Intro to the Linux/UNIX OS Programming

CISC 3350 — CUNY Brooklyn College
Lecture Notes

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

What is the "Workstation Programming" course about?

In this course, you will learn how to write programs used by the Linux operating system and can flawlessly run on various Linux versions. That is, it is about *programming a Linux device*.

Workstation computers, which are computers designed to work with heavy loads of computations, conventionally ran the UNIX operating system; Linux is an OS based on UNIX. As such, in the scope of our course, "Workstation Programming" means the same as "Linux Programming".

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Briefly on Linux and UNIX

Linux's mascot: Tux the penguin

UNIX is a proprietary (non-open-source) operating system developed by AT&T in the 1960s. Its advantage compared to existing OSes was that (1) it allowed several programs to run on the device at the same time, (2) allowed more than one user to access the system at the same time and, most prominently, (3) was able to run on various devices.

This 3rd trait is called **portability**, and the reason that UNIX excelled in it is that UNIX was written in the C language. Once a C source code is compiled on a machine, it is translated into the machine language that the device understands, so it can readily run on that machine.

In the 90s, Richard Stallman, a software programmer, wished to create a free, open-source OS similar to UNIX. He started working on open-source programs he collectively named GNU, but couldn't devise a well-working kernel (the heart of an operating system.) It was the Finnish software engineer Linus Torvalds who coded a productive kernel he called **Linux**. Joining forces, the Linux operating system was released in 1991.

Follow-up question: According to the paragraph above, what is the source of the name "Linux"?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Intro to the Linux/UNIX OS Programming

Linux is a very flexible and powerful operating system: it runs on a variety of types of devices, from appliances to phones to computers to servers, making it the most widely used operating system in the world.

Fun Class Activity: Which of the following are Linux-based operating systems? Which are UNIX-based operating systems? Which are neither?
(***Hint***: Google them out!)

1. Android
2. iOS
3. ChromeOS
4. MacOS
5. Ubuntu
6. Windows

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Linux is nowadays maintained, updated, and upgraded by a loose-knit community of programmers worldwide, including Linus Torvalds himself. Newer versions of the Linux kernel are released every several months.

An operating system that includes the Linux kernel is called a **Linux distribution**, a **Linux distro**, or a **Linux flavor**. Examples include Ubuntu, Debian, RedHat, and CentOS.

As we already mentioned, Linux is based on the Unix OS. It inherited a great deal of its features, such as the ability to run several programs simultaneously, and is also written in the C language. However, Linux isn't intended to mimic UNIX fully; it supports many features that UNIX doesn't.

Speaking about C, because, in this course, we will write programs that imitate programs that execute on Linux devices, we need to acquire some C-programming skills. As such, the next 2.5 weeks will be dedicated only to learning how to program in the C language and using frequent Linux commands and text editors!

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Any questions so far?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

System Programming

"Workstation Programming", or "Linux Programming", is an example of **System Programming**, which is the practice of creating, editing, and maintaining an operating system.

The programs that system programmers use in their job are called **system software**: these are **utilities** (= commands = low-level programs) that, on one hand, interact with the kernel (= the heart of the operating system), and, on the other hand, use the functions of C libraries, e.g., the `string.h` library, which is a higher-level code.

Examples of system software include the command prompt/terminal app, text editors, compilers, debuggers, basic commands (e.g., `whoami`, `cd`, `pwd`,) antivirus software, databases that the OS uses, network services, etc.

A system programmer not only uses existing utilities but also, when needed, writes new, useful utilities.

The homework assignments in this course will provide you with the opportunity to both use in-built Linux utilities and create utilities yourself! The utilities you'll create will closely imitate some commonly used Linux commands, including `cat`, `uniq`, and `grep`. This course will provide you with the skills of a system programmer!

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

System Programming

If I don't plan to work in the field of system programming, why do I need to study this subject?

