阿男的 Linux 内核世界

阿男

Dec 22, 2016

Contents

1 **开篇** 5

2 **进程** 7

什么是进程 9

Parent Process 与 Child Process 13

小结 15

僵尸进程 16

为什么要有 zombie process? 18

Chapter 1

开篇

按照和大家的约定，阿男要开始给大家开这个 Linux 内核专栏了。开这个专栏是非常开心的事情，因为操作系统也是阿男两个最喜欢的领域之一（另一个是编译原理）。

这个专栏的方向无比清晰：让大家知道操作系统是怎么工作的。

阿男觉得学习操作系统和编译原理是最超值的，因为这种学习会彻底改变你在从事编程这件事的时候，你眼前的世界的样子。

首先，很多表面上复杂的东西，其实理解了背后的工作原理，就会发现其实非常简单，正则表达式就是最好的例子（参见阿男之前写的专栏：『阿男导读』*正则表达式背后的有限状态自动机*）。

然后你能知道问题的边界在哪里，把手头的问题放入一个更为层次丰富，更为透明的世界里，你会看得更清楚。所以把层次了解的丰富一些，更方便自己对手头的问题作出判断。

最后，深入的学习当然还有个好处就是，你手头的工具变得丰富了。当你解决一个问题的时候，你可以更为合理地选择更多层面的工具。

阿男感受过这种体验，很棒。相信大家也会体验到。那么我们接下来就一起学习操作系统吧！这个专栏可能会延续很久，没关系，我们有的是时间学习。

Chapter 2

进 程

进程（process）可能是 Linux 世界下最有生命力的存在了。所谓 process 就是活的程序，正在运行中的代码。

计算机必须要有软件，人类才可能使用它。软件和硬件工程师们合作创造了 BIOS（Basic Input/Output System），BIOS 本身是软件，是程序，但是是固化在芯片里的代码（现代的 BIOS 也可以编程，刷新，升级），随着计算机启动最先加载这些代码。

BIOS 负责将计算机里面的各个硬件加载，运转起来。

科学家们给这一过程非常形象地起了一个名字：Bootstrap。

什么是 Bootstrap？这个词初见于童话故事《吹牛大王历险记》：在书中，敏豪森男爵吹牛说，自己骑着马掉进了沼泽，但他天生神力，两腿夹着马，双手抓着自己的小辫子，把自己和马一起从沼泽里拉到了半空！[1]

计算机的启动过程就比较类似于这样，但是为什么后来这个启动过程叫做 Bootstrap（自己的鞋跟），而不是拉着自己的小辫子，这就说不清了。历史上一个说法是，这个德国民间故事后来传到美国，就被再加工，变成了拉鞋跟了。

当 BIOS 完成 Bootstrap 后，会把控制权交接给操作系统。此时 Linux 操作系统的话，会接收控制权，然后执行 Bootloader，比如 GRUB 或者 LILO，让你

[1] The Adventures of Baron Munchausen，《吹牛大王历险记》是 18 世纪德国著名的儿童文学作品。

选择要运行的 Linux 系统[2]。

对于没有安装 Boot Loader 的 Linux 系统，则跳过选择这一步，直接开始 Linux 系统的加载过程。

Linux 系统加载的过程是这样的：首先，一个编号为 0 的 process 产生了，它是所有后续 proce ss 的 parent。这个编号为 0 的 process 会产生一个编号为 1 的 process，然后自我毁灭（准确来讲，是变成 swapper）。这个 id 为 1 的 process 就是 init，它是所有后续 process 的 parent。Li nux 下面的 process，是 parent 与 child 这样的层级关系。

一个 parent process 可以克隆自己，拥有有多个 children，而每个 child 只有一个 parent。每个 process 的编号叫做 process id，简称 PID。每个 process 的 parent process 的编号叫做 parent process id，缩写是 PPID。

