第六章 进程控制

冯 晶 E-mail: gfeng@whu.edu.cn

1.进程的建立与运行

- 进程标识

  每个进程都有一个非负整型的唯一进程ID。

一些特殊进程：

进程ID 0	调度进程，常常被称为交换进程；
进程ID 1	通常是init进程，在启动时由内核调用；

```
top - 20:12:32 up 7 min,  1 user,  load average: 0.01, 0.19, 0.15
Tasks:  78 total,   1 running,  77 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  1.0%sy,  0.0%ni, 99.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    255260k total,   171692k used,    83568k free,    10852k buffers
Swap:   524280k total,        0k used,   524280k free,   104720k cached

_
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2479	root	20	0	2272	968	784	R	1.0	0.4	0:01.05	top
1855	root	20	0	2412	836	700	S	0.3	0.3	0:00.90	vmware-guestd
1	root	20	0	2112	656	564	S	0.0	0.3	0:00.79	init
2	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
4	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
5	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
6	root	15	-5	0	0	0	S	0.0	0.0	0:00.02	events/0
7	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	khelper
58	root	15	-5	0	0	0	S	0.0	0.0	0:00.03	kblockd/0
61	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kacpid
62	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kacpi_notify
130	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	cqueue/0
132	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	ksuspend_usbd
137	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	khubd
140	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kseriod
175	root	20	0	0	0	0	S	0.0	0.0	0:00.00	pdflush
176	root	20	0	0	0	0	S	0.0	0.0	0:00.02	pdflush

1.进程的建立与运行

- 获取进程信息

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

没有出错返回

```
pid_t    getpid(void);
pid_t    getppid(void);
uid_t    getuid(void);
uid_t    geteuid(void);
gid_t    getgid(void);
gid_t    getegid(void);
```

返回：调用进程的进程ID
返回：调用进程的父进程ID
返回：调用进程的实际用户ID
返回：调用进程的有效用户ID
返回：调用进程的实际组ID
返回：调用进程的有效组ID

1.进程的建立与运行

- 创建新进程

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

pid_t    fork(void);
```

调用一次，返回两次：在子进程中返回0；在父进程中返回子进程的进程ID；

出错则返回-1

1.进程的建立与运行

- 创建新进程

为什么子进程中只需返回0，而无需返回父进程的进程ID？

子进程中可以调用getppid()函数来获得父进程的进程ID

为什么在父进程中要返回子进程的进程ID？

一个父进程可能有多个子进程，所以在调用fork()函数创建新进程时就需要保存新创建的子进程的进程ID

1.进程的建立与运行

- 创建新进程

子进程和父进程继续执行fork()之后的命令；

子进程是父进程的复制品，子进程获得父进程的数据空间、堆、栈的复制品；

现在很多的实现并不做一个父进程数据段和堆的完全copy，只对需要修改的数据段或堆进行copy;

```
#include    <sys/types.h>
#include    "ourhdr.h"

int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* child */
        glob++;                 /* modify variables */
        var++;
    } else
        sleep(2);               /* parent */

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

1.进程的建立与运行

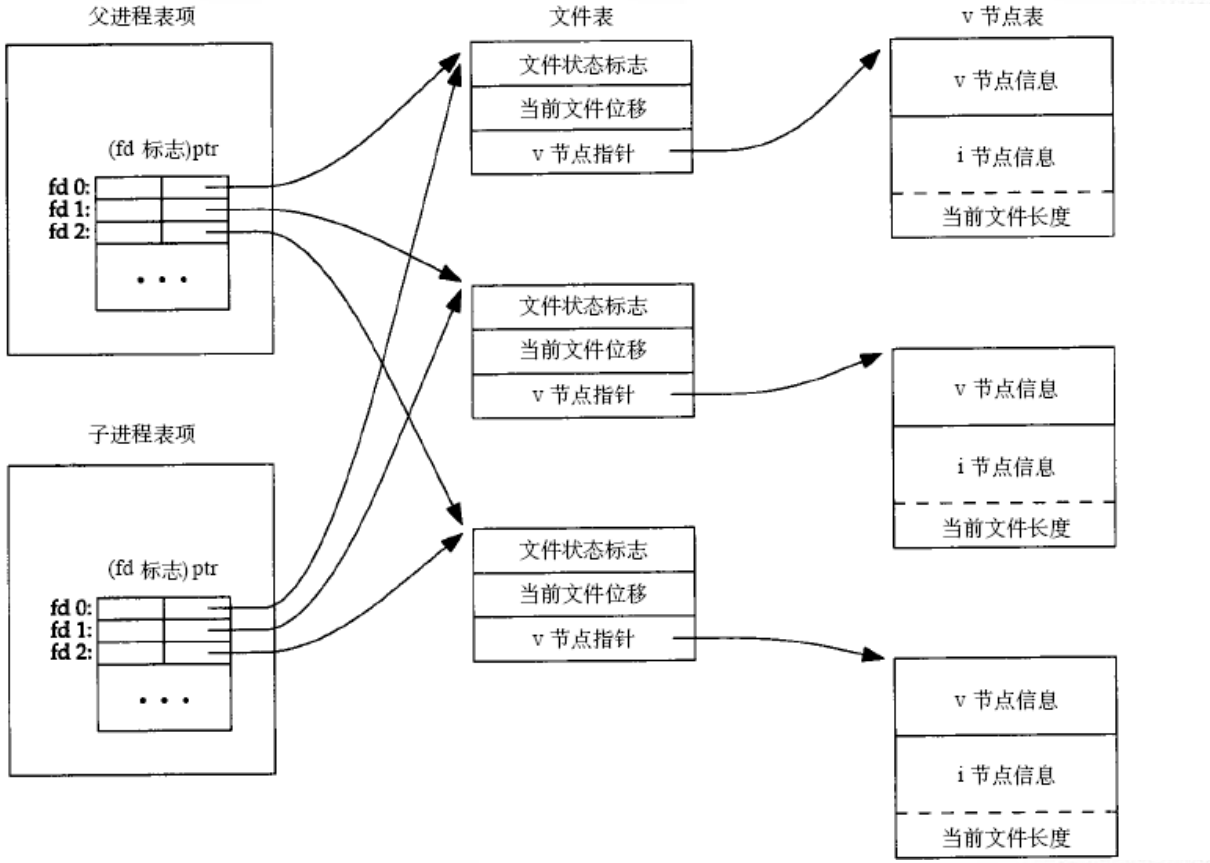
- 创建新进程

一般来说，在fork()之后是父进程先执行还是子进程先执行是不确定的，这取决与内核使用的调度算法；

