Linux进程

本文会简单讲述进程创建和进程间通信

简单说一下，进程是操作系统调度资源的基本单位，比如将可执行程序跑起来就变成了进程。

进程是由数据(全局变量、静态变量、只读变量...)、代码和堆栈(局部变量、动态分配的内存...)组成的。

进程的模式有用户模式、内核模式。

可以输入ps命令来查看正在运行的进程

```
NAME
       ps - report a snapshot of the current processes.

SYNOPSIS
       ps [options]
```

```
hwx@N3ptune:~$ ps
  PID TTY          TIME CMD
18186 pts/1    00:00:00 bash
18215 pts/1    00:00:00 ps
```

ps -aue命令可以查看更为详细的信息

```
hwx@N3ptune:~$ ps -aue
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      4396  0.4  0.6 1242268 98548 tty1     Ssl+ 16:54   0:46 /usr/lib/xorg/Xorg -background none :0 -seat seat0 -auth /var/ru
root      5940  0.0  0.0   9980  1844 pts/0    Ss+  16:55   0:00 /init --second-stage
1036      5966  0.0  0.0  16352  3012 pts/0    SNl  16:55   0:00 /system/bin/logd
root      5967  0.0  0.0   7164  2624 pts/0    S    16:55   0:00 /system/bin/debuggerd
root      5968  0.0  0.0  10712  2864 pts/0    S    16:55   0:00 /system/bin/debuggerd64
root      5969  0.0  0.0  18952  6564 pts/0    Sl   16:55   0:00 /system/bin/vold --blkid_context=u:r:blkid:s0 --blkid_untrusted_
root      5977  0.0  0.0  10456   556 pts/0    S    16:55   0:00 debuggerd64:signaller
root      5979  0.0  0.0   6908   428 pts/0    S    16:55   0:00 debuggerd:signaller
root      5984  0.0  0.0   7624   892 pts/0    S    16:55   0:00 /sbin/healthd
root      5986  0.0  0.0  22092  4636 pts/0    Sl   16:55   0:00 /system/bin/anboxd
root      5987  0.0  0.0   9332  3236 pts/0    SL   16:55   0:00 /system/bin/lmkd
hwx       5988  0.0  0.0  10180  2620 pts/0    S    16:55   0:00 /system/bin/servicemanager PATH=/sbin:/vendor/bin:/system/sbin:/
hwx       5989  0.0  0.0  53044 15592 pts/0    S<l  16:55   0:00 /system/bin/surfaceflinger
root      5990  0.0  0.0   8328  2728 pts/0    S    16:55   0:00 /system/bin/sh /system/bin/dprocess.sh
root      5991  0.0  0.8 2177420 131832 pts/0  Sl   16:55   0:01 zygote64
root      5992  0.0  0.7 2161716 120576 pts/0  Sl   16:55   0:01 zygote
1041      5993  0.0  0.0  29572  9164 pts/0    Sl   16:55   0:00 /system/bin/audioserver
1019      5995  0.0  0.0  19580  7528 pts/0    Sl   16:55   0:00 /system/bin/drmserver
root      5996  0.0  0.0  10400  2792 pts/0    S    16:55   0:00 /system/bin/installd
1017      5997  0.0  0.0  14496  4980 pts/0    S    16:55   0:00 /system/bin/keystore /data/misc/keystore
1046      5998  0.0  0.1  33280 18748 pts/0    Sl   16:55   0:00 media.codec mediacodec
```

top命令可以实时查看正在运行的程序

```
NAME
       top - display Linux processes

SYNOPSIS
       top -hv|-bcEHiOSs1 -d secs -n max -u|U user -p pid -o fld -w [cols]

       The traditional switches `-' and whitespace are optional.
```

