**What is Cross-compiler?**

A cross-compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. A cross-compiler is necessary to compile multiple platform codes from the developer's host machine.

**Why use a cross-compiler?**

* Speed: The target platform is often much slower than the host in terms of computing power. Most embedded hardware is primarily designed to be low-cost and energy-efficient, not high-performance. Modern emulators are much faster than actual hardware because they are virtually simulated on high-powered desktop hardware.
* Capability: Compiling is resource-intensive, and embedded hardware may not have the capability to self-compile.
* Availability: Bringing Linux to a hardware platform it has never run on before requires a cross-compiler. Finding a fully-featured prebuilt native environment is difficult, even on long-established platforms such as ARM or MIPS. However, setting up a host machine to build a new package for your target machine is easy.
* Flexibility: The full set of Linux files is very large, and some packages may not be used on the target machine. Providing a large system with all packages is not a good idea for a target machine with limited resources. A cross-compiler helps to only include necessary files for a small customized system.
* Convenience: User interfaces on headless boxes are often difficult to use. On a powerful host machine, you can easily edit, test, and do many other things.

**What is Toolchain?**

A toolchain is a set of programming tools used to perform a complex software development task or to create a software product, typically a computer program or a set of related programs.

**What does it include?**

A simple toolchain for software development may include a compiler and linker (to turn source code into an executable program), libraries (to provide interfaces with the operating system), and a debugger (to test and debug the created programs).

Components included in the GNU toolchain are:

* GNU make: an automation tool for compilation and build
* GNU Compiler Collection (GCC): a suite of compilers for several programming languages
* GNU C Library (glibc): core C library including headers, libraries, and dynamic loader
* GNU Binutils: a suite of tools including linker, assembler and other tools
* GNU Bison: a parser generator, often used with the Flex lexical analyzer
* GNU m4: an m4 macro processor
* GNU Debugger (GDB): a code debugging tool
* GNU Autotools (GNU Build System): Autoconf, Automake and Libtool

**COMPILING PROCESS in C**

**How many steps?**

Ảnh có chứa biểu đồ, văn bản, ảnh chụp màn hình, hàng

Mô tả được tạo tự động

The steps for a program to be compiled into executable code:

1. **Preprocessing**:

The tasks that will be performed by the preprocessor :

* Expanding included files
* Replacing macros
* Removing disabled code and comments

At this stage, the preprocessor will create a .i file through the command prompt:

gcc -E file\_name.c -o file\_name.i

The content of the “.i” file will be the content of the c file after going through preprocessor directives such as #define, #include, #ifdef, #endif, etc. Preprocessor directives are instructions for the preprocessor to replace text, insert files, conditionally compile, etc. before the compiler compiles the source code.

1. **Compilation**

The compilation step is performed from the output of the preprocessor. The compiler will compile from high-level code language to assembly code.

At this stage, the Compiler will create a .s file through the command prompt:

gcc -S file\_name.c -o file\_name.s

The content of file\_name.s will be the result of compiling from file .c to assembly code.

1. **Assembly**

The assembly code will be converted to machine language by the assembler. This file contains machine-level instructions.

At this stage, the assembler will create a .o file through the command prompt:

gcc -c file\_name.c -o file\_name.o

The content of file\_name.o will only be machine code and unresolved function calls.

1. **Linking**

The linker is the final step in creating output for the compiled object. It will link the files by replacing undefined symbols with correct addresses. Each symbol can be defined in other object files or in libraries.

gcc source.o mylib.o

The result can be a shared (or dynamic) library or an executable code.

**What is Objdump?**

We basically cannot read the .o file in normal text editors because the content inside is binary code. We can use the ObjDump tool to check.

