

Chapter 8: Operating Systems I (File System)

References:

- Brause: Betriebssysteme, 2. Aufl. (in German). Springer, 2001.
- Bach: The Design of the UNIX Operating System. Prentice Hall, 1986.
- Leffler/McKusick/Karels/Quarterman: The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley, 1989.
- Tanenbaum: Modern Operating Systems. Prentice Hall/Pearson, 2001.
- Harbison / Steele Jr.: C — A Reference Manual, 4th Ed. Prentice-Hall, 1995.
- David A. Curry: Using C on the UNIX System. O'Reilly, 1989.
- MSDN Library: Visual Studio 6.0 release.
- SUN Solaris Online Documentation.
- Transtec Catalogue, Yellow Guide, Chap. 4: Mass Storage (<http://www.transtec.co.uk>)
German Version in <http://www.transtec.de>.
- Seagate: <http://www.seagate.com/products/discsales/>
- IBM: <http://www.storage.ibm.com/>
- The PC Guide: Hard Disk Performance: <http://www.pcguides.com/ref/hdd/perf/index.htm>
- Storage Review: <http://www.storagereview.com>, <http://198.76.30.88/jive/sr/>
- Patterson / Keeton: Hardware Technology Trends and Database Opportunities. SIGMOD'98. <http://www.cs.berkeley.edu/~pattarn/talks/sigmod98-keynote-color.pdf>

Overview

1. Motivation for Operating Systems

2. Disk Hardware

3. File System Calls

4. Implementation

Introduction (1)

- It is difficult to work directly with the I/O hardware.
- E.g. in order to read a block from the disk, basically the following must be done:
 - ◇ The disk address of the requested block (head, track/cylinder, sector) is stored in device registers of the disk controller,

The bare disk understands of course no file names. The controller is a small computer that manages the real hardware. Depending on the architecture of the main computer, controller interface registers can be accessed like standard memory locations at special addresses (“memory-mapped I/O”). Some CPUs (like the Pentium) have a second address space for I/O and special I/O instructions.

Introduction (2)

- Reading a block from the disk, continued:
 - ◇ Another device register of the disk or DMA controller must be set to the memory address of a buffer (where the data should be stored).

DMA means “direct memory access”: The controller can write its data to main memory without help from the CPU.

- ◇ Then the “read” code is stored in the command register of the disk controller.

This is the signal to start its work (actually, many more parameters are first set besides the disk and the memory address).

Introduction (3)

- Reading a block from the disk, continued:
 - ◇ then the CPU can do something different — the disk controller causes an interrupt when it is done.

An interrupt is like a procedure call that is triggered by the computer hardware, and not a “CALL” instruction.
 - ◇ When the interrupt arrives, the status register of the disk controller must be read to check whether the operation was successful.
- Of course, the details depend on the specific disk controller, and already a floppy disk is different.

Introduction (4)

- It makes no sense if every program that needs to access data on the disk contains program code that works with the disk controller on this level.
 - ◇ It is not necessary that every programmer “invents the wheel again”.
 - ◇ Such programs depend very much on the hardware (run only with one specific disk controller).
 - ◇ Programs might destroy data if they have incompatible policies for finding free disk space.

Introduction (5)

- Therefore, computers are equipped with operating systems. Their main tasks are:
 - ◇ Offering a user-friendly interface to the hardware that makes programs device-independent.
 - ◇ Managing access to the hardware for concurrent processes/users, simulating more hardware than the the computer really has.
 - ◇ Loading programs into memory for execution.

Overview

1. Motivation for Operating Systems

2. Disk Hardware

3. File System Calls

4. Implementation

Disks (1)

- A disk consists of a stack of circular plates (“platters”) each coated on one or both sides with magnetic recording material.

ST318405LW (Seagate Cheetah 36XL, 18.4 GB, \$260):
2 platters of 3.5inch diameter, coated on both sides.

- The platters are mounted to a rotating spindle.

ST318405: platters rotate with 10000 rpm(revolutions/min).

- There is one read-write head for each magnetic surface.

E.g. the ST318405 has 4 heads.

Disks (2)

- The heads are mounted to an arm-assembly which can move them in and out.

The arm-assembly looks like a comb. Only all heads together can be moved. Only one head can read or write at the same time.

- The heads fly on an air cushion.

E.g. 70 nm above the surface (the number is relatively old). When the disk is switched off, the heads are moved to a safe parking position.

- The data is written on each surface in the form of concentric circles called tracks.

ST318405: 19036 tracks per surface, 76144 in total.
The track density is 24406 tracks/inch (TPI).

Disks (3)

- The tracks with the same distance from the center on all surfaces together are called a cylinder.
- Each track is divided into sectors (small arcs of the circle). A sector is the smallest unit of information that can be read from or written to the disk. It often consists of 512 Bytes.

Modern disks have more sectors on the outer tracks (“multiple zone recording”), since the outer tracks are longer. It is quite typical that the outermost tracks have double the number of sectors as the innermost ones. The ST318405 has on average 471 sectors/track.

Disks (4)

- Blocks can be defined as collection of consecutive sectors.

The sector size is often 512 Byte, but it is more economical for the operating system to read/write larger units. The block size is often 8 KB under UNIX, and e.g. 2 KB under DOS (“cluster size”). Block size is usually determined during formatting as a multiple of the fixed sector size.

- Blocks are the smallest unit that the operating system can read or write from/to the disk.

Disks (5)

- Modern disk drives have a disk controller built-in.
- It translates relatively high-level commands (such as read the sector with address defined by cylinder, surface, and sector number) into the commands for the real hardware.
- The disk controller attaches address information (track and sector number) and checksums to the sectors.

By reading the data, the controller can check that it is on the right track and has found the correct sector (embedded servo technology).

Disks (6)

- It is not economically feasible to produce 100% defect-free media. Disks have a limited number of replacement sectors.

During initialization of the disk, each sector is tested and bad sectors are found. The controller manages a defect map which uses one of the spare sectors in place of a bad sector.

- Disks have a cache (RAM) for recently read sectors.

If the sector is still in the cache, it can be send immediately to the computer without the mechanical delay needed to read a sector from the disk. Usually, the controller also reads following sectors into the cache (read ahead) since these are immediately available and often required next. The ST318405 has 4 MB Buffer Cache (multisegmented).

Disk Capacity

- Disk manufactureres define 1 MByte as $1000 * 1000$ Byte, otherwise 1 MByte is $1024 * 1024$ Byte.
- The operating system needs part of the disk space for control information (not available for user data).
- So the disk will appear as smaller than advertised.
- The needed disk space seems to grow constantly.

A rule of thumb is that the needed disk space doubles every year. Since disk space becomes cheaper over time, it is also not a good idea to buy disk space for many years in advance. One author recommends to buy the space needed in the next two years.

Reading a Sector (1)

- The operating system sends the read request over the disk interface to the disk controller.
- The arm is moved to the right cylinder (**seek time**).

The average seek time is the time needed to move the arm one third of the maximal distance.

For the ST318405, it is 5.4ms (read) / 6.2ms (write).

The disk needs 0.8ms (read)/1.2ms (write) to position the head on the next track and 10.5/11ms for the full distance.

- Then the disk waits until the needed sector shows up under the read-write head (**latency time**).

In average, half a turn is needed. So 10000rpm give an average latency of $60s / (10000 * 2) = 3ms$.

Reading a Sector (2)

- Then it reads the data.

Modern disks can read an entire track in one revolution (interleave factor 1:1). For the example disk, the data can be read with $(18352\text{MB}/76144) * 167(\text{turns/s}) = 40\text{MB/s}$. The speed is higher on the outer tracks and lower on the inner tracks. Seagate specifies: Internal Transfer Rate 320–490 MBits/s, Internal Formatted Transfer Rate 31–50 MByte/s, Average Formatted Transfer Rate 43 MByte/s.

- Then the data is transferred to the computer's main memory.

The disk has an Ultra160 SCSI (pronounced “scuzzy”) interface which allows to transfer 160 MB/s.

Other interfaces: IDE/ATA: 2–4MB/s, Ultra ATA: 33MB/s, Ultra ATA/66: 66MB/s. Fibre Channel: up to 400 MB/s.

Disk Performance (1)

- Consecutive blocks can be accessed especially fast.

Consecutive means first the following sectors on the same track, then another track (surface) in the same cylinder, and then an adjacent cylinder. Normally the sectors are ordered such that when we move to an neighbouring cylinder, no or only a minimal latency time is needed.

- Reading 10 MB which are stored in one piece takes e.g. 0.3s.

In the ST318405, a track contains on average 241128 Bytes, so 44 tracks must be read. This requires one random seek (5.4 ms), latency time (3.0 ms), 10 seeks to the next cylinder ($10 * 0.8 \text{ ms}$), and 11 * 3 head switches (to adjust the head position on another surface, $33 * 0.8 = 26.4 \text{ ms}$), and 44 revolutions for reading ($44 * 6 = 264 \text{ ms}$), in total 307 ms.

Disk Performance (2)

- Textbooks say that 10MB/s can be effectively read from a disk (if the data is stored in consecutive blocks).

The above computation gives 30MB/s.

- Reading the same amount of data from randomly scattered blocks needs about 100 times as much time.

Assume that we need to read 5000 blocks of 2KByte.

The time needed is $5000 * (5.4 + 3.0 + 0.05) \text{ ms} = 44.5 \text{ s}$.

Disk Performance (3)

- Some authors say that if a disk has constantly more than 50 independent accesses per second, it becomes a bottleneck.

This leaves 20ms for every access. The disk is faster, but when the requests are randomly distributed, and you keep the disk operating near its limits, a queue will build up.

- Multiple disks can at least do the seek in parallel. So if the data cannot be stored together, it might be an option to store it on different disks.