在终端输入top，输入q可以退出

```
top - 19:39:14 up  2:44,  1 user,  load average: 0.93, 0.57, 0.30
Tasks: 380 total,   1 running, 379 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.4 us,  0.4 sy,  0.0 ni, 99.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15875.1 total,  11305.9 free,   1526.0 used,   3043.2 buff/cache
MiB Swap:  16384.0 total,  16384.0 free,      0.0 used.  13490.2 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 6089 hwx       20   0  571968  51704  42616 S   6.2   0.3   1:28.97 deepin-system-m
    1 root      20   0  167216  11880   7864 S   0.0   0.1   0:02.48 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
    5 root      20   0       0      0      0 I   0.0   0.0   0:00.78 kworker/0:0-events
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_highpri
    9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
   10 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
   11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
   12 root      20   0       0      0      0 S   0.0   0.0   0:00.05 ksoftirqd/0
   13 root      20   0       0      0      0 I   0.0   0.0   0:04.02 rcu_sched
```

Linux系统上可以管理多个进程，进程被分时分片处理

下面演示在程序中如何创建进程:

1. system函数可以在命令行中执行命令，可以借此来创建一个进程，不作赘述

```
NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);

DESCRIPTION
       The system() library function uses fork(2) to create a child process that executes the shell command specified in com-
       mand using execl(3) as follows:

           execl("/bin/sh", "sh", "-c", command, (char *) 0);
```

滚动截图

2. fork函数、vfork函数 创建子进程: 当前进程是父进程，被创建进程是子进程，创建完后父子进程通知执行

```
NAME
       fork - create a child process

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

3. exec家族

```
NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *path, const char *arg, ...
                       /* (char  *) NULL */);
       int execlp(const char *file, const char *arg, ...
                       /* (char  *) NULL */);
       int execle(const char *path, const char *arg, ...
                       /*, (char *) NULL, char * const envp[] */);
       int execv(const char *path, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                       char *const envp[]);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       execvpe(): _GNU_SOURCE
```

这些函数都可以创建一个进程，暂且不展开描述

进程有很多种状态，例如运行时、睡眠、挂起、等待、死亡、僵尸...

下面代码演示

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

int main(void)
{
    printf("This is parent process,pid: %d\n",getpid());
    sleep(2);
    printf("Create child process\n");
    int ret = fork();
    if (ret) {
        printf("This is parent process: %d\n",getpid());
        while(1) sleep(1);
    }
    else {
        printf("This is child process: %d\n",getpid());
        while(1) sleep(1);
    }
    return 0;
}
```

运行这个程序，程序已经打印出了进程号

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ gcc fork.c -o fork
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ ./fork
This is parent process,pid: 31185
Create child process
This is parent process: 31185
This is child process: 31188
```

这里有必要明确一下fork函数的返回值

```
RETURN VALUE
       On success, the PID of the child process is returned in the parent, and 0 is returned in the child.  On failure, -1 is
       returned in the parent, no child process is created, and errno is set appropriately.
```

根据fork函数的返回值来看是在子进程中还是父进程中，调用成功的话，父进程中返回值是子进程的ID，子进程中是0。实际上这里fork返回了两次。

fork做了两件事，第一件是复制结构，将父进程的数据结构都复制了一份。

第二件事就是唤醒新进程，让子进程运行起来

运行如下代码:

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

int main(void)
{
    printf("This is parent process,pid: %d\n",getpid());
    sleep(2);
    printf("Create child process\n");
    int ret = fork();
    int n = 10;
    if (ret) {
        printf("This is parent process: %d\n",getpid());
        while(1) {
            printf ("n = %d\n",n++);
            sleep(1);
        }
    }
    else {
        printf("This is child process: %d\n",getpid());
        while(1) {
            printf("n = %d\n", n += 2);
            sleep(1);
        }
    }
    return 0;
}
```

在代码中定义了一个变量n，初始值为10，子进程的n同样初始为10

```
This is parent process: 2042
n = 10
This is child process: 2045
n = 12
n = 14
n = 11
n = 16
n = 12
```

下面谈谈僵尸进程

有一种情况，父进程创建了子进程，父进程先于子进程结束，子进程资源没有被释放，就会变成僵尸进程，持续占用系统资源(内核中进程树会保存进程的数据，树中节点会保存进程的一些数据)。

子进程结束前，会向父进程发送SIGCHILD信号，父进程收到信号后，回收子进程资源，然后父进程再结束。父进程可以写一个wait函数，等待子进程发送SIGCHILD信号