The basic syntax of Objdump is:

objdump [options] objfile...

options :

< -f > Display the contents of the overall file header

< -p > Display object format-specific file header contents

< -t > Display the contents of the symbol table (or tables)

< -D> Display assembler contents of all sections

**STATIC VÀ DYNAMIC LIBRARY**

**The difference between static and dynamic libraries?**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, biểu đồ

Mô tả được tạo tự động

**Compare:**

**Resource use :**

Static libraries result in larger executable files because they contain more code. The additional code from the library cannot be shared with other programs in the system, increasing the use of the file system and memory at runtime. Multiple processes running the same program linked statically will still share the code.

Static apps require fewer runtime relocations (time spent moving code from one memory location to another before running), reducing startup time and requiring less private resident set size memory. Code generated from static linking may be more efficient than dynamic linking because it does not have to search for addresses when calling functions.

**Security :**

Dynamically linked libraries provide ABI (application binary interface) compatibility that can be updated without changing the executable files that depend on those libraries.

In addition, security measures such as load address randomization cannot be used with statically linked executable files. This reduces the security of the application.

**Compatibility :**

Static linking was introduced to provide executable files that are not dependent on the version of libraries provided by the OS. However, most libraries depend on other libraries. With static linking, this dependency becomes inflexible and as a result, both backward and forward compatibility is lost. Static linking is only guaranteed to work on the system where the executable file was built.

**Create Static Library**

1. Create the object file for the static library

gcc -c lib/foo.c -o lib/static/libfoo.o

1. Creator static library

ar rcs lib/static/libfoo.a lib/static/libfoo.o

|  |  |
| --- | --- |
| **Local linking**  **Conpile:**  gcc app.c -Ilib -Llib/static -lfoo -o app\_static\_local  **Run**  ./ app\_static\_local | **Global linking**  **Install:**  sudo install -m 755 lib/foo.h /usr/include  sudo install -m 755 lib/static/libfoo.a /usr/lib/  **Compile:**  gcc app.c -lfoo -o app\_static  **Run**  ./ app\_static  **Remove library:**  sudo rm /usr/lib/libfoo.a/usr/include/foo.h |

Create Dynamic library

1. Create the object file for the dynamic library

gcc -c -fPIC lib/foo.c -o lib/dynamic/libfoo.o

1. Creator dynamic library

gcc -shared lib/dynamic/libfoo.o -o lib/dynamic/libfoo.so

|  |  |
| --- | --- |
| **Local linking**  **Compile :**  gcc app.c -Ilib -Llib/dynamic -lfoo -o app\_dynamic\_local  **Run:**  LD\_LIBRARY\_PATH=lib/dynamic ./app\_dynamic\_local | **Global linking**  Install :  sudo install -m 755 lib/foo.h/usr/include  sudo install -m 755 lib/dynamic/libfoo.so /usr/lib/  **Compile :**  gcc app.c -lfoo -o app\_dynamic  **Run :**  ./app\_dynamic  **Remove library :**  sudo rm /usr/lib/libfoo.so/usr/include/foo.h |

**Makefile**

**What is Makefile?**

A Makefile is a text file that contains shell commands that instruct how to compile and link (or build) a set of source code files. It helps to make this work faster.

**What is the Target?**

A target is a file or an action that you want the makefile to perform. Each target can have dependencies, which are files or other targets that need to be built before building the current target. Each target can also have commands, which are shell commands to create or update the target

**What is Marco?**

The Make program allows you to use macros, which are used similarly to variables. Macros are defined in a makefile using the = symbol.

Using macros helps to avoid repeating text and makes the makefile easier to modify.

**What are the Rules?**

Rules are the main components of a Makefile. They tell Make how to create or update certain files, called targets, from other files, called prerequisites, using shell commands.

A rule has the following general form:

target [target …] : [prerequisites …]

[command … ]

Rules have two forms to make it easier to write for multiple files:

Implicit Rules: You don’t need to write commands because implicit rules automatically run according to a set of implicit rules. To change the command, you only need to change the FLAG value in the rule set when using it. Flags are used according to a predefined rule table.