我们在比较新的 Linux 操作系统里面，使用 ps 命令查看 pid 为 1 的 process 时，可能会和上面说的的不太一样。比如阿男的 Fedora 24 系统：

```
$ ps -fp 1
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Dec25 ?        00:00:03 /usr/lib/systemd/systemd
--switched-root --system --deserialize 23
```

pid 为 1 的 process 是 systemd，不是 init，这是因为 systemd 现在会接管系统。关于 systemd，这篇文章不想展开，如果大家有兴趣，阿男以后可以看专栏。

总之我们知道了，有一个 pid 是 1 的 process，它是所有后续 process 的 parent，而它的 parent 的 id 是 0，启动后就自毁了。

对于很多初学者来讲，这个 parent process 克隆自己产生 children processes 的过程是很开脑洞的：首先 parent process 会复制自己，创建一个或多个新的 children processes，这些 children processes 里面的所有内容都和 parent process 一摸一样。

像什么？有点类似于 “单细胞分裂”。

生成的 child 里面对应的数据，代码，都是和 parent 一样的，也就是说在内存里面他们都是一样的。

[2]GRUB 和 LILO 是两个比较常用的 Boot Loader，用在同时安装了多个操作系统的机器上，供用户选择要启动哪个操作系统）

这个过程叫做 fork，在 Linux 世界里就是 fork() 函数做这件事，具体的细节阿男会在下篇文章中给大家讲。

那么问题来了，如果 parent 生成的 children 和自己一样，那岂不是所有的 process 都执行相同的 program 了?

但实际上我们的系统是在执行很多不同的 program 啊?

这个问题，大家可以先带在身上，阿男后续给大家讲。

感觉这篇文章讲的东西够大家消化一下了，下篇阿男继续带大家学习 process。

什么是进程

我们在上篇文章里，学习了计算机的启动过程，以及 Linux 是如何接管系统并且创建各个 pro cess 的过程。

我们知道了 process 就是 “活的” 程序，这些 processes 就是实际在运行的代码。

然后我们学习了 process 是如何产生的：从一个初始的 process，复制自身，产生了新的 proc esses。这个 process 把自己复制的过程叫做 fork，我们这篇文章里就具体看一下 fork() 函数的使用方法。

首先是 fork() 的定义：

```
FORK(2)                                                    Linux
Programmer's Manual
FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
```

```
    fork() creates a new process by duplicating the calling process.
The new process is referred to as the child process. The calling process is
      referred to as the parent process.
```

文档写的很清楚，fork() 函数的作用，就是把调用这个 fork() 函数的 process 自身，复制一份，成为自己的 child process，自己就是创建的这个 child process 的 parent process。

这个函数比较有意思的一点是，它会分别针对 parent process 和 child process 进行返回。如果 child 创建成功的话，则：

- fork() 会对 parent process 返回 child process 的 id
- fork() 会对 child process 返回 0

这样，我们在 process 中就可以通过判断 fork 的返回，来确定自己是 child 还是 parent 了。

接下来写段代码来实际使用一下 fork() 函数：

```c
/* exit() */
#include <stdlib.h>

#include <sys/types.h>

/*
 * fork()
 */
#include <unistd.h>

/*
 * stderr
 * stdout
 * fprintf
 */
#include <stdio.h>

void err_sys(const char* fmt, ...);

int main () {
```

```
    pid_t pid;
    if ((pid = fork()) < 0) { /* 创建 child process */
        err_sys("fork error");
    } else if (pid == 0) { /* 这里是 child process */
        puts("child process here.");
        puts("child process will sleep for 10 seconds...");
        sleep(10);
    } else { /* 这里是 parent process */
        puts("parent process here.");
        printf("parent get the child pid: %d\n", pid);
    }

    /* 父亲孩子共用的代码部分 */
    sleep(3);

    if (pid == 0) { /* child process */
        puts("child process exit.");
    } else { /* parent process */
        puts("parent process exit.");
    }

    return 0;
}

void err_sys(const char* fmt, ...) {
    va_list ap;
    fprintf(stderr, fmt, ap);
    exit(1);
}
```

这段代码的核心是这里：

```
pid = fork()
```

从上面这行代码开始，这个 process 就会创建出一个 child process 来，和自己一模一样，并且会接着往下执行，因此后续的代码都要考虑自己是 paren t process 还是 child process 了：

```
if (pid == 0)
```