1. At any point during your career, you might be asked to analyze or modify the functionality of an operating system or its components and utilities, so having already touched on this subject will minimize your future efforts in familiarizing yourself with the topic (and could also lead to a decent raise! 💵 💵 💵)
2. As we already learned, Linux is the most commonly used operating system in the world. Chances are that you will interact with a Linux device outside of your job. If you come up with an idea of making the device work for you better by creating a new Linux utility for your own use (or, perhaps, release it for the world to enjoy!) these skills will always come in handy.
3. Even if you don't get to work with system development, you can make your programs work more efficiently by understanding how an operating system executes them.
4. The algorithms that we will cover in this course in future topics that operating systems use to solve various problems may inspire your own programming practice.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

3 Cornerstones of System Programming

The three ingredients that a system programmer uses to code utilities are:

1. **System Calls** (also: **syscalls**): These are very basic C functions that ask the OS to provide various services to the utility. Examples: the `open()` system call to open a file and `chown()` to change the owner of a file.
   90% of all syscalls are shared by all Linux distros. Linux has about 300 syscalls in total. In contrast, Windows defines ~2000 different syscalls!
2. **The C Library (`libc`)**: In addition to having your utility call syscalls, you will also resort to calling library functions, e.g., `strlen()` to find the length of a string. Each library function is built up of syscalls. Library functions exist for your convenience and free you from spending extra time on implementing things like `strlen()` yourself. On modern Linux, the C libraries are provided by GNU libc (`glibc`). Thank you, Richard Stallman!
3. **The C Compiler**: Once you have written the source code, you will proceed to compile it and create an **executable**/**binary** file (= a program). You can run it either by double-clicking it or typing its name in the command prompt/terminal window. The compiler's job is to convert the C code into machine code, consisting of a sequence of `0`s and `1`s that the computer understands how to execute. The C compiler, `gcc`, is provided on Linux by the GNU Compiler Collection and stands for "**G**NU **C** **C**ompiler".

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

API vs. ABI

In our programming classes, we learned that **API** means "Application Programming Interface", and that it consists of all the functions, methods, classes, and constants that are available via the various libraries of the language. In fact, we can call any function defined in a library that we import, regardless of the computer on which we write that program. In general, an API defines the compatibility (= portability) of your program at the **source code level** (also called **source compatibility**). As long as the function you call indeed exists in the API of the language, your program will compile and run on the device.

On the other hand, **ABI**, "Application Binary Interface", is the set of machine code and instructions that your specific device recognizes. It defines the compatibility of your program at the **binary (executable) level**. Every operating system, version, and distro has its own ABI (the ABI of Linux is different from the Windows ABI.) For example, if you copy a Windows program (the executable, not the source code,) e.g., MS Word, into a Linux machine, the program won't execute correctly (or at all) since Linux won't recognize the machine instructions in the given file.

As such, if you want a program that was created on one device to run on another, you must copy the source code over to the other device and compile it there instead of running the executable file that was created on the first device.

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

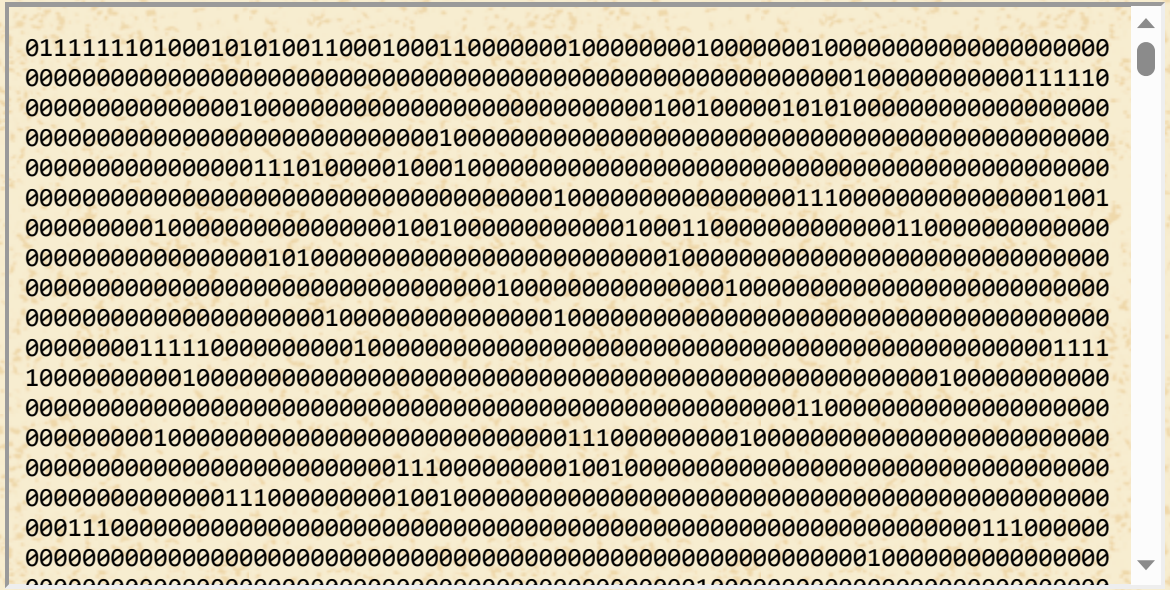
API vs. ABI


```
/* Program writing a message to
 *    the console (screen).
 *
 *    Created for CISC 3350,
 *    Brooklyn College, 1/30/2023
 *    Licensed under CC0 1.0
 */

// Library defining perror() and other i/o actions:
#include <stdio.h>
// Library defining EXIT_FAILURE, EXIT_SUCCESS:
#include <stdlib.h>
// Library defining the 'write' system call:
#include <unistd.h>
// Library defining strlen():
#include <string.h>

int main (void)
```