```
$ a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89子进程的变量值改变了
pid = 429, glob = 6, var = 88父进程的变量值没有改变
```

1.进程的建立与运行

- 父、子进程对打开文件的共享



1.进程的建立与运行

- 父、子进程对打开文件的共享

在fork()之后处理文件描述符有两种常见的情况：

1. 父进程等待子进程完成；

2. 父、子进程各自执行不同的程序段，父、子进程各自关闭不需要使用的文件描述符，而不干扰对方使用的文件描述符。这在网络服务进程中经常使用。

1.进程的建立与运行

- fork()失败

使fork()失败的主要原因：

1. 系统已经有了太多的进程，资源不够；

2. 实际用户ID的进程总数超过了系统限制；

1.进程的建立与运行

- 何时需要使用fork()？

1. 一个进程希望复制自己，使父、子同时执行不同的代码段，这在网络服务进程中最常见；

2. 一个进程要执行一个不同的程序，这对shell是常见的情况，子进程从fork()返回后立即调用exec()来执行另一个程序；

2.进程的控制

- 进程的终止

正常终止：

1. 在main()函数内执行return语句；
2. 调用exit()函数；
3. 调用_exit()系统调用函数，此函数由exit()调用，处理UNIX特定的细节；

异常终止：

1. 调用abort()，它产生SIGABRT信号；
2. 当进程接收到某个信号时；（指针越界，除0错误等）

2.进程的控制

- 进程的终止

对于任何一种终止情形，最后都会执行内核中的同一段代码，来为相应进程关闭所有打开的文件描述符，释放所有的寄存器等等；

对于正常终止，可以通过获取退出状态参数（exit status）来实现；

对于异常终止，内核会产生一个指示其异常的终止状态；

2.进程的控制

- 进程的终止
  - 对于任意一种进程终止状态，父进程都希望子进程能够通知它是如何终止的，父进程能够用wait()或waitpid()函数取得其终止状态；
  - 一个已经终止的，但是其父进程尚未对其进行善后处理（如获取子进程的有关信息、释放它仍占用的资源）的进程被称为僵尸进程（zombie）
  - 僵尸进程占用资源很少，但是会占用内核的进程表等资源，过多的僵尸进程会导致系统有限数目的进程表被用光，无法创建新的进程。

```
top - 20:12:32 up 7 min,  1 user,  load average: 0.01, 0.19, 0.15
Tasks:  78 total,   1 running,  77 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  1.0%sy,  0.0%ni, 99.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    255260k total,   171692k used,    83568k free,    10852k buffers
Swap:   524280k total,        0k used,   524280k free,   104720k cached

_
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2479	root	20	0	2272	968	784	R	1.0	0.4	0:01.05	top
1855	root	20	0	2412	836	700	S	0.3	0.3	0:00.90	vmware-guestd
1	root	20	0	2112	656	564	S	0.0	0.3	0:00.79	init
2	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
4	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
5	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
6	root	15	-5	0	0	0	S	0.0	0.0	0:00.02	events/0
7	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	khelper
58	root	15	-5	0	0	0	S	0.0	0.0	0:00.03	kblockd/0
61	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kacpid
62	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kacpi_notify
130	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	cqueue/0
132	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	ksuspend_usbd
137	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	khubd
140	root	15	-5	0	0	0	S	0.0	0.0	0:00.00	kseriod
175	root	20	0	0	0	0	S	0.0	0.0	0:00.00	pdflush
176	root	20	0	0	0	0	S	0.0	0.0	0:00.02	pdflush