如果 pid 为 0，那么自己就是 child process；如果不为 0，就是 parent process。

所以说此时我们这段代码已经分别在 parent 和 child process 里面运行着，我们的大脑也要 “分裂一下” 。接下来继续看代码，这个程序针对 child process 和 parent process 有不同的代码：

```
} else if (pid == 0) { /* 这里是 child process */
    puts("child process here.");
    puts("child process will sleep for 10 seconds...");
    sleep(10);
} else { /* 这里是 parent process */
    puts("parent process here.");
    printf("parent get the child pid: %d\n", pid);
}
```

如上代码所示，如果是 child，则 sleep 10 秒钟。如果是 parent，则打印出 child process 的 id (从 fork 返回得来)。接下来是 parent 和 child 都会执行的代码：

```
sleep(3);
```

sleep 3 秒，这样我们可以保证 parent process 和 child process 的执行顺序了：parent 和 child 都要睡眠 3 秒，但是 child process 还要多睡 10 秒钟，所以肯定是 parent 先退出，child 后退出。

我们在下篇文章中要用到这个执行顺序来讲解，所以要通过 sleep 来控制一下，到时候再详细说。接下来我们编译上面的代码：

```
cc process.c -o process
```

然后运行代码：

```
$ ./a.out
parent process here.
parent get the child pid: 85797
child process here.
child process will sleep for 10 seconds...
parent process exit.
```

从上面的输出，我们看到程序的整个执行过程。

Parent Process 与 Child Process

这篇文章我们接着学习 Process。我们学习了 fork() 函数，并且写了 sample code 来使用 fork() 函数创建一个 child process：

```
/* exit() */
#include <stdlib.h>

#include <sys/types.h>

/*
 * fork()
 */
#include <unistd.h>

/*
 * stderr
 * stdout
 * fprintf
 */
#include <stdio.h>

void err_sys(const char* fmt, ...);

int main () {
    pid_t pid;
    if ((pid = fork()) < 0) { /* 创建 child process */
        err_sys("fork error");
    } else if (pid == 0) { /* 这里是 child process */
        puts("child process here.");
        puts("child process will sleep for 10 seconds...");
        sleep(10);
    } else { /* 这里是 parent process */
        puts("parent process here.");
        printf("parent get the child pid: %d\n", pid);
    }
```

```
    /* 父亲孩子共用的代码部分 */
    sleep(3);

    if (pid == 0) { /* child process */
        puts("child process exit.");
    } else { /* parent process */
        puts("parent process exit.");
    }

    return 0;
}

void err_sys(const char* fmt, ...) {
    va_list ap;
    fprintf(stderr, fmt, ap);
    exit(1);
}
```

这篇文章里面，我们就仔细地分析一下这段代码的执行过程。首先把它编译成可执行文件：

```
[weli@fedora process]$ cc process.c -o process
```

编译完成后，我们把代码执行起来：

```
[weli@fedora process]$ ./process
parent process here.
parent get the child pid: 22418
child process here.
child process will sleep for 10 seconds...
```

然后马上用 ps 命令来查看 process 的运行情况：

```
[weli@fedora process]$ ps -ef | grep process | grep -v grep
weli     22417 18153  0 21:00 pts/2    00:00:00 ./process
weli     22418 22417  0 21:00 pts/2    00:00:00 ./process
```

如上所示，我们可以用 ps 看到 parent 和 child 两个 process。并且从日志和 ps 的

输出可以看出来，编号为 22418 的是 child process，它的 parent 是 id 为 22417 的 process。因为我们写的代码是让 parent process 先退出，然后 child 要多等 10 秒再推出，因此我们可以看看 parent process 退出后，child process 的状态是什么样子的：

```
[weli@fedora process]$ ps -ef | grep process | grep -v grep
weli     22418     1  0 21:00 pts/2    00:00:00 ./process
```

