**Cross-compilation, Toolchains**

**What is Cross-compiler?**

Là một compiler có khả năng tạo ra executable code cho một platform khác với nền tảng biên dịch đang chạy. Một cross-compiler cần thiết cho việc compile code cho multiphe platform từ máy chủ nhà phát triển.

**Why use a cross-compiler?**

* **Speed** : Tagret platform luôn chậm hợn nhiều so với host theo cường độ làm việc. Hầu hết các phần cứng embedded có mục đích lớn nhất là chi phí thấp, tiết kiệm năg lượng, not high performance. Modern emulators thật sự rất nhanh hơn nhiều so với phần cứng thực vì nó được mô phỏng ảo trên high powered desktop hardware
* **Capability** : Compiling rất tốn tài nguyên , Hard ware embedded không đủ khả năng điể có thể tự compile.
* **Availability** : Mang Linux lên một hardware platform mà nó chưa bao giờ chạy trước đây yêu cầu một cross-compiler. Tìm kiếm một prebuilt native environment đầy đủ tính năng rất khó , ngay cả trên các long-established platforms như là ARM or MIPS. Tuy nhiên, chúng ta có thể dễ dàng setup một host machine để build một new package cho your target machine.
* **Flexibility** : đầu đủ các tệp của linux rất lớn, phần nó các packages không đc sử dụng trên target machine. Cung cấp 1 system lớn với đầy đủ các packages không phải một ý kiến tốt trên target machine với tài nguyên giới hạn. Cross-compiler giúp chỉ đưa những file cần thiết với một small customized system.
* **Convenience** : Giao diện người dùng trên headless boxes thường khá khó dùng. Ơ trên một powerful host machine, bạn có thể dễ dàng edit, test và làm được nhiều việc khác.

**What is Toolchain?**

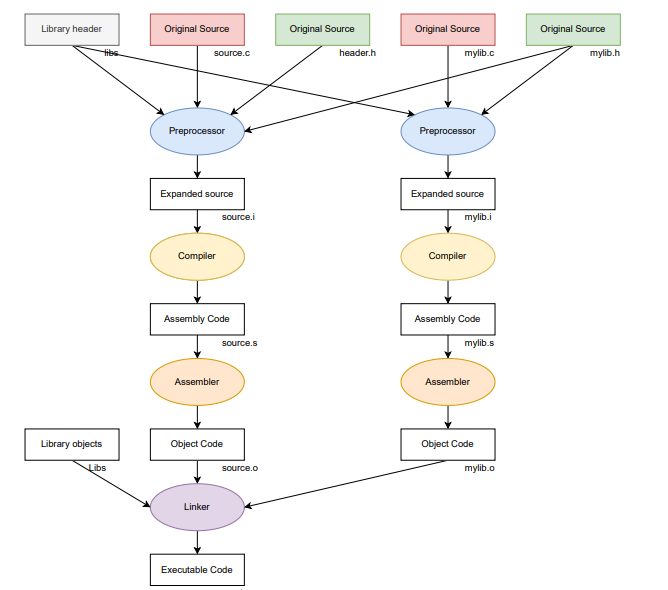
Toolchain là một tập hợp các công cụ lập trình được sử dụng để thực hiện một nhiệm vụ phát triển phần mềm phức tạp hoặc để tạo ra một sản phẩm phần mềm, thường là một chương trình máy tính hoặc một tập hợp các chương trình liên quan.

**What does it include?**

Một toolchain đơn giản để development software có thể bao gồm trình biên dịch và trình liên kết (để biến mã nguồn thành chương trình có thể thực thi), các thư viện (cung cấp giao diện với hệ điều hành) và trình gỡ lỗi (để kiểm tra và gỡ lỗi các chương trình đã tạo).

