第7章

进程和信号

1

7.1 进程的基本概念

- 什么是进程
  - 操作系统中最基本、重要的概念。
  - 多种定义：
    - “一个其中运行着一个或多个线程的地址空间和这些线程所需要的系统资源。”
    - “正在运行的程序。”
  - 所有采用多道程序设计的操作系统都建立在进程的基础上。
  - 从理论角度看，进程是对正在运行的程序过程的抽象。
  - 从实现角度看，进程是一种数据结构，目的在于清晰地刻画动态系统的内在规律，有效管理和调度进入计算机系统主存储器运行的程序。
  - 进程和程序的关系：
    - 程序是指令的有序集合，静态的概念。
    - 进程是一个具有独立功能的程序关于某个数据集合在处理机上的一次执行过程，动态的概念。
  - 进程可以申请和拥有系统资源，是一个活动的实体，包括：
    - 当前的活动（通过程序计数器的值和寄存器的内容来表示）
    - 程序代码、数据、变量（占用着系统内存）
    - 打开的文件（文件描述符）
    - 环境。

2

7.1 进程的基本概念

- 什么是进程
  - 进程的特征：
    - 动态性：进程的实质是程序的一次执行过程，进程是动态产生、动态消亡的。
    - 并发性：任何进程都可以同其他进程一起并发执行。
    - 独立性：进程能够独立运行，是系统分配和调度资源的基本单位。
    - 异步性：因进程的独立性，各进程按各自独立的、不可预知的速度向前推进。
    - 结构性：从结构上看，进程大体上由程序段、数据段和进程控制块三部分组成。

3

7.1.2 Linux进程环境

- 程序的入口
  - int main(int argc, char *argv[]);
  - 当内核起动C程序时，首先调用一个特殊的起动例程（编译连接程序将该例程设置为可执行程序的起始地址）。
  - 起动例程从内核取得命令行参数和环境变量值，然后调用main函数，并将命令行参数传递给它。

Program 7-1 ex_main.c

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int i;
6     for(i = 0; i < argc; i++)
7         printf("argv[%d]: %s\n", i, argv[i]);
8     return 0;
9 }
```

for(i=0; argv[i] != NULL; i++)

4

7.1.2 Linux进程环境

- 进程的终止
  - 5种方式
    - 正常终止
      - 从main返回
      - 调用exit
      - 调用_exit或_Exit
    - 异常终止
      - 被一个系统信号终止
      - 调用abort，它产生SIGABRT信号
  - 起动例程会在main函数return后立即调用exit函数。
    - exit(main(argc, argv));

5

7.1.2 Linux进程环境

- 退出函数
  - _exit和_Exit，正常终止一个程序，立即进入内核.
  - exit，正常终止一个程序，先执行一些清除处理(包括调用执行各终止处理程序，关闭所有标准I/O流等)，然后进入内核。

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);
#include <unistd.h>
void _exit(int status);
```

exit(0) 等价于 return(0)

  - exit和_Exit是ISO C标准，而_exit为POSIX.1标准。
  - 参数status为退出状态。大多数shell都提供检查一个进程终止状态的方法。
  - 如果(a)调用这些函数时不带终止状态，或(b)main执行了一个无返回值的return语句，或(c)main执行隐式返回，则该进程的终止状态是未定义的。

6

7.1.2 Linux进程环境

- atexit函数
  - ANSI C规定，一个进程可以登记多至32个由exit自动调用的函数，这些函数被称为终止处理程序（exit handler），使用atexit函数登记。

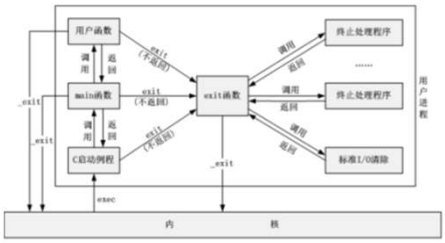
```
#include <stdlib.h>
int atexit(void (*function)(void));
```

  - 参数　　一个函数地址.
  - 返回值　　成功返回0，失败返回非0值.
  - exit函数以登记这些函数的相反顺序调用它们。同一函数如若登记多次，则也将被调用多次。

7

7.1.2 Linux进程环境

- atexit函数
  - C程序的起动/终止的方式和过程



  - 内核使程序执行的唯一方法是调用一个exec函数。进程正常终止的唯一方法是显式或隐式地(调用exit)调用_exit。
  - 示例

8

7.1.2 Linux进程环境

- atexit函数

程序清单 7-2 ex_atexit.c

```
1 #include <stdio.h>
2
3 static void my_exit1(void);
4 static void my_exit2(void);
5
6 int main(void)
7 {
8     if(atexit(my_exit2) != 0)
9     {
10        printf("register my_exit2 failed\n");
11        return -1;
12    }
13
14    if(atexit(my_exit1) != 0)
15    {
16        printf("register my_exit1 failed\n");
17        return -1;
18    }
19
20    printf("main is done\n");
21    return 0;
22 }
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_atexit ex_atexit.c
jianglinmei@ubuntu:~/c$ ./ex_atexit
main is done
first exit handler
second exit handler
```

```
23
24 static void my_exit1(void)
25 {
26     printf("first exit handler\n");
27 }
28
29 static void my_exit2(void)
30 {
31     printf("second exit handler\n");
32 }
```

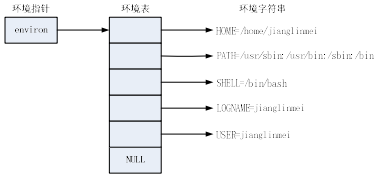

9

7.1.2 Linux进程环境

- 环境表
  - 每个进程在启动时都接收到一张环境表：一个字符指针数组。全局变量environ记录了该指针数组的地址。

```
extern char **environ;
```

  - 全局变量environ称为环境指针，其所指向的数组称为环境表，数组中的每个指针指向的字符串称为环境字符串。
  - 环境字符串具有约定的形如“name=value”的格式。



10

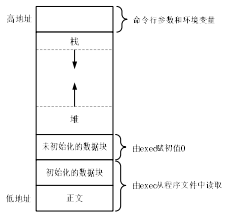
7.1.2 Linux进程环境

- C程序的存储空间布局
  - 存储空间的组成部分
    - 正文段　　由CPU执行的机器指令构成，共享+只读。
    - 初始化数据段　　数据段，由程序中已赋初值的静态变量构成。
    - 非初始化数据段　　bss段，在程序开始执行之前，内核将此段初始化为0。
    - 栈　　自动变量、参数、函数调用的场景信息（如返回地址、寄存器值）。
    - 堆　　进行动态存储分配，位于非初始化数据段顶和栈底之间。

11

7.1.2 Linux进程环境

- C程序的存储空间布局



  - 使用size命令可查看一个可执行程序的正文段、数据段和bss段的长度

```
jianglinmei@ubuntu:~/c$ size /bin/bash
   text    data     bss     dec     hex filename
 799889   18452   20232  838573  ccbad /bin/bash
```


12

7.1.2 Linux进程环境

静态库和共享库

- 库，就是可复用的二进制可执行代码的有序集合。
- 静态库和共享库的区别
  - 使用静态库的程序，在编译连接过程即载入静态库的代码并将静态库的代码置入编译出的可执行程序。文件名格式: libxxxx.a。
  - 使用共享库的程序，在编译连接过程仅对共享库作简单引用，在程序运行时才将共享库的代码载入内存。 文件名格式: libxxxx.so.major.minor。
- 静态库
  - 使用ar命令将多个二进制目标代码文件打包成静态库。使用Linux下的nm命令可列出库中的符号清单。
  - 示例

13

13

7.1.2 Linux进程环境

静态库和共享库

Program 7-3 ex_static.h

```
1 extern int sum(int a, int b);
2 extern int average(int a, int b);
```

Program 7-4 ex_static1.c

```
1 int sum(int a, int b)
2 {
3     return a + b;
4 }
```

Program 7-5 ex_static2.c

```
1 int average(int a, int b)
2 {
3     return (a + b) / 2;
4 }
```

```
jianglinmei@ubuntu:~/c$ gcc -c -o ex_static1.o ex_static1.c
jianglinmei@ubuntu:~/c$ gcc -c -o ex_static2.o ex_static2.c
jianglinmei@ubuntu:~/c$ ar rcs libmystatic.a ex_static1.o
ex_static2.o
jianglinmei@ubuntu:~/c$ nm libmystatic.a

ex_static1.o:
00000000 T sum

ex_static2.o:
00000000 T average
```

14

14

7.1.2 Linux进程环境

静态库和共享库

Program 7-6 ex_test_static.c

```
1 #include <stdio.h>
2 #include "ex_static.h"
3
4 int main(void)
5 {
6     printf("3 + 5 = %d\n", sum(3, 5));
7     printf("average of 3 and 5 is: %d\n", average(3, 5));
8     return 0;
9 }
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_test_static ex_test_static.c -L. -lmystatic
jianglinmei@ubuntu:~/c$ ./ex_test_static
3 + 5 = 8
average of 3 and 5 is: 4
```

