进程管理

1. 创建进程

Linux 系统里，创建进程有两种目的：

- 将一个程序分成多个进程处理 (web 服务器接收多个请求)
- 创建另外一个程序 (从 bash 启动另外一个程序)

两种目的分别对应 Linux 提供的 `fork()` 与 `execve()` 函数，分别对应 `clone()` 与 `execve()` 两个 **系统调用**

1.1 fork() 函数

`fork()` 将基于发出调用的进程 (称为 **父进程**)，创建一个新进程 (称为 **子进程**)，流程为：

- 为子进程申请内存空间，将父进程的内存复制给子进程
- 父进程与子进程分裂为两个进程，执行不同的代码，这里靠 `fork()` 返回的值来区分父子进程

下面使用在 Linux 上运行 **fork.cpp**

```
#include <unistd.h> // 存放与系统调用相关的函数, getpid()
#include <err.h>
#include <bits/stdc++.h>

// getpid() 获取当前进程的 ID

static void child() {
        std::cout << "[Child] pid is " << getpid() << '\n';
        exit(EXIT_SUCCESS);
}

static void parent(pid_t pid) {
```

```
        std::cout << "[Parent] pid is " << getpid() << " [Child] pid is " <<
pid << '\n';
        exit(EXIT_SUCCESS);
}

int main() {
        pid_t ret = fork();// 由此创建子进程
        if (ret == -1)
                err(EXIT_FAILURE, "fork() failed");
        if (ret == 0) {
                // 子进程的 fork() 返回 0
                child();
        } else {
                // 父进程的 fork() 返回子进程的 进程ID
                parent(ret);
        }
        // 正常情况下两个进程在各自的函数里已经成功退出，或者 fork() 失败时退出
        err(EXIT_FAILURE, "shouldn't reach here");
}
```

运行结果为：

```
syz@syz:~/projects/class3$ g++ fork.cpp -o fork
syz@syz:~/projects/class3$ ./fork
[Parent] pid is 1023 [Child] pid is 1024
[Child] pid is 1024
syz@syz:~/projects/class3$
```