**COMPILING PROCESS in C**

**How many steps?**



Các bước để một program biên dịch thành executable code:

1. **Preprocesing** :

Các công việc sẽ được thực hiện bởi preprocessor :

* Mở rộng các included files
* Thay thế các marcro
* Xóa bỏ các mã bị vô hiệu và comments

Ở bước này preprocessor sẽ tạo ra file .i qua command prompt:

gcc –E file\_name.c –o file\_name.i

Nội dung của file .i sẽ là nội dung của file c sau khi qua chỉ thị tiền xử lý như là #define, #include, #ifdef, #endif, v.v. Các chỉ thị tiền xử lý là các lệnh cho preprocessor để thay thế văn bản, chèn file, điều kiện biên dịch, v.v. trước khi trình biên dịch biên dịch mã nguồn

1. **Compilation :**

Bước compilation được thực hiện từ output của preprocessor. The compiler sẽ biên dich từ ngôn ngữ code bậc cao về assembly code

Ở bước này Complier sẽ tạo ra file .s qua command prompt:

gcc –S file\_name.c –o file\_name.s

Nội dung của file\_name.s sẽ là kết quả của việc biên dịch từ file.c thành assembly code.

1. **Assembly :**

The assembly code sẽ được converted thành machine language bởi assembler. Tệp này chứa machine-level instruction.

Ở bước này assembler sẽ tạo ra file .o qua command prompt:

gcc –c file\_name.c –o file\_name.o

Nội dung của file\_name.o sẽ chỉ là mã máy và cái function call chưa được giải quyết.

1. **Linking :**

Linker là bước cuối cùng để tạo đầu ra cho đối tượng được biên dịch. Nó sẽ link các file bằng cách thay thế undefined sysmbols bằng correct addresses. Mỗi symbol có thể được xác định ở các object file khác hoặc trong libraries.

gcc source.o mylib.o

Kết quả có thể là 1 thư viện dùng chung ( or dynamic) hoặc là 1 executable code.

**What is Objdump?**

Chúng ta về cơ bản là không thể đọc được file.o trong các norrmal text editor nay more vì content bên trong là binary code. Chúng ta có thể sử dụng opbjdump tool để có thể kiểm tra.

The basic syntax of Objdump is :

objdump [options] objfile...

options :

< -f > để display the contents of the overall file header

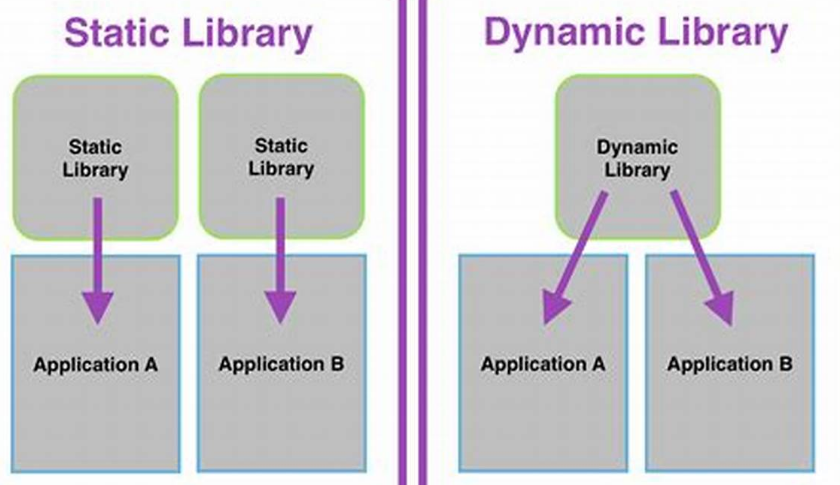
< -p > Display object format specific file header contents

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

< -D> Display assembler contents of all sections

**STATIC VÀ DYNAMIC LIBRARY**

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



**Compare:**

**Resource use :**

Static library dẫn đến executable files lớn hơn do chứa nhiều code hơn. Các additional code đến từ library không thể share sang các program khác trong system, tang mức sử dụng file system và memory khi run time. Nhiều processes đang chạy cùng program được liên kết tĩnh sẽ vẫn share the code.