从上面的输出可以看到，如果我们等到 parent process 退出，而 child process 没有退出的时候，此时 ps 用查看，就可以正在运行的 process 只有一个了。而且我们看此时这个在运行的 child process，它的 ppid 变成了 1。也就是说，因为 parent process 已经退出不存在了，所以 1 号进程称为了 child process 的 parent，这是内核对 process 继承关系的管理办法。最后我们看一下，等到两个进程都执行完后的状态：

```
[weli@fedora process]$ ./process
parent process here.
parent get the child pid: 22618
child process here.
child process will sleep for 10 seconds...
parent process exit.
[weli@fedora process]$ child process exit.

[weli@fedora process]$ ps -ef | grep process | grep -v grep
```

可以看到，此时 ps 已经看不到 parent 和 child 两个 process 了，因为它们都执行完成了。

小结

我们在这篇文章里看到了 parent 和 child process 的执行过程。并且我们知道了，如果 parent process 先退出，那么还在执行的 child process 的 parent process 就变成了 1 号 process。下篇文章中，阿男给大家讲反过来的情况，就是 child process 先结束的情况。

僵尸进程

这篇文章里面，阿男仍然是接续上一篇的内容，带大家继续学习 Process。上一篇文章里我们使用了一段代码来展示了 parent process 和 child process 的关系：我们看到了 parent process 退出后，还在运行的 child process 的 parent 变成了 pid 为 1 的 process。那如果是 child process 先退出，会是什么情况呢？我们这篇文章就讨论这种情况。这次我们要用到的代码是这样的：

```
/* exit() */
#include <stdlib.h>

#include <sys/types.h>

/*
 * fork()
 */
#include <unistd.h>

/*
 * stderr
 * stdout
 * fprintf
 */
#include <stdio.h>

void err_sys(const char* fmt, ...);

int main () {
    pid_t pid;
    if ((pid = fork()) < 0) { /* create a process */
        err_sys("fork error");
    } else if (pid == 0) { /* child process */
        /* Child process 直接退出 */
        exit (0);
    } else {
        /* Parent process 睡眠 60 秒，保证 child process 先退出 */
```

```
        fputs("parent process goes to sleep...", stderr);
        sleep (60);
    }
    return 0;
}

// 错误处理函数
void err_sys(const char* fmt, ...) {
    /* va_list 可变长参数，使用 `man stdarg` 命令学习相关文档。*/
    va_list ap;
    fprintf(stderr, fmt, ap);
    exit(1);
}
```

可以看到，我们这次所使用的代码其实和之前使用的差不多，而最大的区别就是，这次的代码是 child process 先退出，然后 parent process 要 sleep 60 秒才退出。因此我们可以看看再这样的情况下，和 child process 先退出会有什么不同。首先我们编译这段代码：

```
[weli@fedora process]$ cc zombie.c -o zombie
```

然后我们运行编译后的代码：

```
[weli@fedora process]$ ./zombie
parent process goes to sleep...
```

此时我们使用 `ps` 命令查看这个程序的两个 process 的运行状况：

```
[weli@fedora process]$ ps -ef | grep zombie
weli      9203  5502  0 19:53 pts/0    00:00:00 ./zombie
weli      9204  9203  0 19:53 pts/0    00:00:00 [zombie] <defunct>
```

此时通过 pid 和 ppid 的关系可以看到，parent process 的 id 是 9203，child process 的 id 是 9204。此时 parent process 在正常运行的状态，而 child process 的状态是 `<defunct>`。也就是说，child process 虽然已经退出了，但是它并没有完全消失，而是还留下了一个 defunct 状态的 “壳”，我们管这种状态的进程叫做 zombie process，也就是僵尸进程。

为什么叫它 zombie process？因为这个 process 的资源已经已经被释放干净了，只是个没有生命力的空壳而已。我们可以使用 lsof 命令来验证这点：

```
[weli@fedora process]$ lsof -p 9204
[weli@fedora process]$
```