15

15

7.1.2 Linux进程环境

静态库和共享库

- 共享库
  - 不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的副本.
  - 程序第一次执行或者第一次调用某个库函数时，用动态连接方法将程序与共享库函数相连接。因此，减少了每个可执行文件的长度，但增加了一些运行时间开销。
  - 可以用库函数的新版本代替老版本而无需对使用该库的程序重新进行编译和连接。
  - 使用Linux命令ldd可以查看一个二进制可执行程序或共享库所依赖的共享库。

```
jianglinmei@ubuntu:~/c$ ldd ex_test_static
        linux-gate.so.1 =>  (0x006c4000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x0026d000)
        /lib/ld-linux.so.2 (0x00fa8000)
```

16

16

7.2 进程的结构

进程控制块和进程表

- Linux内核通过进程控制块(PCB)对进程进行控制和管理。PCB由一个task_struct结构体定义.
- PCB通常是系统内存占用区中的一个连续存储区，它存放着内核用于描述进程情况及控制进程运行所需的全部信息，如:
  - 进程标识符， PID
  - 进程当前状态
  - 进程的程序和数据地址
  - 进程资源清单
  - 进程优先级
  - CPU现场保护区
  - 用于实现进程间通信所需的信息
  - 其他信息：如父进程的PID、有效用户ID、有效组ID、进程占用CPU的时间、进程退出码、当前目录节点、执行文件节点等
- Linux内核将所有进程控制块组织成指针数组形式，形如:

```
struct task_struct *task[NR_TASK];
```

  - 该指针数组即进程表，其中记录了指向各PCB的指针。NR_TASK规定了最多可同时运行进程的个数。内核以PID作为进程表项的索引。

17

17

7.2 进程的结构

PID

- 每个进程都有一个唯一非负整数作为进程标识。因其唯一性，常用于构成其他的标识符以保证该标识符的唯一性。
- 三个特殊进程
  - PID为0调度进程。内核的一部分，也被称为交换进程或系统进程，无对应的程序文件。
  - PID为1的init进程。在系统自举过程结束时由内核调用（第一个用户进程），其对应的程序文件是/sbin/init。
    - 负责在内核自举后启动一个Linux系统;
    - 通常读与系统有关的初始化文件(/etc/rc*文件)，并将系统引导到某一个状态(例如多用户)。
    - init进程决不会终止。
    - 是一个普通的用户进程，但以超级用户特权运行。
  - PID为2的kthreadd内核进程，也是一个内核线程，用于在后台执行一些任务。
    - 内核线程是独立运行在内核空间的标准进程，但：没有独立的地址空间，从来不切换到用户空间去。
    - kthreadd由内核从init进程产生，用于衍生出其它的内核线程。

18

18

7.2 进程的结构

- PID
  - 一般进程均由一个“父进程”创建，被父进程创建的进程称为“子进程”。
    - PID为0的交换进程是其他所有进程的祖先进程。
    - init进程是所有其它用户进程的祖先进程。
    - kthreadd内核线程是其他所有内核线程的父进程。
  - 使用“pstree”命令和“ps ax -o pid,ppid,command”命令可以清楚地查看进程间的父子关系。

```
jianglinmei@ubuntu:~$ ps ax -o pid,ppid,command
  PID  PPID COMMAND
    1     0 /sbin/init
    2     0 [kthreadd]
    3     2 [ksoftirqd/0]
    5     2 [kworker/u:0]
    6     2 [migration/0]
......
  644     1 /usr/sbin/sshd -D
774     1 cron
......
18573 18408 sshd: jianglinmei@pts/0
18574 18573 -bash
```

19

19

7.2 进程的结构

- PID
  - 下列函数可返回进程的一些其他标识符。

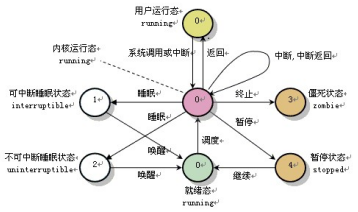
```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);          /* 返回：调用进程的进程ID          */
pid_t getppid(void);         /* 返回：调用进程的父进程ID        */
uid_t getuid(void);          /* 返回：调用进程的实际用户ID      */
uid_t geteuid(void);         /* 返回：调用进程的有效用户ID      */
gid_t getgid(void);          /* 返回：调用进程的实际组ID        */
gid_t getegid(void);         /* 返回：调用进程的有效组ID        */
```

20

20

7.2 进程的结构

- 进程的状态
  - 在多道程序系统中，进程在处理器上交替运行，状态也不断地发生变化。
  - Linux进程状态和各状态间的转换关系



  - Linux是一个分时操作系统。当一个进程的运行时间片用完后，系统调度程序就会切换到其它的进程去执行。而当进程在内核态运行时需要等待某个资源，该进程就会自愿地放弃CPU的使用权进入睡眠状态。

21

21

7.2 进程的结构

- 进程的状态
  - 使用“ps ax -o pid,stat,command”命令可查看进程所处的状态。

```
jianglinmei@ubuntu:~$ ps ax -o pid,stat,command
PID   STAT   COMMAND
    1 Ss     /sbin/init
   37 SN     [ksmd]
  892 Ss1    gdm-binary
 2049 Ss+    bash
 8204 S<     udevd --daemon
12682 R+     ps ax
```

  - 常见的状态字符

STAT字符	说明
S	睡眠。通常是在等待某个事件的发生。
R	运行/可运行。即在运行队列中，处于正在运行或即将运行状态。
D	不可中断的睡眠（等待,不响应异步信号）。通常是在等待输入或输出完成。
T	停止。通常是被shell作业控制所停止，或处于调试器控制下。
Z	僵尸（zombie）进程。
N	低优先级任务。
s	进程是会话期首进程。
+	进程属于前台进程组。
l	进程是多线程的。
<	高优先级任务。

22

22

7.3 进程控制

- system
  - 执
  - co
  - re
  - 说

```
jianglinmei@ubuntu:~/c$ gcc -o ex_system ex_system.c
jianglinmei@ubuntu:~/c$ ./ex_system
Running ps with system
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Nov16 ?        00:00:01 /sbin/init
....
1001     20397 20396  0 18:53 pts/1    00:00:01 -bash
1001     20706 20397  0 19:13 pts/1    00:00:00 ./ex_system
1001     20707 20706  0 19:13 pts/1    00:00:00 sh -c ps -ef
1001     20708 20707  0 19:13 pts/1    00:00:00 ps -ef
Done.
```

Program 7-7 ex_system.c

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("Running ps with system\n");
7     system("ps -ef");
8     printf("Done.\n");
9     exit(0);
10 }
```

23

23

7.3 进程控制

- exec
  - 执行一个程序

前6个函数通过调用execve实现，execvpe则是GNU的扩展.

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., 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[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

  - path　　待运行的程序全路径名（命令字符串）.
  - file　　待运行的程序名，通过PATH环境变量搜索其路径.
  - arg　　命令参数.
  - …　　可选的一到多个命令参数，要求以NULL结束.
  - agrv　　命令参数指针数组，要求以NULL结束.
  - envp　　传递给待运行程序的环境变量指针数组，以NULL结束.
  - return value　　成功时不返回，出错时返回-1，并设置errno变量.
  - 说明：1), 调用exec函数后，进程完全被新程序替换，新程序从其main函数开始执行。2), exec并不创建新进程，只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。3), 调用exec前后进程ID不变。

24

24

7.3 进程控制

exec

- 说明：
  - 调用exec函数后，进程完全被新程序替换，新程序从其main函数开始执行。
  - exec并不创建新进程，只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。
  - 调用exec前后进程ID不变。
  - 执行新程序的进程还保持了原进程的下列特征：
    - 进程ID和父进程ID。
    - 实际用户ID和实际组ID。
    - 添加组ID。
    - 进程组ID。
    - 会话期ID。
    - 控制终端。
    - 闹钟尚余留的时间。
    - 当前工作目录。
    - 根目录。
    - 文件权限创建屏蔽字。
    - 文件锁。
    - 进程信号屏蔽。
    - 未决信号。
    - 资源限制。
    - 用户态运行时间、内核态运行时间、子进程用户态运行时间和子进程内核态运行时间。
  - 默认情况下，已打开文件的文件描述符仍保持打开，除非调用fcntl函数设置了exec关闭标志。

7.3 进程控制

exec

- 用法和示例

Program 7-8 exec函数的基本用法#include <unistd.h>

```
/* 命令行参数指针数组范例，注意数组应以命令名本身（即argv[0]）作为第一个元素 */
char *const ps_argv[] = {"ps", "ax", 0};

/* 环境变量指针数组范例 */
char *const ps_envp[] = {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* 各个exec函数的调用方法范例，以下函数不可能同时成功调用，
 * 因一旦一个调用成功就不会返回，其后的语句将不再有机会得到执行。 */
execl("/bin/ps", "ps", "ax", 0);                    /* 假设ps命令在/bin目录下 */
execlp("ps", "ps", "ax", 0);                        /* 假设/bin目录在PATH环境变量中 */
execle("/bin/ps", "ps", "ax", 0, ps_envp);
execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execvpe("ps", ps_argv, ps_envp);
execve("/bin/ps", ps_argv, ps_envp);
```


7.3 进程控制

exec

- 用法和示例

Program 7-9 ex_exec.c

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main()
5 {
6     printf("Running ps with execlp\n");
7     execlp("ps", "ps", "ax", NULL);
8     printf("Done.\n");
9     exit(0);
10 }
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_exec ex_exec.c
jianglinmei@ubuntu:~/c$ ./ex_exec
Running ps with execlp
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Nov16 ?        00:00:01 /sbin/init
......
1001     21758 21757  1 22:28 pts/2    00:00:01 -bash
1001     21874 21758  0 22:29 pts/2    00:00:00 ps -ef
```

并无一个名为 “ex_exec” 的进程。

7.3 进程控制

fork

- 创建一个新进程.

```
#include <unistd.h>
pid_t fork(void);
```

- 返回值　　成功时返回两次。子进程的返回值是0，父进程的返回值为子进程的PID。失败时返回-1，并设置errno，典型错误：
  - E_AGAIN　　子进程超过CHILD_MAX限制，或系统缺少资源.
  - ENOMEM　　进程表无足够空间.
- 说明
  - 在调用fork函数之后，子进程和父进程以不确定的次序继续执行fork之后的指令。
  - 子进程是父进程的复制品。子进程获得父进程数据空间、堆和栈的复制品。
  - 如果正文段是只读的，则父、子进程共享正文段。
  - 父、子进程之间的区别
    - PID & PPID.
    - 子进程的用户态运行时间、内核态运行时间、子进程用户态运行时间和子进程内核态运行时间被设置为0.
    - 父进程设置的锁，子进程不继承.
    - 子进程的未决告警被清除.
    - 子进程的未决信号集设置为空集.