Static app cần ít run-time reloations ( time moving code from memory location to another before running) làm giảm thời gian khởi động và yêu cầuu ít private resident set size memory. Code được tạo từ static linking có thể sẽ hiệu quả hơn so với dynamic linking do cần phải tìm địa chỉ khi call function

**Security :**

Dynamically linked libraries được cung cấp khả năng tương thích ABI ( application binary interface ) có thể được cập nhập mà ko thay đổi executable file phụ thuộc vào libraries đó.

Ngoài ra, các biện pháp bảo mật như là load address randomization không thể sử dụng với statically linked executable file. Điều này làm giảm tính bảo mật của ứng dụng.

**Compatibility :**

Static linking xuất hiện để cung cấp executable files không phụ thuộc vào version of libraries được cung cấp bởi OS. Tuy nhiên, đa phần libraries phụ thuộc các other libraries.Với static linking, sự phụ thuộc này sẽ trở nên không linh động và như một kết quả là cả hai khả tương thích trước và sau điều mất. Static linking sẽ được đảm bảo chỉ làm việc ở the system , nơi đã the executable file được built.

**Create Static Library**

1. Creater the object file for static library

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

1. Creater 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

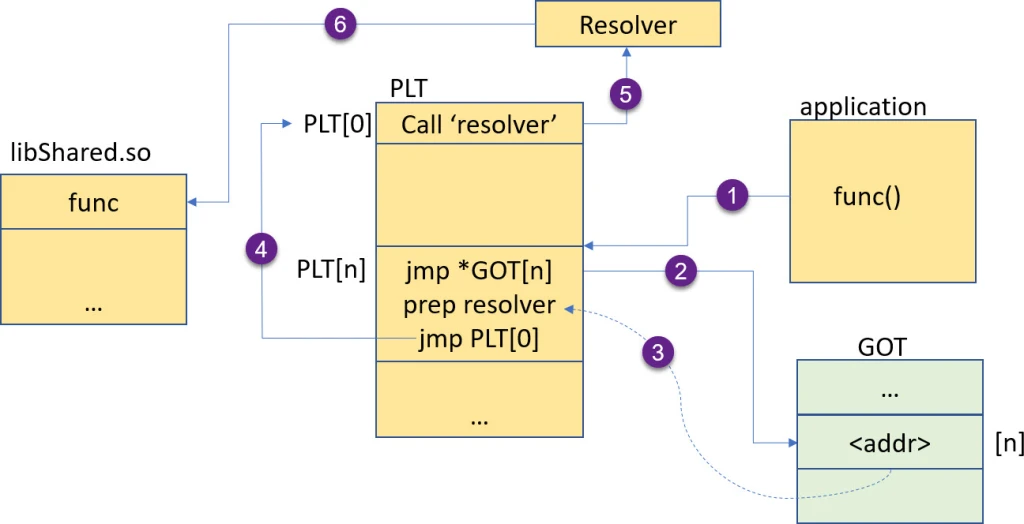
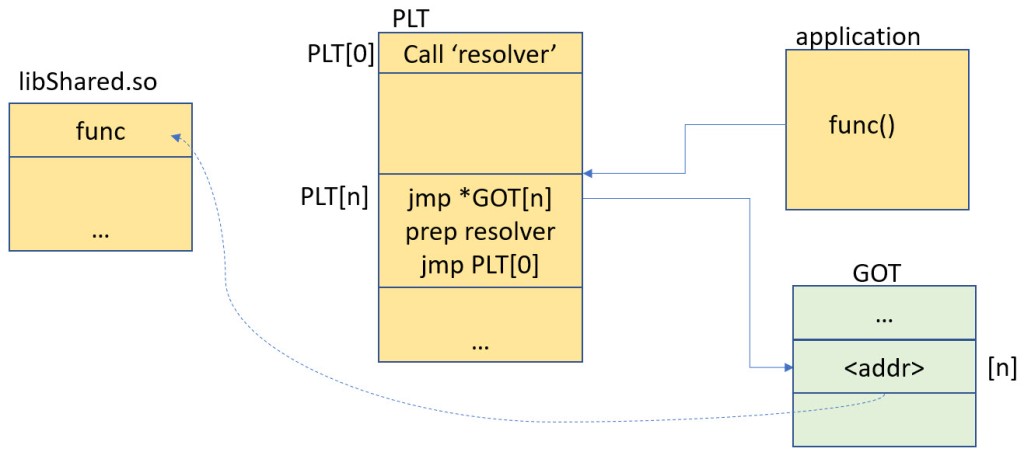
1. Creater the object file for dynamic library