The source code of prog1.c.



The binary code of prog1.c, represented as a string of 0s and 1s.

The source code on the left will compile on ANY Linux machine because the functions and code used in it belong to the Linux API. However, the binary code on the right will run only on some Linux machines. Every device type has its own ABI, and only if the ABIs of both machines are the same, you can copy over the executable/binary file from one device to another and successfully run it on both!

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Linux Standards

To make UNIX programming orderly and organized, UNIX developers created **programming standards**, which are documents that instruct system programmers how to implement certain aspects of their utilities. When an operating system follows a standard, it is called **standard-compliant**.

Linux, although being UNIX-based, isn't required to comply with any of UNIX standards (which, in some cases, might cost money if the developers want their system to be officially recognized as standard-compliant.) However, wherever possible, Linux *strives* to comply with two such UNIX standards:

1. **POSIX** (= "Portable Operating System Interface", named by Richard Stallman) was proposed in the mid-1980s by IEEE, defines UNIX APIs, shells (command interpreters), and utilities, and its latest version is `IEEE Std 1003.1-2024`. To be officially recognized as a compliant, the developers of an OS must pay more than $10K for a license.
2. **SUS** (= "Single Unix Specification") was proposed in the early 1990s by the Open Group (from the merging of two organizations: OSF and X/Open), and the latest version is `SUSv4-2018`. The license of this standard is actually free, and it incorporates the latest POSIX standard. Still, not all Linux distros choose to officially comply with it.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Linux Standards

The Linux Standard Base (LSB) is a joint project of several Linux distributions under the Linux Foundation, which attempts to extend POSIX and SUS to provide a standard to all Linux distros. This attempt is still unsuccessful.

Despite that, the Linux kernel guarantees the stability of system calls on all Linux distributions.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The gcc Command

Before we start creating any program, let's familiarize ourselves with how to compile C programs using a Linux terminal.

Once the source code for a C program was created, e.g., in the file `program.c`, you would type the following command to the right of `$` to compile the program:

```
$ gcc -Wall -Wextra -O2 -g -o program program.c
```