Depending on the maximal transfer rate of the interface, also the transfer can be done interleaved. Larger computers have several disk interfaces which can work in parallel.

Disk Performance (4)

- Expected future performance improvements:
 - ◇ Disk Capacity: 27–60%/year
I.e. the disk capacity doubles every 1.5–2 years.
 - ◇ Transfer Rate: 22–40%/year
I.e. the transfer rate doubles every 2–3.5 years.
 - ◇ Rotation/Seek Time: 8%/year
I.e. the rotation and seek time halves every 7–10 years.
 - ◇ \$/MB: > 60%/year
I.e. the price halves in less than 1.5 years.

Disk Performance (5)

- This shows that the difference between random and sequential accesses will become greater and greater.
- In 1970, this factor was about 30, in 2000 it is about 140.

Data of the IBM 3330 (1970): Capacity: 93.7 MB, average seek: 30ms, next track: 10ms, max. seek: 55ms, 3600 rpm, 13 KB/track, 19 heads, 411 cylinders, 806 KB/s transfer rate.

Overview

1. Motivation for Operating Systems

2. Disk Hardware

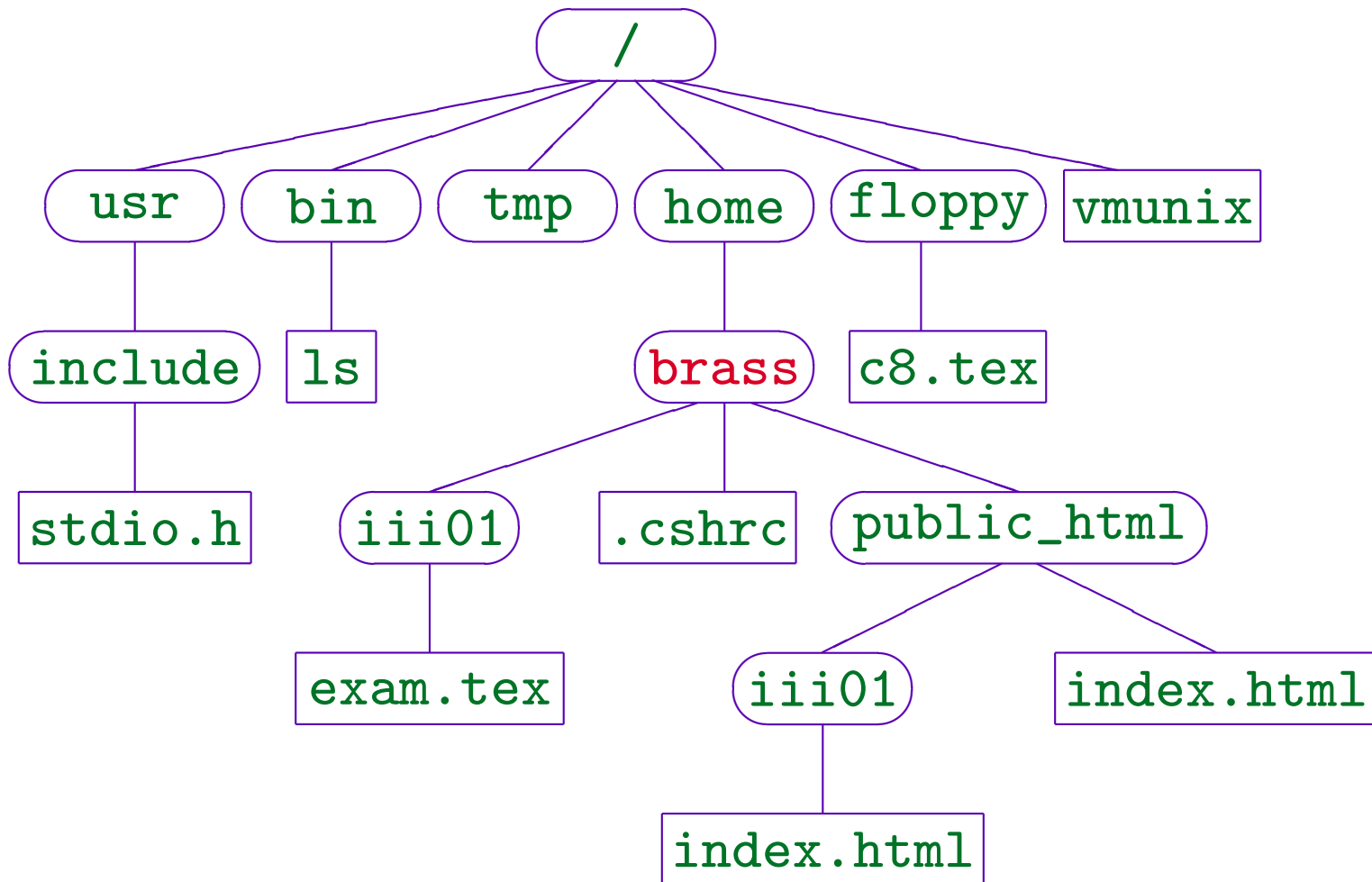
3. File System Calls

4. Implementation

File System Fundamentals (1)

- A file is a sequence of characters/bytes.
- In text files, lines are ended with LF (ASCII 10) under UNIX and the sequence CR, LF (ASCII 13, 10) under DOS/Windows. But this is only convention.
- Files are identified within a directory by a name (file name).
- Directories can be nested within other directories (“subdirectories”). In this way, a tree structure is constructed.

File System Fundamentals (2)



File System Fundamentals (3)

- Under UNIX, the system has a unique root directory. I.e. there is only one directory tree that contains all files in the system.

The directory tree on a disk (or CD-ROM, floppy disk etc.) can be “mounted” over a directory in the existing file system. It thereby becomes a subtree of the global directory tree.

- Under DOS/Windows, the system has several devices (e.g. “A:”, “C:”), each of which has a root directory. I.e. each device has its own directory tree.

Windows NT/2000 has a single namespace for all its objects. E.g. “A:” is only an alias for “\Device\Floppy0”. However, the internal names cannot be used via the Win32 API.

File System Fundamentals (4)

- Files and directories in the tree can be identified with an **absolute path name** that starts with the root directory and lists all intermediate directories on the path “down” in the tree until the file/directory one wants to identify, e.g.

```
/home/brass/iii01/exam.tex
```

- Under UNIX, “/” is used to separate elements on the path, under MS-DOS/Windows, it is “\”, e.g.

```
C:\Stefan\iii01\exam.tex
```

File System Fundamentals (5)

- Every process (program in execution) has a current working directory under UNIX.
- One can also use **relative path names** (which do not begin with “/”). They are interpreted with respect to the current working directory.
- If the current working directory is e.g. `/home/brass`, the relative path name

`iii01/exam.tex`

means `/home/brass/iii01/exam.tex`.

File System Fundamentals (6)

- A special case of relative path names is a simple file name. If e.g. the current working directory is `/home/brass/iii01`, the file can be identified as `exam.tex`
- The notation “`..`” is used for the next higher directory in the file system hierarchy.
- E.g.: If the current directory is `/home/brass/iii01`,
`../public_html/index.html`
refers to `/home/brass/public_html/index.html`.

File System Fundamentals (7)

- MS-DOS/Windows manages one current working directory for each device/drive.
- Since there is also a current device, relative path names work the same as under UNIX.
- However, if one changes to another drive, one does not necessarily get into to root directory of that device, but into the current directory.
- Programs are often started via shortcuts. By right-clicking on them, and selecting “Properties”, one can change the “Start in” working directory.

File System Fundamentals (8)

- Under UNIX uppercase and lowercase in filenames is important, under MS-DOS/Windows filenames are case-insensitive.
- File names often have an “**extension**” (a few letters after the “.”) that indicates the type of the file.
- However, it is only convention and not enforced by the system that e.g. a file with extension “.**c**” contains “C” code.

Sometimes extensions are ambiguous, e.g. “.**pl**” is used for Perl code and for Prolog code. Sometimes also the first few bytes of the file indicate the type.

Access Rights (1)

- In a multi-user system like UNIX, a single disk contains files of different users.
- Normally, users should not be able to change files created by other users (no **write access**).
- Users should also not be able to create and delete files in all directories, but basically only in their **“home directory”** (e.g. **/home/brass**).

In order to create a file in a directory (or to delete a file in a directory) one needs write access to the directory.

Access Rights (2)

- Some files contain confidential information. E.g. students should not have **read access** to **exam.tex**.
- For programs, also the **execute access** is important.

If a user has execute, but no read access to a program, he/she cannot copy it. Directories can also be blocked. Under UNIX, in order to pass through a directory, one needs “execute access” for the directory.

- Under UNIX, users belong to one or more groups. Each file has **rwX** rights not only for the file owner and for all others, but also for the members of one group (usually the primary group of the file owner).

Access Rights (3)

- In summary, for the UNIX security model, each file contains the following information:
 - ◇ Who owns the file?
 - ◇ Which group has rights in the file?
 - ◇ Which read/write/execute rights has the owner?
 - ◇ Which read/write/execute rights has the group?
 - ◇ Which r/w/x rights have all other users?
- The access rights are encoded in 9 bits, e.g. **0751** (**rwxr-x--x**) means r/w/x for the owner, r/x for the group, and only execute access for all others.

Access Rights (4)

- Under UNIX, there is a **super-user** (administrator, **root**), who can do everything.
- Programs normally have the rights of the user who executes them.
- However, there are special programs (“setuid programs”) which are executed with the rights of the owner of the program (often **root**).

Although a user may not have direct write access for a file, he/she might be able to modify the file in controlled ways by using a program. E.g. a game might manage a highscore list.