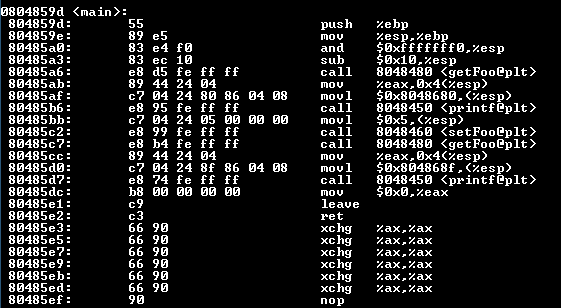
gcc -c -fPIC lib/foo.c -o lib/dynamic/libfoo.o

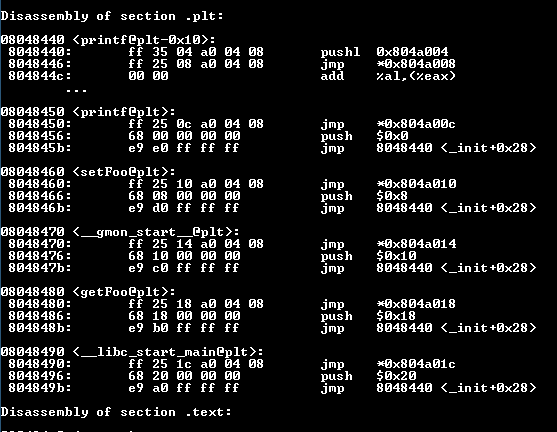
1. Creater dynamic library

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 |
|  |  |

[](https://mcuoneclipse.files.wordpress.com/2021/06/on-demand-binding-calls.jpg)





**MAKEFILE**

**What is Makefile?**

1 Makefile là một tệp văn bản chứa các command shell hướng dẫn về cách biên dịch và liên kết ( hoặc xây dựng ) a set ( một tập hợp ) of source code file. Giúp công việc trên trở lên nhanh chóng hơn.

**What is target ?**

A target là một file hoặc 1 hành động mà bạn muốn makefile thực hiện. Mỗi target có thể có dependencies, tức là files hoặc other targets cần được xây dựng trước khi xây dung target hiện tại. Mỗi target có thể cũng có commands, các đó là shell commands để tạo ra hoặc cập nhập the target.

**What is Marco ?**

The Make program cho phép bạn sử dụng macros , cái được sử dụng tương tự như biến

Marcos được định nghĩa ở trong 1 makefile dưới dạng = , sử dụng marco giúp ra tránh lặp lại các đoạn văn bản và Makefile sẽ dễ dàng sửa đổi.

**What is Rules?**

Quy tắc (rules) là các thành phần chính của một Makefile. Chúng nói cho Make biết cách tạo ra hoặc cập nhật các tệp nhất định, gọi là mục tiêu (targets), từ các tệp khác, gọi là phụ thuộc (prerequisites), bằng cách sử dụng các lệnh shell (command)

Một quy tắc có dạng chung như sau:

target [target …] : [prerequisites …]

[command … ]

**Rules** có 2 dạng là hỗ trợ dễ dàng viết cho nhiều file là :

Implicit Rules: không cần viết command vì implicit rules tự động chạy theo một bộ quy tắc ngầm, muốn thay đổi câu lệnh chỉ cần thay đổi cái trị FLAG trong bộ quy tắc khi sử dụng, Flags được sử dụng theo 1 bẳng quy tắc được quy định sẵn.

Static Pattern Rules :có thể viết mọi thứ ngắn hơn khi viết mutil target

Target …: target-pattern: prereq-patters

command

**What is automatic variables?**

Khi sử dung implicit rules và pattern rules thì sẽ ko biết rõ rang về tên của tệp , **automatic variables** hỗ trợ việc lấy file name từ targer và prerequites

$@ target name

$? All prerequisites newer than the target

$^ all prerequisites

What is .PHONY?