如上所示，我们使用 lsof 命令来查看这个 zombie process 所使用的文件资源，因为只要是在运行的 process，都会打开一些文件的，至少这个 process 自身所对应的程序文件是一定要打开的。我们可以对比一下使用 `lsof` 命令查看正在运行的 parent process 的情况：

```
[weli@fedora process]$ lsof -p 9203
COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
zombie  9203 weli  cwd    DIR  253,2     4096 523330
/home/weli/projs/kernel-learning/process
zombie  9203 weli  rtd    DIR  253,0     4096      2 /
zombie  9203 weli  txt    REG  253,2     8912 523335
/home/weli/projs/kernel-learning/process/zombie
zombie  9203 weli  mem    REG  253,0  2093616 795514 /usr/lib64/libc-2.23.so
zombie  9203 weli  mem    REG  253,0   172080 795403 /usr/lib64/ld-2.23.so
zombie  9203 weli    0u   CHR  136,0      0t0      3 /dev/pts/0
zombie  9203 weli    1u   CHR  136,0      0t0      3 /dev/pts/0
zombie  9203 weli    2u   CHR  136,0      0t0      3 /dev/pts/0
```

从上面的输出可以看到，正在运行的 parent process 至少在使用着自己所对应的程序文件`/home/weli/projs/kernel-learning/process /zombie`，而实际上已经退出的 child prorcess 就是一个空壳，实实在在就是一个 zombie！那么为什么 child process 执行完成后，内核还要保留这样一个 zombie process 呢？这个问题作为本篇文章的读后思考问题，阿男下篇文章为大家进行讲解。

为什么要有 zombie process?

阿男在上篇文章里给大家留了一个问题：child process 退出后，此时 parent process 还没退出，为什么 child process 此时要保留一个 zombie process?

答案是：这是为了进程间同步而考虑。

Linux 提供了一个函数叫做 `waitpid`:

```
WAIT(2)                                                    Linux
Programmer's Manual
```

```
WAIT(2)

NAME
       wait, waitpid, waitid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);
       pid_t waitpid(pid_t pid, int *status, int options);
```

这个函数可以让 caller 等待某一个进程（通过 pid 来识别）的状态改变，在没有检查到状态转变之前，caller 会一直 hold 在那里不往下执行。

我们可以思考一下，因为程序退出也是一种 “状态的改变”，因此保留一个 zombie process，这样如果有一个 waitpid 的 caller process 就可以捕获这个状态改变了。

对于退出的 child process 来讲，如果 parent process 想调用 waipid 来等待 child process 的退出，那么我们可以想一下，waitpid 在设计实现上肯定需要 child process 在退出后保留一个 zombie，而不是直接完全退出，这样 waitpid 才能检测这种退出状态的发生。最后，waitpid 是会负责回收 zombie process 的。

那么我们在之前，parent process 先退出，child process 后退出的情况下，为什么没有看到过 zombie process?

其实很简单，因为 parent process 的 parent 是 bash shell，parent process 退出后，bash shell 就处理掉相关的 zombie process 了。

另一方面，child process 因为在 parent process 退出后仍然在运行，我们之前学习了，这种情况下，id 为 1 的 process 会成为 child process 的 parent。因此，child process 退出后，id 为 1 的 process 就也把 child process 的 zombie 释放了。所以这种情况下并不是没有 zombie process，而是 zombie 被回收的过程是瞬间，所以我们在 `ps` 里面看不到。

只有当我们的 parent process 没退出，child process 已经退出的情况下，才会在 `ps` 的输出里看到 child process 的 zombie。因为内核预定由 parent process 负责回收 child process 的 zombie，这样 parent process 的 waitpid 才能捕获 child

process 的退出状态。

但如果我们的 parent process 一直不退出，也不回收 zombie，就会在 `ps` 的输出里面看到 child process 的 zombie。

这也是为什么服务器性质的代码往往会制造一堆 zombie。比如 http 服务器，加入服务程序长时间运行，生成了好多 children process，等 children 运行完成后又不回收，就会看到进程列表里面好多 zombie。这种情况下只能 kill 掉主进程，这样所有的 children 的 parent 就变成了 id 为 1 的 process，而 id 为 1 的 process 会把所有的 zombie 都释放掉。

在下一篇文章里，阿男给大家讲解 waitpid 的具体使用方法。