Operating Systems – spring 2022
Tutorial-Assignment 10

Instructor: Hans P. Reiser

Question 10.1: Disk Space Allocation

a. How does contiguous allocation work? What are the advantages and problems of this approach?

Solution:
With contiguous allocation, a (contiguous) array of blocks is allocated for each newly created file. A file cannot become larger than the pre-allocated space. The filesystem only needs to store the start block and the size of each file. Contiguous allocation allows for both sequential and direct access: If a `seek` *to position* l *shall be performed, the corresponding block number can be retrieved by adding* $l \div$ *blocksize to the starting block. However, contiguous allocation has some severe drawbacks: As files cannot grow beyond the initially reserved size, one has to carefully choose the number of blocks to allocate to each file: If the number is too small, the space for the file will probably not suffice, if it is too large, we will suffer from internal fragmentation. Compaction can be used to reduce external fragmentation, but is rather expensive.*

b. How does linked allocation work? What is its major problem?

Solution:
Linked allocation solves the major problem of contiguous allocation, as files no longer need to occupy a contiguous range of blocks. Similar to contiguous allocation, the file system stores a pointer to the first (and possibly also to the last) block of each file, but each block now contains a pointer to the next block that belongs to the same file. There's no need to pre-allocate disk blocks for files, blocks can be allocated on demand. External fragmentation is no longer a problem. However, linked allocation only allows for sequential access: To find the i*th block of a file, the entire list of blocks must be walked. If the link to the next block is stored within each block, then walking the list of blocks requires lots of I/O and disk seeks. If a pointer is corrupted for some reason, wrong blocks might be retrieved. Such errors are hard to detect and to correct.*

c. What is the basic idea of a File Allocation Table (FAT)?

Solution:
The FAT approach is a variant of the linked allocation approach, while it mitigates the main problem mentioned above: It stores the list information in a separate data structure (the FAT) (that might be cached in RAM). Still, the entire list has to be walked to find a specific byte offset in a file, but walking the list doesn't require disk accesses if the FAT is cached (even if the FAT is not cached, the amount of disk seeks is significantly reduced (why?)). Using a FAT still has some disadvantages: The size of the FAT does not depend on the number or size of files, but on the size of the hard disk (there is one entry for each block). For large disks, the FAT might occupy a large amount of disk space, and might also be too large to be completely cached in RAM.

d. How does indexed allocation work?

Solution:
Indexed allocation groups the pointers to disk blocks occupied by a specific file together into a single data structure, the index block. There is one index block for each file. Once an index block is loaded to main memory, it is easy to find a specific block of the file: The ith pointer in the index block specifies the location of the ith block of the file.

e. Using indexed allocation, the maximum size of a file depends on the size of the index block. Discuss various approaches that allow for very large files without increasing the size of an index block.

Solution:
The basic idea is to use more than one index block for large files. Different approaches are possible how these blocks are organized: One approach is to construct a chain of index blocks for large files: The first index block contains pointers to n disk blocks, its last entry points to another index block, which again points to n disk blocks and might point to another index block if the number of referenced disk blocks does not suffice. Another approach is similar to multi-level page tables: A first-level index block does not point to disk blocks, but to a number of second-level index blocks, which can point to a third level of blocks, and so on. Entries of blocks of the last level will then point to the actual disk blocks occupied by the file. These two approaches can also be combined: Some entries of a file's index block can point directly to the file's disk blocks (direct blocks), one entry can point to a block of pointers to disk blocks (single indirect block), another entry can point to a block of pointers to blocks with pointers (double indirect block) . . .

f. Consider a filesystem that uses inodes to represent files. Assume that disk blocks are 8 KiB in size and a pointer to a disk block is 4 bytes long. An inode contains 12 pointers to direct blocks, and one pointer to a single, double, and triple indirect block, respectively. What is the maximum size a file can have?

Solution:
*With 4 byte pointers, we can store $8KiB/4bytes = 2Ki = 2048$ pointers in each block. The single indirect block allows to address 2048 blocks, the double indirect block allows to address 2048 * 2048 blocks, and the triple indirect block allows to address 2048 * 2028 * 2048 blocks. In addition, 12 blocks can be accessed directly. The maximum file size can thus be calculated as:*

$$(12 + 2048 + 2048^2 + 2048^3) * 8\textit{KiB} \approx 64\textit{TiB}$$

Question 10.2: I/O Techniques

Which actions does the CPU need to perform in device-to-memory data transfers in each of the three primary I/O models?

a. Programmed I/O (polling)

Solution:
The CPU coordinates the entire data transfer. It issues an I/O command to the device, polls for a response, fetches the data directly from the device registers, and stores it in main memory.

b. Interrupt-driven I/O

Solution:
The CPU initiates the I/O transaction, and then switches to another task while waiting for an interrupt from the device. Upon receiving the device interrupt, the CPU fetches the data directly from the device registers and stores it in main memory.

c. DMA

Solution:
The CPU initiates the I/O transaction, and provides a location in physical memory for the results. The CPU performs other work while waiting for a DMA interrupt, which indicates that the whole I/O transaction has completed. At that time, the CPU has access to the data in main memory without ever having communicated with the actual device.

d. Operating systems group device interfaces into different categories to support a modular structure of the kernel code. An operating system offers a (small) number of different interfaces and abstractions that each device driver can use.