Việc thêm .PHONY vào mục tiêu sẽ ngăn Make hiểu nhầm target with a file name

**Shell**

**What is shell?**

Shell là một dòng lệnh thông dịch của hệ điều hành, hướng dẫn hệ điều hành thực thiện bất kỳ tác vụ và lệnh cần thiết. Shell command có thể ở trong scripts để thực hiện nhiều tác vụ và công việc phức tạp.

**What is Prompt?**

Lệnh gõ trên terminal, thao tác với hệ thống thông qua các câu lệnh trực tiếp trên terminal, sau dấu $

**Shell types:**

* Bourne Shell (sh): a historically first shell
* C Shell (csh): an C style shell
* Korn Shell (ksh): a mixed of Bourne Shell and C Shell
* Bourne Again Shell (bash): an 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 Distributed Version Control System – DVCS , Git cung cấp cho mỗi lập trình viên kho lưu trữ (repository) riêng chứa toàn bộ lịch sử thay đổi.

**Ảnh có chứa văn bản, biểu đồ, ảnh chụp màn hình, Hình chữ nhật

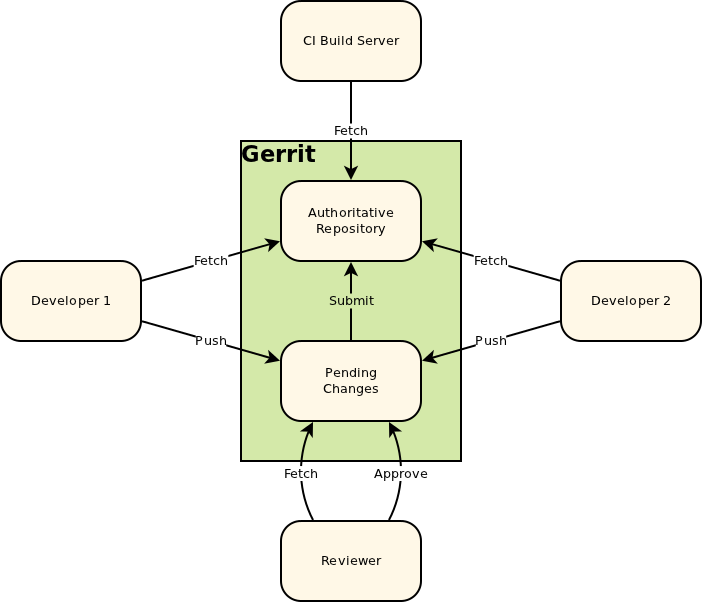
Mô tả được tạo tự động**

**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 branch, which you need to work
* git status: display all information about current branch
* git add: include the changes of repo into 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 (or –u ) <remote> <name of branch>: if your branch is new created, not exist in remote

**What is gerrit ?**

Gerrit là một công cụ hỗ trợ việc quản lý và review source code dựa trên nền tảng web, sử dụng git làm version control

Các developers sẽ Fetch source từ Authoritative Repository. Mỗi khi hoàn thành một task thay vì push lại repository này, họ phải push vào một Pending Changes repository. Điều này đảm bảo rằng mọi commit cần được Approve từ Reviewer(việc push back Authoritative Repository chỉ khi có permission đặc biệt). Một khi change đã được approved, reviewer có thể submit để merge change sang Authoritative Repository

**U-boot (Das U-Boot)** là một bootloader có mã nguồn mở được sử dụng rộng rãi trong các hệ thống nhúng nhỏ. Nó hỗ trợ sẵn cho các kiến trúc bao gồm 68k ,arm, blackfin,microblaze ppc và x86

1 boot rom

Khởi động 1 mã reset vector đc nhà sản xuất ghi để trỏ vào vùng khởi động, Rom startup code khởi tạp CPU, memory controller, on-chip devices, nó cũng cấu hình memory map. Rom code có thể load SPL image từ boot devices, tải vào bộ nhớ trong hoặc SRAM từ MCC, UART or SDCard,NAND Flash

SPL

Boot ROM copy content của “MLO” file đến static ram, do resource memory giới hạn , BootLoader sẽ lớn hơn nhiều, Vì có giới hạn bộ nhớ trong bộ nhớ trong, SPL trong trường hợp này, tệp MLO trước tiên được tải vào SRAM, tệp này sẽ khởi tạo DRAM và sao chép u-boot từ MMC hoặc Thẻ SD vào DRAM.