Static Pattern Rules: You can write everything shorter when writing multiple targets

Target …: target-pattern: prereq-patters

command

**What are automatic variables?**

When using implicit rules and pattern rules, the file name is not clear. Automatic variables support getting the file name from the target and prerequisites.

$@ Target name

$? All prerequisites newer than the target

$^ All prerequisites

$< The name of the first prerequisite. If the target got its recipe from an implicit rule, this will be the first prerequisite added by the implicit rule

$| The names of all the order-only prerequisites, with spaces between them.

**What is .PHONY?**

Adding .PHONY to a target will prevent Make from confusing the target with a file name.

**Shell Script**

**What is the Shell?**

A shell is a command-line interpreter of the operating system, instructing the operating system to perform any necessary tasks and commands. Shell commands can be in scripts to perform multiple tasks and complex work.

**What is Prompt?**

Commands are entered on the terminal, interacting with the system through direct commands on the terminal, after the $ sign.

**Shell types:**

* Bourne Shell (sh): a historically first shell
* C Shell (csh): a C-style shell
* Korn Shell (ksh): a mix of Bourne Shell and C Shell
* Bourne Again Shell (bash): a GNU version of Korn Shell
* Debian Almquist shell (dash): a smaller Bash shell
* Tenex-extended C Shell (tcsh): an Advanced C Shell
* Z shell (zsh): the most advanced shell

**Git – Gerrit**

**What is Git?**

Git is a Distributed Version Control System (DVCS), providing each programmer with their own repository containing the entire history of changes. **Ảnh có chứa văn bản, biểu đồ, ảnh chụp màn hình, Hình chữ nhật

Mô tả được tạo tự động**

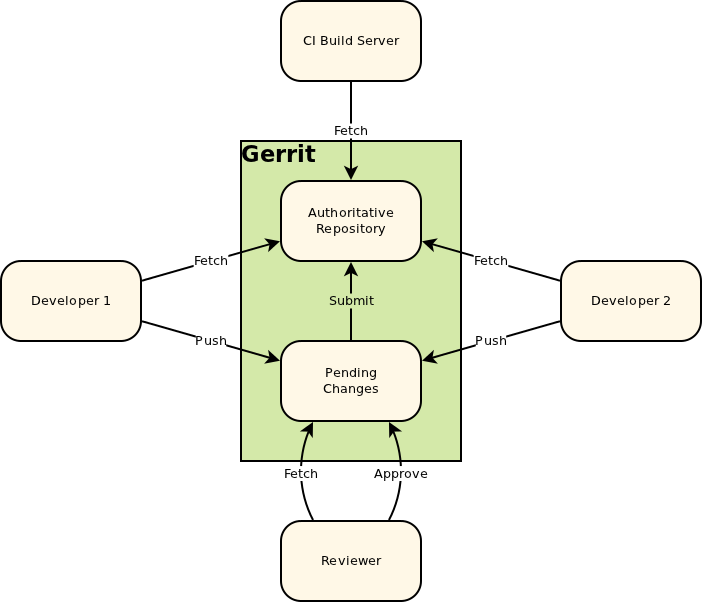
**Git basic command**

* git init: create a repo in local
* git clone <link of repo> : download source code from remote repo
* git branch <branch-name>: create a new branch
* git checkout <branch-name>: switch to a branch, which you need to work
* git status: display all information about the current branch
* git add: include the changes of the repo into the next commit
* git commit: save your changes in local
* git push <remote> <branch-name>: upload your commits to the remote repository
* git pull <remote>: get updates from the remote repo
* (= git fetch + git merge)
* git log: display all the commit history
* git revert <commit id>: undo the commit, but will create a new commit without deleting.
* git merge <branch name>: integrate your branch with master branch
* git reset --hard <commit id>: delete all commit until commit id
* git tag <commit Id>: make tag for the specified commit
* git push –set-upstream <remote> <name of branch>: if your branch is newly created, not exist in remote
* git rebase <branch> is a command that allows you to take all the changes that were committed on one branch and replay them on a different branch

**What is Gerrit?**

Gerrit is a web-based tool that supports source code management and review, using git as the version control system.

