks but the largest file that can be allocated is three blocks.

7. Additional considerations

Wear-leveling (there is a maximum number of erase cycles) (match to hardware not file system)

Optimizing for speed (know your hardware, predict the future)

Clustering (disk size, speed, internal fragmentation)

Legacy (backwards compatibility)

Reliability (detect errors, prevent errors, recover from errors)

Low-voltage interrupt

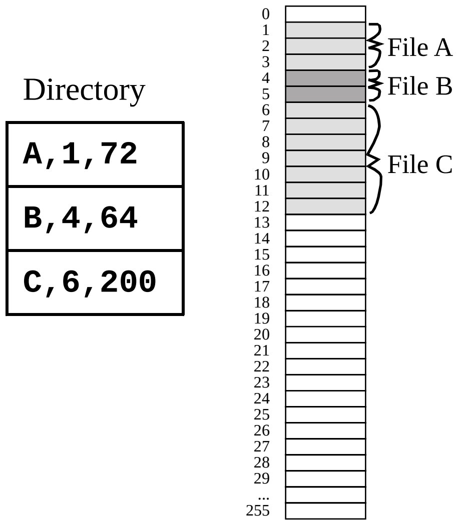


Figure 14.12. File system for Question 14.1.

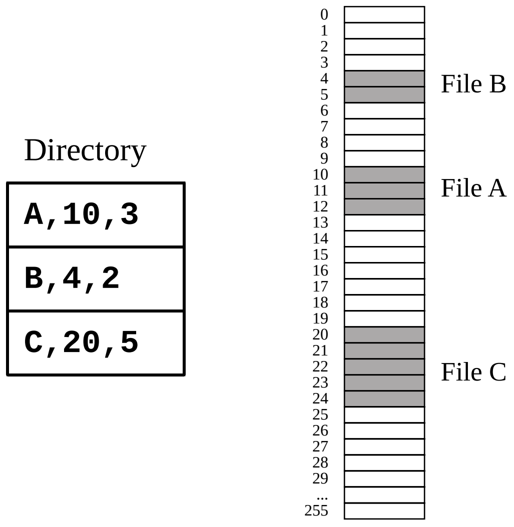


Figure 14.13. File system for Question 14.3.

Question 14.1. Consider a file system that uses **contiguous allocation** to define the set of blocks allocated to each file as shown in Figure 14.12. There are 8192 bytes on this disk, made up of 256 blocks, where each block is 32 bytes. This file system is used to record important "black box" information. Therefore, the file system is initialized to empty when the device is manufactured. Each time the system is turned on a new file is created. While running important data are stored into that file (open file, append data at the end, close file). Files are never deleted. Block 0 contains the directory and not available for data. Each directory entry has three fields: name, block number of the first block, and total number of bytes stored. The example in the figure shows file A with 3 allocated blocks (1,2,3 containing 32,32,8 bytes), file B with 2 blocks (4,5 containing 32,32 bytes) and file C with 7 blocks (6,7,8,9,10,11,12 containing 32,32,32,32,32,32,8). All 32 bytes of each data block can contain data for the file.

Part a) Does this file system have any external fragmentation? Justify your answer.
Part b) Assume a file has **n** data blocks. It takes one *block read* to fetch the **directory**. On **average** how many more *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of additional *block reads* that it takes to read a single byte in the file (worst case)?
Part c) Describe a simple mechanism to manage free blocks in this system. Be as explicit as possible, describing how many bytes in the directory are needed to manage the free space. Describe what the free space looks like after the disk is erased/formatted. Describe what the free space looks like when the disk is full.
Part d) File names are a single character. How many files can be stored? Justify your answer.
Part e) Assume you have **n** files each with of random size. Quantify the number of wasted bytes due to internal fragmentation. You may assume **n** is less than the number determined in part d).

