Linux多进程开发

- 进程概述 57
- 进程状态转换 43
- exec函数族 18
- 进程控制 21
- 进程间通信
  - 概述 4
  - 匿名管道 59
  - 有名管道 22
  - 内存映射 16
  - 信号
    - 信号的概念
      - 产生信号四种方式
        - 前台进程输入
        - 硬件异常
        - 进程状态改变
        - 执行kill命令
      - 使用信号的两个主要目的是:
        - 让进程知道已经发生了一个特定的事情,通知中断
        - 强迫进程执行它自己代码中的信号处理程序。
      - 信号的特点
        - 简单
        - 不能携带大量信息
        - 满足某个特定条件才发送
        - 优先级比较高
      - 查看系统定义的信号列表: kill -l
      - 前 31 个信号为常规信号，其余为实时信号。

```
yygoaheadhyygoahead-virtual-machine:~/linux/chapter2/mmap$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

前31个信号为常规信号，后边31个信号是预留信号。

      - 常见信号

编号	信号名称	对应事件	默认动作
2	SIGINT	用户通过ctrl+c组合键键时，用户终端向正在运行中的由该终端启动的程序发出的信号	终止进程
3	SIGQUIT	用户按下ctrl+\组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出信号	终止进程并产生core文件
4	SIGILL	CPU检测到某进程执行了非法指令	终止进程并产生core文件
5	SIGTRAP	该信号由断点指令或其他 trap指令产生	终止进程并产生core文件
6	SIGABRT	调用abort函数时产生该信号	终止进程并产生core文件
7	SIGBUS	非法访问内存地址，包括内存对齐出错	终止进程并产生core文件
8	SIGFPE	在发生致命的运算错误时发生，不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误	终止进程并产生core文件

编号	信号名称	对应事件	默认动作
9	SIGKILL	无条件终止进程，该信号不能被忽略，处理和阻塞	终止进程，可以杀死任何进程
10	SIGUSR1	用户定义的信号，即程序员可以在程序中定义并使用该信号	终止进程
11	SIGSEGV	指示进程进行了无效内存访问（段错误）	终止进程并产生core文件
12	SIGUSR2	另外一个用户自定义信号，程序员可以在程序中定义并使用该信号	终止进程
13	SIGPIPE	Broken pipe向一个没有读端的管道写数据	终止进程
14	SIGALRM	定时器超时，超时的时间 由系统调用alarm设置	终止进程
15	SIGTERM	程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止，通常用来要示程序正常退出。执行shell命令kill时，缺省产生这个信号	终止进程
16	SIGSTKFLT	Linux早期版本出现的信号，现仍保留向后兼容	终止进程

编号	信号名称	对应事件	默认动作
17	SIGCHLD	子进程结束时，父进程会收到这个信号	忽略这个信号
18	SIGCONT	如果进程已停止，则使其继续运行	继续/忽略
19	SIGSTOP	停止进程的执行，信号不能被忽略，处理和阻塞	为终止进程
20	SIGTSTP	停止终端交互进程的运行，按下<ctrl-z>组合键时发出这个信号	暂停进程
21	SIGTTIN	后台进程读终端控制台	暂停进程
22	SIGTTOU	该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生	暂停进程
23	SIGURG	套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达，	忽略该信号
24	SIGXCPU	进程执行时间超过了分配给进程的CPU时间，系统产生该信号并发送给进进程	终止进程

      - 信号的状态
        - 信号的三种状态：产生、未决、递达。
        - SIGKILL 和 SIGSTOP 信号不能被捕捉、阻塞或者忽略，只能执行默认动作。

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(){
4      char * buf;
5      strcpy(buf,"hello");
6      return 0;
7  }
```

编写一个存在指针越界的程序

      - core 文件调试
        - gcc core.c -g 对其进行可调试编译
        - 更改core文件存储大小

```
yygoaheadhyygoahead-virtual-machine:~/linux/chapter2/signal$ ulimit -a
core file size          (blocks, -c) 0
```

ulimit -c 1024

        - 运行错误程序，生成调试文件core
        - gdb中查看core文件内容

gdb a.out

```
(gdb) core-file core
[New LWP 113842]
Core was generated by `./a.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000000004004de in main () at core.c:5
5           strcpy(buf,"hello");
```

      - kill函数

```
/*
  #include <sys/types.h>
  #include <signal.h>
  int kill(pid_t pid, int sig);
      功能：给某个进程pid，发送某个信号sig(linux6类)
      参数：
          pid:将标进程号
              >0:将信号发送给指定进程
              =0: 发给当前进程组的所有进程
              =-1: 将信号发送给所有有权限接收信号的进程
              <-1:给进程组的组号等于|pid|绝对值的进程组发送信号
          sig:信号的编号或名称(建议使用名称)，0表示不发送任何信号
  #include <signal.h>
  int raise(int sig);
      功能：给当前进程发送信号
      参数：
          sig:要发送的信号
      返回值：
          成功0，失败为-1
  #include <stdlib.h>
  void abort(void);
      功能：发送SIGABRT信号给当前的进程，杀死当前的进程
*/
```