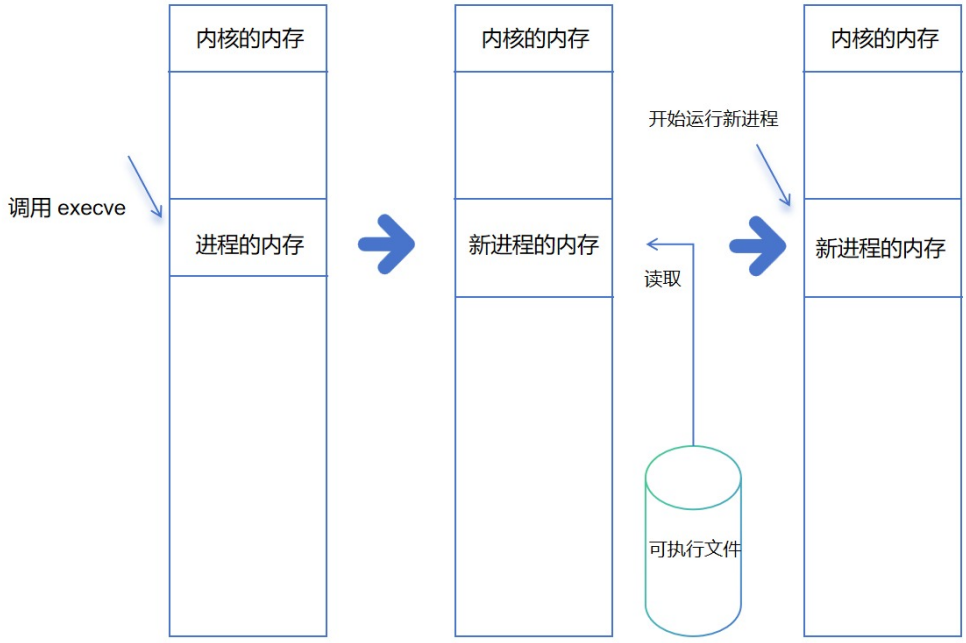
> 本质上 `fork()` 就是通过返回值的不同来实现一个程序分裂为多个线程

1.2 execve() 函数

内核运行新进程的流程一般是：

- 读取可执行文件，读取创建进程的内存映像所需的信息
- 用新进程的数据覆盖当前进程的内存
- 从最初的命令开始运行新的进程

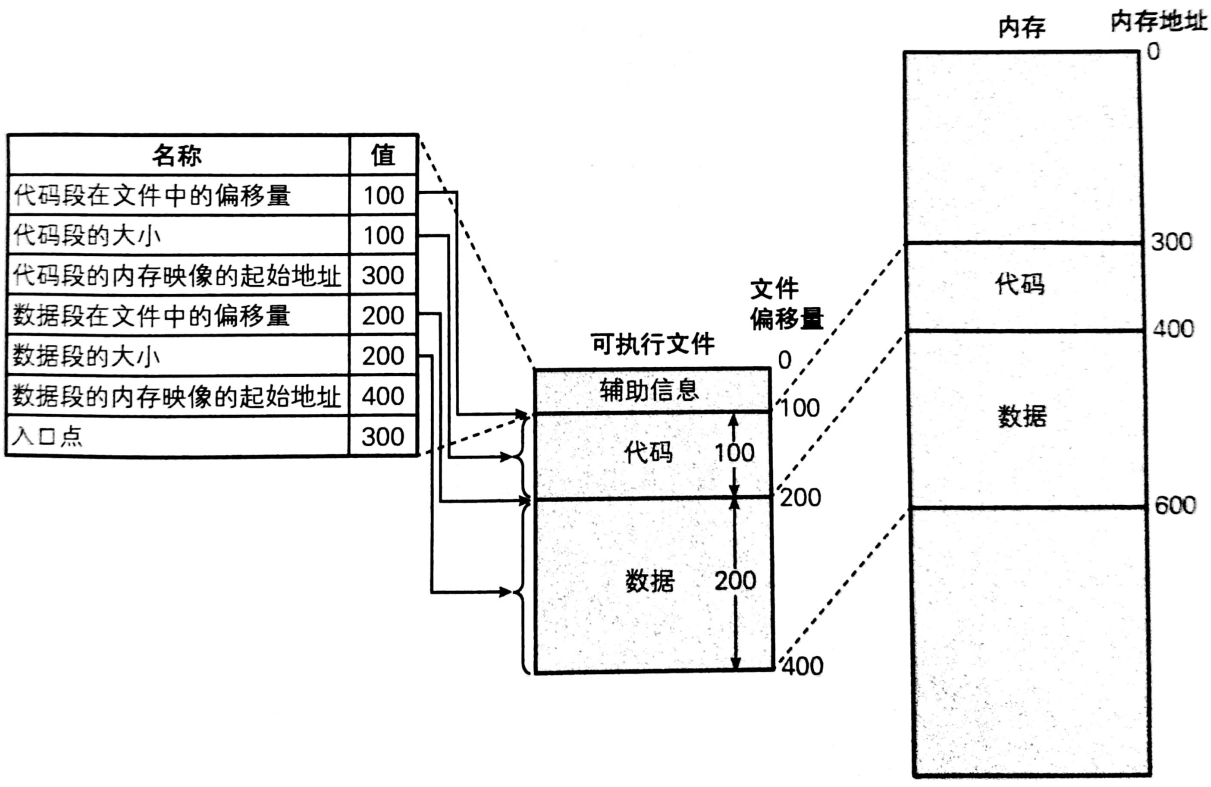
> 启动新进程，并非新增一个进程，而是替换了当前进程



首先第一步，读取可执行文件，可执行文件；可执行文件中不仅包含代码和数据，还包含了开始运行程序时所需的数据：

- 代码在可执行文件中的偏移量，大小，以及内存映像的起始位置
- 包含代码以外的变量等数据段在文件中的偏移量，大小以及内存映像的起始地址
- 程序执行的第一条指令的内存地址

对应关系图如下所示：



系统从地址 `300` 开始运行文件.

Linux 可执行文件的结构遵循 ELF (Executable and Linkable Format, 可执行与可链接格式)，可以通过 `readelf` 命令获取其相关信息。

例如，我们可以获取 `/bin/sleep` 的 ELF 信息。

- 使用命令 `readelf -h /bin/sleep`

```
syz@syz:~/projects/class3$ readelf -h /bin/sleep
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x2b80
  Start of program headers:          64 (bytes into file)
  Start of section headers:          33352 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

- 使用命令 `readelf -S /bin/sleep`

可以获取代码段，数据段在文件中的偏移量，大小和起始地址。

输出：