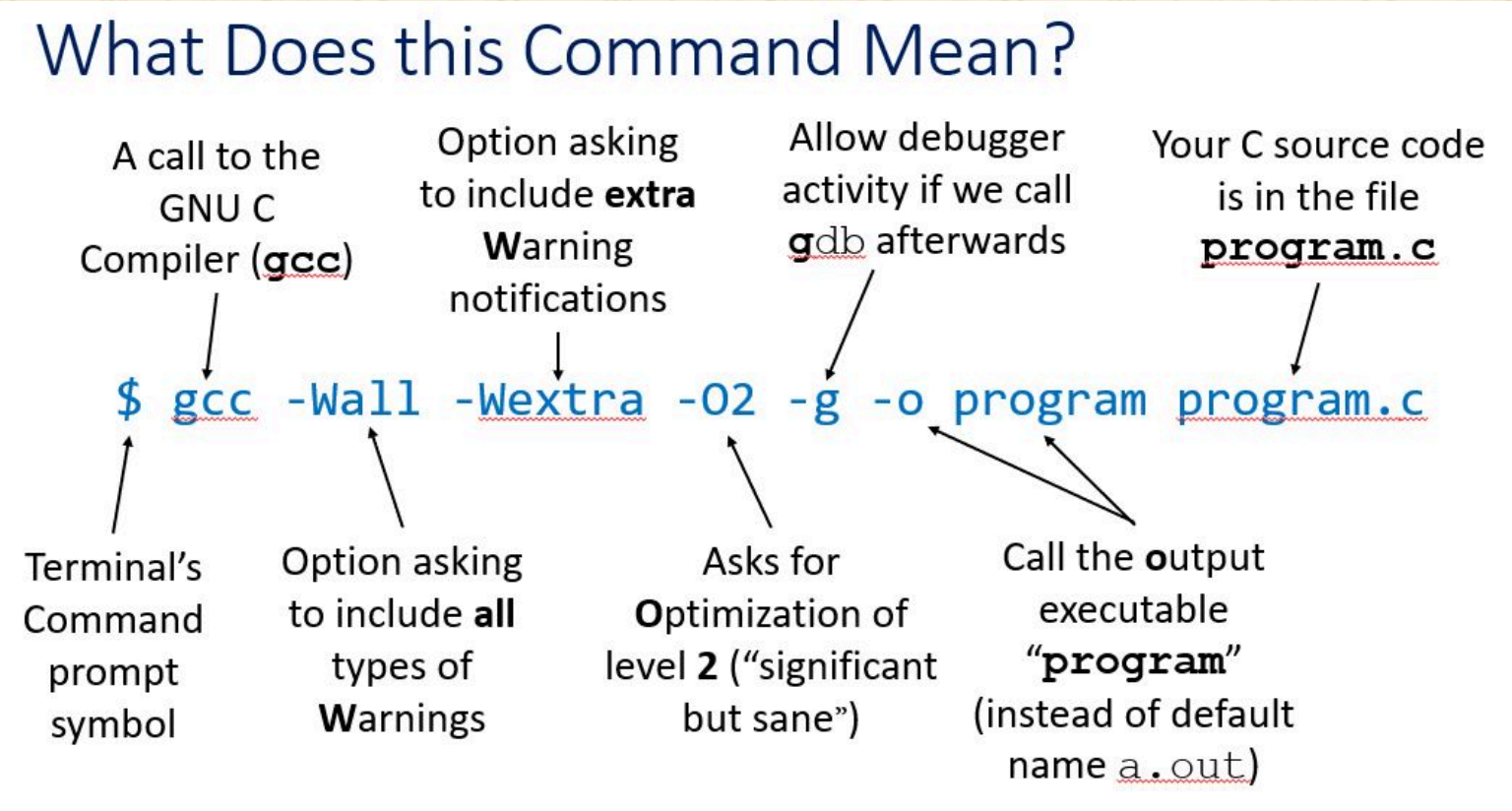
As we mentioned earlier, `gcc` is the compiler of the C language. Moreover, as the above shows, `gcc` is also a command that you can invoke via the terminal to compile source code.

After this command starts running, it will either print error/warning messages to the screen or, if there are no syntax errors in the source code, it will create an executable that you can later run!

Fun question: How do you compile a Java program, e.g., `prog.java`, via a terminal or command prompt?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The gcc Command



gcc command explanation. Miriam Briskman, CC BY-NC 4.0.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The gcc Command

Any questions so far?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem

The Google Files logo

The rest of these lecture notes will briefly introduce some of the topics that we will cover this semester! First, we will speak about files in Linux, which we will discuss further in Topics 4, 5, 6, and 12.

Linux treats every entity as a file, even if it doesn't look like one (directories, disks, attached devices, etc.) To access the content of a file, you should open it for reading or writing.

Every file has a name (= **filename**). In addition, every file is associated with a unique ID number called an **inode number** (also called: **i-number**). This number distinguishes a file from any other files stored on the Linux device.