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

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

       pid_t wait(int *wstatus);

       pid_t waitpid(pid_t pid, int *wstatus, int options);

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                      /* This is the glibc and POSIX interface; see
                         NOTES for information on the raw system call. */
```

如下是代码演示

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

int main(void) {
    if (fork()) {
        printf("create child process\n");
        wait(0);
    }
    else {
        for(int i=0;i<5;i++) {
            printf("child process: %d\n",i);
            sleep(1);
        }
    }
    printf("end parent process\n");
    return 0;
}
```

使用wait函数就是要等待子进程打印完所有数字，父进程才结束

最后看看守护进程

守护进程是一个独立的进程，最常见的用途就是记录其他进程的情况，保存系统日志

终端输入ps axj可以查看系统中的守护进程

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ ps axj
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     1     1     1 ?           -1 Ss       0   0:02 /sbin/init splash
    0     2     0     0 ?           -1 S        0   0:00 [kthreadd]
    2     3     0     0 ?           -1 I<       0   0:00 [rcu_gp]
    2     4     0     0 ?           -1 I<       0   0:00 [rcu_par_gp]
    2     6     0     0 ?           -1 I<       0   0:00 [kworker/0:0H-events_highpri]
    2     9     0     0 ?           -1 I<       0   0:00 [mm_percpu_wq]
    2    10     0     0 ?           -1 S        0   0:00 [rcu_tasks_rude_]
    2    11     0     0 ?           -1 S        0   0:00 [rcu_tasks_trace]
    2    12     0     0 ?           -1 S        0   0:00 [ksoftirqd/0]
    2    13     0     0 ?           -1 I        0   0:10 [rcu_sched]
    2    14     0     0 ?           -1 S        0   0:00 [migration/0]
    2    15     0     0 ?           -1 S        0   0:00 [idle_inject/0]
    2    16     0     0 ?           -1 S        0   0:00 [cpuhp/0]
    2    17     0     0 ?           -1 S        0   0:00 [cpuhp/1]
    2    18     0     0 ?           -1 S        0   0:00 [idle_inject/1]
    2    19     0     0 ?           -1 S        0   0:00 [migration/1]
    2    20     0     0 ?           -1 S        0   0:00 [ksoftirqd/1]
```

TPGID为-1的话，就说明是守护进程

如果要把一个进程变成守护进程，要先kill其父进程，同时摆脱终端的控制

要摆脱终端的控制，就要关闭三个文件描述符号：标准输入设备，标准输出设备，标准错误输出设备，然后重定向当前进程IO操作到/dev/null (黑洞设备)。然后要创建新的会话，摆脱原有会话进程组的控制。

这里要提到进程的组织形式：多个进程组成一个进程组，多个进程组组成一个会话。这里不详细解释会话是什么。

守护进程创建编程模型：

第一种:

1. 创建新会话 setsid
2. 改变当前工作目录 chdir
3. 重设当前文件权限 umask
4. 关闭文件 fclose

第二种：

1. 重设文件权限 umask
2. 创建子进程 fork
3. 结束父进程
4. 创建新会话 setsid

5. 防止子进程成为僵尸进程 忽略SIGCHILD SIGUP信号

6. 改变当前工作目录 chdir

7. 重定向文件描述符号 open dup(fd,0) dup(fd,1)

下面演示创建守护进程

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>
#include <stdbool.h>
#include <fcntl.h>
#include <stdlib.h>

int main(void)
{
    // 重设文件权限
    umask(0);
    // 创建子进程 结束父进程
    int ret = fork();
    if (ret < 0) {
        printf("create process failed: %m\n");
        exit(-1);
    }
    if (ret > 0) {
        printf("parent process end\n");
        exit(0);
    }
    if (0 == ret) {
        printf("pid: %d\n",getpid());
        // 创建新会话
        setsid();
        // 防止子进程成为僵尸进程 忽略SIGCHILD、SIGUP信号
        signal(SIGCHLD,SIG_IGN);
        signal(SIGHUP,SIG_IGN);
        // 改变当前工作目录
        chdir("/");
        // 重定向文件描述符号 open、dup函数
        int fd = open("/dev/null",O_RDWR);
        dup2(fd,0);
        dup2(fd,1);
    }
    //模拟守护进程工作
    while (true) {
        sleep(1);
    }
    return 0;
}
```