7.3 进程控制

fork

- 两种用法
  - 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。常用于网络服务。
  - 一个进程要执行一个不同的程序。在这种情况下，子进程在从fork返回后立即调用exec。
  - 子进程在fork和exec之间可以更改自己的属性，如I/O重新定向、用户ID、信号排列等。
- 示例

Program 7-10 ex_fork.c

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 int main()
6 {
7     pid_t pid;   /* store PID */
8     char *message;
9     int n = 2;   /* counter */
10
11    printf("fork program starting\n");
12    pid = fork(); /* create process */
13    switch(pid)
14    {
15    case -1:    /* error */
16        perror("fork failed");
17        exit(1);
18    case 0:     /* child process */
19        message = "This is the child";
20        n = 5;
21        break;
22    default:    /* parent process */
23        message = "This is the parent";
24        n++;
25        break;
26    }
27
28    for(; n > 0; n--) {
29        puts(message);
30        sleep(1);
31    }
32
33    exit(0);
34 }
```

```
jianglinmei@ubuntu:~/c$ ./ex_fork
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
jianglinmei@ubuntu:~/c$ This is the child
This is the child
```

7.3 进程控制

vfork

- 创建共享虚拟存储空间的新进程.

```
#include<sys/types.h>
#include<unistd.h>
pid_t vfork(void);
```

- 返回值　　同fork().
- 说明
  - vfork创建的新进程的目的是exec另一个程序。
  - vfork不将父进程的地址空间完全复制到子进程中，在子进程调用exec或exit之前，它在父进程的空间中运行，应注意在此期间不要改变数据。
  - vfork保证子进程先运行，在子进程调用exec或exit之后父进程才可能被调度运行。
- 示例

31

7.3 进程控制

Program 7-11 ex_vfork.c

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 int main()
6 {
7    pid_t pid;              /* 用于保存PID */
8    char *message;          /* 用于保存消息字符串 */
9    int n = 2;              /* 计数变量 */
10
11   printf("fork program starting\n");
12   pid = vfork();          /* 用vfork创建子进程 */
13   switch(pid)
14   {
15   case -1:                /* 出错 */
16      perror("fork failed");
17      exit(1);
18   case 0:                 /* 子进程 */
19      message = "This is the child";
20      n = 5;
21      break;
22   default:                /* 父进程 */
23      message = "This is the parent";
24      n++;
25      break;
26   }
27
28   for(; n > 0; n--) {
29      puts(message);
30      sleep(1);
31   }
32
33   exit(0);
34 }
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_vfork
ex_vfork.c
jianglinmei@ubuntu:~/c$ ./ex_vfork
fork program starting
This is the child
This is the child
This is the child
This is the child
This is the parent
```


32

7.3 进程控制

进程的终止状态

- 不管进程如何终止，最后都会执行内核中的同一段代码。这段代码为相应进程关闭所有已打开的文件描述符，释放它所使用的内存，等等。
- 对任意一种终止情形，父进程都应当得到通知，知道子进程是如何终止的。
  - 对于正常终止的情况，传向exit或_exit的参数，或main函数的返回值，指明了它们的退出状态（exits tatus），内核以该“退出状态”作为进程的“终止状态”。
  - 在异常终止情况下，内核会产生一个指示其异常终止原因的终止状态。
  - 终止进程的父进程可使用wait或waitpid函数取得其终止状态
- 如果父进程在子进程之前终止，其父进程会改为 “init”进程。称子进程被init进程领养。
- 进程终止的时候，内核将保持其PCB，直到父进程调用wait或waitpid。
- 一个已经终止、但是其父进程尚未对其进行善后处理的进程被称为僵尸进程（zombie）.

33

7.3 进程控制

wait & waitpid

- 等待进程终止.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- status　　输出参数，用于获取子进程的退出状态，可为空。
- pid　　要等待的子进程的PID，含义：
  - < -1　　等待进程的组ID等于pid的绝对值的所有子进程.
  - -1　　等待所有子进程.
  - 0　　等待进程的组ID等于调用进程的组ID的所有子进程.
  - > 0　　等待pid进程.
- options　　常见的选项为WNOHANG，意为不阻塞调用者进程.
- 返回值
  - 成功时，返回已结束子进程的PID；对于waitpid，若指定了WNOHANG选项，但没有子进程终止，则返回0
  - 出错时返回-1，并设置errno变量

34

7.3 进程控制

wait & waitpid

- 在父进程中调用wait或waitpid可能发生的情况
  - 阻塞（如果其所有子进程都还在运行）
  - 带子进程的终止状态立即返回(如果一个子进程已终止，正等待父进程获取其终止状态)
  - 出错立即返回（如果它没有任何子进程）
- 测试止状态的宏

宏	说明
WIFEXITED(status)	如果子进程正常结束，则取非零值。
WEXITSTATUS(status)	如果WIFEXITED非零，则得到子进程的退出码。
WIFSIGNALED(status)	如果子进程因未捕获的信号而终止，则取非零值。
WTERMSIG(status)	如果WIFSIGNALED非零, 则得到引起子进程终止的信号代码。
WIFSTOPPED(status)	如果子进程已意外终止，则取非零值。
WSTOPSIG(status)	如果WIFSTOPPED非零, 则得到引起子进程终止的信号代码。

35

7.3 进程控制

wait & waitpid

- 示例[7-12 ex_wait.c]

Program 7-12 ex_wait.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main()
```

```
22   case 0:
23      message = "This is the child";
24      n = 5;
25      exit_code = 37;/* set child exit status 37 */
26      break;
27   default:
28      message = "This is the parent";
29      n++;
30      exit_code = 0;/* set parent exit status 0 */
31      break;
32   }
33
34   for(; n > 0; n--)
35   {
36      puts(message);
      eep(1);

      d != 0)      /* parent process */

      t stat_val;
      d_t child_pid;

      ild_pid = wait(&stat_val); /* wait child */

      rintf("Child has finished: PID = %d\n", child_pid);
      WIFEXITED(stat_val))
        printf("Child exited with code %d\n",
      US(stat_val));
      se
        printf("Child terminated abnormally\n");

