DEVLINUX
ACADEMY

LINUX PROCESS

“Code Your Embedded Future”

https://devlinux.vn

Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097

DEVLINUX ACADEMY

"Code Your Embedded Future"

AGENDA

- I. Introduction
- II. Command-line Arguments
- III. Memory Layout
- IV. Operations on Process
- V. Process Management
- VI. Orphane & Zombie Process

https://devlinux.vn

Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097

DEVLINUX
ACADEMY

INTRODUCTION

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

What is the Program & Process?

Program and process are two related terms.

- We need to distinguish beetween program and process
- Each process has a code that is used to identify it called Process ID (PID), which is a positive and unique integer for each process on the system.

Program

A program is a group of instructions that perform a specific task, represented by executable files located on a computer's hard drive.

Process

A process is a program that is executing and using system resources.

DEVLINUX
ACADEMY

COMMAND-LINE ARGUMENTS

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Command-line Arguments

Every program starts from main.
When running the program, the environment parameters (command line arguments) will be passed as two arguments in the main() :

- argc
- argv[]

argc

Number of parameters passed into main() function

argv[]

Array of pointers to point to the parameters passed into the main() function

Example

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    // In ra số lượng đối số dòng lệnh được truyền vào
    printf("Number of arguments: %d\n", argc);

    // In ra nội dung của từng đối số dòng lệnh
    for (i = 0; i < argc; i++) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }

    return 0;
}
```

DEVLINUX
ACADEMY

MEMORY LAYOUT

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

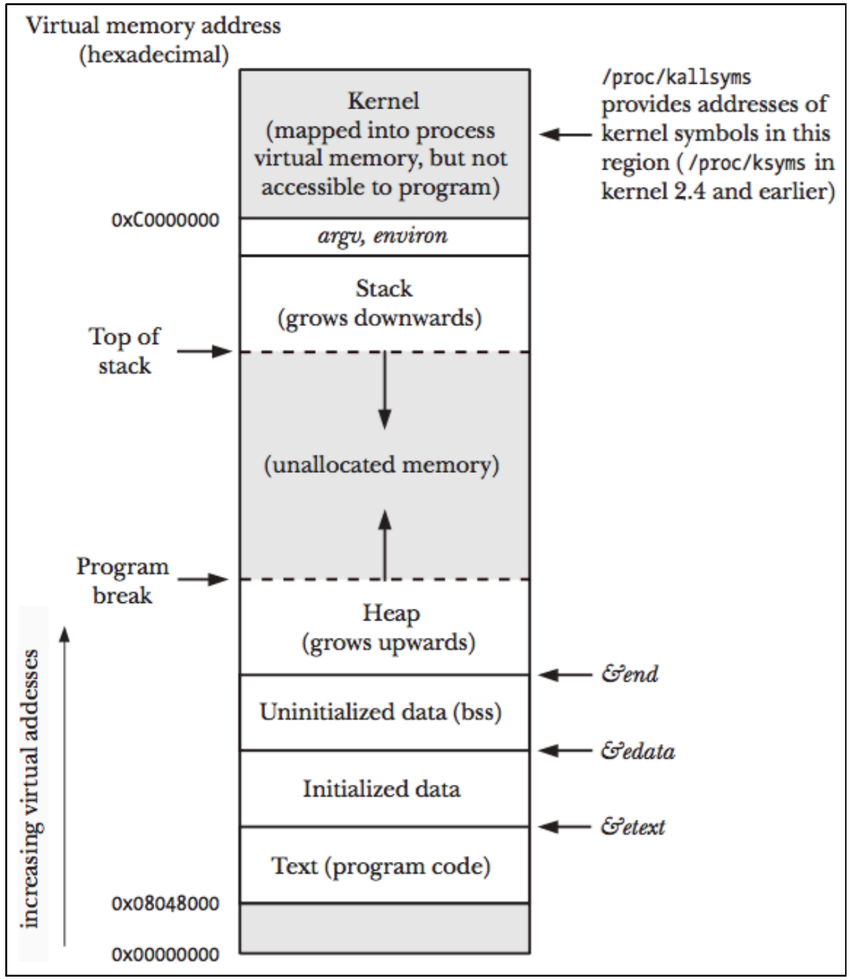
“Code Your Embedded Future”

Memory Layout

The memory allocated to each process is usually divided into different parts. These are commonly called segments or memory partitions.

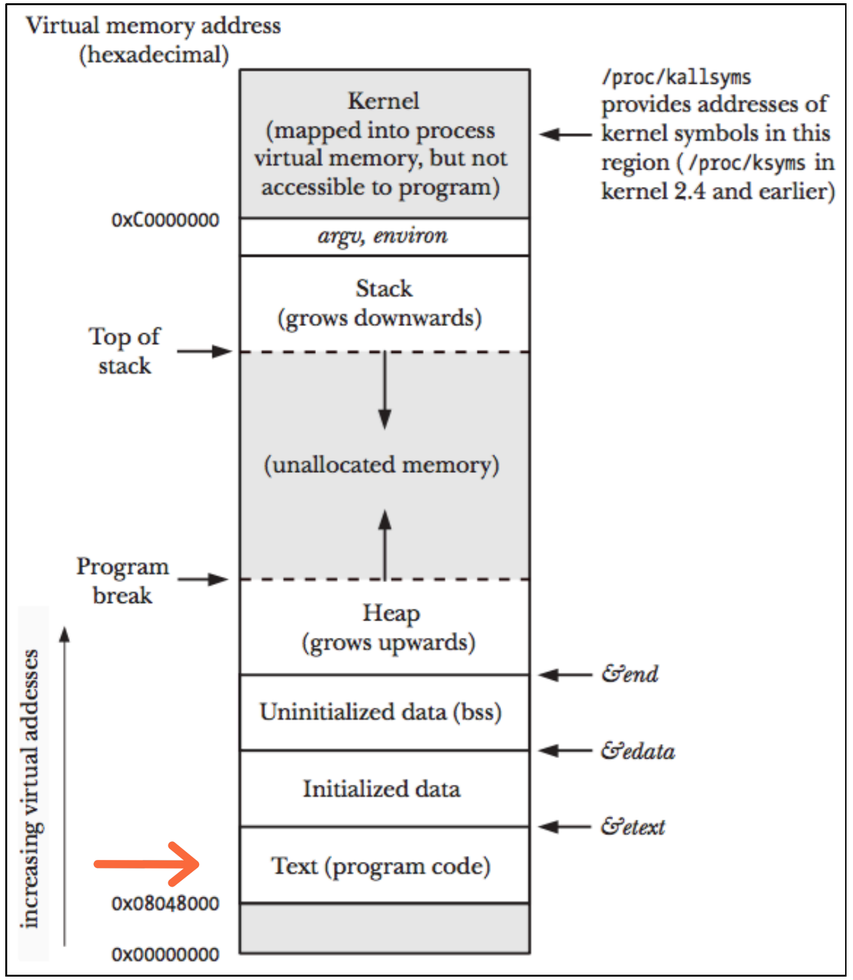
The main segments include:

- Stack
- Heap
- Uninitialized data (.bss)
- Initialized data (.data)
- Text (.text)



Text Segment

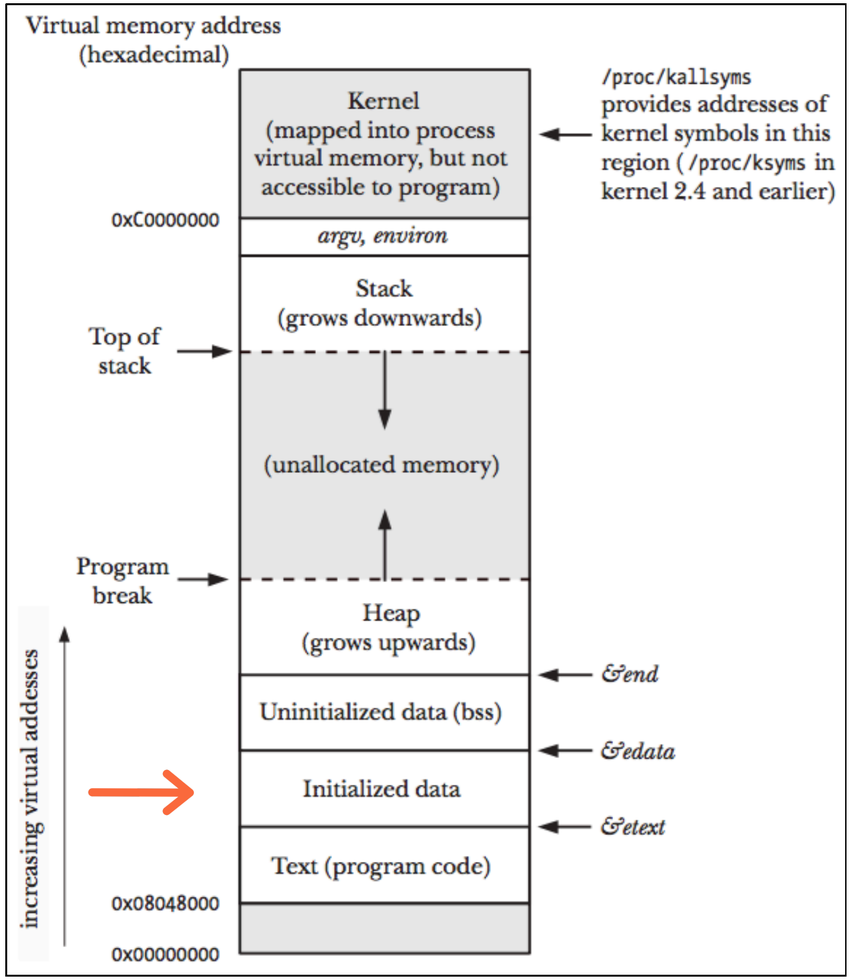