Developers will fetch the source from the Authoritative Repository. Whenever they complete a task, instead of pushing back to this repository, they must push to a Pending Changes repository. This ensures that all commits must be approved by a Reviewer (pushing back to the Authoritative Repository only when there is special permission). Once the change has been approved, the reviewer can submit to merge the change into the Authoritative Repository.



**U-boot (Das U-Boot)** is an open-source bootloader widely used in small embedded systems. It supports architectures including 68k, ARM, Blackfin, Microblaze, PPC, and x86.

Its main function is to initialize hardware and load other components of the OS (Linux kernel, root file system, device tree) into IRAM and hand over control to the Linux kernel.

1. **SoC ROM Bootloader**

When the system boots, control belongs to the vector, an assembly code pre-written by the manufacturer. The vector then points to the memory address containing the first boot code, specifically the BootRom. The main function of the BootRom is to copy the context in the “MLO” file, also known as the Second Program Loader (SPL), into IRAM and execute it.

The Bootrom’s memory is small, so Rom code is also limited in initializing some hardware necessary for loading SPL onto the system, such as MMC/eMMC, SDcard, and NAND flash. These hardware are collectively called boot devices. Rom code selects the boot device (load from memory card, flash, etc.) depending on how pins are configured via switches/jumps on the hardware.

1. **Second Program loader (SPL)**

The SPL (Second Program Loader) has the main task of continuing to set up necessary components such as the DRAM controller, eMMC, etc. It then loads U-boot to the CONFIG\_SYS\_TEXT\_BASE address of RAM. The main function of the SPL is to load U-boot into RAM.

1. **U-boot**

After loading into RAM, U-boot will perform relocation. It moves to the Relocaddr address of RAM (usually the last address of RAM) and jumps to the relocated U-boot code. U-boot will check if the uEnv.txt file exists.

If it does, it will load it into RAM. uEnv.txt is a boot script that defines configuration parameters and kernel parameters. These parameters are defaulted in U-boot and can be modified through the uEnv.txt file. Next, U-boot will continue to load the kernel and device tree into RAM at the pre-configured address in the U-boot source code or in the uEnv.txt file.

Then it passes all kernel parameters and hands over execution to the kernel.

1. **Kernel**

The core of any operating system, controlling everything in the system, the kernel will mount the file specified in grub.conf.

Then the kernel will run the /sbin/init program (the first program to run, with PID 1).

The kernel will set up a root file system using the Initial RAM Disk (initrd) until kernel mounts the root file system.

1. **Init**

The system executes the runlevel program.

At one point, it will look for the init file, usually in /etc/inittab to determine the Linux run level.

**User-Space, Kernel-space, and System call**

Modules run in kernel space, while applications run in user space. Processes running in user space cannot access kernel space. User space processes can only access a small part of the kernel through an interface provided by the kernel - the system calls.

Unix systems are designed to take advantage of this hardware feature, using these two levels. All current processors have at least two levels of protection and some, such as the x86 line, have more levels; when there are multiple levels, the highest and lowest levels are used. In Unix, the kernel executes at the highest level (also called supervisor mode), where everything is allowed, while applications execute at the lowest level (called user mode), where the processor restricts direct access to hardware and unauthorized memory access.

We commonly refer to these execution modes as kernel space and user space.

**System calls**

A system call is a way for a program to request a service from the kernel. The system call interface includes several functions that the OS exports to applications running on it.

These functions allow operations such as opening file systems, creating network connections, and reading and writing from files.

System calls are divided into 5 main parts :

* **Process Control**
* **File management**
* **Device management**
* **Information Maintenance**
* **Communication**

1. **Process control**

System calls perform tasks such as process creation, process termination, etc.