      exit_code);
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_wait
ex_wait.c
jianglinmei@ubuntu:~/c$ ./ex_wait
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 24664
Child exited with code 37
```


36

7.4 信号

简介

- 信号是软件中断，提供了一种处理异步事件的方法，可作为进程间通信的一种机制。
- 每个信号都有一个以SIG开头名称，例如SIGINT、SIGALRM等。这些信号名称在signal.h中以宏的形式定义，对应一个正整数（信号编号）。
- 以下情形可以产生一个信号
  - 当用户在终端键入某些组合键时，如Ctrl+C键产生中断信号（SIGINT）。
  - 硬件异常产生信号。例如：除数为0、非法的内存访问等。
  - 进程调用kill函数将信号发送给另一个进程或进程组。
  - 用户在shell命令行，使用kill命令将信号发送给其他进程。
    - kill -<信号名> <PID>
    - killall -<信号名> <命令名>
  - 当检测到某种软件异常时产生信号。
    - 波特率不符产生SIGURG
    - 写无读进程的管道产生SIGPIPE
    - 闹钟超时产生SIGALRM

37

37

7.4 信号

简介

- 信号是一种典型的异步事件，产生信号的事件对进程而言是随机出现的。
- 当信号产生时，可有三种处理方式：忽略、捕获或执行默认操作。
  - 忽略信号。
    - 大多数信号均可忽略。
    - SIGKILL和SIGSTOP不能被忽略，因为它们为超级用户提供了一种使进程终止或停止的可靠方法。
    - 某些由硬件异常产生的信号（例如非法的内存访问或除以0）也不应被忽略。
  - 捕获信号。让内核在信号产生时调用事先设置好的回调函数。
  - 执行系统默认动作。

38

38

7.4 信号

简介

- 信号列表（未捕获时会引起进程终止的信号）

信号名称	说明
SIGABORT	调用abort函数时产生，进程将异常终止。
SIGALRM	超时。一般由alarm设置的定时器产生。
SIGFPE	浮点运算异常，如除以0和浮点溢出。
SIGHUP	终端关闭或断开连接。由处于非连接状态的终端发给控制进程或由控制进程在自身结束时发给每个前台进程。
SIGILL	非法指令。通常由一个崩溃的程序或无效的共享内存模块引起。
SIGINT	程序终止，一般由从终端敲入的中断字符（Ctrl+C）产生。
SIGKILL	终止进程(此信号不能被捕获或忽略)，一般在shell中用它来强制终止异常进程。
SIGPIPE	向管道写数据时没有与之对应的读进程时产生。
SIGQUIT	程序退出。一般由终端敲入的退出字符（Ctrl+\）产生。
SIGSEGV	无效内存段访问。一般是因为对内存中的无效地址进行读写引起，如数组越界、解引用无效指针。
SIGTERM	kill命令默认发送的信号，要求进程结束运行。UNIX在关机时也用此信号要求服务停止运行。
SIGUSR1	用户定义信号1，用于进程间通信。
SIGUSR2	用户定义信号2，用于进程间通信。

39

39

7.4 信号

简介

- 信号列表（未捕获时不会引起进程终止的常见信号）

信号名称	说明
SIGCHLD	子进程停止或退出时产生，默认被忽略
SIGCONT	如果进程被暂停则继续执行。
SIGSTOP	停止执行（此信号不能被捕获或忽略）
SIGTSTP	终端挂起。通常因按下Ctrl+Z组合键而产生。
SIGTTIN	后台进程尝试读操作。shell用以表明后台进程因需要从终端读取输入而暂停运行。
SIGTTOU	后台进程尝试写操作。shell用以表明后台进程因需要产生输出而暂停运行。

40

40

7.4 信号

捕获信号

- 最简单的捕获信号的方式是调用sign

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- signum　准备捕获或忽略的信号.
- handler　捕获到信号后由系统调用的回调函数。特殊值：
  - SIG_IGN　忽略信号.
  - SIG_DFL　恢复默认行为.
- 返回值
  - 成功时返回原先定义信号处理函数，如果原先未定义则返回SIG_ERR，并设置errno为一正数值。
  - 出错时，如果给出的是一个无效的信号或不可捕获或不可忽略的信号，则返回SIG_ERR，并设置errno为EINVAL。
- 进程启动时，所有信号的状态都为系统默认或忽略。
- 调用exec函数会将原先设置为要捕捉的信号都更改为默认动作。
- 当一个进程调用fork创建子进程时，其子进程继承了父进程的信号处理方式。
- shell程序会自动将后台进程中对中断和退出信号的处理方式设置为忽略。
  - 当用户在命令行按Ctrl+C或Ctrl+\键时就不会影响到后台进程。

41

41

7.4 信号

捕获信号

- 示例

Program 7-13 ex_signal.c

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void gotit(int sig)                    /* 定义信号回调函数 */
6 {
7     printf("signal %d is captured.\n", sig);
8     signal(SIGINT, SIG_DFL);           /* 设置回默认动作 */
9 }
10
11 int main()
12 {
13     signal(SIGINT,
14     while(1)
15     {
16         printf("Hello
17         sleep(1);
18     }
19 }
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_signal ex_signal.c
jianglinmei@ubuntu:~/c$ ./ex_signal
Hello World!
Hello World!
^Csignal 2 is captured.
Hello World!
Hello World!
^C
```

42

42

7.4 信号

发送信号

- 进程可以调用kill函数给自己或其他进程发送信号。

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- pid　　接收信号的进程PID.
- sig　　要发送的信号.
- 返回值　　成功0；失败-1，并设置errno。常见的errno值为:
  - EINVAL　　给定的信号无效.
  - EPERM　　发送进程权限不够.
  - ESRCH　　目标进程不存在.
- 发送方进程应具有相应的权限，满足以下两个条件之一:
  - 接收信号进程和发送信号进程的所有者相同.
  - 发送信号的进程的所有者是超级用户.

7.4 信号

发送信号

- 示例

Program 7-14 ex_kill.c

```
 1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     pid_t pid;
```

```
16    switch(pid)
17    {
18    case -1:
19        perror("fork failed");
20        exit(1);
21    case 0:
22        message = "This is the child";
23        n = 5;
24        break;
25    default:
26        message = "This is the parent";
27        n++;
28        break;
29    }
30
31    for(; n > 0; n--)
      {
          puts(message);
          sleep(1);
      }

      f (pid != 0)
      {
          kill(pid, SIGINT);      /* 发送SIGINT信号 */
      }

      return 0;
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_kill
ex_kill.c
jianglinmei@ubuntu:~/c$ ./ex_kill
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
```


7.4 信号

alarm、pause和sleep函数

- alarm函数可为进程设置一个定时器，在设定的时间到达时，产生SIGALRM信号。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- seconds　　指定几秒后产生SIGALRM信号，0表示取消设置.
- 返回值　　成功时返回以前设置的定时器时间的余留秒数；失败时返回0。
- 每个进程只能有一个定时器。
- alarm设置的定时器并非周期性的定时器，即调用一次只产生一次SIGALRM信号。
- alarm函数设置的定时器并不能精确定时执行。

7.4 信号

alarm、pause和sleep函数

- 调用pause函数可使调用进程挂起直至捕捉到一个信号.

```
#include <unistd.h>
int pause(void);
```

- 返回值　　直到进程捕获到一个信号并从该信号的信号处理程序中返回时，pause函数才返回，返回值为-1，并设置errno为EINTR。
- sleep函数使进程睡眠若干秒。

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

- 返回值　　返回值为所设时间的余留秒数。
- sleep函数使调用进程挂起直到
  - 已经过了seconds所指定秒数
  - 捕捉到一个信号并从信号处理程序返回.

7.4 信号

alarm、pause和sleep函数

- 示例

```
jianglinmei@ubuntu:~/c$ gcc -o
ex_alarm_pause ex_alarm_pause.c
jianglinmei@ubuntu:~/c$ ./ex_alarm_pause
Begin...
The timer is timeout.
End...
```

Program 7-15 e

```
 1 #include <unistd.h>
2 #include <signal.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 static void onTimer(int sig) /* handler */
7 {
8     printf("The timer is timeout.\n");
9 }
10
11 static unsigned int mysleep(unsigned int seconds)
12 {
13     if (signal(SIGALRM, onTimer) == SIG_ERR)
14     {
15         printf("Fail to call signal()\n");
16         return seconds;
17     }
18
19     alarm(seconds);          /* set alarm clock */
20     pause();                 /* pause to wait signal */
21     /* close clock and return the remaining seconds*/
22     return alarm(0);
23 }
```

```
25 int main()
26 {
27     printf("Begin...\n");
28     mysleep(10);
29     printf("End...\n");
30
31     return 0;
32 }
```


7.4 信号

abort

- abort函数的功能是使程序异常终止。

```
#include <stdlib.h>
void abort(void);
```

- abort函数将SIGABRT信号发送给调用进程。进程不应忽略此信号。
- 进程捕捉SIGABRT后可在进程终止之前执行必要的清理操作。当信号处理程序返回时，abort即终止进程。

7.4 信号

信号集

- 数据类型sigset_t用于存放一个信号集。头文件signal.h中声明了sigset_t和五个信号集处理函数。

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
int sigismember(const sigset_t *set, int sig);
```

- set　　信号集.
- sig　　信号.
- 返回值
  - 成功时，sigismeber函数返回1或0；其他函数返回0。
  - 失败时返回-1，并设置errno，错误代码EINVAL，表示给定的信号无效。
- 函数sigemptyset初始化由set指向的信号集，使其排除所有信号。
- sigfillset初始化由set指向的信号集，使其包括所有信号。

7.4 信号

sigaction

- sigaction是一个比signal更健壮的用于捕获信号的编程接口。

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oldact);
```

- sig　　准备捕获或忽略的信号.
- act　　将要设置的信号处理动作.
- oldact　　用于取回原先的信号处理动作.
- 返回值　　成功时返回0；失败时返回-1，并设置errno变量。EINVAL表示无效的、不可捕获或不可忽略的信号。
- 如果act指针非空，则表示要修改sig信号的处理动作。如果oldact指针非空，则系统在其中返回该信号的原先动作。

7.4 信号

sigaction

- struct sigaction.

```
struct sigaction {
        void (*sa_handler)(int);   /* 信号处理函数，同signal */
        sigset_t sa_mask;          /* 回调过程中将被屏蔽的信号集 */
        int sa_flags;              /* 可决定回调行为的位标志值 */
        ......                     /* 未列出的其他不重要的成员*/
}
```

- sa_flags
  - SA_NOCLDSTOP　　子进程停止时不产生SIGCHLD信号。
  - SA_RESETHAND　　在信号处理函数入口处将对此信号的处理方式重置为SIG_DFL。
  - SA_RESTART　　重启可中断的函数而不是给出EINTR错误。
  - SA_NODEFER　　捕获到信号时不将它添加到信号屏蔽字中，即不自动阻塞当前捕获到的信号。
- 一旦对给定的信号设置了一个动作，那么在用sigaction改变它之前，该设置就一直有效。

7.4 信号

sigaction

- 示例

Program 7-16 ex_sigaction.c

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void gotit(int sig)          /* handler */
6 {
7     printf("signal %d is captured.\n", sig);
8
9     sleep(20);
10 }
11
12 int main()
13 {
14     struct sigaction act;
15
16     act.sa_handler = gotit;
17     sigemptyset(&act.sa_mask);
18     sigaddset(&act.sa_mask, SIGQUIT);/* shield SIGQUIT */
19     act.sa_flags = 0;
20
21     sigaction(SIGINT, &act, 0); /* set handler for SIGINT */
22     while(1) {
23         printf("Hello World!\n");
24         sleep(1);
25     }
26 }
```

