# fork函数

## 概述

Linux中提供了两种创建进程的方式，一种是fork函数，另一种是exec系统调用。
编程中，可以通过一下函数获取当前进程的pid，ppid：

```
pid_t getpid(void);
pid_t getppid(void);
```

## fork创建进程

```
/**
        函数原型
        #include <sys/types.h>
        #include <unistd.h>

        pid_t fork(void);

        参数说明：
             fork()的返回值会返回两次。一次是在父进程中，一次是在子进程中。
             在父进程中返回创建的子进程的ID，
             在子进程中返回0
             在父进程中返回-1，表示创建子进程失败，并且设置errno
             所以可以通过返回值判断当前程序是在子进程中还是父进程中。
             进程创建失败的两个主要原因：
                  1、系统中的进程数达到上限，errno设置为EAGAIN
                  2、内存空间不足，errno设置为ENOMEM

 */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid = fork(); //创建子进程

    if (pid > 0) //如果大于0，表示此时在当前进程中
    {
        printf("当前主进程创建了一个子进程pid ---> %d\n", pid);
        printf("当前进程是main主进程 --> pid = %d\n", getpid());
        printf("当前主进程的父进程是 --> ppid = %d\n", getppid());
    }
    else if (pid == 0) //如果=0，表示此时执行的是子进程
    {
        printf("这是fork创建的子进程 --> pid = %d\n", getpid());
        printf("子进程的父进程 --> ppid = %d\n", getppid());
    }

    //这是一段两个进程都会执行的代码
    for (size_t i = 0; i < 3; i++)
```

```
    {
        printf("%d\n", (int)i);
    }

    return 0;
}
```

执行结果如下：

```
kiko@Hkiko:~/lesson/myprocess$ ./myfork
当前主进程创建了一个子进程pid ---> 69677
这是fork创建的子进程 --> pid = 69677
当前进程是main主进程 --> pid = 69676
子进程的父进程 --> ppid = 69676
当前主进程的父进程是 --> ppid = 64033
0
0
1
1
2
2
kiko@Hkiko:~/lesson/myprocess$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
kiko       64033   61491  0 08:08 pts/9    00:00:00 /bin/bash
kiko       69714   64033  0 09:18 pts/9    00:00:00 ps -f
kiko@Hkiko:~/lesson/myprocess$
```



## 深入理解fork

- 早期fork的策略
  调用fork函数时，内核会复制所有的内部数据结构，复制进程的页表项，然后把父进程的地址空间复制到子进程的地址空间中，这种操作很耗时。
  备注：每一个进程都有一个虚拟地址空间。结构图如下：

- 写时复制策略
  为了避免复制时的系统开销。将前提假设的很简单：如果有多个进程要读取他们自己的那部分资源的副本，那么复制是不必要的。每个进程只保存一个指向这个资源的指针就可以了。
  只要没有一个进程修改自己的“副本”，每个进程就好像独占那个资源。如果某个进程想要修改那份资源，这是就要开始复制那份资源了，并把复制的副本提供给这个线程。这个进程之后就可以反复修改持有的这个副本，且对之前那一份资源没有任何影响。这就是写时复制---> 只有在进程写操作是才会复制。
- 写时复制好处
  如果进程从来没有修改过那份资源，则不进行复制，减少了系统开销。

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    int shareNum = 10; //一份共享资源

    pid_t pid = fork(); //创建子进程

    if (pid > 0)
    {
        //父进程
        printf("父进程 ---> %d\n", shareNum);
    }
    else if (pid == 0)
    {
        //子进程
        shareNum = 100;
        printf("子进程 ---> %d\n", shareNum);
    }

    return 0;
}
```

先执行子进程数据的修改，对父进程没有造成影响。

```
kiko@Hkiko:~/lesson/myprocess$ ./myfork1
子进程 ---> 100
父进程 ---> 10
```



- 总结

```
父子进程之间的关系：
        区别：
            1.fork()函数的返回值不同
                父进程中：>0 返回的子进程的ID
                子进程中：=0
            2.pcb中的一些数据
                当前的进程的id pid
                当前的进程的父进程的id ppid
                信号集