Access Rights (5)

- Besides the 9 `rwX` bits, there are three more bits:
 - ◇ `04000`: `setuid`

The program is executed as if the caller would belong to the group to which the program belongs.
 - ◇ `02000`: `set group ID`

The program is executed as if the caller would belong to the group to which the program belongs.
 - ◇ `01000`: `sticky bit`

The program remains in memory/on the swap device after it is executed — in case it is soon called again. This is seldom used in modern systems. For directories, the sticky bit means that users cannot delete or rename files created by other users in this directory even if they have write permission for the directory. This is used for shared directories like `/tmp`. This feature was not contained in older UNIX systems.

Access Rights (6)

- Windows 95/98/ME is a single-user system.

It is possible to declare files as read-only or system files, but since whoever works with the computer owns all files, he/she can change the file attributes.

- Windows NT/2000 is a multi-user system.
- It uses “access control lists” for files, in which one can specify the rights of every user.

This supports more detailed access right specifications than UNIX.

- Windows NT/2000 also supports auditing.

I.e. one can request that the system records who has one what to which file.

UNIX System Calls (1)

- In order to read or write a file, one must first open it. This loads information about the file into memory and returns a “file descriptor” (small non-negative number, index into file table):

```
fd = open(path, flags, rights);
```

- `const char *path` is the name of the file to be opened. It can include a directory prefix.
- `int flags` determines the mode in which the file is opened, e.g. `O_RDONLY` means that the file is opened for read-only access.

UNIX System Calls (2)

- Constants like `O_RDONLY` are defined in `sys/file.h` (on Berkeley systems) or `sys/fcntl.h` (System V).

Flags can be combined with bit-or, e.g. `O_WRONLY|O_CREAT|O_TRUNC` is the usual way to open a file for writing: If it exists, it is truncated (reduced to size 0), if not, it is created. `O_WRONLY|O_CREAT|O_EXCL` means that the file must not yet exist, or the open fails. There are also `O_RDWR` (open for reading and writing), `O_APPEND`, and possibly other flags.

- The third argument specifies the file permissions if the file is created.

See “Access Rights” above. E.g. `0644` means that the owner of the file can read and write it, while the group and all others can only read it (`rw-r--r--`). Modern UNIX systems have an “umask” that the user can set to make file permissions more restrictive.

UNIX System Calls (3)

- If the file cannot be opened, `open` returns `-1`.
- It also sets the global variable “`extern int errno;`” to a number that explains the reason of the error.
- The variable and error numbers are declared in `errno.h`.
- One can also use the library routine `perror` to print a detailed error message.

`perror(const char *msg)` writes “`msg: sys-err-msg`” to `stderr`.

Some systems have an array `sys_errlist` of strings with error messages that can be indexed with `errno` (`sys_nerr` is the array size).

One can use also `char *strerror(int errno)` to access error messages.

UNIX System Calls (4)

- Normally, the following three file descriptors are already open when the program starts:
 - ◇ 0: Standard input

This is usually the keyboard, but it can also be redirected to take input from a file.
 - ◇ 1: Standard output

This is usually the screen, but it can also be redirected to a file.
 - ◇ 2: Standard error output

This is usually the screen. When the normal output is redirected to a file, the user should still see any error messages, therefore there is a second output channel. It is possible that this is also redirected to a file, but then the user probably knows that there are errors.

UNIX System Calls (5)

- In old versions of UNIX, `open` had only two arguments and required that the file already exists.
- One had to use a different system call for creating a file (which still exists in modern UNIX versions):

```
fd = creat(path, rights);
```

`const char *path` is the name of the new file, `int rights` are the access rights.

- The function returns a file descriptor `fd` for the new file (open for writing), or `-1` in case of an error.

It is no error if the file already exists: It is truncated in this case.

Standard Library (1)

- Normally, one does not directly use the system calls.
- Instead of calling `open`, one uses the standard library function

```
fp = fopen(path, mode);
```

- It returns a file pointer (`FILE *fp`) instead of a file descriptor.
- However, internally `fopen` calls `open` and stores the file descriptor in the `FILE` data structure.

One can extract it with `fd = fileno(fp)`. Under MS VC++, the names of non-standard functions are prefixed with “_” (e.g. `_fileno`).

Standard Library (2)

- If `fopen` returns a null pointer indicating an error, one can also use `errno`, `perror`, `strerror` to get a detailed error message.
- The `mode` parameter is a string and can be e.g. `"w"` (open for writing) or `"r"` (open for reading).
 - "a" opens the file for appending data, `"r+"` for reading and writing (file must exist), `"w+"` for reading and writing (file is overwritten). On non-UNIX systems, one can append `"b"` (binary) to the mode-string in order to suppress translation of line ends and `Ctrl+Z` (EOF).
- The `FILE *` variables `stdin`, `stdout`, and `stderr` are declared in `stdio.h`.

UNIX System Calls (6)

- One can read bytes from a file into a buffer with the system call:

```
n = read(fd, buf, bufsize);
```

The parameters have the following meaning:

- ◇ `int fd`: file descriptor of a file open for reading,
- ◇ `char *buf`: address of a character array,
- ◇ `int bufsize`: size of the array (the number of characters that `read` should read).
- ◇ `int n`: The return value is the number of characters actually read (0 at end of file, -1 if error).

UNIX System Calls (7)

- `write` has the same parameters as `read` (but data is transferred from the buffer to the file):

```
n = write(fd, buf, bufsize);
```

The parameters have the following meaning:

- ◇ `int fd`: file descriptor of a file open for writing,
- ◇ `char *buf`: address of a character array that contains the bytes to be written,
- ◇ `int bufsize`: number of bytes to be written,
- ◇ `int n`: The return value is the number of characters actually written (should be equal to `bufsize`).

Standard Library (3)

- E.g. the following standard library output functions internally call `write`:

- ◇ `int fputc(int c, FILE *fp)`

This function writes character `c` to file `fp`. `int putc(int c, FILE *fp)` also writes `c` to `fp`, but is usually implemented as a macro and a bit more efficient. `int putchar(int c)` is equivalent to `putc(c, stdout)`. The functions return the character written or `EOF` in case of an error.

- ◇ `int fputs(const char *s, FILE *fp)`

This function writes the characters of the null-terminated string `s` to the file `fp`. It returns `EOF` if an error happens. The function `int puts(const char *s)` writes `s` to `stdout`, but in contrast to `fputs`, it appends a newline at the end.

Standard Library (4)

- Standard library output functions, continued:

- ◇ `int fprintf(FILE *fp, const char *format, ...)`

This function writes the arguments “...” to the file `fp` where the string `format` specifies number, type, and output format of the arguments. Normal characters in `format` are written unchanged to the output, `fprintf` only interprets “format elements” that begin with a percent sign. E.g. “%d” prints an `int` argument in decimal form, “%ld” is for `long int`. Unsigned values can be printed in decimal (`%u`), in octal (`%o`), or in hexadecimal (`%x` and `%X`). Floating point (`double`) values can be printed with `%f`, `%e`, `%E`, `%g`, `%G`. Strings are printed with `%s` and characters with `%c`. It is possible to specify a minimum output field width, e.g. `%4d` will print at least four characters. For floating point types, one can also specify the number of digits after the decimal point, e.g. `%8.4f`. The function returns `EOF` if an error happens. `printf(format, ...)` writes to `stdout`.

Standard Library (5)

- Standard library output functions, continued:

- ◇ `size_t fwrite(void *buf, size_t s, size_t n, FILE *fp)`

This function writes `n` elements from the array `buf` to the file `fp`. Each element is `s` bytes large, one normally uses `sizeof(T)` for that argument. The number of elements written is returned.

- The standard library functions buffer the output, e.g. `putc` normally stores the character only in an array, and does not immediately call `write`.

For standard output, the output characters are usually written when a newline character `'\n'` is printed. One can also call `fflush(FILE *fp)` to make sure that the buffer contents is actually written.

Standard Library (6)

- Standard library input functions:

- ◇ `int fgetc(FILE *fp)`

This function returns the next character of the file `fp` or `EOF` at the end of file (or when an error happened, one can use `feof(FILE *fp)` and `ferror(FILE *fp)` to distinguished between the two conditions). `getc` is the same as `fgetc`, but it is usually implemented as a macro. `int getchar(void)` is equivalent to `getc(stdin)`.

- ◇ `int ungetc(int c, FILE *fp)`

This function pushes the character `c` back into the input stream, i.e. it will be returned with the next call to `getc` and similar functions. It is only guaranteed that a single character can be buffered in this way, i.e. before the call to `ungetc`, one must have read a character.

Standard Library (7)

- Standard library input functions, continued:

- ◇ `char *fgets(char *buf, int n, FILE *fp)`

This function reads characters from `fp` until the end of the line (or EOF) and stores them in the array `buf` (including the newline character and a terminating null character). The function respects the buffer size `n`, i.e. it will read at most `n-1` characters and use the last array position for the null character. The function returns `buf` in the positive case and `NULL` if an error happened. There is also a function `char *gets(char *buf)`, which reads from `stdin`, but it is dangerous because too long input lines can cause a buffer overflow (in contrast to `fgets`, it does not store the final newline in the buffer).

Standard Library (8)

- Standard library input functions, continued:

- ◇ `int fscanf(FILE *fp, const char *format, ...)`

This function reads input characters and converts them e.g. to numbers as specified in the format string. The format elements are as explained under `fprintf` above, they must match in number and type the pointer arguments in the “...” part. E.g. `%d` converts a sequence of decimal digits to an `int` value. White space in the input in front of the digits is skipped, the conversion stops at the first non-digit character after the digits. This character will remain in the input. The format element `%s` reads a string, it skips white space in front of the string, and ends when a white space character (or EOF) is read. The format element `%c` stores the next input character in the corresponding variable, it does not skip white space. The function returns the number of successfully processed format elements. Invalid characters remain in the input.