Question 14.2. One way to manage free-space on a disk is to implement a **bit vector**. Each block is 32 bytes long, and there are 256 blocks .For each block on our 8-kibibyte disk, there will be a single bit, specifying whether the block is free (1) or allocated. In C, we can define 256 bits as a byte-array with 32 entries.

```
unsigned char BitVector[32];  // 256 bits
```

Similar to the directory, the BitVector will exist both in RAM, as the above C definition, and on the disk as block 1. The format operation will initialize 254 of these bits to 1, performing:

```
  BitVector[0] = 0x3F; // blocks 0,1 used (directory, BitVector)
  for(i=1;i<32;i++) BitVector[i]=0xFF; // blocks 8-255 are free
  eDisk_WriteBlock(BitVector,1);       // update disk copy
```

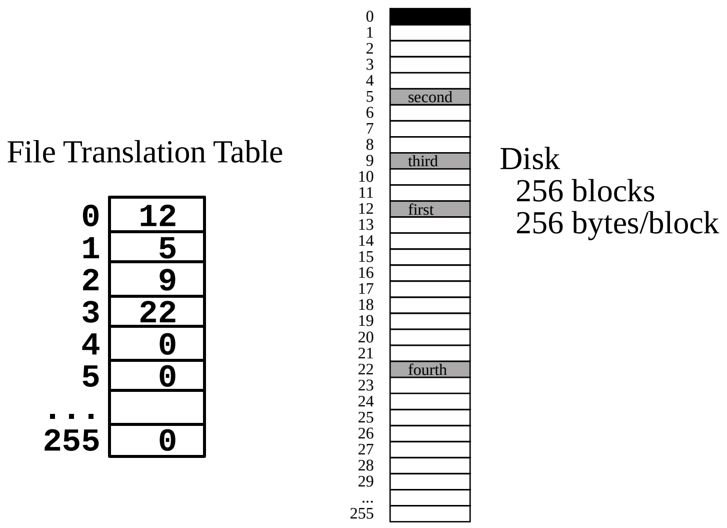
Part a) Write a helper function that allocates a free block updating the disk copy of BitVector.

```
// allocate a free block, returns a block number of a free block
// Output: block number 2 to 255 if successful and 0 if full
unsigned char AllocateBlock(void){
  eDisk_ReadBlock(BitVector,1);        // fresh RAM copy
```

Part b) Write a helper function that deallocates a block updating the disk copy of BitVector.

```
// deallocate a free block
// Input: block number 2 to 255
void DeallocateBlock(unsigned char blockNum){
  eDisk_ReadBlock(BitVector,1);        // fresh RAM copy
```

Question 14.3. Consider a file system that uses contiguous allocation as illustrated by Figure 14.13. The block size is 32 bytes and all 256 blocks can be used to store data. The directory is not stored on the disk. Each directory entry contains the file name (e.g., A, B, C), the start block (e.g., File B starts at block 4) and the number of blocks used in the file (e.g., File C has 5 blocks). The file sizes are always a multiple of 32 bytes. I.e., a file can contain only 32, 64, 96, … 8192 bytes. For example, File A is 3*32=96 bytes, File B is 2*32=62 bytes and File C is 5*32=160 bytes. Does this system have internal fragmentation? Explain your answer.
Question 14.4. Consider a file system that uses a **file translation table** (**FTT**) to define the set of blocks allocated to each file. There are 65536 bytes on this disk, made up of 256 blocks, where each block is 256 bytes. Block 0 contains the directory and not available for data. Each file has its own **FTT**, which is a null-terminated list of block numbers assigned to the file. The example in the figure shows a file with 4 allocated blocks, with the first block at 12, and the last block at 22. The directory entry includes the file name, the total number of bytes and the block number of its **FTT**. All 256 bytes of each data block can contain data for the file. For example, the figure shows a file with 1024 bytes of data, stored in 5 blocks (**FTT** and 4 data blocks).