2.进程的控制

- 进程的终止
  - 系统调用wait()就是等待当前进程的任一个子进程终止，获取子进程终止的状态。wait()系统调用执行完之后，一个已死亡的僵尸子进程不再存在。wait()系统调用的最重要功能是销毁子进程的僵尸，得到子进程终止的状态并回收僵尸子进程占用的系统资源。
  - 如果执行wait()时，还没有子进程终止，那么，父进程会进入睡眠状态等待，直到有一个子进程终止。如果执行wait()时，已经有子进程终止，那么，wait会立刻返回。

2.进程的控制

- 进程的终止

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

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

成功返回终止子进程的进程ID
出错则返回-1

2.进程的控制

- 进程的终止

  子进程的终止有两种情况，

  – 一种是进程自愿终止（自杀），在程序中调用函数exit()或者在main()函数中的return ；

  – 另一种情况是异常终止（被杀），其他进程或者操作系统内核向进程发送信号将进程杀死。

第一种情况可以获得进程正常终止的返回码，第二种情况，可以获得进程被杀死的信号值。

2.进程的控制

- 进程的终止

调用wait()或waitpid()的进程可能会：

1. 阻塞（如果其所有子进程都还在运行）；
2. 带子进程的终止状态立即返回（如果一个子进程已终止，正等待父进程存取其终止状态）；
3. 出错立即返回（如果它没有任何子进程）；

```
int
main(void)
{
    pid_t   pid;
    int     status;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        abort();                /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        status /= 0;            /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

正常终止子进程为真

异常终止子进程为真

暂停子进程为真

2.进程的控制

- waitpid() vs wait()

1. waitpid()等待一个特定的进程，而wait()则返回任意一个终止子进程的状态；

2. waitpid()提供一个wait的非阻塞版本；

3. waitpid()支持作业控制；

```c
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {          /* first child */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /* We're the second child; our parent becomes init as soon
           as our real parent calls exit() in the statement above.
           Here's where we'd continue executing, knowing that when
           we're done, init will reap our status. */

        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)    /* wait for first child */
        err_sys("waitpid error");

    /* We're the parent (the original process); we continue executing,
       knowing that we're not the parent of the second child. */

    exit(0);
}
```

通过fork( )两次来避免僵死进程

2.进程的控制

- 进程中执行另一个程序

```
#include <unistd.h>
int execl(const char *pathname, const char * arg0,
    ... /* (char *) 0 */);
int execlv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0,
    ... /* (char *)0 , char *const envp[] */)
int execve(const char *pathname, char *const argv[], char
    *const envp[]);
int execlp(const char *filename, const char *arg0,
    ... /* (char *)0 */)
int execvp(const char *filename, char *const argv[]);
```

```c
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

2.进程的控制


```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash

                                    47 more lines that aren't shown

HOME=/home/sar
```

2.进程的控制

- 调用shell命令

```
#include <stdlib.h>

int system(const char *cmdstring);
```

```c
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                                /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

2.进程的控制


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

int
main(void)
{
    int      status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

   if ((status = system("nosuchcommand")) < 0)
       err_sys("system() error");
   pr_exit(status);

   if ((status = system("who; exit 44")) < 0)
       err_sys("system() error");
   pr_exit(status);

   exit(0);
}
```

2.进程的控制

- 构造一个简单的shell解释程序

有了fork()，exec()和wait()调用，可以很容易构造一个简单的交互式shell。

给出提示符，从键盘输入命令，执行输入的命令。等待命令执行完毕之后shell再给出提示符才允许输入下一条命令。

3.守护进程

- 什么是守护进程
  - 守护进程（daemon）是生存期长的一种进程，它们常在系统引导装入时启动，在系统关闭时终止；
  - 因为没有控制终端，守护进程是在后台运行的；
  - 守护进程常常用作为服务器进程；

PPID	PID	PGID	SID	TTY	TPGID	UID	COMMAND
0	1	0	0	?	-1	0	init
1	2	1	1	?	-1	0	[keventd]
1	3	1	1	?	-1	0	[kapmd]
0	5	1	1	?	-1	0	[kswapd]
0	6	1	1	?	-1	0	[bdflush]
0	7	1	1	?	-1	0	[kupdated]
1	1009	1009	1009	?	-1	32	portmap
1	1048	1048	1048	?	-1	0	syslogd -m 0
1	1335	1335	1335	?	-1	0	xinetd - pidfile /var/run/xinetd.pid
1	1403	1	1	?	-1	0	[nfsd]
1	1405	1	1	?	-1	0	[lockd]
1405	1406	1	1	?	-1	0	[rpciod]
1	1853	1853	1853	?	-1	0	crond
1	2182	2182	2182	?	-1	0	/usr/sbin/cupsd

ps -axj