Contains the machine code instructions of the program.
This segment is read-only.



https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

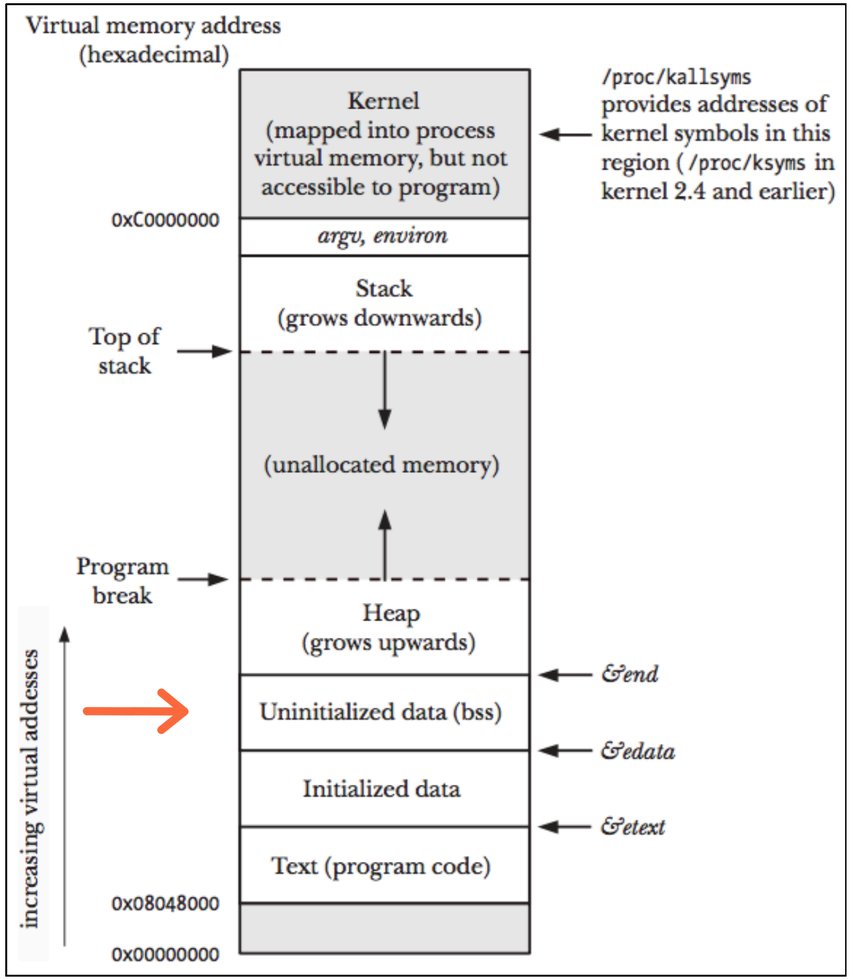
Initialized Data Segment

Contains global and static variables that have been explicitly initialized.
This segment has read and write permissions.



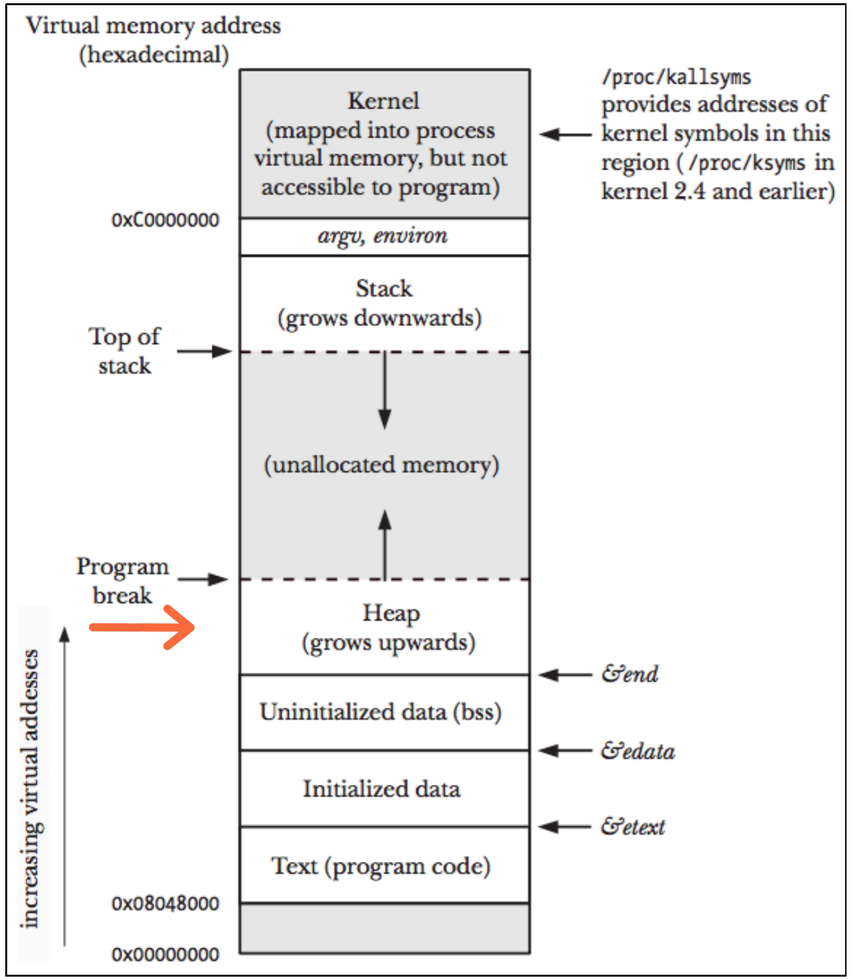
Uninitialized Data Segment

Contains uninitialized global and static variables.
This segment has read and write access.



Heap Segment