```
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  ...
  [16] .text             PROGBITS         00000000000025e0  000025e0
       0000000000002ae2  0000000000000000  AX       0     0     16
  ...
  [26] .data             PROGBITS         0000000000009000  00008000
       0000000000000098  0000000000000000  WA       0     0     32
```

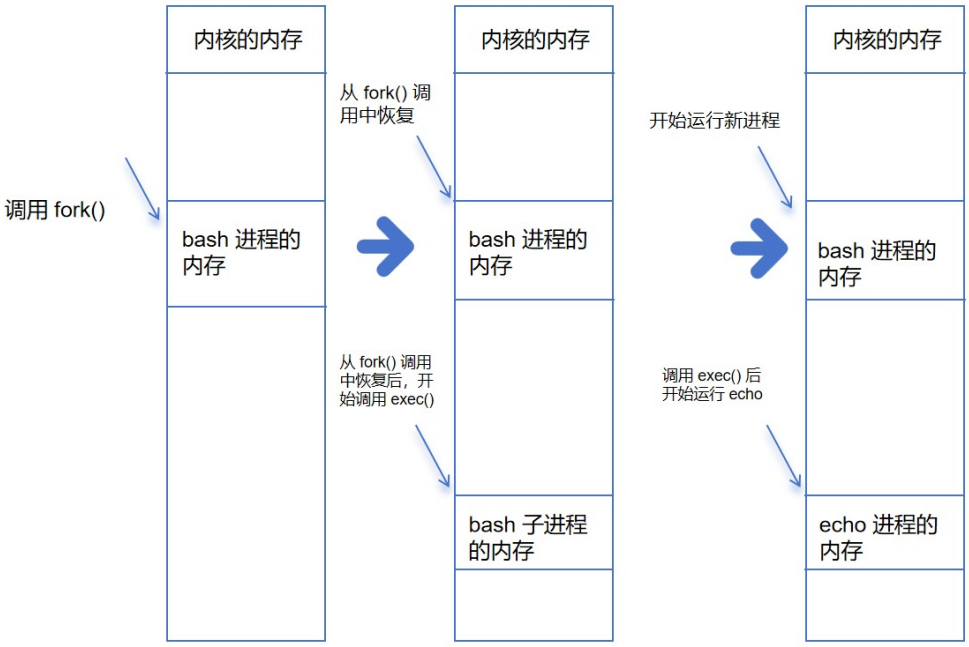
有如下特征：

- 输出的数据为两行为一组
- Name 字段，`.text` 表示的是代码段的信息，`.data` 表示的是数据段的信息
- Address：内存映像的起始地址

- Offset：在文件中的偏移量
- SIze：大小

> 注意这里，代码段的内存映像的起始地址与程序入口点不一致

如果需要新建一个新进程，一般采用 `fork and exec` 的方式，父进程调用 `fork()` 创建子进程，再由子进程调用 `exec()` ，下图展示了 `bash` 进程创建 `echo` 进程的流程。



我们可以编写一段如下程序，体验一下 **fork and exec**

```
syz@syz:~/projects/class3$ cat fork-and-exec.cpp
#include <unistd.h>
#include <bits/stdc++.h>
#include <err.h>

// int execve(const char *filename, char *const argv[], char *const envp[]);
// filename : 新程序的路径和文件名
// argv 包含了要传递给新程序的命令行参数，其中第一个参数通常是程序名。
// envp 包含了新程序的环境变量。

static void child() {
        char *args[] = {"/bin/echo/", "hello", NULL};
        std::cout << "[child] pid is " << getpid() << std::endl;
        execve("/bin/echo", args, NULL);
        // echo 自带退出功能，因此不会往下执行了。
        err(EXIT_FAILURE, "exec() failed");
}

static void parent(pid_t pid_c) {
        std::cout << "[parent] pid is " << getpid() << " and [child] pid is " << pid_c << std::endl;
        exit(EXIT_SUCCESS);
}

int main() {
        pid_t ret = fork();
        if (ret == -1)
                err(EXIT_FAILURE, "fork() failed");
        if (ret == 0) child();
        else parent(ret);

        err(EXIT_FAILURE, "shouldn't reach here");
}
```

运行结果为：

```
syz@syz:~/projects/class3$ ./fork-and-exec
[parent] pid is 1366 and [child] pid is 1367
[child] pid is 1367
```

同时，`python` 里也有 `os.exec()` 函数来请求 `execve()` 系统调用

2. 结束进程

结束进程可以使用系统库的 `_exit()` 函数，但是 `C/C++` 一般使用 `exit()` 来进行进程退出，该函数完成自身的处理之后会调用 `_exit()` 函数。