Standard Library (9)

- Standard library input functions, continued:

- ◇ `int scanf(const char *format, ...)`

This is the same as `fscanf`, but it reads from `stdin`. The function `sscanf` reads from a character array in memory.

- ◇ `size_t fread(void *buf, size_t s, size_t n, FILE *fp)`

This function reads up to `n` elements from the file `fp` into the array `buf`. Each element is `s` bytes large, one normally uses `sizeof(T)` for that argument. The number of elements actually read is returned.

UNIX System Calls (8)

- Open files have a current read/write position where `read/write` start. After each call to `read/write`, it is incremented by the number of bytes read or written.
- One can set the position (i.e. jump in the file):

```
position = lseek(fd, offset, origin);
```

The parameters have the following meaning:

- ◇ `int fd`: file descriptor of an open file,
- ◇ `long offset`: number of bytes to move,

E.g. in Solaris, the type of this parameter is `off_t` defined in `sys/types.h`. This will be a 64-bit number in future in order to overcome the 4GB limit for the file size.

UNIX System Calls (9)

- Parameters of `lseek`, continued:

- ◇ `int origin`: `SEEK_SET` (offset is relative to the beginning of the file), `SEEK_CUR` (relative to current position), `SEEK_END` (relative to file end).

The constants are defined as 0, 1, 2 in `sys/file.h` (Berkeley) or `sys/fcntl.h` (System V) or `unistd.h` (Solaris).

- ◇ `long position`: The function returns the new position (relative to the beginning of the file) or `-1L` in case of an error.

The next `read/write` starts from this position. Under Solaris, the return type is `off_t`.

Standard Library (10)

- The function `fseek` can be used to set the current read/write position for a file (it calls `lseek`):

```
err = fseek(fp, offset, origin);
```

The parameters have the following meaning:

- ◇ `FILE *fp`: file pointer of an open file,
- ◇ `long offset`: number of bytes to move,
- ◇ `int origin`: `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`,

See `lseek`, e.g. `SEEK_SET` means absolute file position.

- ◇ `int err`: The function returns 0 if successful.

Standard Library (11)

- The function `ftell` can be used to get the current read/write position for a file:

```
pos = ftell(fp);
```

The parameters have the following meaning:

- ◇ `FILE *fp`: file pointer of an open file,
- ◇ `long pos`: The current file position (or `-1L` in case of an error).

For files that are read/written in text mode (with translation of line ends), the file position might not be what one expects. However, it is guaranteed that if one uses the result of an `ftell` in a later `fseek`, one returns to the same position.

Standard Library (12)

- As a special case of `fseek`, the read/write position can be moved back to the beginning of the file with:

```
rewind(fp);
```

It is equivalent to `(void) fseek(fp, 0L, SEEK_SET)`.

- The following function closes a file and uses the same file pointer for opening a new file:

```
fp = freopen(path, mode, fp);
```

This is basically equivalent to `fclose(fp)` followed by `fp = fopen(path, mode)`, but it results in the same file pointer (or `NULL` if the open fails).

UNIX System Calls (10)

- If an open file will be no longer accessed in this program, it should be closed:

```
err = close(fd);
```

The parameters have the following meaning:

- ◇ `int fd`: File descriptor to be closed.
 - ◇ `int err`: 0 means successful, -1 means error.
- When the program terminates, all open files are automatically closed.

However, the number of possible file descriptors is limited, so one cannot open very many different files without closing some of them.

Standard Library (13)

- When one is done with a file, one should close it:

```
err = fclose(fp);
```

The parameters have the following meaning:

- ◇ `FILE *fp`: file pointer of an open file,

After the call to `close`, the file pointer cannot be used.

- ◇ `int err`: The function returns `0` if successful and `EOF` in case of an error.

- Closing a file that was written also outputs any remaining characters in the buffer.

UNIX System Calls (11)

- Every program has a current (working) directory which is used for interpreting relative path names.
- The current directory can be changed with the following system call:

```
err = chdir(path);
```

- The parameter is declared as “`const char *path`”. It can be a relative path.
- Return value: `0` if successful, `-1` if error.
- The current directory can be queried with `getcwd`.

UNIX System Calls (12)

- Files can be deleted with the following system call:

```
err = unlink(path);
```

- The parameter is declared as “`const char *path`”.
- Return value: `0` if successful, `-1` if error.
- If the file is still open, it vanishes only after it is closed.

UNIX also permits that the same file has several names in possibly different directories. The file is really deleted only when the last link to it is removed.

- `unlink` cannot be used for deleting directories.

Standard Library (14)

- The standard library offers the following function for deleting files:

```
err = remove(path);
```

- The parameter is declared as “`const char *path`”.
- The function returns `0` if successful and a nonzero value in case of an error.
- The file should not be open when it is removed, or else the result is implementation-defined.

Standard Library (15)

- The standard library also has a function for renaming files:

```
err = rename(oldname, newname);
```

- The parameters are declared as “`const char *oldname`” and “`const char *newname`”.
- The function returns 0 if successful and a nonzero value in case of an error.
- The file should not be open when it is renamed, and there should be no file with the new name, or else the result is implementation-defined.

Standard Library (16)

- It is possible to create a temporary file with:

```
fp = tmpfile();
```

- The function returns the file pointer `FILE *fp` of an open file (or `NULL` in case of an error).
- The file is opened with the mode `"w+b"` (open for reading and writing in binary mode).
- The file will be removed when it is closed or when the program exits.

Standard Library (17)

- The standard library also has a routine to create a file name for a temporary file (which is guaranteed not to conflict with any existing file):

```
filename = tmpnam(buf);
```

- The parameter `char *buf` must point to a character array of at least `L_tmpnam` characters.

It is also valid to pass a `NIL` pointer in which case `tmpnam` uses its own buffer.

- `tmpnam` returns a pointer to the new file name (normally `buf`), or `NULL` if it fails.

UNIX System Calls (13)

- File permissions are changed with this call:

```
err = chmod(path, rights);
```

- The parameters are declared as “const char *path” and “int rights”.

The return value 0 means successful, -1 means error.

- The following system call can be used (by the superuser) to set the owner and group of a file:

```
err = chown(path, owner, group);
```

UNIX System Calls (14)

- One can check whether a file is accessible in a given mode with this system call:

```
err = access(path, mode);
```

- “`const char *path`” is the file to be checked.
- “`int mode`” is 4 for read access, 2 for write access and 1 for execute access.

`sys/file.h` may define the constants `F_OK` (mere existence test), `R_OK` (read access), `W_OK` (write access), `X_OK` (execute access). The flags can also be combined with bit-or.

- Return value: 0 (access possible) or -1 (forbidden).

UNIX System Calls (15)

- It is possible to get information about a file (“file status”) with this call:

```
err = stat(path, buf);
```

- `const char *path` specifies the file for which the status is requested.
- `buf` is the address of a structure of type `struct stat` that is filled by the call.

The structure is defined in `sys/stat.h`. It may be necessary to include `sys/types.h` first.

- The function returns `0` if successful, `-1` if not.

UNIX System Calls (16)

- “`struct stat`” has e.g. the following fields:
 - ◇ `st_mode`: Access rights and file type.
 - ◇ `st_uid`: Owner of the file.
 - ◇ `st_gid`: Group to which the file belongs.
 - ◇ `st_size`: Size of the file in bytes.
 - ◇ `st_atime`: Last time the file was accessed (r/x).
 - ◇ `st_mtime`: Last time the file was modified (w).
 - ◇ `st_ctime`: Last time the file information (i-node) was changed.

UNIX System Calls (17)

- The field `st_mode` contains the access rights bits plus file type bits defined as constants in `sys/stat.h`.

E.g. the bit `S_IFDIR` is set if the file is a directory, `S_IFREG` if it is a regular file.

- The library function `ctime` translates the times in `struct stat` (of type `time_t`) to a readable form.

The function `gmtime` translates `time_t` to a structure `struct tm` defined in `time.h`. Fields are, e.g., `tm_mday` (1..31), `tm_mon` (0..11), `tm_year`.

- One can specify the file also via a file descriptor:

```
err = fstat(fd, buf);
```

UNIX System Calls (18)

- Directories under UNIX are normal files that contain file names and i-nodes (references to the internal file data).

In older UNIX variants, file names were restricted to 14 characters. A directory consisted of a sequence of records that were each 16 Bytes long. The first two bytes contained the i-node number (or 0 for deleted entries) and the other 14 bytes the file name. In modern UNIX variants, the records have variable size to permit long file names. A directory always contains entries for the special file names “.” (the directory itself) and “..” (the next higher directory in the file system hierarchy). 4.2BSD UNIX introduced long file names (up to 255 bytes). There, directory entries consist of: The size of the entry, the length of the filename, the inode, the filename, possibly empty space (e.g. when the next entry was deleted or to give better alignment).

UNIX System Calls (19)

- Usually, one can read directories with the normal `open` and `read` system calls.

Under Solaris, read operations are not permitted on directories (depending on the type of file system, maybe it is excluded only for NFS file systems accessed via the network, e.g. `read` worked for `"/`).

- The library routines `opendir`, `readdir`, `closedir` simplify the parsing of directory data structures.

See `dirent.h` or `sys/dir.h`. These functions can be used under Solaris.

- However, one can change directories only with calls like `creat`, `link` and `unlink` (in order to protect the consistency of the data structure).

UNIX System Calls (20)

- Directories are created with

```
err = mkdir(path, rights);
```

In old UNIX versions, `mkdir` was not a system call, but one had to execute the system program `mkdir` instead. This program used the `mknod` system call, but had to become root (the super-user) for that purpose, so it was a “setuid” program.

- Directories (that must be empty) are deleted with

```
err = rmdir(path);
```

As with `mkdir`, `rmdir` was not a system call in older UNIX versions. One had to execute the system program `rmdir`. This program used internally `unlink`, but one had to be root to use `unlink` on directories, so `rmdir` was another setuid program.