Part a) Does this file system have any external fragmentation? Justify your answer.
Part b) Assume a file has **n** data blocks. It takes one *block read* to fetch the **FTT**. On **average** how many more *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of additional *block reads* that it takes to read a single byte in the file (worst case)?
Part c) Consider the linked allocation scheme described in section 3. Assume the directory is in memory and the file has **n** data blocks. On **average** how many *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of *block reads* that it takes to read a single byte in the file (worst case)?
Part d) Assume you are given the following function that reads a 256-byte block from disk

```
int eDisk_ReadBlock(unsigned char *pt,  // result returned by reference
  unsigned char blockNum);              // which block to read
```

Write a C function that returns a byte from a file at a random location. Do not worry about error handling (e.g., `eDisk_ReadBlock` error or address too big). The inputs to the function are `numFTT` (the block number of the file's **FTT**) and `address` (the byte address, where 0 is the first byte, 1 means second byte etc.). You can use two buffers.

```
unsigned char FTTbuf[256];  // place to store FTT
unsigned char Databuf[256]; // place to store data
```

The prototype of the C function you have to write is

```
unsigned char eFile_Read(unsigned char numFTT, unsigned short address);
```

Question 14.5. Consider a file system that manages a 16 Megabyte (2^{24} bytes) EEPROM storage for a battery-powered embedded system. You are free to select from a range of EEPROM chips with different block sizes. The block size can be any power of 2 from 1 to 2^{24} bytes. **Chip$_n$** has a total of 16 Megabytes with block size 2^n bytes. **Chip$_n$** can perform a 2^n byte block-write operation in 1 ms regardless of block size. For bandwidth reasons, therefore, you wish to choose a large block size. A block will be completed allocated to a file (you are not allowed to split one block between two files.) 16 bytes of each block are used by the file system to manage pointers, type, size, and free space. However, if the file were to contain 1 byte of data, an entire block would be allocated, and the remaining 2^n-17 bytes would be wasted. File sizes in this system are uniformly distributed from 50,000 to 150,000 bytes (this means any file size from 50,000 to 150,000 bytes is equally likely with an average size of 100,000 bytes). **You are asked to choose the largest block size with the constraint that the average internal fragmentation be less 5% of the total number of bytes stored.** Show your work.

Solutions to checkpoints

Solution 14.1: The largest contiguous part of the disk is 8 blocks. So the largest new file can have 8*512 bytes of data (4096 bytes). This is less than the available 16 free blocks.

Solution 14.2: First fit would put the file in block 1 (block 0 has the directory). Best fit would put the file in block 10, because it is the smallest free space that is big enough. Worst fit would put it in block 14.

Solution 14.3: 2 Gibibytes is 2^{31} bytes. 512 bytes is 2^9 bytes. 31-9 = 22, so it would take 22 bits to store the block number.

Solution 14.4: 2 Gibibytes is 2^{31} bytes. 32k bytes is 2^{15} bytes. 31-15 = 16, so it would take 16 bits to store the block number.

Solution 14.5: There are 16 free blocks, they can all be linked together to create one new file.

Solution 14.6: You could store a byte count in the directory or in each block.

Solution 14.7: 16+9=25. 2^{25} is 32 Mebibytes, which is the largest possible disk.

Solution 14.8: There are $2^{31}/2^{10}=2^{21}$ blocks, so the 22-bit block address will be stored as a 32-bit number. One can store 1024/4=256 index entries in one 512-byte block. So the maximum file size is 256*512 = $2^8*2^9 = 2^{17}$ = 128 kibibytes. You can increase the block size or store the index in multiple blocks.

Solution 14.9: There are 15 free blocks, they can create an index table using all the free blocks to create one new file.

Solution 14.10: Each directory entry now requires 10 bytes. You could have 50 files, leaving some space for the free space management.

Solution 14.11: There are 15 free blocks, they can create FAT using all the free blocks to create one new file.

http://www2.cs.uregina.ca/~hamilton/courses/330/notes/allocate/allocate.html