```
jianglinmei@ubuntu:~/c$ ./ex_sigaction
Hello World!
Hello World!
^Csignal 2 is captured.
^\ <exit>
```


第8章

进程间通信

8.1 IPC简介

- IPC是类UNIX系统中的一个专业术语。
- 进程间通信就是在不同进程之间传递或交换信息。需要一个不同进程都能访问的存放数据的 “公共场所”，即数据存放介质。
- 3类“公共场所”
  - 进程的用户空间是互相独立的，但共享存储区可公用。
  - 系统空间是公用，但是只有内核具有访问权限。
  - 外设，两个进程可以通过磁盘上的普通文件、或者通过“注册表”（Windows系统）、或者通过第三方数据库进行信息的相互交换。

8.1 IPC简介

- 三种常用的IPC技术
  - 管道
    - （普通）管道，有两个局限：
      - 半双工，只能单向传送数据；
      - 只能在同源进程（在进程创建上具有亲缘关系）间使用。
    - 命名管道(FIFO)，可以在不相关的进程之间进行通讯。
  - SysV IPC
    - 三种类型：消息队列、信号量和共享内存，它们在实现上具有很大的相似性。
    - 最初在AT&T公司的System V.2 UNIX版本中引入。
  - 套接字
    - 主要用于网络通信，可以在不同主机的进程间相互交换信息。

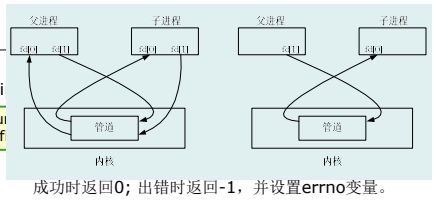
55

55

8.2 管道

- pipe函数
  - 可通过调用pi

```
#include <u
int pipe(int f
```



- filedes
- 返回值　成功时返回0；出错时返回-1，并设置errno变量。
- 说明
  - filedes[0]为读而打开，filedes[1]为写而打开。filedes[1] 的输出将作为filedes[0]的输入。写入写端的数据将被内核缓存，直到从读端读出。
  - 通常，调用pipe函数的进程会接着调用fork，以创建一个父进程与子进程之间的IPC通道。
  - fork之后有两个操作选择，这取决于所需建立的管道的数据流向。根据不同流向，在父子进程中要关闭相应端口。
  - 在通信过程中的读、写规则：
    - 当读一个写端已被关闭的管道时，在所有数据都被读取后，read返回0，以指示到了文件结束处。
    - 如果写一个读端已被关闭的管道，则产生SIGPIPE信号。如果忽略该信号或者捕获该信号并从其处理程序返回，则write出错返回，errno设置为EPIPE。
  - 在写管道时，已写但尚未被读走的字节数应小于或等于PIPE_BUF(4096B).

56

56

8.2 管道

- pipe函数
  - 示例[8-

Progr

```
1 #include <sys/wait.h
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <string.h>
6
7 int main(int argc, cha
8 {
9     int pipefd[2];
10    pid_t cpid;
11    char buf;
12
13    if (argc != 2) {
14        fprintf(stderr, "
15        exit(EXIT_FAIL
16    }
17
18    if (pipe(
19        perror
20        exit(E
21    }
```

```
22
23    cpid = fork();            /* create child process */
24    if (cpid == -1) {
25        perror("fork");
26        exit(EXIT_FAILURE);
27    }
28
29    if (cpid == 0) {          /* read pipe in child */
30        close(pipefd[1]);     /* close write end */
31
32        while (read(pipefd[0], &buf, 1) > 0)
33            write(STDOUT_FILENO, &buf, 1);
34
35        write(STDOUT_FILENO, "\n", 1);
36        close(pipefd[0]);
37        exit(EXIT_SUCCESS);
38    } else {                  /* write argv[1] to pipe in prarent */
39        close(pipefd[0]);     /* close read end */
40        write(pipefd[1], argv[1], strlen(argv[1]));
41        close(pipefd[1]);     /* close write end, so EOF will be read */
42        wait(NULL);           /* wait the exit of child */
43        exit(EXIT_SUCCESS);
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_pipe ex_pipe.c
jianglinmei@ubuntu:~/c$ ./ex_pipe "this is a string from father"
this is a string from father
```

57

57

8.2 管道

popen和pcolse函数

- 这两函数用于实现：创建管道，fork子进程，然后关闭管道的不使用端，在子进程中exec一个shell以执行一条命令，然后等待命令的终止。

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- command　将在子进程中执行的命令行字符串。
- type　打开方式，应为"r"或"w"二者之一，含义与fopen的第二个参数一样。
- 返回值
  - 成功时，popen返回一个文件流指针，pclose返回command的终止状态。
  - 出错时，popen返回NULL，pclose返回-1。
- 说明
  - 函数popen先调用fork函数创建子进程，然后调用exec函数以执行command，最后返回一个标准I/O文件指针。
  - popen打开的文件流是全缓存的。
  - 函数pclose关闭标准I/O流，并等待command命令执行结束，最后返回shell的终止状态。
  - popen函数执行command的方式同system()函数，相当于在Shell下执行：“sh –c command”。

58

58

8.2 管道

- popen和pcc
  - 示例[8-2

Program 8-2 ex_p

```
1 #include <ctype.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(void)
6 {
7     int c;
8     while( (c = getchar(
9         if( isupper(c) )
10            c = tolower(c
11        if( putchar(c) ==
12            printf("fail in
13            exit
14        }
15        if(c ==
16            fflus
17    }
18    exit(0);
19 }
```

Program 8-2 ex_popen.c

```
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(void)
6 {
7     char    line[BUFSIZ];
8     FILE*   fpRead;
9
10    if( (fpRead = popen("./ex_popen_filter", "r")) == NULL) {
11        printf("Fail to call popen()!\n");
12        exit(-1);
13    }
14
15    for( ; ; ) {
16        fputs("PipeIn> ", stdout);
17        fflush(stdout);
18        if( fgets(line, BUFSIZ, fpRead) == NULL ) /* read pipe */
19            break;
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_popen ex_popen.c
jianglinmei@ubuntu:~/c$ ./ex_popen
PipeIn> Hello, world!
hello, world!
PipeIn> Data from FILTER.
data from filter.
PipeIn>                    [note: press Ctrl + D]
```

```
32      exit(0);
33 }
```

59

59

8.3 命名管道（FIFO）

先入先出队列（First In First Out Queue）
FIFO也是一种文件类型。FIFO的路径名存在于文件系统中，创建命名管道类似于创建文件。
可调用mkfifo函数来创建一个命名管道。

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- pathname　要打开或创建的文件的名字。
- mode　存取访问权限，同open函数的mode参数。
- 返回值　成功时返回0，出错时返回-1并设置errno。
- 说明
  - 新创建的FIFO文件的所有者是当前进程的有效用户，文件的所属组是当前进程的有效组或父目录的所属组。
  - 一旦用mkfifo创建了一个FIFO文件，就可用系统I/O调用(close, read, write, unlink, etc.) 来操作它。
  - 未指定O_NONBLOCK时，以读方式打开FIFO将阻塞到某个其他进程以写方式打开该FIFO。反之，以写方式打开FIFO将阻塞到某个其他进程以读方式打开它。
  - 指定了O_NONBLOCK，则以读方式打开FIFO会立即返回。但是，以写的方式打开没有读端的FIFO将以失败返回，并设置errno为ENXIO。

60

60

8.3 命名管道（FIFO）

示例[8-4 ex_fifo.c & 8-5 ex_fifo_writer.c]

Program 8-4 ex_fifo.c

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main(void)
9 {
10    int fd;
11    char buf;
12    char *fifofile = "/tmp/myfifo1234";
13
14    if(access(fifofile, F_OK) != 0) {    /* file doesn't exist */
15       if(mkfifo(fifofile, 0744) == -1) {   /* create FIFO */
16          printf("Fail to create fifo.\n");
17          exit(EXIT_FAILURE);
18       }
19    }
20
21    if( (fd = open(fifofile, O_RDONLY)) == -1) {
22       printf("Fail to open fifo.\n");
23       exit(EXIT_FAILURE);
24    }
25
26    while (read(fd, &buf, 1) > 0)  /* read FIFO */
27       write(STDOUT_FILENO, &buf, 1);
28    write(STDOUT_FILENO, "\n", 1);
29
30    close(fd);
31    unlink(fifofile);                  /* delete FIFO file */
32    exit(EXIT_SUCCESS);
33 }
```

61

61

8.3 FIFO

示例[8-4 ex_fifo.c & 8-5 ex_fifo_writer.c]