        共同点：
            某些状态下：子进程刚被创建出来，还没有执行任何的写数据的操作
```

- 用户区的数据
- 文件描述符表

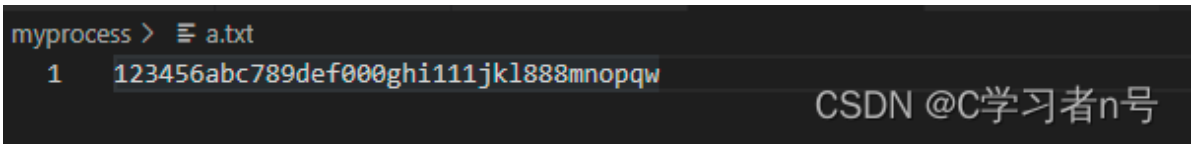
父子进程对变量是不是共享的？
- 刚开始的时候，是一样的，共享的。如果修改了数据，不共享了。
- 读时共享（子进程被创建，两个进程没有做任何的写的操作），写时拷贝。

## 文件描述符

fork产生的子进程与父进程相同的文件文件描述符指向相同的文件表，引用计数增加，共享文件偏移指针。
如下代码：共享文件偏移指针

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    //打开一个文件
    int fd = open("a.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);
    if (fd == -1)
        return 0;

    pid_t pid = fork(); //创建子进程

    if (pid > 0)
    {
        //父进程
        write(fd, "123", 3);
        write(fd, "456", 3);
        write(fd, "789", 3);
        write(fd, "OOO", 3);
        write(fd, "111", 3);
        write(fd, "888", 3);
    }
    else if (pid == 0)
    {
        //子进程
        write(fd, "abc", 3);
        write(fd, "def", 3);
        write(fd, "ghi", 3);
        write(fd, "jkl", 3);
        write(fd, "mno", 3);
        write(fd, "pqw", 3);
    }

    return 0;
}
```

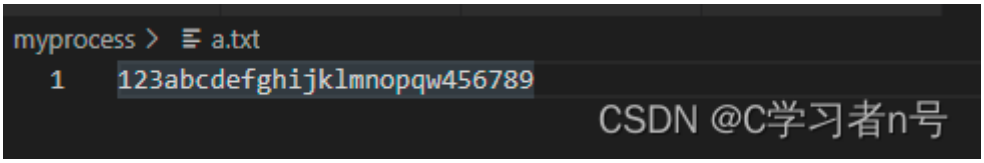
文件中的内容：




如下代码：一个进程关闭文件，不会影响其他的进程

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    //打开一个文件
    int fd = open("a.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);
    if (fd == -1)
        return 0;

    pid_t pid = fork(); //创建子进程

    if (pid > 0)
    {
        //父进程
        write(fd, "123", 3);
        write(fd, "456", 3);
        write(fd, "789", 3);
        close(fd);
        write(fd, "000", 3);
        write(fd, "111", 3);
        write(fd, "888", 3);
    }
    else if (pid == 0)
    {
        //子进程
        write(fd, "abc", 3);
        write(fd, "def", 3);
        write(fd, "ghi", 3);
        write(fd, "jkl", 3);
        write(fd, "mno", 3);
        write(fd, "pqw", 3);
    }

    return 0;
}
```

文件内容：




## GDB多进程调试

使用 GDB 调试的时候，GDB 默认只能跟踪一个进程，可以在 fork 函数调用之前，通过指令设置 GDB 调试工具跟踪父进程或者是跟踪子进程，默认跟踪父进程。

设置调试父进程或者子进程：`set follow-fork-mode [parent（默认）| child]`
查看调试父进程还是子进程：`show follow-fork-mode`

设置调试模式：`set detach-on-fork [on | off]`
查看调试模式：`show detach-on-fork`
默认为 on，表示调试当前进程的时候，其它的进程继续运行，如果为 off，调试当前进程的时候，其它进程被 GDB 挂起。

查看调试的进程：`info inferiors`
切换当前调试的进程：`inferior id`
使进程脱离 GDB 调试：`detach inferiors id`

# exce函数族

## exec函数族概述

1、exec函数族是一种不同于fork函数创建子进程的方式。它的本意不是创建新的进程，而是用exec指定的程序的虚拟地址空间去替换执行这个函数所在进程的虚拟地址空间。主要是用户区的替换。但是子进程的pid是不会变的。
2、exec函数族的函数主要是执行指定的可执行文件或者是shell命令。
3、该函数用法：首先在main的主进程中用fork函数创建一个子进程，然后在这个子进程中调用exec函数族的函数。当执行exec函数族的函数之后，当前这个子进程的进程映像就被exec指定的执行程序替换掉了。所以exec函数之后的代码都不会执行。
4、返回值介绍。函数族中的函数在执行成功的时候都不会有返回值，因为，成功之后进程映像就已经替换了，无法得到返回值。如果失败就会返回-1，并且设置对应errno值。
5、参数介绍。函数族中的函数主要有2中。
第一种就是指定的可执行文件或者是shell命令，这个参数在函数的第一个参数。
第二种就是指定程序执行时需要的参数，它是一个可变参数类型或者是字符串数组。如果是可变参数，最后需要加上NULL值作为参数结束的标志。
这种参数要说明一点，如果真的需要执行参数，那么它的第一个一般是设置成可执行文件名，他虽然没有什么意思，但是都是这么设置的。
其实还有第三种，这种是放在第一二种参数的后面，通常是字符串数组类型。