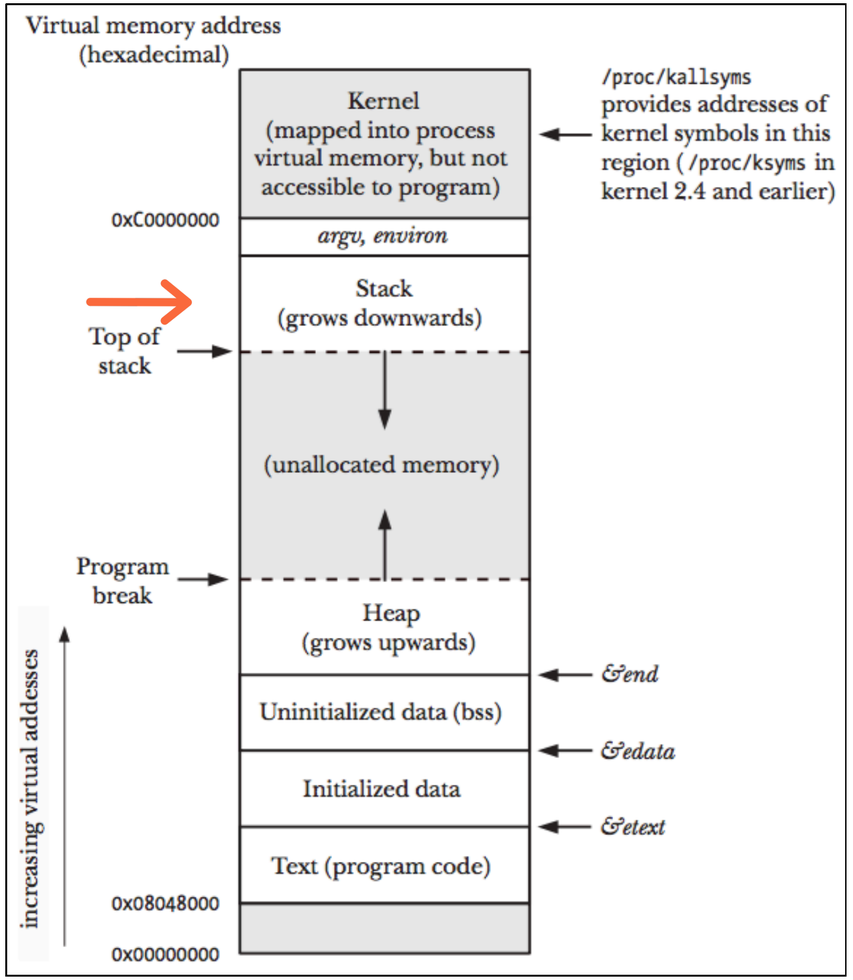
This segment is used for automatic memory allocation. Use functions like alloc(), malloc(), calloc().
The end point of the Heap region is called program break.
This segment has read and write permissions.



https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

Stack Segment

The stack can be extended by allocating or freeing stack frames.
This segment has read and write access.



DEVLINUX
ACADEMY

OPERATIONS ON PROCESS

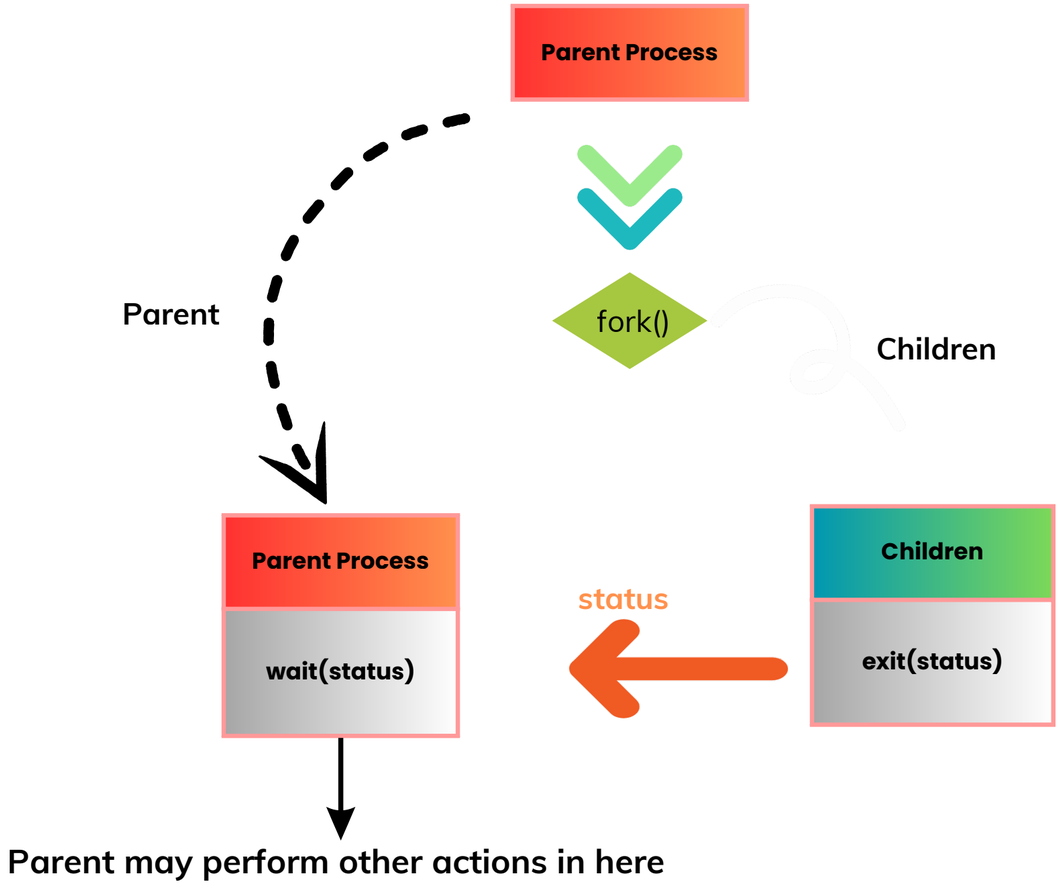
https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

System call fork()

Create a new process.

- Use system call fork().
- The process calling fork() is called the parent process.
- New process are created by parent process is called child process.
- Init process is the first process that is run, is the parent of all other processes, and has a pid of 1



Exec Family

In many cases, you need to run process A and want to run a program B from process A or from its child processes. This can be done using a series of program execution functions.