What are the three basic categories typically found in an operating systems, and what are the differences?

Solution:
The main categories are character, block, and network devices.

Traditionally, character device were primarily intended for serial, character-oriented devices such as keyboard, printer, mouse, modems, in many cases slow devices, and devices that do not support seeking. On the other hand, block devices were intended for devices such as hard disks that internally operate on blocks of data and usuallly also support arbitrary seeking.

In Linux, the main difference between character devices and block devices is that block devices make use the kernel's buffer cache, whereas the interaction with character devices is direct.

A block device in Linux has to support random access (note that this is not necessarily always the case: on Linux, a tape device (that does not support random access) is a character device, but it could be a block device on other operating systems). The main part of the device API consists of request queues that need to be handled by the device driver. Block devices are mainly used for disks.

Many devices in Linux (basically everything except disks and network devices) are character devices. The main part of the API of a character device driver consists of a read and a write operation. It is possible, but not mandatory, that a character device supports random access by implementing a seek method.

Details also depend on the operating systems. For example, recent versions of OpenBSD

don't have block devices any more. MacOS provides a uncached character device (/dev/rdisk) and a cached block device (/dev/disk) for each disk.

Network devices require a quite different API and thus are using a separate abstraction. They are neither character nor block devices.

Question 10.3: Memory mapped files

a. Assume you have a file of size 1 GiB. The following program counts the number of 1 KiB blocks in a file that start with the character “h”.

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

// 1 GiB
#define SIZE (1024*1024*1024*1L)

int buffer[SIZE];

int main() {
    int fd = open("test.bin", O_RDONLY);
    if(fd<0) { perror("open failed: "); return 1; }

    int n = read(fd, buffer, SIZE);
    if(n<SIZE) { printf("Could not read %d bytes\n", SIZE); return 1; }

    int cnt=0;
    for(size_t i=0; i<SIZE; i+=1024) {
        if(buffer[i] == 'h') cnt++;
    }
    printf("Counted 'h' characters in file: %d\n", cnt);
    close(fd);
    return 0;
}
```

How can you re-write the program such that it uses the `mmap` system call instead of the `read` system call? What are the differences between the two approaches?

Solution:

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

// 1 GiB
#define SIZE (1024*1024*1024*1L)

int main() {
    int fd = open("test.bin", O_RDONLY);
    if(fd<0) { perror("open failed: "); return 1; }

    char *ptr = mmap ( NULL, SIZE, PROT_READ, MAP_SHARED, fd, 0 );
    if(ptr == MAP_FAILED) { perror("mmap failed"); return 1; }
```

```
    int cnt=0;
    for(size_t i=0; i<SIZE; i+=1024) {
        if(ptr[i] == 'h') cnt++;
    }
    printf("Counted 'h' characters in file: %d\n", cnt);
    munmap(ptr, SIZE);
    close(fd);
    return 0;
}
```

The `read` version requires a buffer in user space, and the `read` system call copies the file content from kernel memory (buffer cache, to which the file is read) to user space. When the `read` system call returns, the full file content has been read from disk.

The `mmap` version instead maps the file into the address space of the application. This means that no buffer in user space is needed. Also, the `mmap` system call does not read anything from disk. Instead, it sets up page table entries that tell the system to read the data from file when the virtual address is accessed (in the page fault handler). The data is directly mapped into the user address space, there is no need to copy data from kernel memory to application memory.

A possible disadvantage of the `mmap` approach is that you might have multiple page fauls (up to one for each 4 KiB page). In practice, the operating system will usually use some read-ahead strategy (like starting the disk I/O already when `mmap` is invoked, or fetching more pages when memory towards the end of the last prefetch is accessed).

b. Try to experimentally verify the performance difference between both approaches!

Solution:
Example run on skel:

```
$ time ./file-read
Counted 'h' characters in file: 2
./file-read  0.00s user 0.85s system 99% cpu 0.854 total

$ time ./file-mmap
Counted 'h' characters in file: 2
./file-mmap  0.03s user 0.06s system 98% cpu 0.092 total
```

In this specific example, it can be observed that the `mmap` version is significantly faster. The main reason for this is most likely the time it takes to copy the file content from kernel space to user space.