Every file that is open within a C program is also associated with a unique integer called the **file descriptor** (also: **`fd`**). This number distinguishes an open file from other open files by the same program. C's `open()` syscall would tell us what the `fd` of a file is:

```c
int fd = open("data.txt", ...);
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem

Regular files (non-special files, e.g., text, source code, PDFs, HTML, executables, and even directories) are organized linearly: as sequences of bytes. Such files don't have any special structure in them, not even 'records' or 'rows'.

We can write into or read from a file not only at its top but at any position (= location inside the file = **offset**) inside it.

If we attempt to write content at a position beyond the end of the file (= **`EOF`**), the file will be filled with null characters between the existing content and the new content we write. For example, writing "`World`" at position 10 inside a file containing "`Hello`" will result in a file containing: "`Hello\0\0\0\0\0World`".

An attempt to write content before the 0th position (0 is the smallest position of a file, standing for the 'top' of the file) will result in an error. That is, using negative positions isn't allowed.

Writing to the middle of the file will overwrite what's currently there. For example, writing "`wow`" at position 1 of a file containing "`Hello`" will result in "`Hwowo`".

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem

You can open the same file more than once simultaneously (e.g., once for reading and once for writing.) In this case, every instance of the open file will have its own `fd`.

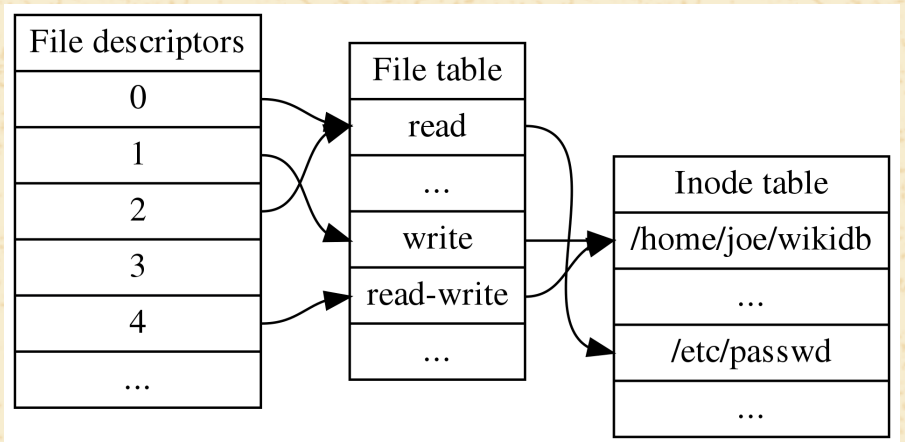
We already defined inode numbers. But what are inodes?

An **inode** (= information node) is a data structure used by the Linux operating system to store **metadata** about files. It is implemented by the Linux OS and is stored somewhere on the disk. Examples of such metadata include the i-number, timestamp of creation, timestamp of modification, owner's username, file type (regular, directory, device, etc.,), length of the file in bytes, location on the disk of the file's content, etc.

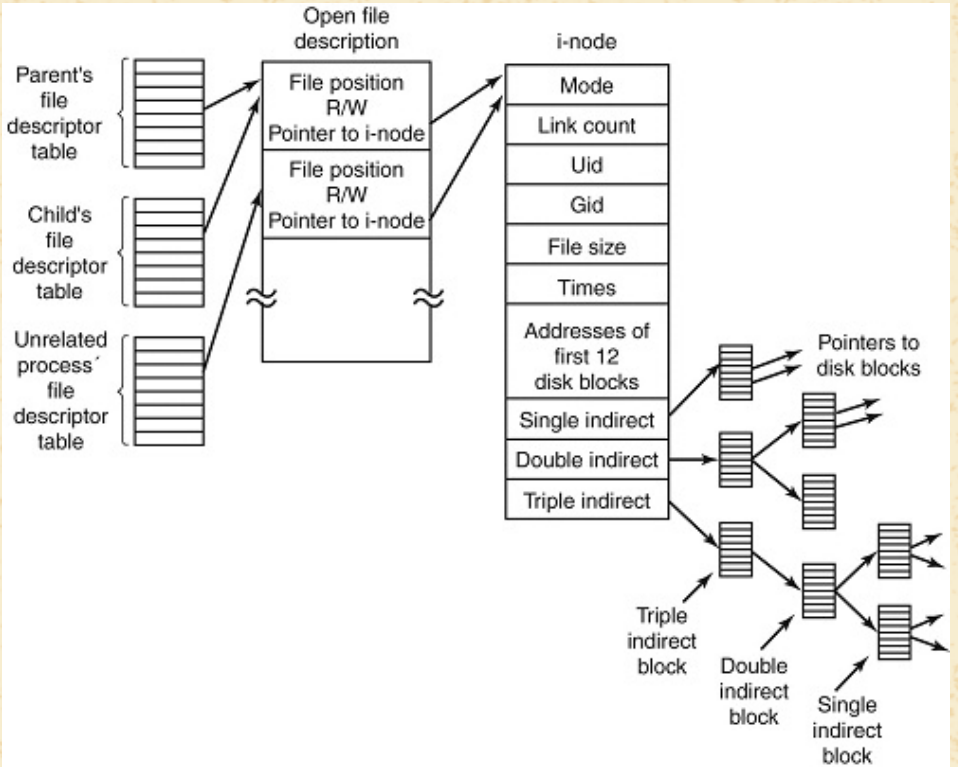
Surprisingly, the filename isn't stored in the inode: we will learn where it is stored in the next slides. Contents of files aren't stored in inodes, either; they are stored at different places on the disk, many times in a non-contiguous way (the content of a file is chopped into parts, and each part is stored at a different place on the disk.) The inode only keeps a list of addresses on the disk where each chunk of the content is stored.

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem



Connection between a program's file descriptors table and inode tables. Taken from Qwertyus, CC BY-SA 4.0, via Wikimedia Commons.



Structure of the inode table. Taken from geeksforgeeks.org.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem

As users, we open files by referring to their filenames.

Since there might be more than 1 file with the same name on the computer (e.g., two files called `hello.txt` but located in different folders,) the Linux operating system needs to understand which file to open by relating it to its inode number.

A **directory** (folder) is a map between filenames and inode numbers. To find the inode associated with a file, the OS accesses a directory and finds which inode is related to the given filename. This means that the directory file is the place where the filename is stored! A directory file is stored on the disk just as other files are.

Nested directories make up a **path** (each part of a path is called a **dentry** = **d**irectory **entry**.) An example of a path: `/public_html/hello.txt`.

A "fully qualified" path starts at `/` (root directory) and is called an **absolute path**. A path that starts with any other directory is called a **relative path**.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem: Links

We can create links to files! A file can have several links, but the inode would be the same!

This kind of link is called a **hard link**. To create one, you would type:

```
$ ln file1 hardlinkForFile1
```

This will create a link named `hardlinkForFile1` for the file `file1`. You can confirm that both `file1` and `hardlinkForFile1` are mapped to the same inode number by listing the i-numbers of all the files in the current folder:

```
$ ls -i
```

Both `file1` and `hardlinkForFile1` are hard links to the same file. As long as there exists at least one hard link to a file, the OS won't delete the content of the file from the disk. Once the last hard link is deleted, the file and its inode data structure will be deleted from the disk. The number of existing hard links to a file is called the **link count**.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem: Links

To create a link that can access files located on other systems/devices, we use **symbolic links** (also called **symlinks**, **soft links**, and **junctions**). You can create one by typing:

```
$ ln -s file1 symlinkForFile1
```

A symlink's data contains the path to the file, which could be across filesystems, or nonexistent (resulting in a **broken link**).

A symbolic link includes an inode of its own! To access the content of a file via a symlink, the OS first opens the inode to extract the path and then opens the inode of the target file to find where the content is stored on disk. It's a 2-step process!

To delete either a hard or soft link, simply use the `rm` command:

```
$ rm linkForFile1
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Files and the Filesystem: Special Files

Besides regular and directory files, here are special file types recognized by Linux:

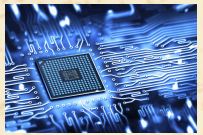
1. **Block device files** represent devices that are connected to the computer whose data can be accessed in chunks (**blocks**) of multiple bytes at a time. A block is usually a power-of-two multiple of the sector size (512 bytes), e.g., 4096 and 8192. Examples: hard drives (e.g., `/dev/sda`), solid-state drives, and USB flash drives. Each such device is implemented as an array of bytes: you can access the disk at any position (= supports **random access**).
2. **Character device files** represent devices that are connected to the computer whose data can be accessed one byte at a time. Examples: keyboard, mouse, terminal (e.g., `/dev/tty`), speakers, etc. Each such device is implemented as a queue of bytes: you can access only the first byte at the front of the queue (= supports **sequential access**).
3. **Named pipes (FIFOs)** are a data structure that allows two programs to communicate with each other (= **inter-process communication** = **IPC**). There are two types of pipes: **regular pipes** are implemented as memory variables, and **named pipes** are implemented as files on the disk; one program writes data into the pipe (file), and another program reads that data from the pipe.
4. **Sockets** are another IPC technique implemented as files. Unlike pipes, sockets allow programs running on different machines/devices to communicate with one another through a network to which both machines are connected.

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Processes



By Jamesmathew1840139, CC BY-SA 4.0, via Wikimedia Commons

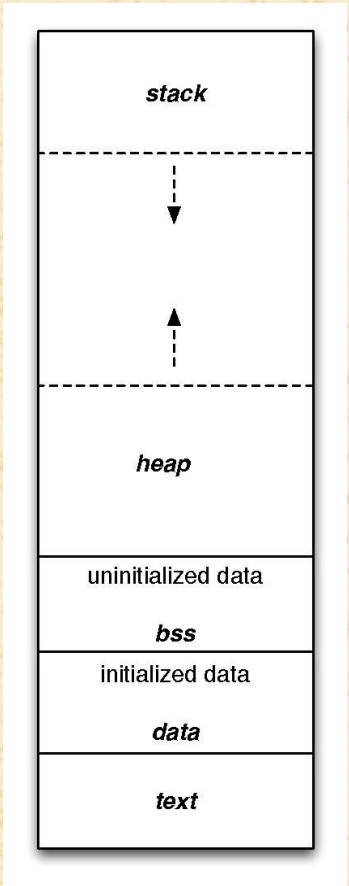
Processes are programs whose machine code was copied to main memory (RAM) to run these programs. In other words, processes are running programs. Each process's memory chunk consists of the following sections:

- The **text** section, in which the binary code of the program (the process's instructions) and read-only data (constants) are stored. This section is fixed in size (depending on how long the binary code is.)
- The **data** section, in which global and static variables that were initialized are stored. It has a fixed size, too. (***How do we know that it's fixed in size?***)
- The **bss** (= **b**lock **s**tarted by **s**ymbol) section, in which global and static variables that *weren't yet* initialized are stored. It has a fixed size, too.
- The **stack** section, in which local variables, such as those defined inside functions, are stored, and
- The **heap** section, in which variables that were dynamically allocated are stored. Example: when you use the `new` operator in Java or the `malloc()` function in C, your variable (or object) is stored on the heap.

The Linux kernel is responsible for allocating memory for processes, deciding in which order processes run, and ensuring their proper execution. We will cover more on processes in Topics 7 and 9.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Processes



Memory sections of a process. Taken from Dougct, CC BY-SA 3.0, via Wikipedia.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Threads and Process Hierarchy

Taken from Ecotextile News

Each process consists of one or more **threads** (= units of execution.) Threads are covered in Topic 8.

A thread has its own stack, processor state, and instruction pointer, but the code (instructions) of a process, the address space, and the heap are shared by all threads.

Back to processes: each process is identified by a unique positive integer - process ID (`pid`).

`init` is the grandparent of all the processes: it has the `pid` of 1.

Every process, including `init`, can call the `fork()` syscall to create a child process. Useful processes that run in the background, e.g., music playing in the background, are called **daemons**. Linux runs a lot of daemons at any moment!

If a parent process terminates before its child, `init` will become its parent. Parent processes usually wait on terminating children, which allows the child processes to be destroyed. If a process that is about to terminate isn't waited upon by its parent, it is called a **zombie**. `init` will eventually wait for all zombies and terminate them.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Users and Groups

Taken from The New York Times

Each Linux user has a unique positive integer called **user ID** (`uid`) in addition to the username. The `root` user has the `uid` of 0.

Each process is associated with one `uid` (the user who starts running the program.) All the child processes inherit the `uids` of their parent processes.

Usernames and their `uids` are stored in the file `/etc/passwd`. You can find your user information (e.g., username, `uid`, home folder, default shell [e.g., `bash`, `sh`, or `ksh`]) by typing the following Linux command:

```
$ getent passwd "$( id -u )"
```

Each user belongs to a login group with a unique **group ID** (`gid`). This will be the user's primary group.

The user may also belong to a few supplemental groups, listed in `/etc/group`.

We will speak more about Users and Groups in Topic 7.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Permissions

Image by Rbac via Online Web Fonts

Each file is associated with an owner user, owning group, and 3 sets of 3 permission bits (in the form: `rwxrwxrwx`,) all of which are stored in the file's inode.

(r)ead, (w)rite, and e(x)ecute; "`-`" instead of a letter is a permission that doesn't apply (e.g., the owner isn't allowed to execute the file.) The 1st set of 3 letters represents the permissions of the owner; the 2nd set represents the permissions of the group members; and the 3rd set represents the permissions of all the other users.

Fun follow-up question: What permissions does the string `rwxrw-rw-` represent?

In the context of directories, reading means listing the directory's contents, writing means being able to add or remove links inside the directory, and executing means being able to use the directory in paths when typing Linux commands, e.g., `ls thisDir`.

Linux also supports access control lists (ACLs) that provide greater and more detailed permissions control. We will learn more about permissions in Topic 12.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Signals

Taken from Maslosoft.com

Signals are a mechanism for one-way asynchronous inter-process communication. The Linux kernel can send a signal to a process in case, e.g., of an error, and processes can send signals to one another and even to themselves. Typically, signals alert a process about some event (e.g., segmentation fault, Ctrl+C pressed, etc).

The Linux kernel has about 30 signal types. The exact number and the types of signals depend on every Linux device architecture.