在终端中编译运行

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ gcc daemon.c -o daemon
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ ./daemon
parent process end
pid: 24902
```

现在关闭终端，在命令行输入ps -axj，可以看到运行起来的守护进程还在运行

```
    1 24902 24902 24902 ?           -1 Ss    1000   0:00 ./daemon
    2 24914     0     0 ?           -1 I        0   0:00 [kworker/10:1-events]
    2 24917     0     0 ?           -1 I        0   0:00 [kworker/6:2]
    2 24952     0     0 ?           -1 I        0   0:00 [kworker/3:0-events]
    2 25014     0     0 ?           -1 I        0   0:00 [kworker/2:0-events]
 5438 25117  5468  5468 ?           -1 Sl    1000   0:00 /usr/lib/deepin-daemon/soundeffect
23929 25217 25217 23929 pts/1    25217 R+    1000   0:00 ps -axj
```

这里要用kill杀掉进程

接下来谈进程间通信

顾名思义，要实现不同进程间进行传递信息。

这里可以分为两大类，一个是不同主机上的进程间进行通信(网络通信)，另一个是同一主机上的进程进行通信。

先谈同一主机上进程通信，此时又可以分为两类：父子进程间通信和非父子进程间通信。

上文提到，fork子进程会拷贝父进程的数据，因此父子进程间通信还是比较简单的。

第一种通信方式，较为朴素，使用普通文件，进程A将要传递的信息放入这个文件，进程B再去读这个文件即可。父子进程间可通过文件描述符号，非父子进程之间就只能通过具体文件来通信。

第二种方式，文件映射虚拟内存 mmap

第三种方式，匿名管道(父子间进程使用)和有名管道(非父子进程间使用)

除此之外，还有信号、共享内存、消息队列、信号量和网络可用于通信。

本文主要讲前3种

下面代码简单演示了第一种

```c
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>

// 通过文件进行通信
// 父进程往文件里写入数据
// 子进程读出来并打印

int main(void)
{
    int fd = open("test.dat",O_CREAT,0666);
    if (-1 == fd) {
        printf("创建文件失败\n");
        exit(-1);
    }
    printf("创建文件成功\n");
    if (fork())
    {
        int n = 0;
        while(true) {
            int fd = open("test.dat",O_WRONLY);
            write(fd,&n,4);
            close(fd);
            sleep(1);
            n++;
        }
    }
    else
    {
        int m;
        while(true) {
            sleep(1);
            int fd = open("test.dat",O_RDONLY);
            read(fd,&m,4);
            close(fd);
            printf(">> %d\n",m);
        }
    }
}
```

运行程序，可以看见子进程将父进程写入文件的数据都打印了出来

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ ./file
创建文件成功
>> 1
>> 1
>> 2
>> 3
>> 4
>> 5
>> 7
>> 7
>> 8
>> 9
>> 10
>> 11
>> 12
>> 13
>> 14
>> 15
>> 16
>> 17
```

下面讨论管道

管道也是FIFO结构，分为两种，匿名管道和有名管道。

父子进程使用匿名管道

1. 创建文件描述符号
2. 将文件描述符号变成管道

```
NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);
```

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

// 使用匿名管道
int main(void)
{
    // 创建管道描述符号
    int fd[2];
    int ret = pipe(fd);
    if (ret == -1) {
        printf("创建管道失败\n");
        exit(-1);
    }
    printf("创建管道成功\n");
    // 父子进程之间使用管道
    if (fork()) {
        char buff[1024] = {0};
        while(true) {
            scanf("%s",buff);
            write(fd[1],buff,strlen(buff));
        }
    }
    else {
        char temp[1024] = {0};
        while(true) {
            ret = read(fd[0],temp,1023);
            if (ret > 0) {
                temp[ret] = 0;
                printf(">> %s\n",temp);
            }
        }
    }
    close(fd[0]);
    close(fd[1]);
    return 0;
}
```

上述代码的功能是父进程接收用户输入后传入管道，子进程从管道读出并显示

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ ./pipe
创建管道成功
hello
>> hello
bye
>> bye
```