Program 8-5 ex_fifo_writer.c

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 int main(int argc, char* argv[])
10 {
11    int fd;
12    char *fifofile = "/tmp/myfifo1234";
13
14    if (arg
15       fpri
16       exi
17    }
18
19    if( (fd
20       prin
21       exi
22    }
23
24    write(fd, argv[1], strlen(argv[1]));/*write FIFO*/
25
26    close(fd);     /* close write-end, EOF will be read */
27    exit(EXIT_SUCCESS);
28 }
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_fifo ex_fifo.c
jianglinmei@ubuntu:~/c$ gcc -o ex_fifo_writer ex_fifo_writer.c
jianglinmei@ubuntu:~/c$ ./ex_fifo &  [run at background]
[1] 31381
jianglinmei@ubuntu:~/c$   ./ex_fifo_writer "Hello, FIFO."
Hello, FIFO.
[1]+  完成                  ./ex_fifo
```

62

62

8.4 SysV IPC

标识符和关键字

- 每种SysV IPC结构，内核都有一个标识符（非负整数）予以标识和引用。
- 与文件描述符不同，SysV IPC标识符是进程无关并连续递增的。
- 使用ipcs命令和ipcrm命令可分别查看和删除系统中的SysV IPC。
- 创建SysV IPC结构时，必须指定一个关键字（key），关键字的数据类型为key_t，通常在头文件<sys/types.h>中规定为长整型。
- 创建SysV IPC的函数都带两个的参数：一个ket_t型的key和一个整型的flag。
- 创建新的IPC结构，则必须① 指定key为IPC_PRIVATE；或② 在flag中设置IPC_CREAT位。
- 访问现存的IPC结构，key必须与创建该IPC时所指定的关键字一致，并且不应指定IPC_CREAT。
- 在进程间共用一个SysV IPC结构的方法：
  - 创建一个IPC结构时以IPC_PRIVATE作为参数，然后将返回的标识符存放在某处（例如一个文件）以便其它进程取用。
  - 在父进程指定IPC_PRIVATE创建一个新IPC结构，所返回的标识符在fork后可由子进程使用，然后子进程可将此标识符作为exec函数的一个参数传给一个新进程。
  - 指定一个具体的关键字。该关键字如果已与一个IPC结构相结合，并且创建时flag参数中同时指定了IPC_CREAT和IPC_EXCL位，则创建函数会返回EEXIST。此时应删除已存在的IPC结构，然后再创建它。

63

63

8.4 SysV IPC

访问权限结构

- SysV IPC为每一个IPC结构都设置了一个ipc_perm结构。该结构规定了访问权限和所有者。

```
struct ipc_perm {
    key_t        __key;     /* 键 */
    uid_t        uid;       /* 所有者有效用户ID */
    gid_t        gid;       /* 所有者有效组ID */
    uid_t        cuid;      /* 创建者有效用户ID */
    gid_t        cgid;      /* 创建者有效组ID */
    unsigned short mode;    /* 访问权限 */
    unsigned short __seq;   /* 序列号 */
};
```

- 在创建IPC结构时，除seq以外的所有字段都会被赋初值。
- IPC结构的创建进程或超级用户的进程可以调用ctl函数（semctl、shmctl或msgctl）修改uid、gid和mode字段。
- mode字段的值类似于open函数的mode参数，但是对于任何IPC结构都不存在执行权限。

64

64

8.5 信号量

简介

- 信号量是一个计数器，用于多进程存取共享资源时的同步操作。使用信号量执行共享资源的访问控制应遵循“获取”和“释放”规则。
- 需获取共享资源时：
  ① 测试控制该资源的信号量；
  ② 若信号量的值为正，则进程可以使用该资源，将信号量值减1，表示它使用了一个资源单位；
  ③ 若此信号量的值为0，则进程进入睡眠状态，直至信号量值大于0时被唤醒，返回①。
- 使用完共享资源时：
  ① 使该信号量值增1；
  ② 如果有进程正在睡眠并等待此信号量，则唤醒它们。
- 获取和释放信号量的操作分别被称为“P操作”和“V操作”。信号量的测试、增1和减1操作应当是原子操作。

65

65

8.5 信号量

简介

- 信号量的初值可以是任一正值，该值说明有多少个共享资源单位可供使用。
- 最常用的信号量是二值信号量，它控制单个资源，初始值为1。
- SysV的信号量较复杂：
  - SysV以信号量集（多个信号量组成的集合）而非单一的信号量来处理信号量；
  - 创建信息量（semget）与对其赋初值（semctl）分开；
  - 即使已没有进程在使用某个信号量集，它仍然存在于系统，因此必须在使用完后，手动清除已创建的信号量集。

66

66

8.5 信号量

semget函数

- 创建一个新的信号量集或是获得一个已存在的信号量集.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

- key　用于标识一个信号量集的键，通常为一长整数.
- nsems　信号量的数目.
- semflg　标志位，低9位用于指定访问权限。IPC_CREAT表示创建信号量集，可按位或IPC_EXCL。
- 返回值　成功时，返回信号量集的标识符（一个非负整数）。出错时，返回-1并设置errno。常见的错误值如下:
  - EACCES　无访问权限.
  - EEXIST　在同时指定IPC_CREAT和IPC_EXCL时，具有指定键的信号量集已存在.
  - EINVAL　nsems < 0 或大于信号量数的极限值SEMMSL.
  - ENOENT　键不存在.

67

8.5 信号量

semop函数

- 用于改变信

```
struct sembuf {
    unsigned short sem_num;/* 信号量在信号量集中的序号*/
    short sem_op;          /* 要做的操作*/
    short sem_flg;         /* 选项标志*/
}
#include <sys/
#include <sys/
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

- semid　由semget返回的标识符.
- sops　指向一个结构体数组首元素的指针。该数组的每个元素指示了要对信号量集中的哪个信号量做何操作.
- nsops　结构体数组的元素个数.
- 返回值　成功时，返回0。出错时，返回-1并设置errno.
- 说明
  - struct sembuf的成员sem_op通常使用两个值：-1表示执行P操作，用来等待一个信号量变为可用；+1是表示执行V操作，用来通知一个信号量可用。
  - struct sembuf的成员sem_flg通常设置为SEM_UNDO。该标志让操作系统跟踪当前进程对信号量所做的改变，如果占有信号量的进程终止时没有将其释放，操作系统将自动释放该信号量。
  - sops所指示的所有动作会按数组元素的顺序全部“原子性”地执行，从而可以避免多个信号量的同时使用所引起的竞争条件。

68

8.5 信号量

semctl函数

- 可对信

```
union semun {
    int              val;    /* Value for SETVAL */
    struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Array for GETALL, SETALL */
    struct seminfo  *__buf;  /* Buffer for IPC_INFO */
};
#include <
#include <
#include <
int semctl(int semid, int semnum, int cmd, ...);
```

- semid　由semget返回的标识符.
- semnum　信号量在信号量集中的序号，当cmd为IPC_RMID时无效.
- cmd　要执行的动作.
- …　依赖于cmd参数的可选参数，应为union semun类型.
- 返回值　成功时，返回一个非负数。出错时，返回-1并设置errno.
- 常用的cmd值:
  - IPC_SET　初始化信号量的值（该值表示可用共享资源的数量），应通过第四个参数semun联合体的val成员来传递该值，并且应在第一次使用信号量集之前设置.
  - IPC_RMID　删除一个信号量集，信号量集的所有者或创建者才有权删除。如果没有显示删除信号量集，它会一直存在于系统中。删除后，任何进程继续使用该信号量集将得到一个EIDRM错误.

69

8.5 信号量

示例[8-6 ex_semaphore.c]

Program 8-6 ex_semaphore.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/types.h>
 5 #include <sys/ipc.h>
 6 #include <sys/sem.h>
 7
 8 static int set_semvalue(void);
 9 static int semaphore_p(void);
10 static int semaphore_v(void);
11 static void del_sem_set(void);
12
13 /* 定义自己的semun联合体 */
14 union semun {
15     int              val;
16     struct semid_ds  *buf;
17     unsigned short   *array;
18     struct seminfo   *__buf;
19 };
20
```


70

8.5 信号量

```
21 /* defines global variable sem_id to store the identified of semaphore set */
22 static int sem_id;
23
24 int main()
25 {
26     int i;
27     pid_t pid;
28     char ch;
29
30     /* create semaphore set */
31     sem_id = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
32     if(sem_id == -1) {
33         fprintf(stderr, "Failed to create semaphore set. \n");
34         exit(EXIT_FAILURE);
35     }
36     if(!set_semvalue()) {          /* set semaphore value */
37         fprintf(stderr, "Failed to initialize semaphore\n");
38         exit(EXIT_FAILURE);
39     }
40
41     pid = fork();                  /* create child process */
```


71

8.5 信号量

```
42     switch(pid)
43     {
44     case -1:
45         del_sem_set();         /* delete semaphore set */
46         exit(EXIT_FAILURE);
47     case 0:                    /* child process */
48         ch = 'O';
49         break;
50     default:                   /* parent process */
51         ch = 'X';
52         break;
53     }
54
55     srand((unsigned int)getpid());  /* seed for random number */
56
57     for(i=0; i < 10; i++){         /* Using semaphore to control critical region */
58         semaphore_p();             /* P operation，obtain semaphore */
59         printf("%c", ch);          /* output 'X' in parent，' O' in child */
60         fflush(stdout);
61         sleep(rand() % 4);
62
63         printf("%c", ch);
64         fflush(stdout);
65         sleep(1);
66         semaphore_v();             /* V operation，release semaphore */
67     }
```

72

8.5 信号量