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid=fork();
    if (pid>0){
        //父进程
        printf("parent process\n");
        sleep(2);
        printf("kill child process\n");
        kill(pid,9);
    }else{
        //子进程
        int i=0;
        for (;i<5;i++){
            printf("child process\n");
            sleep(1);
        }
    }
}
```

```
yygoaheadhyygoahead-virtual-machine:~/linux/chapter2/signal$ gcc kill.c -o kill
yygoaheadhyygoahead-virtual-machine:~/linux/chapter2/signal$ ./kill
parent process
child process
child process
kill child process
```

子进程未全部执行完就被杀死了

      - alarm函数

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
    功能：设置定时器，函数调用时开始倒计时，倒计时结束时，
        内核会给当前进程发送一个信号，SIGALARM，该函数不阻塞
    参数：
        seconds:倒计时的时长，单位为秒，如果参数设置为0，定时器无效，
            不发送信号，取得以前设置的闹钟时间
    返回值：倒计时剩余的时间
        之前没有定时器返回0，
        之前有定时器返回之前定时器的剩余时间
SIGALARM:默认终止当前的进程，每个进程有且只有一个定时器。
        alarm(10);//返回0
        alarm(5);//此时会覆盖上一个定时器，并且返回上一个定时器剩余时间


*/
#include <unistd.h>
#include <stdio.h>
int main(){
    int second=alarm(5);
    printf("second-%d\n",second);
    sleep(2);
    second=alarm(5);//重新从5秒计时
    printf("second-%d\n",second);
    while (1){
        printf("still alive\n");
        sleep(1);
    }
    return 0;
}
```

```
yygoaheadhyygoahead-virtual-machine:~/linux/chapter2/signal$ ./alarm
second=0
second=3
still alive
still alive
still alive
still alive
still alive
Alarm clock
```

      - 计时0.5秒

```
/*
    实际运行的时间并不是alarm的1秒，在程序结束后还会有清空缓存区操作等，
    由于计时器的1秒中还包括io操作，所以实际上比真正要统计数要少很多，而且
    在程序运行结束后还会做新缓存区，因此可能实际只运行了实际时间是1秒，但
    是最终等程序结束后刷新完缓冲区总共用了3.4秒
    实际时间=内核时间+用户时间+真假的时间
    定时器与进程的状态无关，无论阻塞还是就绪计时仍在进行
*/
#include <unistd.h>
#include <stdio.h>
int main(){
    alarm(1);
    int i=0;
    while (1){
        printf("i-%d\n",i++);
    }
    return 0;
}
```

另外可以重定向至文件中，降低io次数

./alarm>>a.txt

      - setitimer函数

```
/*
    #include <sys/time.h>
    int getitimer(int which, struct itimerval *curr_value);
    int setitimer(int which, const struct itimerval *new_value,
                  struct itimerval *old_value);
        功能：设置定时器，可以替代alarm,精度可以达到us,可以实现周期性的定时
        参数：
            which:定时器以什么时间计时
                ITIMER_REAL：真实时间，时间到达发送SIGALRM
                ITIMER_VIRTUAL:用户时间，时间到达发送SIGVTALRM
                ITIMER_PROF：进程在用户态和核心态消耗的时间，时间到发送SIGPROF
            new_value:设置定时器属性
                struct itimerval { // 定时器的结构体
                    struct timeval it_interval; //每个阶段的时间，间隔时间
                    struct timeval it_value; //延时多久执行定时器
                };
                struct timeval { // 时间的结构体
                    time_t      tv_sec; //多少秒
                    suseconds_t tv_usec;//多少微秒数
                };
            old_value：记录上一次定时的时间参数，不使用指定NULL
        返回值：成功0，失败-1
*/
```

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
//终止与setitimer 一起
int main(){
    struct itimerval new_value;
    //设置间隔时间
    new_value.it_interval.tv_sec=2;
    new_value.it_interval.tv_usec=0;
    //设置延时时间
    new_value.it_value.tv_sec=3;
    new_value.it_value.tv_usec=0;
    int ret=setitimer(ITIMER_REAL,&new_value,NULL);//非阻塞
    if (ret==-1){
        perror("setitimer");
        exit(0);
    }
    while (1){
        printf("still alive\n");
        sleep(1);
    }
    return 0;
}
```

该函数是非阻塞的函数，因此会直接向下运行，而不是等待到超时三秒

而超时三秒后实际上发送一个SIGALRM信号，造成进程终止，而后边的周期设时并没有起到作用，需要后边进行信号捕捉。