Uboot

Sau khi đc nạp vào RAM thì Uboot Uboot khởi tạo interface đc yêu cầu, cái sẽ giúp loading trong kernel,, Uboot giải nén kernel linux vào RAM , rồi trao quyền điều khiển cho Kernel, Uboot cũng cung cấp cho Kernel device trê and RAM Disk images

Kernel

Bây giờ kernel được tải vào bộ nhớ và được khởi tạo. initrd image được biên dịch và gắn vào bộ nhớ. Nó phục vụ như một temporary root file systten và giúp kernel khởi động đúng cách mà không cần gắn bất kỳ root file system. Bây giờ tất cả các trình điều khiển đã được tải vào bộ nhớ và kernel đã khởi động, kernel sẽ mount the root filesystem ở chế độ chỉ đọc và bắt đầu quá trình đầu tiên.

Init

Là first process started bởi kernel ( khởi tạo process) Nó là parent của all processes. The PID of init process = 1, process chạy đếb khi máy tính dừng, nó chịu trách nhiệm cho trạng thái của system

Run level khởi động mà hệ thống khởi động , có thể là a single user mode. Graphic, network

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

Module chạy ở kernel space, trong khi Applications chạy trong user space. Các quá trình chạy trong user space cũng không thể truy cập đến the kernel space. User space processes chỉ có thể truy cập một phần nhỏ của kernel qua một giao tiếp tiếp xúc bởi kernel- the system calls

Các hệ thống Unix được thiết kế để tận dụng tính năng phần cứng này, sử dụng hai cấp độ như vậy. Tất cả các bộ xử lý hiện tại đều có ít nhất hai cấp độ bảo vệ và một số, như dòng x86, có nhiều cấp độ hơn; khi có nhiều cấp độ, cấp độ cao nhất và thấp nhất được sử dụng. Trong Unix, kernel thực thi ở mức cao nhất (còn gọi là chế độ giám sát), nơi mọi thứ được cho phép, trong khi các ứng dụng thực thi ở mức thấp nhất (được gọi là chế độ người dùng), nơi processor quy định quyền truy cập trực tiếp vào phần cứng và truy cập trái phép vào memory.

Chúng ta thường gọi các chế độ thức thi là kernel space and user space.

**System calls**

1 system call là 1 cách có lập trình mà một chương trình yêu cầu 1 dịch vụ từ kernel

System call interface bao gồm một số các chứng năng mà OS xuất sang các ứng dụng đang chạy trên đó.

Các function này cho phép hoạt đông như hệ thống tiệp mở, tạo các các kết nối mạng , đọc và viết từ các file

System call đc chia ra làm 5 phần chính:

**+ Process Control**

**+ File management**

**+ Device management**

**+ Information maintenance**

**+ Communication**

**1.Process control**

System gọi thực hiện các nhiệm vụ of proceses creation , proceses termination ,vv

**Fork() :**

1 tiến trình mới được tạo ra bởi system call fork(). Một tiến trình mới có thể đc tạo ra với fork() không cần chạy lại chương trình mới, trương trình con đơn giản tiếp tục thực thi chính xác giống chương trình là tiến trình đang chạy đầu tiên( chương trình cha) .

**Exit() :**

system call được sử dụng để kết thúc tiến trình thực thi. Hệ điều hành sẽ lấy lại tài nguyên đang được sử dụng bởi tiến trình sau khi exit() system call.