Standard Library (18)

- The C standard library has no functions to read directories or query file attributes.

One can also not create or delete directories. One can determine the file size by using `fseek` to position at the end and then query the position with `ftell`.

- Thus, directories can only be processed in a system-dependent way.

Since C originated in a UNIX environment, C implementations sometimes offer UNIX system calls as far as possible. E.g. MS VC++ has `_stat` (or even `stat` if `__STDC__` is not defined). Of course, one can also use the Win32 API functions `FindFirstFile`, `FindNextFile`, `FindClose`, `GetFileAttributes`, `GetFileInformationByHandle`, `GetFileSize`, `GetFileTime`, `GetFileType`, `GetFullPathName`, etc. (see `winbase.h`).

UNIX System Calls (21)

- Further I/O system calls:
 - ◇ **chroot**: Change the file system root directory.
 - ◇ **mount**: Add disk to the directory tree.
 - ◇ **umount**: Remove disk from the directory tree.
 - ◇ **mknod**: Create file system entry for special device.
 - ◇ **link**: Create a second name for the same file.
 - ◇ **dup**: Duplicate a file descriptor.
 - ◇ **fcntl**: “File control”, used e.g. for locking.
 - ◇ **ioctl**: Device-specific operations.

Overview

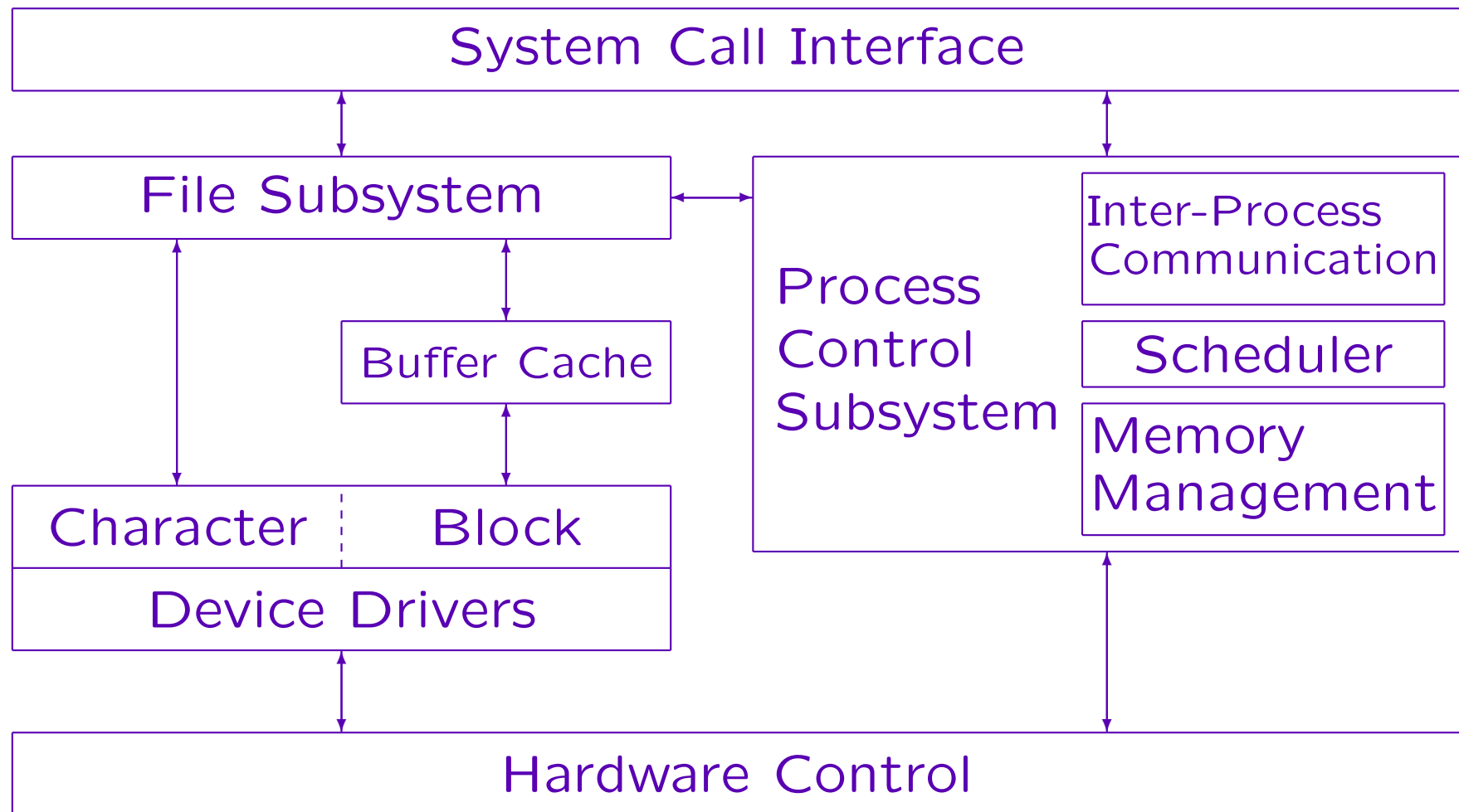
1. Motivation for Operating Systems

2. Disk Hardware

3. File System Calls

4. Implementation

UNIX System Kernel [Bach 1986]



The Buffer Cache (1)

- In order to work with a disk block (i.e. read/write bytes in it), the entire block must be in memory.
- The purpose of the buffer cache is:
 - ◇ Give file system number (i.e. the disk/disk partition) and block number within the file system,
 - ◇ return the main memory address of a buffer that contains the block.

It is really a copy of the block. The block data are read into the buffer, but the original version of the block is still on the disk.

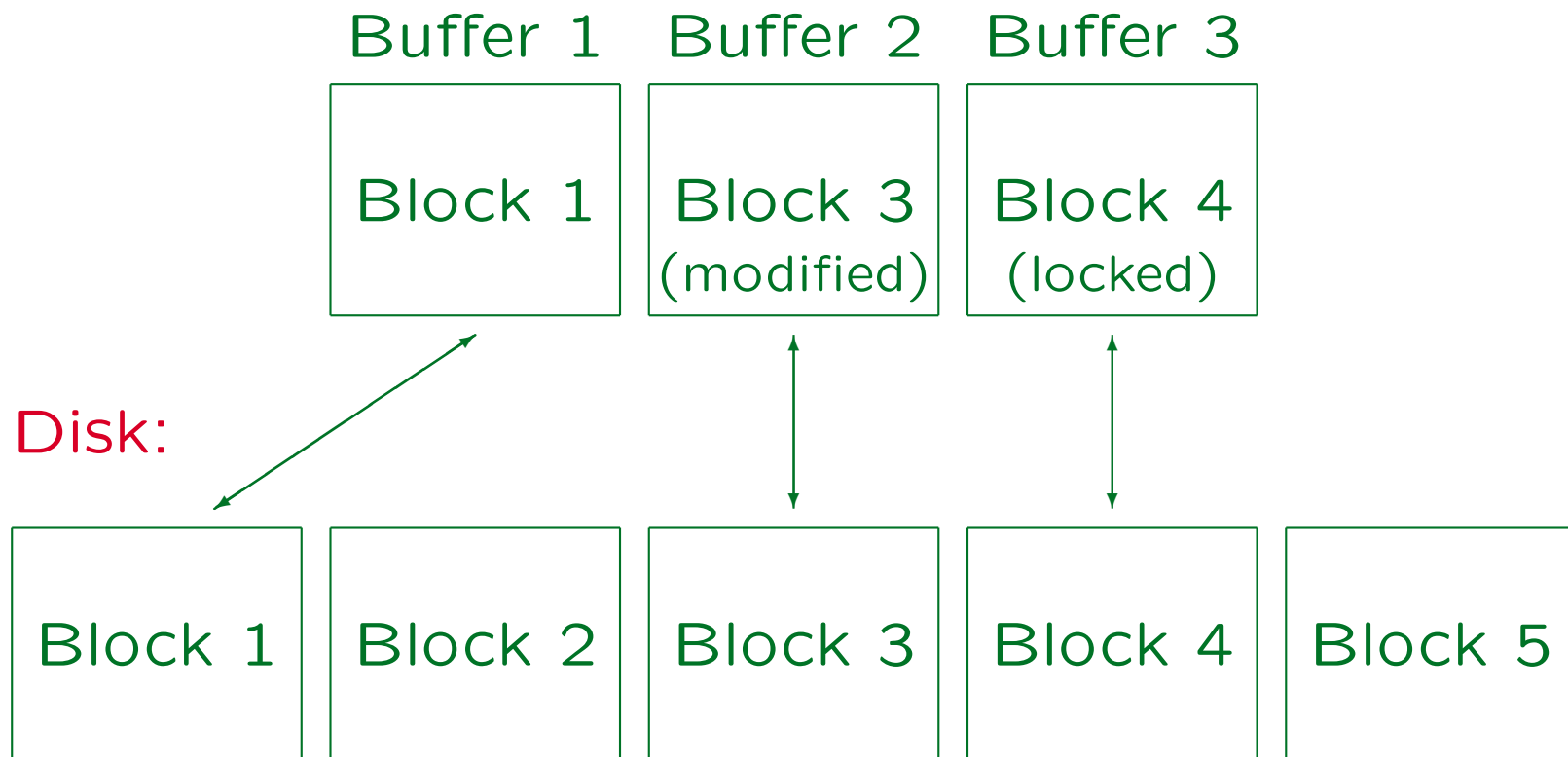
The Buffer Cache (2)

- It is also possible to change the buffered version of the block in which case it must be written back to the disk (maybe with some delay).
- The buffer cache functions are also linked to process management (buffers in use must be locked).