```
68
69    if(pid > 0) {                  /* parent process */
70        wait(NULL);                 /* wait child process's exit */
71        del_sem_set();              /* delete semaphore set */
72    }
73
74    printf("\n%d - finished\n", getpid());
75    exit(EXIT_SUCCESS);
76 }
77
78 /* set semaphore's value */
79 static int set_semvalue(void)
80 {
81    union semun sem_union;
82
83    sem_union.val = 1;
84    if(semctl(sem_id, 0, SETVAL, sem_union) == -1)
85        return 0;
86
87    return 1;
88 }
89
```

73

73

8.5 信号量

```
90 /* P operation，obtain semaphore*/
91 static int semaphore_p(void)
92 {
93    struct sembuf sem_b;
94
95    sem_b.sem_num  = 0;
96    sem_b.sem_op   = -1;
97    sem_b.sem_flg  = SEM_UNDO;
98    if(semop(sem_id, &sem_b, 1) == -1) {
99        fprintf(stderr, "semaphore_p failed/n");
100       return 0;
101   }
102
103   return 1;
104 }
105
```

74

74

8.5 信号量

```
106 /* V operation，release semaphore */
107 static int semaphore_v(void)
108 {
109    struct sembuf sem_b;
110
111    sem_b.sem_num  = 0;
112    sem_b.sem_op   = 1;
113    sem_b.sem_flg  = SEM_UNDO;
114    if(semop(sem_id, &sem_b, 1) == -1) {
115        fprintf(stderr, "semaphore_v failed/n");
116        return 0;
117    }
118
119    return 1;
120 }
121
122 /* delete semaphore set */
123 static void del_sem_set(void)
124 {
125    union semun sem_union;
126
127    if(semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
128        fprintf(stderr, "Failed to delete semaphore/n");
129 }
```

75

75

8.5 信号量

示例 [8-6 ex_semaphore.c] result

```
jianglinmei@ubuntu:~/c$ gcc -o ex_semaphore ex_semaphore.c
jianglinmei@ubuntu:~/c$ ./ex_semaphore
XXOOXXOOXXOOXXXXOOXXOOXXOOXXOOXXOOXXOOOO
2707 - finished

2706 - finished
```

76

76

8.6 共享内存

简介

- 共享内存允许两个或多个进程访问同一块存储区，是最快的IPC机制。
- 使用共享内存必须注意多个进程之间对同一内存段的同步存取。一般使用信号量进行互斥访问。
- 使用一块共享内存的过程：
  - 由一个进程分配；
  - 需要访问这个共享内存段的每一个进程都将这个共享内存绑定到自己的地址空间中；
  - 完成通信后，所有进程则将共享内存与自己的地址空间分离开来；
  - 由一个进程释放该共享内存段。
- Linux每个进程都维护一个从物理页面地址到虚拟页面地址之间的映射。
- 分配共享内存就是由内核分配一或多个新的内存页面。一个进程如需使用这个共享内存段，则必须建立进程本身的虚拟地址到共享内存页面地址之间的映射，该过程称为绑定。当对共享内存的使用结束之后，解除这个映射关系，该过程称为分离。

77

77

8.6 共享内存

简介

- 内核为每个共享内存段设置了一个shmid_ds结构用以管理共享内存。

```
struct shmid_ds {
    struct ipc_perm shm_perm;   /* 所有者和权限标识 */
    size_t          shm_segsz;  /* 以字节为单位的段的长度 */
    time_t          shm_atime;  /* 最后绑定时间 */
    time_t          shm_dtime;  /* 最后分离时间 */
    time_t          shm_ctime;  /* 最后更改时间 */
    pid_t           shm_cpid;   /* 创建者进程的PID */
    pid_t           shm_lpid;   /* 最后绑定或分离进程的PID */
    shmatt_t        shm_nattch; /* 绑定数 */
    ...
};
```

- shmatt_t是一个至少比unsigned short更大的整数类型.

78

78

8.6 共享内存

shmget函数

- 分配一个共享内存段.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

- key　用于标识一个共享内存的键，通常为一整数。
- size　共享内存的字节数，常向上取整到PAGE_SIZE的整数倍。
- shmflg　标志位，低9位用于指定访问权限。IPC_CREAT表示创建信号量集，可按位或IPC_EXCL。
- 返回值　成功时，返回共享内存的标识符（一个非负整数）。出错时，返回-1并设置errno。常见的错误值如下：
  - EACCES　无访问权限
  - EEXIST　在同时指定IPC_CREAT和IPC_EXCL时，具有指定键的信号量集已存在.
  - EINVAL　size小于SHMMIN或大于SHMMAX，或key所指共享内存已存在，但size比已存在的共享内存更大.
  - ENOENT　键不存在.

79

79

8.6 共享内存

shmat函数

- 将共享内存绑定到当前进程的内存空间.

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- shmid　由shmget返回的标识符.
- shmaddr　绑定的地址（本进程地址空间的地址）.
- shmflg　标志位，常用的值为SHM_RND和SHM_RDONLY.
- 返回值　对于shmat，成功时返回共享内存的地址；出错时，返回(void*)-1并设置errno。
- 说明
  - 如果shmaddr指定为NULL，系统将选择一个合适的地址来绑定共享内存。这是推荐的做法。
  - 如果shmaddr不为NULL，shmflg指定了SHM_RND，则实际绑定地址为shmaddr 向下舍入到最近的SHMLBA(=PAGE_SIZE)的倍数的位置。目前，Linux下SHMLBA等于PAGE_SIZE（内存页面大小）。
  - 如果shmaddr不为NULL，shmflg未指定了SHM_RND，shmaddr必须为一页对齐（即页面大小的整数倍）的地址。
  - 如果shmflg指定了SHM_RDONLY，则共享内存以只读的方式绑定到本进程的地址空间，否则为“读写”方式。
  - shmat成功时更改与共享内存关联的shmid_ds 结构，将其shm_atime成员 设置为当前时间，shm_lpid成员设置为当前进程的PID，shm_nattch成员的值则增1。

80

80

8.6 共享内存

shmdt函数

- 将已绑定的共享内存与当前进程的内存空间相分离。

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

- shmaddr　绑定的地址（本进程地址空间的地址）
- 返回值　成功时返回0，失败时返回-1并设置errno
- 说明
  - shmdt如果成功也会更改与共享内存关联的shmid_ds 结构，将其shm_dtime成员设置为当前时间，shm_lpid成员设置为当前进程的PID，shm_nattch成员的值则减1。
  - 如果shm_nattch的值变成了0，且共享内存段标记为删除，则相应的共享内存段被删除。
  - 用fork创建的子进程将继承已绑定的共享内存段。但在子进程中调用exec系列函数后，所有已绑定的共享内存段会与新进程分离。
  - 当进程调用_exit函数退出的时候，所有已绑定的共享内存段也会自动从进程分离出去。

81

81

8.6 共享内存

shmctl函数

- 可对共享内存进行多种类型的控制。

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- shmid　由shmget返回的标识符
- cmd　要执行的动作
- buf　用于设置（cmd为IPC_SET）或获取（cmd为IPC_STAT）共享内存段的信息
- 返回值　成功时返回非负数，失败时返回-1并设置errno
- 说明
  - 常用的cmd命令是IPC_RMID，用于删除一个共享内存段。
  - 在进程中调用 exit 和 exec 会使进程分离共享内存段，但不会删除这个内存段。在结束使用每个共享内存段的时候都应当使用 shmctl的IPC_RMID命令进行释放。

82

82

8.6 共享内存

示例[8-7 ex_sharememory.c]

Program 8-7 ex_sharememory.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/sem.h>
7
8 static int set_semvalue(void);
9 static int semaphore_p(void);
10 static int semaphore_v(void);
11 static void del_sem_set(void);
12
13 /* define oure own semun union */
14 union semun {
15     int              val;
16     struct semid_ds     *buf;
17     unsigned short      *array;
18     struct seminfo      *__buf;
19 };
20
21 /* define global variable sem_id to store the identifier of semaphore set */
22 static int sem_id;
23 /* define global variable shm_id to store the identifier of shared memory */
24 static int shm_id;
```

83

83

8.6 共享内存

```
25
26 int main()
27 {
28     int i;
29     pid_t pid;
30     char ch1, ch2;
31     char* pData = NULL;
32
33     /* create semaphore set */
34     sem_id = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
35     if(sem_id == -1) {
36         fprintf(stderr, "Failed to create semaphore set. \n");
37         exit(EXIT_FAILURE);
38     }
39     if(!set_semvalue()) {                    /* set semaphore value */
40         fprintf(stderr, "Failed to initialize semaphore\n");
41         exit(EXIT_FAILURE);
42     }
43     shm_id = shmget(IPC_PRIVATE, 4096, 0666 | IPC_CREAT);
44     if(shm_id == -1) {
45         fprintf(stderr, "Failed to create sharememory. \n");
46         del_sem_set();
47         exit(EXIT_FAILURE);
48     }
49
```

84

84

8.6 共享内存

```
50    pid = fork();                          /* create child process */
51    if(pid == -1) {
52       perror("fork failed");
53       shmctl(shm_id, IPC_RMID, 0);        /* delete shared memory */
54       del_sem_set();
55       exit(EXIT_FAILURE);
56    }
57    else {
58       srand((unsigned int)getpid());      /* seed for random number */
59       pData = (char*)shmat(shm_id, 0, 0); /* binding */
60
61       if (pid == 0) {                     /* child process */
62          do {
63             semaphore_p();
64             ch1 = *pData;                /* read */
65             ch2 = *(pData + 1);
66             if(ch2 == '@') {
67                *pData = tolower(ch1);  /* write */
68                *(pData + 1) = '#';
69             }
70             if(ch1 == 'Z') break;
71             semaphore_v();
72
73             sleep(1);
74          }while(1);
75       }
```