**Exec()** : 1 chương trình mới sẽ bắt đầu sau khi gọi call exec()

Chạy 1 chương trình mưới ko yêu cầu rằng tiến trình mới phải tạo ra đầuu tiên, bất kì tiến nào cũng có thể call exec() tại bất kỳ thời gian nào. Chương trình đang chạy immediately terminated và chương trình mưới sẽ bắt đầu thực thi với nội dung của tiến trình hiện có.

Sử dụng function này , tạo ra tiến trình con chạy không nhất thiết phải phải giống tiến trình cha. Các lệnh gọi hệ thống loại exec cho phép một quy trình chạy bất kỳ tệp chương trình nào, bao gồm tệp thực thi nhị phân hoặc tập lệnh shell .

**2.File Management**

File Management system calls xử lý các công việc thao tác tệp dữ liệu như là đọc và viết etc.

**Open()** It is the system call to open file

**Read()** this system call open the file in reading code , chúng ta ko thể chỉnh sửa file với system call này. Nhiều tiến trình có thể thực thi read() system call trên đồng thời trên cùng 1 tệp

**Write()** có thể edit file , nhiêìu tiến trình ko thể thực thi write() system call ở trên cùng 1 tệp

**Close()** đóng file

**3.Communication**

Các kiểu of system call được sử dụng đặc biệt cho inter-process communication (IPC)

2 models đc sử dụng cho inter-process communication:

+ Truyền message ( process trao đổi message với 1 process khác)

+ Shared memory ( process shared memory region (vùng , khư vực) to communicate)

**pipe() shmget mmap()**

**+) Pipe()**

**Pipe()** system call đc sử dụng để giao tiếp giữa 2 linux process khác nhau

Mục đích chính được sử dụng cho inter process communication

The pipe() system fuction đc sử để để mở file descriptors

**+) shmget()**

Shmget viết tắt của share memory segment

Mục đính sử dụng chúng là để inter-process communication

System call này đc sử dụng để truy cập the shared memory và truy cập message in order to communicate with process

**+) Mmap()**

This function call đc sử dụng để map ( ánh xạ ) hoặc unmap files or devices vào trong memory

Mmap() system call chịu trách nhiện cho việc ánh xạ nội danh cảu vai đến virtual memory space của process

**4.Device Management**

Device management làm công việc của thiết bị thao tác như là đọc từ device buffers, viết vvào trong device buffer , etc. The Linux system call dưới là ioctl()

Ioctl() là giới thiệu như là Input and Output Control

Là 1 system call của hoạt động vào ra thiết bị cụ thể ( device specific input/output operations) và các hoạt động khác, cái mà không thể được mong đời từ system call thông thường.

**5. Information Maintenance**

Tác dụng xử lý thông tin và nó tryền giữa OS và chương trình người dùng. Ngoài ra, OS giữ thông tin về tất cả các tiến trình của nó và system call được sử dụng để truy câp và thông tin đó. System call dưới là getpid(), alarm() sleep()

**Getpid()** viết tắt là get the process id

Function sẽ return lại process ID của tiến trình đang gọi

Function sẽ luôn luôn thành công và ko trả về giá trị dành riêng để chỉ lỗi

**Alarm()**

This system call sẽ đặt đồng hồ báo thức để gửi 1 tín hiệu khi nó được đặt được đủ

Nó sắp xếp để 1 tín hiệu được gửi đến process đang gọi

**Sleep()**

This system call hoãn việc thực thi của process hiện tại đang chạy trong một khoảng thời gian

Trong khoảng thời gian đó, another process có cơ hội để thực thi.

**What is process?**

Process là một program đang thực thi (executes) The process được tạo ra khi 1 lệnh được thưc thi nên nó có thể được gọi là một trường hợp đang chạy của program. Bất kỳ command nào bạn thực hiện điều bắt đầu bằng một process

**What is threads ?**

Một thread là một cơ chế cấp quyền 1 application để thực hiện kiêm nhiệm mutiliple tasks. Process là một nhóm các thread có liên kết, thực thi cùng nhau trên cùng một môi trường và cùng chia sẻ tài nguyên trên đó với nhau. Nghĩa là các thread trong cùng một process chia sẻ với nhau cùng một memory space và có thể giao tiếp trực tiếp với nhau, sử dụng hết chức năng của multi core.