This will be discussed in a later chapter. When a block must be read from the disk, the corresponding process (program) is “put to sleep” (it must wait). In the meantime, the CPU can execute another process (program). However, while a system call works with a buffer, the system must ensure that no other process can modify it. This is done by means of a “lock”. One also says that the block is “pinned” in the buffer (so that the buffer is not accidentally reused for another block). When the system call is finished, it unlocks/unpins the blocks.

The Buffer Cache (3)

Main Memory:



The Buffer Cache (4)

- Quite often the same disk block is requested again after a short time:

- ◇ E.g. when reading a file in small portions.

If a user requests e.g. the first 100 bytes of a file, the entire first data block (e.g. 8192 bytes) is fetched from the disk. The user will probably soon request the next 100 bytes, which are stored in the same block. When directly using `read` and `write` (not the standard library), it is better to write data in multiples of the blocksize. `BUFSIZ` defined in `stdio.h` is the smallest possible efficient size. The `struct stat` structure contains a preferred block size.

- ◇ The “superblock” of the file system is frequently accessed. It contains e.g. the information about free blocks.

The Buffer Cache (5)

- Reasons for frequent accesses to the same block:
 - ◇ The inode of a file is requested when a file is opened. Later the inode must be written back to disk, but the inode is only part of a block, so the containing block must be read again before the modified version can be written back.

Inodes are explained below. They contain management information about a file, e.g. the date of last access.

- ◇ Certain files and directories are often accessed.

E.g. the current working directory of the user is probably often accessed. There are also certain programs that are often executed (programs are stored in files).

The Buffer Cache (6)

- Therefore, the block data are not immediately deleted from the buffer when the system call is done.
- Instead, the system has a queue of “free” buffers (the “free list”).

These buffers normally still contain valid block data. For every buffer, the system stores the file system number and the block number in the buffer. If the buffer is really empty, it is marked as invalid.

- When the system is done with a block (it is “unlocked”), the buffer is put at the end of the queue.

The Buffer Cache (7)

- When the system needs a free buffer, it takes the one at the front of the queue.
- If during the time while the buffer moves through the queue, its block is requested again, it is removed from the queue and immediately returned.
- In this case, the request for a block can be satisfied without actually reading it from the disk (because it is still in memory).

When the block is unlocked, it is again added at the end of the free list. Often accessed blocks can stay in memory for a long time.

The Buffer Cache (8)

- The system caches recently requested blocks in memory. Caching is often used for storage hierarchies:
 - ◇ Main memory is fast, but small and expensive.

And its contents is lost in case of a power failure or system crash. I.e. main memory is volatile storage. Every byte can be accessed.
 - ◇ The disk is much slower, but larger and cheaper.

Its contents is only lost when the disk fails (happens seldom). The disk is persistent storage. Only whole blocks can be accessed.
- E.g. a PC has today 256 MB RAM (\$0.15–\$1.00 per MB), and a 40 GB disk (\$0.002–0.01/MB).

The Buffer Cache (9)

- In the next level of the storage hierarchy, also the contents of main memory is cached in special “cache” memory (which may be part of the CPU chip or placed between the CPU and main memory).
- Since the size of the cache is restricted, not every block of the disk can be kept in main memory.
- When a new block is requested, normally all available buffers already contain a block.

It would not be advantageous to remove blocks from main memory buffers without need, since more or less every block might be requested again.

The Buffer Cache (10)

- Then the system must select a “victim”, i.e. a block that is removed from its buffer in order to make space for the newly requested block.
- UNIX uses a “Least Recently Used” replacement strategy for its buffer cache.

All blocks behind the first one in the free list have been accessed (became unlocked) after the first one.

- If the block in the buffer was modified and not yet written back to disk, the buffer cannot be reused.

In this case, UNIX will start the write operation and allocate the next buffer on the free list.

The Buffer Cache (11)

- UNIX often uses a “**delayed write**”: The buffer is marked as modified, but the system call completes without actually writing the block to disk.

System calls unlock all buffers before they complete, so the block is put on the free list. The actual write is only started when the block gets to the front of the free list and cannot be reused as explained above. The block is then removed from the free list. When the write later completes, the buffer is put back at the front of the free list.

- The advantage is that a block might be changed several times and written only once at the end.

If the block belongs to a temporary file or a pipe (see a later chapter), then it might not be necessary to write the block to disk at all.

The Buffer Cache (12)

- “A user issuing a `write` system call is never sure when the data finally makes its way to the disk.”

“The standard I/O package available to C language programs includes an `fflush` call. This function call flushes data from buffers in the user address space (part of the package), into the kernel. However, the user still does not know when the kernel writes the data to the disk.”

[Bach 1986]

- Modern UNIX implementations have ways to make sure that data is actually written to disk.

On Solaris, the `open` system call also has e.g. the flag `O_SYNC` which means that subsequent `write` calls will not return until the data is delivered to the hardware. The function `fsync` writes all buffered data of a given file, `sync` schedules a write for all dirty buffers.

The Buffer Cache (13)

Exercise:

- Suppose you have 3 buffers and 10 disk blocks. What happens when the following blocks are requested? Show the free list after each step.

1, 9, 1, 5, 8, 1, 8.

- For simplicity, assume that each buffer is unlocked before the next block is requested.

We also assume that all accesses are read accesses.

- The current buffer contents are blocks 1, 2, 3.

The buffers are in this sequence on the free list.

The Buffer Cache (14)

- At least in older UNIX versions, the kernel contains many fixed-size tables.
- E.g. the number of buffers is fixed.
- There is a configuration file that contains the table sizes. One can change this configuration file and then build a new kernel.
- This is one method of performance tuning.

Also certain programs (e.g. database management systems) might need more resources of a certain kind (e.g. semaphores) than are configured into the standard kernel.

The Buffer Cache (15)

- When a system call requests a block, the buffer cache module must check whether this block is already in memory (i.e. contained in a buffer).
- Since the number of buffers can be quite large, it would be relatively slow to do a linear search.

There is a table (an array) that contains information about every buffer which includes the number of the block (and the file system) that is contained in the respective buffer. A linear search means to check every entry in the buffer table for one that contains the given block and file system number. A faster algorithm is advantageous also because the system has to search for a block rather frequently.

The Buffer Cache (16)

- Therefore, UNIX uses a hash method to locate the buffer that contains a given block.
- Instead of a single long linked list of buffers it has an array with many short linked lists.
- The array is indexed by a hash function computed from the block number and the file system number.

In principle, one can use any function that maps block and file system number to indexes in the array. Of course, the same function is used when a block is entered and when it is later searched. Good hash functions evenly distribute blocks over the entire array. It is normal that different block numbers are mapped to the same array location. Each array element contains the start of a linked list of blocks.

The Buffer Cache (17)

- Below a hash table of size 5 for 4 buffers is shown.

It is quite common that the hash table is slightly larger than the number of entries. In this way the probability of long chains is low. But even if one should have bad luck (or use a bad hash function) and all blocks end up under a single entry of the hash table, the method still works (But it is slow: In this worst case one does a linear search).

- The blocks 4, 8, 15, and 19 are in memory.

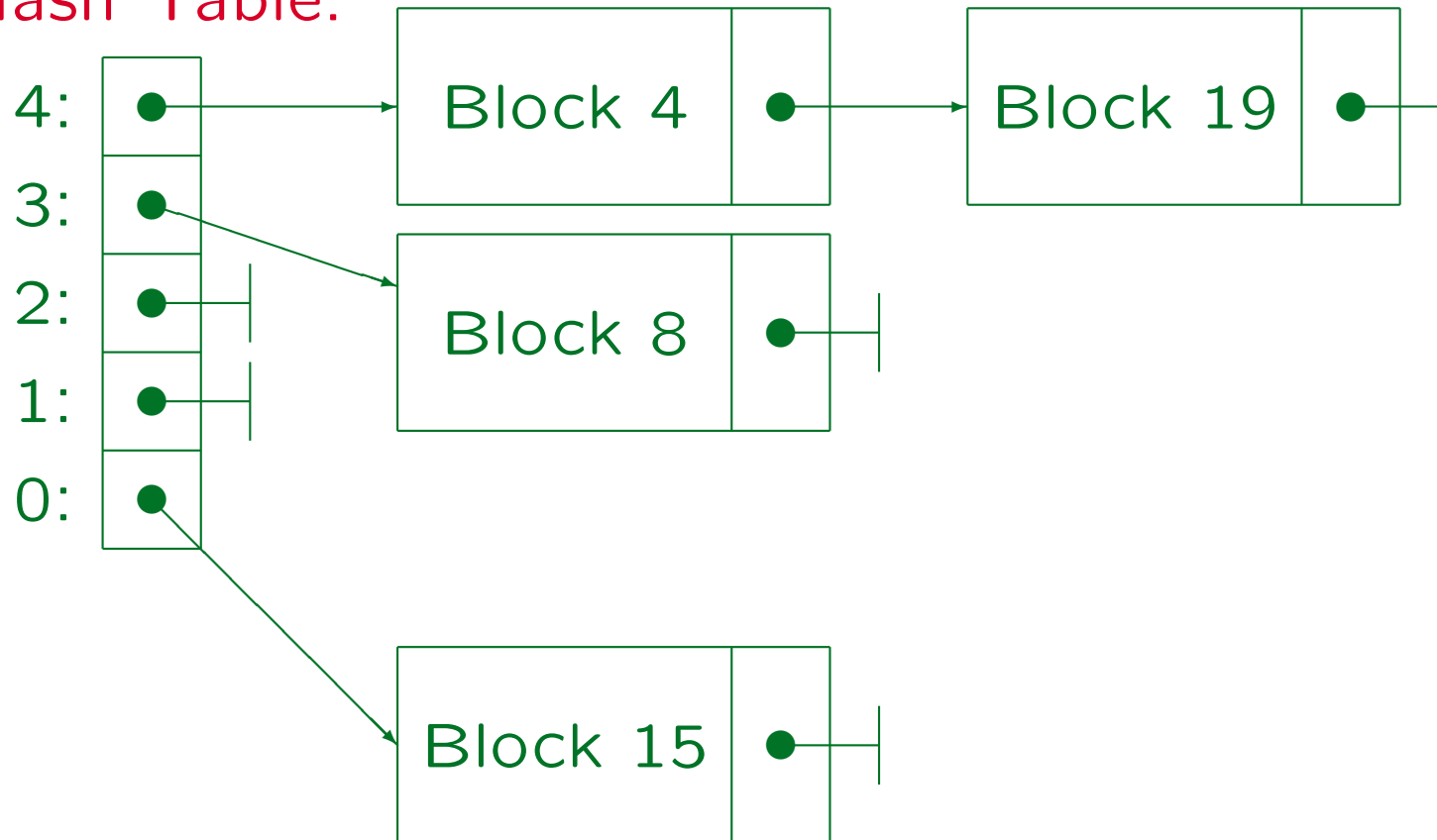
For simplicity, we assume that there is only one disk (file system). So the block number is sufficient to identify each block.