```c
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...);

int execlp(const char *filename, const char *arg, ...);

int execvp(const char *filename, char *const argv[]);

int execv(const char *pathname, char *const argv[]);

int execl(const char *pathname, const char *arg, ...);

None of the above returns on success; all return -1 on error
```

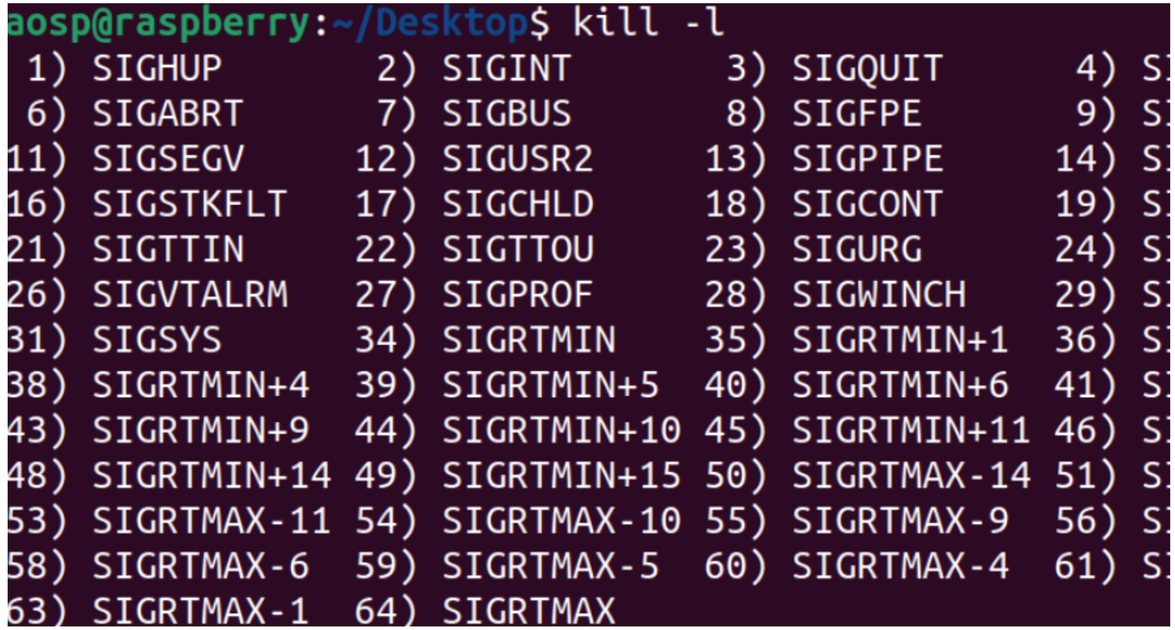
Process Termination

Normal termination

- Normally, a process can terminate its execution by calling the _exit() system call or using the exit() function.

Abnormally termination

- In unusual cases, a process can be terminated by using the kill() system call or the kill command in Linux.



Example

Let take an example by using execl() function

```c
int main(int argc, char *argv[])
{
    printf("Run command <ls -lah> after 2 seconds\n");
    sleep(2);

    execl("/bin/ls", "ls", "-l", "-h", NULL);