**Pthread\_join()** đợi thread được xác định bởi thread kết thúc, tạm dừng thread thực thi đang chạy đến khi thread được join kết thúc. Nếu thread đã kết thúc , thread\_join sẽ return ngay lập tức. Hành động này gọi là joining

**Pthread\_detach()** về cơ bản , 1 thread là joinable nghĩa là khi nó kết thúc, một thread khác sẽ có thể lấy được kết quả trả về của nó qua việc sử dụng pthread\_join() . Thi thoảng chúng ta không quan tâm về the thread’s return. Chúng ta đơn giản là muốn hệ thống tự động dọn sạch và xóa bỏ thread khi nó tắt. Trong trường hợp này, chúng ta có thể đánh dấu thread như là detach.

**What is Thread syncronization ?**

Khi chúng ta bắt đầu chạy 2 hay nhiều thread trong 1 program, có thể xảy ra trường hợp khi nhieiuef luồng cố gắng truy cập vào cùng 1 tài nguyên tạo ra một kết quả không thể lường trược được. Vì thế cần phải đồng bộ hành động của nhiều thread để đảm bảo rằng chỉ có 1 thread có thể truy cập vào tài nguyên trong một thời điểm

**Methods of thread synchronization ?**

**Semaphone** : cấp quyền cho một số thread được thực hiện một phần của code. Semaphone cấp cho các thread nhiều key , trong một thời điểm các thread có key có thể có nhiều thread có thể cùng sử dụng một tài nguyên.

**Mutex :** tạo ra 1 key duy nhất , trong một thời điểm chỉ có 1 thread duy nhất có được key để sử dụng tài nguyên cần được synchronization.

**Race condition:** Kết quả của việc tính toán bị ảnh hưởng nếu có hơn 2 thread cùng thực hiện đoạn code hay cùng sử dụng 1 tài nguyên để tính toán.

**Deadlock** : Hai hay nhiều thread đi vào vòng lặp vô tận để chờ đợi 1 tài nguyên nào đó.

**How to share data between proceses ?**

Một phần của memorry mà nhiều process có thể read or write

Message passing : pipes – sockets – signals

System V API :

Msgget() create 1 new message queue

Msgsnd() pushes a message onto the queue

Msgrcv() pops in the queue

POSIX Message Queue: tối ưu và dễ dàng sử dụng có mức độ ưu tiên liên quan đến message , và message luôn luôn được xếp hàng nghiêm ngặt , người nhận sẽ được nhận theo thứ tự ưu tiên.

POSIX API:

Mq\_open

Mq\_send

Mq\_reveive

Mq\_close

Mq\_unlink

Share memory :

**+) shmget()**

Shmget viết tắt của share memory segment

Mục đính sử dụng chúng là để inter-process communication

System call này đc sử dụng để truy cập the shared memory và truy cập message in order to communicate with process

**+) Mmap()**

This function call đc sử dụng để map ( ánh xạ ) hoặc unmap files or devices vào trong memory

Mmap() system call chịu trách nhiện cho việc ánh xạ nội danh cảu vai đến virtual memory space của process

**Ảnh có chứa văn bản, biểu đồ, ảnh chụp màn hình, Hình chữ nhật

Mô tả được tạo tự động**

Git config use user-specific cấu hình giá trị như là email, user name, file format

**git config --global user.email** [**youremail@example.com**](mailto:youremail@example.com)

Git status hiểu thị danh sách trạng thái của local working and staging area so với

Git checkout create branches and chuyển đổi giữa chúng

**command git checkout -b <branch-name>**

or switch from one branch to another

**git checkout <branch-name>**

Connect the local working to a remote sever

**git remote add origin <host-or-remoteURL>**

**git remote rm <name-of-the-repository> :** sẽ delete connection to a specified remot repository

check branch: git branch – return branch itself

Git status :