- The hash function is the block number modulo 5.

I.e. the remainder of dividing the block number by 5 is the index 0..4 in the hash table, under which the block is entered.

The Buffer Cache (18)

Hash Table:



The Buffer Cache (19)

- Most of the buffers will additionally be on the free list (not shown on the previous slide).

Only buffers that are locked (i.e. currently used) or asynchronously written are not on the free list.

- When a buffer is reused for another block, it must be removed from the current list in the hash table.
- Therefore, each entry in the hash table is really a doubly-linked list.

Every list element has a pointer to the next and to the previous element. This makes it simpler to delete elements from the middle of the list.

The Buffer Cache (20)

- It is very important to make sure that no block ends up two times in the buffer cache.
- Therefore, the buffer that is selected for a new block is already put into the hash table list for the new block before the block is actually read.

Of course, the buffer is marked as “currently being filled” (so that it is clear that it does not yet contain the block).

- Then the read request is passed to the device driver and the current process is put to sleep.

I.e. the CPU does something else. Processes are discussed in more detail in a later chapter.

The Buffer Cache (21)

- The `read` system call first requests the block and then copies the data from the system buffer into the array that the user specified.

In the same way, `write` copies data from the user buffer into the system buffer and then writes the system buffer contents to the disk (with a certain delay).

- It might seem a bit inefficient that the disk controller (DMA chip) does not directly copy the data into the array that the user specified in the `read` system call.

The Buffer Cache (22)

- Reasons for separation of user and system buffer:
 - ◇ Some DMA hardware has alignment requirements for the buffers, which do not exist for the user.
 - ◇ The user might not read or write full blocks.
 - ◇ The user can change the array without informing the system. But cached blocks can only be reused if are still a valid image of the disk block.
 - ◇ When the data arrives, user memory might be not accessible.
 - ◇ System buffers are locked only for a short time.

File Management (1)

- For each file, the operating system must be able to find the sequence of disk blocks that contains the data of the file.

A file contains a sequence of bytes that are stored in a sequence of disk blocks. The last block might be only partially filled: The operating system can detect this because it also stores the file size (number of bytes in the file).

- One option is to store each file in consecutive disk blocks: Then the operating system must store only the position of the first block of the file and the number of blocks.

File Management (2)

- Using consecutive disk blocks is very inflexible:
 - ◇ When files grow, it might be necessary to copy them to a different disk position because the next disk block is already occupied by another file.

This is especially problematic since when a file is being written, the operating system does not know how long it will be.

- ◇ When files are deleted, they leave holes between used disk blocks. These holes can only be reused by a file that is not longer than the original file.

In the long run, the free space on the disk will become fragmented: There will be many small holes, making it difficult to find space for big files.

File Management (3)

- For this reason, operating systems normally do not require that the disk blocks for a file are stored consecutively on the disk.
- This means that the operating system must explicitly store the sequence of disk blocks for each file.
- If the disk blocks for a file are spread over the entire disk, sequential access to the file data is much slower than if the blocks are stored consecutively.

File Management (4)

- For some operating systems (e.g. DOS/Windows) there are “Defragmentation” utility programs that exchange the contents of disks blocks in such a way that all currently existing files are stored in consecutive locations.

The operating system still needs to store the sequence of disk blocks for each file. Immediately after the defragmentation tool was executed, these sequences describe consecutive disk blocks. But new files or modified files need again the general case.

- Some operating systems (e.g. UNIX) try to allocate blocks that are near to each other if possible.

File Management (5)

- File systems for read-only media (e.g. CD-ROMs) can of course use the simple (and efficient) case and require that each file is stored in consecutive blocks.

The problem are only deletions and updates that change the file size. This cannot occur on CD-ROMS.

- This example also shows that operating systems can manage files with different data structures on different devices.

E.g. a floppy disk can use a relatively old MS-DOS FAT file system even under UNIX for compatibility reasons.

File Management (6)

- E.g. the Oracle database management system uses a compromise between both possibilities:
 - ◇ A table (corresponding to a file) is stored in a sequence of extents, where each extent consists of consecutive disk blocks.
 - ◇ Optimal case: Only one extent is required.
 - ◇ But when tables grow, additional extents can be allocated (which can be anywhere on the disk).

The extent size can be specified for each table. If it is chosen too large, disk space is wasted (the complete extent is reserved for the table), if it is chosen too small, many extents are required.

UNIX File Management (1)

- Under UNIX, the data about each file are stored in inodes (“index nodes”).
- Internally, files are identified by the file system number (i.e. the disk/disk partition) and the inode number within the file system.
- Directories map file names to inode numbers, but contain no other data about the file.

Directories do not contain file system numbers because they only reference files on the same disk. Different disks are integrated into the file system hierarchy via a mount table in the system.

UNIX File Management (2)

- Since directories are basically normal files under UNIX, they are also identified via inodes.
- File information is stored separately from directory entries also because the same file can have multiple names and be contained in different directories.

Instead of copying a file (which would create a new file), one can also create a new link pointing to the same file. This means that the file hierarchy under UNIX is not really a tree, but only a (hopefully) acyclic graph with one root node, from which all other nodes are reachable. After creating the link, both names for the file have equal status: UNIX does not distinguish between an original name and an alias. However, modern UNIX variants have besides these “hard links” also “symbolic links” (which are distinguishable from the file itself).

UNIX File Management (3)

- Inodes contain the following information:
 - ◇ file owner and group,
 - ◇ file type,
 - Can be: regular file, directory, character device (e.g. keyboard), block device (e.g. disk), pipe (for inter-process communication).
 - ◇ file access permissions (rwx bits),
 - ◇ file access times (last read, write, inode change),
 - ◇ number of links to the file,
 - ◇ file size,
 - ◇ information to find the data blocks (see below).

UNIX File Management (4)

- The disk contains an array of inodes in a number of blocks that were reserved for this purpose when the UNIX file system was created.

A disk must first be low-level formatted by instructing the disk controller to write every sector (including e.g. the track and sector numbers and checksum information). Under Solaris, this is done with the program `format`. This program can also be used to split the disk into several “partitions” (logical disks). After that, it is still necessary to initialize the UNIX file system and e.g. write the “superblock” (see below). This is done with the program `mkfs`. At this point, one can choose how large the array for inodes should be. This defines the maximal number of files that can be stored on the disk or disk partition. All blocks that are reserved for inodes cannot be used for file data, so it is also not good to make the array extremely large.

UNIX File Management (5)

- In older UNIX systems, disks were divided as follows:



- The first block contains code or information for loading the operating system kernel.

Of course, this is really used only for one disk (the boot device). Nevertheless, the first block is always reserved for this purpose.

UNIX File Management (6)

- The second block is the “superblock” that contains information about the file system, such as the number of blocks reserved for inodes, and the number of data blocks.

In System V, as described here, the superblock also contains information to find free inodes and free data blocks. In 4.3BSD UNIX, this is done differently, and the superblock is never modified after the creation of the file system.

- Then follow blocks reserved for inodes.
- The rest of the disk is used for data blocks.

UNIX File Management (7)

- Disks appear smaller than advertized. E.g. a 9 GB disk might have only space for 8 GB of user data:
 - ◇ As noted above, 1 GB is 2^{20} byte for operating systems, and 10^6 byte for disk manufacturers.

E.g. the OS shows a 9 GB disk as 8.38 GB large.
 - ◇ Some space is occupied by management information and is not available for user data.

E.g. inodes and indirect blocks (see below). This overhead is probably only 1–3%.
 - ◇ Under UNIX, there is a space-reserve for the superuser (5–10%).

UNIX File Management (8)

- Given an inode number, the system can easily compute the corresponding disk address:

- ◇ E.g. inodes are 80 bytes large, and disk blocks 8192 bytes.

So 102 inodes fit into one block, 32 bytes at the end of each block remain unused. An inode should not cross a block boundary: Then the system would have to read two blocks in order to get one inode. Original inode size: 64 bytes. In Linux Ext2 file system: 128 bytes.

- ◇ E.g. inode number 210 is the 6th inode in the 3rd inode block (5th block on disk).

Inodes are counted from 1 (0 is used for deleted directory entries). The inode is stored bytes 400–479 in this block (counted from 0).

UNIX File Management (9)

- UNIX uses a caching mechanism for inodes that is very similar to the buffer cache for blocks.

There is also a hash table to quickly find an inode if it is in memory, and a free list to determine which inode table entry should be reused.

- For every inode that is in memory, a reference count is stored, which is the number of times the file is currently open. When the reference count is decremented to 0 by closing the file, the inode table entry is added at the end of the free list.