Processes can control what happens when a signal is received. They can choose to do one of the following: (1) accept the default action (usually terminate/core-dump the process), (2) ignore it (silently drop it), or (3) handle it with a predetermined action (signal handler), then resume interrupted action.

Two exceptions of processes that can't be controlled: `SIGKILL` (always terminates processes) and `SIGSTOP` (always stops processes).

Signals are covered in Topic 10.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Error Handling

Taken from Boston Web Marketing

A function return value tells whether an error occurred during its execution.

- A function in C returns `0` if no errors happened (the execution was successful), and non-zero values in case of an error.

A special variable named **`errno`**, which is defined in the C library `<errno.h>` describes what kind of error happened.

```
extern int errno; // This is how errno is defined for us in <errno.h>.
```

After calling a function that changes `errno`, we can check the content of `errno` to find if anything abnormal happened during the function's execution. The check must be done *right after* the function call in our code (since, otherwise, we may instead see error codes that other functions after our function call have set.) Each `errno` value corresponds to a different type of error. Example: When `errno = 1`, this stands for the `EACCESS` = "Permission denied" error.

The C library provides functions translating `errno` to its description: `perror()`, `strerror()`, and `strerror_r()`.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Error Handling

The C program errno.c demonstrates the uses of the three functions `perror()`, `strerror()`, and `strerror_r()`. This and the following slide show their syntax and usage examples:

```c
#include <stdio.h>
void perror (const char *str); // perror() is already defined in <stdio.h>

if (close (fd) == -1) // -1 means that an error occurred during the execution of close()
    perror ("close"); // Prints the error message (e.g., "close: Permission denied")

/* representation of the error described by errno, prefixed by str,
   followed by a colon. It's common to include the name of the function
   in str, so it reports which function failed (here it is "close".) */
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Error Handling


```c
#include <stdio.h> // Needed for calling printf()
#include <string.h>
char* strerror (int errnum); // Defined in <string.h>
int strerror_r (int errnum, char *buf,  size_t len); // Defined in <string.h> as well

/* After calling some function X that may result in an error: */

char * error_string = strerror (errno);
printf ("The error message that function X returned is: %s.\n", error_string);

char buf[1024];
int strerror_r_returned_value = strerror_r (errno, buf, sizeof(buf));
printf ("The error message that function X returned is: %s.\n", buf);
printf ("strerror_r() itself returned with code %d.\n", strerror_r_returned_value);
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Error Handling

Some functions may return 0 and 1 as part of their calculations (e.g., a function that adds numbers.) If so, the returned value can't be used to indicate the presence of an error. In this case, we must check the value of `errno`.

Since `errno` might carry the error number of some previously called function, we should always set `errno` to 0 before each function call:

```
errno = 0;
unsigned long int arg = strtoul (buf, NULL, 0); // Translates error message in 'buf'
                                                // back to the respective error number.
if (errno) // Check if the call to strtoul() experienced an error.
    perror ("strtoul");
```

If we hadn't set `errno` to 0, and `strtoul()` would run successfully, we could've mistakenly concluded that `strtoul()` failed if `errno` contained a number other than 0.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Error Handling

Notice that `errno` can be set by any library function or syscall. What is the problem in the following `if (errno == EIO)` statement?

```c
if (fsync (fd) == -1)
{
    fprintf (stderr, "fsync failed!\n"); // Hint: fprintf() can also change errno.
    if (errno == EIO) // So, what could go wrong in asking if errno == EIO at this point?
        fprintf (stderr, "I/O error on %d!\n", fd);
}
```

In single-threaded programs, `errno` is a global variable. In multithreaded programs, `errno` is stored per thread, and thus is thread-safe (won't be overwritten/corrupted by other threads.)

In a multi-threaded program, use `strerror_r()` instead of `strerror()` because out of both, only `strerror_r()` is thread-safe.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Error Handling

The code of the C program errno.c is given below (scroll to view the full code:)

```
/* A C program that shows how to detect and print
 *    error messages in various ways.
 *
 * We will attempt to issue a system call such
 *    that it fails, intentionally, to test the
 *    methods of C to detect errors, including
 *    (1) checking the returned value of a system
 *    call, and (2) checking the value of the
 *    'errno' variable that C uses to indicate
 *    what specific error happened.
 *
 *    Miriam Briskman, 2/8/2023
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Error Handling

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Intro to the Linux/UNIX OS Programming: Sources

Topic 1: Intro to the Linux/UNIX OS Programming. Briskman, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+1:+intro+to+the+linux/unix+os+programming.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.