```
> 头文件：#include <unistd.h>
> 函数原型：
> int execl(const char*path, const char *arg, ...);
> int execv(const char*path, const char *const arg[]);
> int execle(const char*path, const char *arg, ..., char *const envp[]);
> int execve(const char*path, const char *const arg[], char *const envp[]);
> int execlp(const char*file, const char *arg, ...);
> int execvp(const char*file, const char *const arg[]);
> int execvpe(const char * file, char * cosnt argv[], char * const envp[]);
```

函数名前四位都是exec，标识了函数族exec；
1、第五位标识的是可执行文件或者shell命令所需要的参数是按照什么形式传入的，l（小写L）表示按照list方式传入，也就是可边长的参数；如果是v，那就是按照字符串数组的形式传入。
2、第六位，不是每个函数都有。如果是p，那么就到PATH环境变量中找到第一个参数指定的可执行文件或者是shell命令。如果是e，那么这个函数就一定有第三种参数，它是多个路径的字符串数组，表示它会去指定的这些位置找可执行文件或者是shell命令。
3、还有一点，第一个参数都是可执行文件或者是shell命令的文件名或者是路径+文件名。如果形参是

path，那么就是路径+文件名，如果是file，那么直接写文件名就可以了，这种情况一般是函数名有p或者e的函数中才有，因为它们不需要根据路径找到可执行文件或者shell命令，而是直接搜索环境变量。

总结：exec函数族中的函数执行的功能都是一样的，只是具体的方式有所不同，主要是直接指定文件名还是路径名；是通过可变参数指定执行参数还是通过字符串数组指定执行参数。

## 常用函数之execl

准备一个可执行文件

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("这是一个在子进程里面调用的可执行文件，所在进程为--> %d\n", getpid());

    for (int i = 0; i < argc; i++)
    {
        printf("参数 %d --> %s\n", i, argv[i]);
    }

    return 0;
}
```

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    //创建一个子进程
    pid_t pid = fork();

    if (pid > 0)
    {
        //父进程
        printf("这是一个父进程  --> %d\n", pid);
    }
    else if (pid == 0)
    {
        //子进程
        //执行可执行文件
        // int ret = execl("a", "a", "Hello", "World", NULL); // execl函数
        //执行shell命令
        int ret = execl("/usr/bin/ps", "ps", "-aux", NULL);

        if (ret < -1)
        {
            perror("execl");
        }
        //这句代码不会被执行，因为execl如果执行成功，那么这个子进程的虚拟地址空间的用户区就会被上面的a可执行文件替换掉
        //当前这个子进程的数据就被清理了，只会执行可执行文件的代码
        printf("子进程 ---");
    }
}
```

总结：execl函数是通过可变长参数指定可执行参数的，第一个参数需要指定可执行文件或者shell命令的路径，推荐绝对路径。

## 常用函数之execlp

会到环境变量中查找指定的可执行文件，如果找到了就执行，找不到就执行不成功。

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    //创建一个子进程
    pid_t pid = fork();

    if (pid > 0)
    {
        //父进程
        printf("这是一个父进程  --> %d\n", pid);
    }
    else if (pid == 0)
    {
        //子进程
        //执行可执行文件
        // int ret = execlp("a", "a", "Hello", "World", NULL); // execlp函数,a可执行文件不在PATH环境变量中是执行不成功的。
        int ret = execlp("ps", "ps", "-aux", NULL);

        if (ret < -1)
        {
            perror("execl");
        }
    }
}
```

## 其他函数

```
int execv(const char *path, char *const argv[]);
// argv是需要的参数的一个字符串数组
char * argv[] = {"ps", "aux", NULL};
execv("/bin/ps", argv);
```

```
int execve(const char *filename, char *const argv[], char *const envp[]);
char * envp[] = {"/home/nowcoder", "/home/bbb", "/home/aaa"}
```