In this way, inodes for open files are locked in memory for a possibly long time. This is different than blocks in the buffer cache.

UNIX File Management (10)

- Every process has a user file descriptor table. Used entries reference entries in a global “file table”.
- The file table entries contain especially the current read/write position, the opening mode, and a reference into the inode table.
- When a program executes another program, it inherits all open file descriptors (the file descriptor table is copied).

UNIX File Management (11)

- In this way, all programs that are executed by a single parent process share the read/write position.

E.g. suppose that a shellscript (batch file) executes programs P and Q , and that the standard output of the shellscript is redirected to a file F . The file F is opened only once, and therefore there is only a single entry in the file table. The output of Q will be written into F after the output of P , since both share one read/write position. This would not be possible if the read/write position would be part of the user file descriptor table: Then Q would start writing F at position 0 again (since the position in the shell has not changed, only in P).

- If, on the other hand, two programs independently open the same file, two file table entries are constructed, and each has its own read/write position.

UNIX File Management (12)

- The disk addresses (block numbers) of the first 10 data blocks of the file are contained in the inode.

If the block size is e.g. 4KB, this is sufficient for files up to 40KB.

- If this is not sufficient, there is also space for the disk address of an **indirect block** in the inode: This block contains disk addresses of data blocks.

E.g. if block numbers are 4 byte, and the block size is 4KB, the indirect block can contain up to 1024 block numbers. Together with the ten block numbers in the inode, files of slightly more than 4MB can be managed in this way.

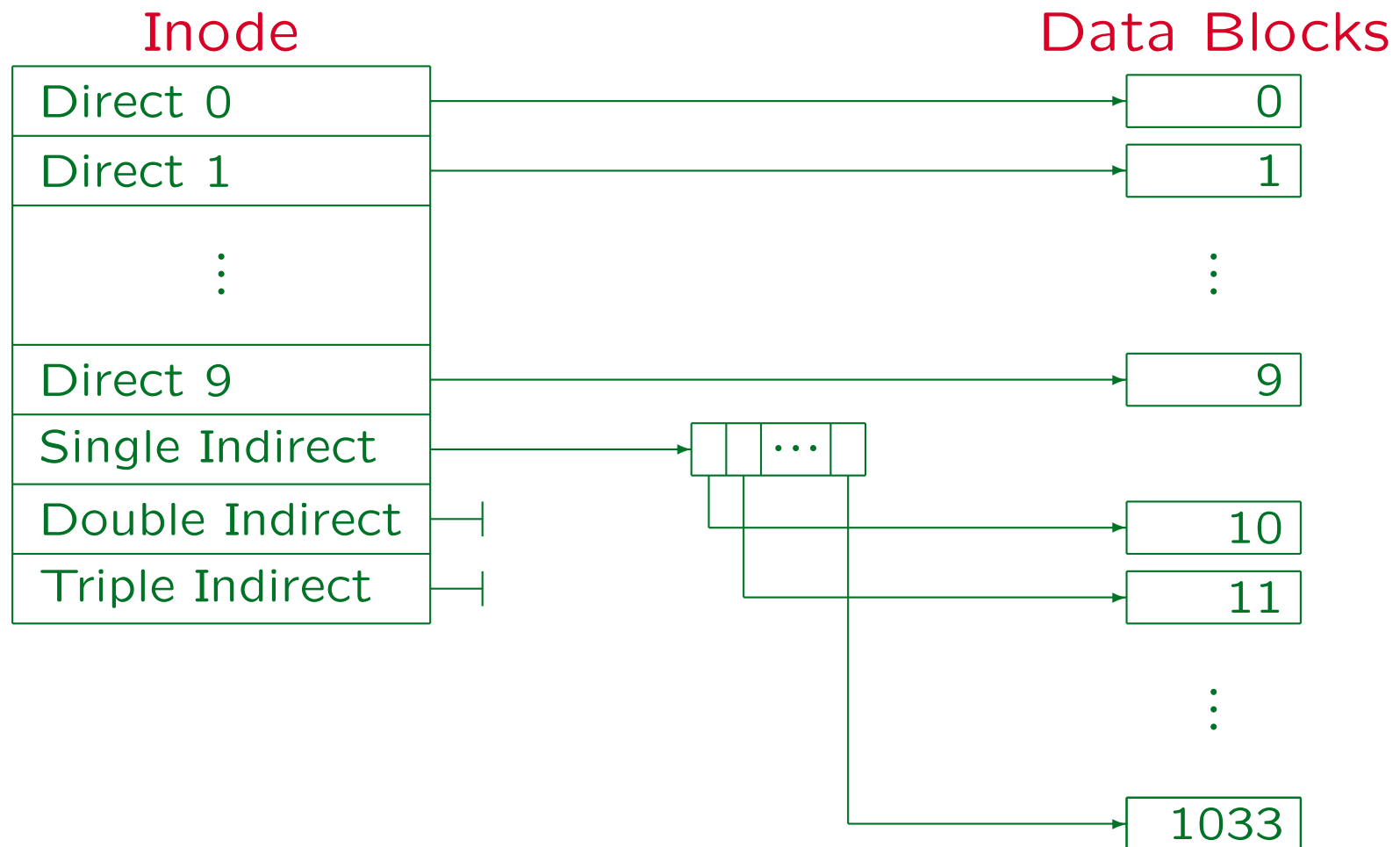
UNIX File Management (13)

- If this is still not sufficient, the inode has also space for the address of a double indirect block, i.e. a block that contains the disk addresses of indirect blocks (which in turn refer to data blocks).

In the example, files of slightly more than 4 GB can be managed in this way (it is more because the ten direct block addresses in the inode and the one indirect block are used in addition to the double indirect block. If the file size field in the inode is 32 bit, the file size is anyway limited to 4 GB. However, more modern UNIX implementations switch to 64 bit, and if the block size were chosen smaller, the double indirect block would still not be sufficient for 4 GB files.

- UNIX can also use a triple indirect block.

UNIX File Management (14)



UNIX File Management (15)

- The UNIX data structure is well suited to non-sequential access to the data blocks.
- E.g. suppose that the user does an `lseek` to byte 100 000, and then reads some bytes.
- If the block size is 4096 bytes, byte 100 000 in the file is byte $100\,000 \bmod 4096 = 1696$ in block $100\,000 / 4096 = 24$ (all are counted from 0).
- In order to find block 24, the system must read the single indirect block and then take entry number 14 there which is the disk address of block 24.

UNIX File Management (16)

- In UNIX, it is possible that files have “holes”.
- E.g. it is possible to create a new file, do an `lseek` to byte 100 000, and only then start writing.
- Then data blocks 0 to 23 will not be allocated.
- It is possible to read bytes before byte 100 000 later.
The system will then return 0 bytes.

Therefore, when the file is later copied (by reading and writing each byte), the copy will need more disk blocks than the original.

UNIX File Management (17)

- When a file is written (such that its size grows) the system must allocate free disks blocks.
- When a file is deleted or overwritten (by first reducing its size to 0), the system must move blocks back into the pool of free blocks.
- Older UNIX systems manage free blocks via a linked list of blocks that contain block addresses of free blocks.

The blocks that implement the linked list are themselves free.

UNIX File Management (18)

- The superblock contains an array with block numbers of free blocks (anchor of the linked list).
- The last entry in this array is the address of a block B that contains itself block numbers of free blocks. The system copies these block numbers into the superblock and then uses block B .
- If the array in the superblock is full and another block B becomes free, B is filled with all block addresses in the array. B then becomes the last (and only) entry in the super block array.

UNIX File Management (19)

- In a freshly created file system, the free disk blocks in the linked list are sorted by their disk position.
- Thus, if a file is sequentially written, it will be stored in contiguous disk blocks.

Assuming that no other files are written in parallel.

- After the disk is used for some time, the blocks on the list of free blocks will be in no particular order.
- Now, even if a file is sequentially written, its disk blocks will be spread randomly over the entire disk.

UNIX File Management (20)

- For this reason, in 4.3BSD UNIX, the disk was split into several cylinder groups.
- Each cylinder group contains its own inodes and data blocks.

It also has a redundant copy of the super block (which is never modified under 4.2BSD UNIX) and a bitmap for finding free blocks.

- The system tries to keep each file within a single cylinder group if possible.

In this way, the inode is not very far from the data blocks. In the old structure, all inodes were stored at the beginning of the disk and the data blocks could be at the other end of the disk.

UNIX File Management (21)

- New files are spread evenly over the cylinder groups.

In this way, free space remains within the cylinder groups for the existing files to grow.

- Free blocks are managed via a bitmap, i.e. an array of bits. If bit number n is 1, then block number n is free, if it is 0, block number n is already in use.

If the blocksize is 4 KB, one bitmap block is required for each 128 MB of storage ($4096 * 8 = 32768$ blocks).

- When a new block is allocated for a file, the system tries to find a block that is near to the previous block (can be efficiently done with bitmaps).

UNIX File Management (22)

- Larger block sizes improve the file system efficiency:
 - ◇ Blocks are stored in one piece on the disk. If the block size is larger, the file consists of fewer blocks, and fewer seeks are needed between reading the blocks.
 - ◇ Fewer indirect blocks are needed.
- The larger the block size, the more space is wasted: The last block of a file is on average only half full.

Also, UNIX systems often contain many small files. There was an investigation (1986?) stating that 45% disk space are wasted already with 4 KB blocks. I do not believe that.

UNIX File Management (23)

- Therefore, 4.3BSD UNIX permits that disk blocks can be split into several “fragments” that can be allocated by different files.
- E.g. suppose that the disk block size is 4 KB and the fragment size is 1 KB.
- Then a file of 10 KB will be stored in 2 complete blocks and one half (2 fragments) of a third block.
- The remaining two fragments of the third block can be allocated by other files.