**Fork() :**

A new process is created by the fork() system call. A new process can be created with fork() without running a new program, the child process simply continues to execute exactly like the first running process (the parent program).

**Exit() :**

The system call is used to terminate the execution of a process. The operating system will reclaim the resources being used by the process after the exit() system call.

**Exec()** :

A new program will start after calling the exec() system call.

Running a new program does not require that a new process be created first, any process can call exec() at any time. The currently running program is immediately terminated and the new program begins execution with the contents of the current process.

Using this function, creating a child process to run does not necessarily have to be the same as the parent process. The exec system calls allow a process to run any program file, including binary executable files or shell scripts.

1. **File Management**

File Management system calls handle tasks such as reading and writing data files.

Open() is the system call to open a file.

Read() is the system call to open the file in reading mode, we cannot edit the file with this system call. Multiple processes can execute the read() system call simultaneously on the same file.

Write() can edit the file, but multiple processes cannot execute the write() system call on the same file. Close() closes the file.

1. **Communication**

There are types of system calls used specifically for inter-process communication (IPC).

Two models are used for inter-process communication:

* message passing (where processes exchange messages with another process)
* shared memory (where processes share a memory region to communicate).

**Pipe()** is a system call used for communication between two different Linux processes. Its main purpose is for inter-process communication. The pipe() system call function is used to open file descriptors.

**Shmget()** stands for a shared memory segment. Its purpose is for inter-process communication. This system call is used to access shared memory and access messages in order to communicate with the process.

**Mmap()** is a function call used to map or unmap files or devices into memory. The mmap() system call creates a new mapping in the virtual address space of the calling process.

1. **Device Management**

Device management handles device operations such as reading from device buffers, writing to device buffers, etc. The Linux system call for this is ioctl().

**Ioctl()** stands for Input and Output Control. It is a system call for device-specific input/output operations and other operations that cannot be expected from regular system calls.

1. **Information Maintenance**

System calls are used to process and transmit information between the OS and user programs. Additionally, the OS keeps information about all its processes, and system calls are used to access that information. The system calls for this are **getpid(), alarm(),** and **sleep().**

**Getpid()** stands for “get the process ID”. This function returns the process ID of the calling process. The function always succeeds and does not return a value reserved to indicate an error.

**Alarm()** sets an alarm clock to send a signal when it is set enough. It arranges for a signal to be sent to the calling process.

**Sleep()** suspends the execution of the current process for a specified period. During that time, another process has the opportunity to execute.

**What are threads?**

A thread is the smallest unit of execution that can be managed by the operating system. A process is a group of related threads that execute together in the same environment and share resources with each other. This means that threads within the same process share the same memory space and can communicate directly with each other and make full use of multiple cores.

**Pthread\_join()**: waits for the thread specified by the thread to terminate, suspending the executing thread until the joined thread terminates. If the thread has already terminated, thread\_join returns immediately. This action is called joining.

**Pthread\_detach():** Basically, a thread is joinable, meaning that when it terminates, another thread can retrieve its return value by using pthread\_join(). Sometimes we don't care about the thread's return. We simply want the system to automatically clean up and remove the thread when it exits. In this case, we can mark the thread as detached.

**What is Thread synchronization?**

We start running two or more threads in a program, there may be a situation where multiple threads try to access the same resource, resulting in unpredictable results. Therefore, it is necessary to synchronize the actions of multiple threads to ensure that only one thread can access the resource at a time.

**Methods of thread synchronization include :**

**Semaphore :** which grants permission for several threads to execute a portion of code. Semaphore grants multiple keys to threads, at any given time, threads with keys can have multiple threads using a resource simultaneously.

**Mutex :** creates a single key, so at any given time, only one thread has the key to use the resource that needs synchronization.

**The race condition :** is when the result of a calculation is affected if more than two threads execute the same code or use the same computational resource.