接着演示有名管道，流程如下：

进程A	进程B
创建管道文件 mkfifo	
打开管道文件	打开管道文件
往管道文件写入数据	从管道文件读取数据
关闭管道	关闭管道
删除管道文件	

```
NAME
       mkfifo, mkfifoat - make a FIFO special file (a named pipe)

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

       int mkfifo(const char *pathname, mode_t mode);

       #include <fcntl.h>           /* Definition of AT_* constants */
       #include <sys/stat.h>

       int mkfifoat(int dirfd, const char *pathname, mode_t mode);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       mkfifoat():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _ATFILE_SOURCE
```

进程A的代码

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

int main(void)
{
    // 创建管道文件
    int ret = mkfifo("test.pipe",0);
    if (-1 == ret) {
        printf("创建管道文件失败:%m\n");
        exit(-1);
    }
    printf("创建管道文件成功\n");
    // 打开管道文件
    int fd = open("test.pipe",O_WRONLY);
    if (-1 == fd) {
        printf("打开管道文件失败:%m\n");
        unlink("test.pipe");
        exit(-1);
    }
    printf("打开管道文件成功\n");
    // 循环写入
    int n = 0;
    char buff[1024] = {0};
    while(true) {
        sprintf(buff,"Hello Linux %d",n++);
        write(fd,buff,strlen(buff));
        sleep(1);
    }
    close(fd);
    unlink("test.pipe");
    return 0;
}
```

进程B

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

int main(void)
{
    // 打开管道文件
    int fd = open("test.pipe",O_RDONLY);
    if (-1 == fd) {
        printf("打开管道文件失败:%m\n");
        unlink("test.pipe");
        exit(-1);
    }
    printf("打开管道文件成功\n");
    // 循环读取
    while (true) {
        char buff[1024] = {0};
        int ret = read(fd,buff,1023);
        if (ret > 0) {
            buff[ret] = 0;
            printf(">>> %s\n",buff);
        }
    }
    close(fd);
    unlink("test.pipe");
    return 0;
}
```

运行结果

```
hwx@N3ptune:~/cprogram/testc/linux-process$ sudo ./fifoA
创建管道文件成功
打开管道文件成功
```

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ ls -l
总用量 80
-rw-r--r-- 1 hwx  hwx   1009 4月    1 11:05 daemon.c
-rwxr-xr-x 1 hwx  hwx  16856 4月    1 17:49 fifoA
-rw-r--r-- 1 hwx  hwx    852 4月    1 17:48 fifoA.c
-rwxr-xr-x 1 hwx  hwx  16672 4月    1 17:50 fifoB
-rw-r--r-- 1 hwx  hwx    679 4月    1 17:51 fifoB.c
-rw-r--r-- 1 hwx  hwx    857 4月    1 15:31 file.c
-rw-r--r-- 1 hwx  hwx    549 3月   31 20:54 fork.c
-rw-r--r-- 1 hwx  hwx  11703 4月    1 15:54 Linux进程.md
-rw-r--r-- 1 hwx  hwx    857 4月    1 15:32 pipe.c
p--------- 1 root root     0 4月    1 17:51 test.pipe
-rw-r--r-- 1 hwx  hwx    316 3月   31 21:21 wait.c
```

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ sudo ./fifoB
打开管道文件成功
>>> Hello Linux 0
>>> Hello Linux 1
>>> Hello Linux 2
>>> Hello Linux 3
>>> Hello Linux 4
>>> Hello Linux 5
>>> Hello Linux 6
```

这里要注意的是，这个程序不能在共享的文件夹下运行，因为共享文件夹下不能创建管道。同时必须两个进程都打开，这个程序才能返回，否则会阻塞。先关闭读取端，会导致写入端结束而先关写入端，不会对读取端造成影响。