    return 0;
}
```

DEVLINUX
ACADEMY

PROCESS MANAGEMENT

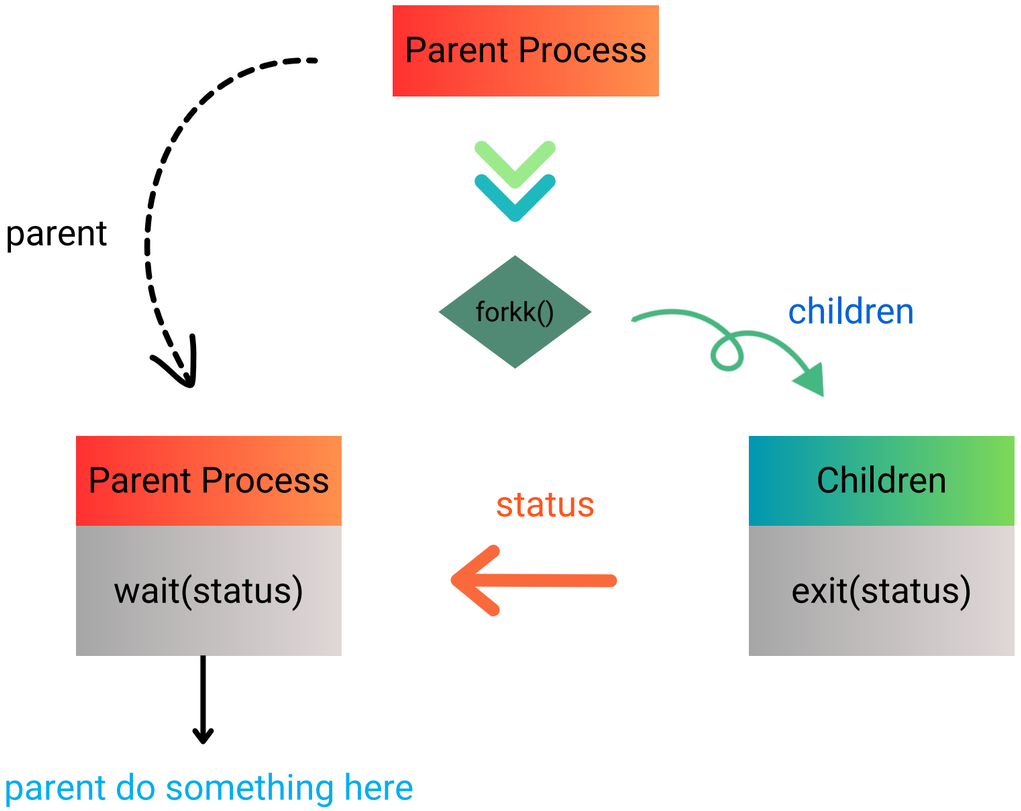
https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Process Management

One of the programming philosophies is “do one thing and do it well”. The process management is basically implemented using a number of system calls. These system calls can be combined to perform more complex problems.

- **fork**()
- e**xit**()
- w**ait**()



System call wait()

The wait() system call is used to monitor the termination status of one of the child processes created by the parent process.

- At the time wait() is called, it will block until a child process terminates. If there exists a child process that has terminated before the time of calling wait(), it will return immediately.
- If the status is other than -1, status argument will point to a value that is an integer, which is information about the end state of the process.
- When wait() is finished, it will return the PID of the child process or return -1 if it fails.

wait()

```
#include <sys/wait.h>

/*
* @param[out] status Trạng thái kết thúc của tiến trình con.
*
* @return     Trả về PID của tiến trình con nếu thành công, trả về -1 nếu lỗi.
*/
pid_t wait(int *status);
```


System call waitpid()

The system call wait() has a limitation:

- If the parent process creates many child processes (many children), the wait() function cannot be used to monitor a particular child process.
- waitpid() solves this problem.

waitpid()

```
#include <sys/wait.h>

/*
* @param[in]  pid     pid  >  0, PID của tiến trình con cụ thể mà wait muốn theo dõi.
*                     pid  =  0, Ít sử dụng.
*                     pid  < -1, Ít sử dụng.
*                     pid == -1, Chờ bất cứ tiến trình con nào thuộc về tiến trình cha - giống wait().
* @param[out] status  Trạng thái kết thúc của tiến trình con.
* @param[in]  options Thông thường chúng ta sẽ sử dụng giá trị NULL ở trường này.
*
* @return     Trả về PID của tiến trình con nếu thành công, trả về -1 nếu lỗi.
*/
pid_t waitpid(pid_t pid, int *status, int options);
```

DEVLINUX
ACADEMY

ORPHANE & ZOMBIE PROCESS

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embeded Future”

Orphane Process

The parent process terminates before the child process, so who will get the child process's termination state ?

Zombie Process

What if the child process terminates before the parent process can call system wait() ?

ZOMBIE

Prevent Zombie Process

There is a process ID (PID) table for each system. The size of this table is finite.

If too many zombie processes are created, this table will be full. As the result of this, the system will not be able to create any new processes.

wait()/waitpid()

Always call wait() or waitpid() in the parent process

SIGCHILD

When the child process terminates, a signal of type SIGCHILD is sent to the parent process.

DEVLINUX
ACADEMY

Thanks for Your Attention!

Liên hệ với chúng tôi

https://devlinux.vn

Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097