**Deadlock** is when two or more threads enter an infinite loop waiting for some resource.

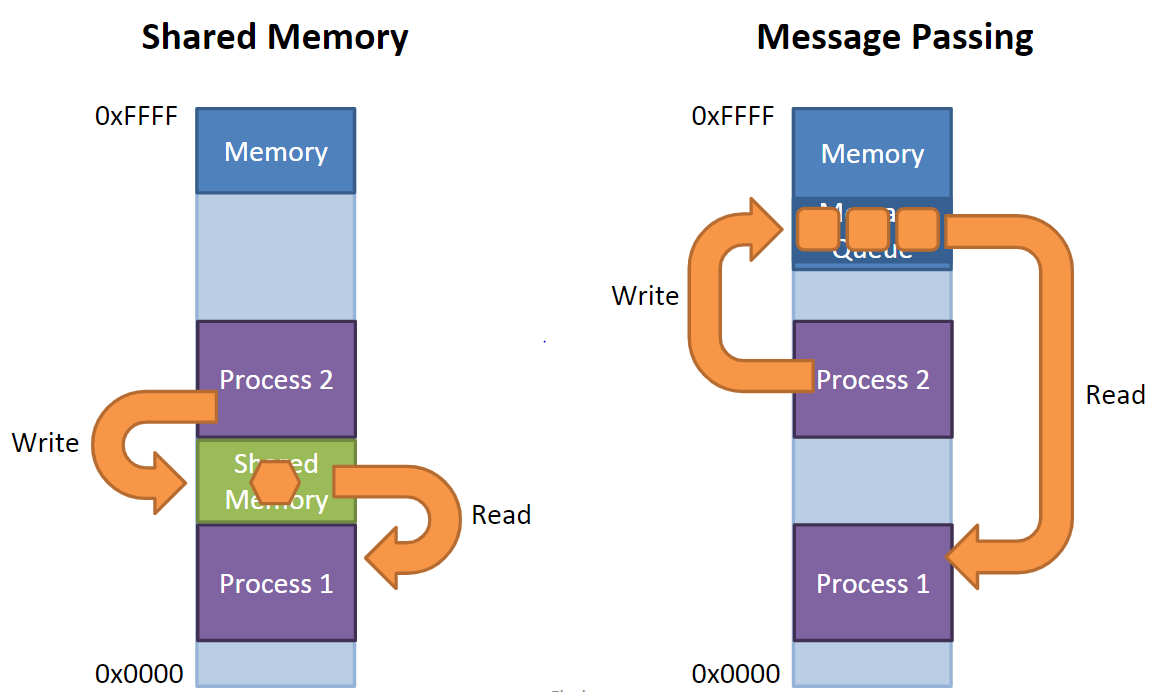
**What is the process?**

A process is a program that executes. The process is created when a command is executed, so it can be called a running instance of a program. Any command you execute starts with a process.

If a parent exits before its children, the children become orphans

If a child exits before the parent calls wait() the child becomes a zombie

**How to share data between processes?**

****

Data can be shared between processes through a portion of memory that multiple processes can read or write, message passing (pipes, sockets, signals), shared memorry.

**System V API :**

* msgget() creates a new message queue
* msgsnd() pushes a message onto the queue
* msgrcv() pops in the queue

**POSIX Message Queue :** optimized and easy to use with priority levels associated with messages and messages always strictly queued, recipients receive in priority order

**POSIX API:**

Mq\_open() ,Mq\_send() ,Mq\_reveive() , Mq\_close(), Mq\_unlink()

**Share memory :**

**Shmget()** stands for a shared memory segment.

Its purpose is for inter-process communication. This system call is used to access shared memory and access messages in order to communicate with the process.

**Mmap()** is a function call used to map or unmap files or devices into memory.

The mmap() system call creates a new mapping in the virtual address space of the calling process.