85

8.6 共享内存

```
76       else {                              /* parent process */
77          for(i=0; i < 26; i++){
78             semaphore_p();
79             *pData = 'A' + rand() % 26; /* write */
80             if(i == 25) *pData = 'Z';
81             printf("%c", *pData);
82             *(pData + 1) = '@';
83             semaphore_v();
84             sleep(1);
85
86             do {
87                semaphore_p();
88                ch1 = *pData;          /* read */
89                ch2 = *(pData + 1);
90                if(ch2 == '#') {
91                   printf("%c", ch1);
92                   fflush(stdout);
93                   semaphore_v();
94                   break;
95                }
96                semaphore_v();
97             }while(1);
98          }
99       }
100
101      shmdt(pData);                       /* detach */
102   }
```

86

8.6 共享内存

```
103
104   if(pid > 0) {                          /* parent process */
105      wait(NULL);                         /* waiting for child process to exit */
106      shmctl(shm_id, IPC_RMID, 0);        /* delete shared memory */
107      del_sem_set();                      /* delete semaphore set */
108   }
109
110   printf("\n%d - finished\n", getpid());
111   exit(EXIT_SUCCESS);
112 }
113 ..... /* omit some codes the same as what are in program 8-6*/
```

```
jianglinmei@ubuntu:~/c$ gcc -o ex_sharememory ex_sharememory.c
jianglinmei@ubuntu:~/c$ ./ex_sharememory
QqUuXxVvTtCcEeKkLlBbCcQqQqRrEeYyYyKkOoGgMmOoLlRrXx
1808 - finished
Zz
1807 - finished
```

87

8.7 消息队列

简介

- 消息队列是消息的链接表，存放在内核中并由消息队列标识符标识。
- 使用消息队列可以从一个进程向另一个进程发送数据块。每个数据块有一个最大长度的限制MSGMAX，所有队列的全部数据块的总长度也有一个上限值MSGMNB。
- 消息队列独立于发送和接收进程而存在。消息队列和队列中的内容保留在文件系统中。
- 内核为每个消息队列设置了一个shmid_ds结构用以管理消息队列。

```
struct msqid_ds {
   struct ipc_perm msg_perm;     /* 所有者和权限标识*/
   time_t          msg_stime;    /* 最后一次发送消息的时间   */
   time_t          msg_rtime;    /* 最后一次接收消息的时间   */
   time_t          msg_ctime;    /* 最后改变时间*/
   unsigned long   __msg_cbytes; /* 队列中当前数据字节数*/
   msgqnum_t       msg_qnum;     /* 队列中当前消息数*/
   msglen_t        msg_qbytes;   /* 队列允许的最大字节数     */
   pid_t           msg_lspid;    /* 最后发送消息的进程的PID */
   pid_t           msg_lrpid;    /* 最后接收消息的进程的PID */
};
```

88

8.7 消息队列

msgget函数

- 创建或获取一个消息队列.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- key　　用于标识一个消息队列的键，通常为一整数。
- msgflg　　标志位，低9位用于指定访问权限。IPC_CREAT表示创建信号量集，可按位或IPC_EXCL。
- 返回值　　成功时，返回消息队列的标识符（一个非负整数）。出错时，返回-1并设置errno。常见的错误值如下：
  - EACCES　　无访问权限.
  - EEXIST　　在同时指定IPC_CREAT和IPC_EXCL时，具有指定键的消息队列已存在.
  - ENOENT　　键不存在.
- 说明
  - 调用msgget函数时，参数msgflg指定IPC_CREAT而未指定IPC_EXCL，如果具有指定键的消息队列已存在，则只是忽略创建动作，而不会出错。

89

8.7 消息队列

msgsnd函数

- 把一条消息添加到消息队列中。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

- msqid　　由msgget返回的消息队列标识符
- msgp　　指向要发送的消息的缓冲区的指针
- msgsz　　消息长度。这个长度不包括长整形成员变量的长度。
- msgflg　　标志。指定IPC_NOWAIT时，表示当队列满或达到系统限制时，函数立即返回（返回值为-1），不发送消息。
- 返回值　　成功时返回0；失败时返回-1，并设置errno变量。
- 说明
  - 未指定IPC_NOWAIT时，　如果队列满或达到系统限制，函数返回-1，错误为EAGAIN。

```
struct my_message {
   long int message_type; /* 消息类型 */
   <anydatatype> data;   /* 要发送的数据*/
} msg;
```

90

8.7 消息队列

msgrcv函数

- 从一个消息队列获取消息

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
               int msgflg);
```

- msqid　由msgget返回的消息队列标识符
- msgp　指向准备接收消息的缓冲区的指针
- msgsz　消息长度。这个长度不包括长整形成员变量的长度。msgp所指向的缓冲区应大于或等于此长度。
- msgtyp　一个长整数，可能情况：
  - 若值为0，获取队列中的第一个可用消息；
  - 若值大于0，获取具有相同类型的第一个消息；
  - 若小于0，获取消息类型小于或等于其绝对值的第一个消息。
- msgflg　标志。指定IPC_NOWAIT时，表示当没有相应类型消息时，函数立即返回（返回值为-1），不接收消息，errno设置为：ENOMSG
- 返回值　成功时返回0；失败时返回-1，并设置errno变量。

91

8.7 消息队列

msgctl函数

- 直接控制消息队列，可对消息队列做多种操作.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- msqid　由msgget返回的消息队列标识符.
- cmd　要采取的动作，常取以下值:
  - IPC_STAT　将内核所管理的消息队列的当前属性值复制到buf（msqid_ds结构）中.
  - IPC_SET　如果进程有足够的权限，就把内核所管理的消息队列的当前属性值设置为buf（msgqid_ds结构）各成员的值
  - IPC_RMID　删除消息队列.
- buf　缓冲区，作用视cmd而定.
- return value　成功时返回0；失败时返回-1，并设置errno变量。
- 说明
  - 如果在进程正阻塞于msgsnd或msgrcv中等待时删除消息队列，则这两个函数将以失败返回。

92

8.7 消息队列

示例[8-8 ex_msgrcv.c & 8-9 ex_msgsnd.c]

Program 8-8 ex_msgrcv.c

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <errno.h>
 5 #include <unistd.h>
 6 #include <sys/msg.h>
 7 /* defines message structure */
 8 struct my_msg_st {
 9     long int my_msg_type;             /* message type*/
10     char some_text[BUFSIZ];           /* the data received */
11 };
12
13 int main()
14 {
15     int running = 1;
16     int msgid;
17     struct my_msg_st some_data;
18     long int msg_to_receive = 0;
19
20     /* Get or create message queue */
21     msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
22     if (msgid == -1) {
23         fprintf(stderr, "msgget failed with error: %d\n", errno);
24         exit(EXIT_FAILURE);
25     }
```

93

8.7 消息队列

```
26
27     while(running) {
28         /*receive message*/
29         if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
30                    msg_to_receive, 0) == -1) {
31             fprintf(stderr, "msgrcv failed with error: %d\n", errno);
32             exit(EXIT_FAILURE);
33         }
34         printf("You wrote: %s", some_data.some_text);
35         if (strncmp(some_data.some_text, "end", 3) == 0) {
36             running = 0;
37         }
38     }
39     /*delete message queue*/
40     if (msgctl(msgid, IPC_RMID, 0) == -1) {
41         fprintf(stderr, "msgctl(IPC_RMID) failed\n");
42         exit(EXIT_FAILURE);
43     }
44     exit(EXIT_SUCCESS);
45 }
```


94

8.7 消息队列

Program 8-9 ex_msgsnd.c

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <errno.h>
 5 #include <unistd.h>
 6 #include <sys/msg.h>
 7
 8 #define MAX_TEXT 512
 9 /* defines message structure */
10 struct my_msg_st {
11     long int my_msg_type;             /* message type */
12     char some_text[MAX_TEXT];         /* the data to send */
13 };
14
15 int main()
16 {
17     int running = 1;
18     struct my_msg_st some_data;
19     int msgid;
20     char buffer[BUFSIZ];
21     msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
22     if (msgid == -1) {
23         fprintf(stderr, "msgget failed with error: %d\n", errno);
24         exit(EXIT_FAILURE);
25     }
```


95

8.7 消息队列

```
26
27     while(running) {
28         printf("Enter some text: ");
29         fgets(buffer, BUFSIZ, stdin);
30         some_data.my_msg_type = 1;
31         strcpy(some_data.some_text, buffer);
32         /* send message */
33         if (msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
34             fprintf(stderr, "msgsnd failed\n");
35             exit(EXIT_FAILURE);
36         }
37         if (strncmp(buffer, "end", 3) == 0) {
38
39
40
41
42
43 }
```

```
jianglinmei@ubuntu:~/c$ ./ex_msgrcv &     [run in background]
[1] 2408
jianglinmei@ubuntu:~/c$ ./ex_msgsnd
Enter some text: Hello, message queue.
You wrote: Hello, message queue.
Enter some text: It's very easy to use it to
You wrote: It's very easy to use it to
Enter some text: transfer message between process.
You wrote: transfer message between process.
Enter some text: end
You wrote: end
[1]+  over                ./ex_msgrcv
```


96