阶段二:

1. 进程的概念 诞生 API函数接口
2. 进程间的通信,管道(有名和无名)、信号、消息队列、共享内存、信号量、套接字
3. 进程间的信号处理、信号集(注册信号和捕获信号)
4. 线程的概念、诞生 API函数接口
5. 线程的处理和线程安全(互斥锁、读写锁、条件变量)
6. 线程池(数据结构+线程使用+线程安全)

# 系统编程：

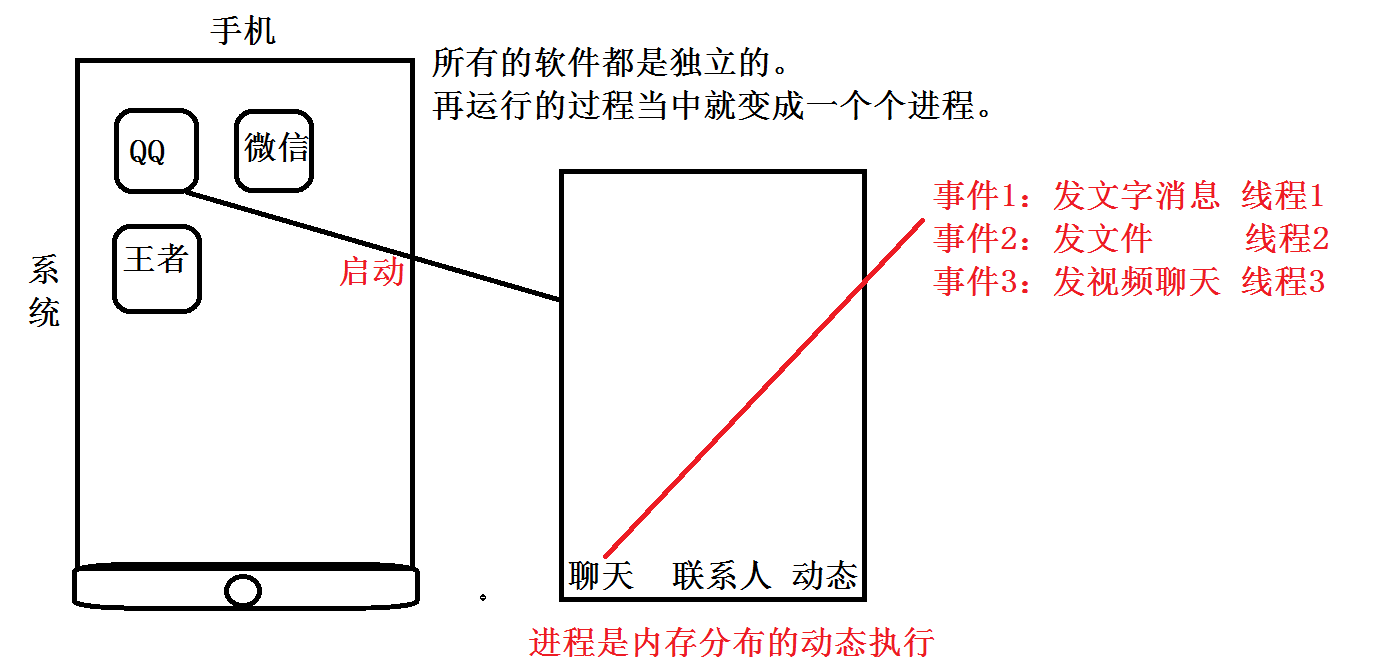
系统编程一般分为:windows系统编程和linux(或者unix)系统编程

进程和线程关系：

进程是系统资源管理的最小单位

线程是系统调度的最小单位

线程作用于进程，在同一个进程当中至少要有一个线程，进程当中的线程共享资源



# 进程

## 进程的概念

进程和程序的差别：（进程组成包括程序、数据、进程控制块PCB）

程序就是一个静态的软件(待执行),类似于拍电影的剧本

进程就是一个软件的运行(已执行),类似于电影执行的过程

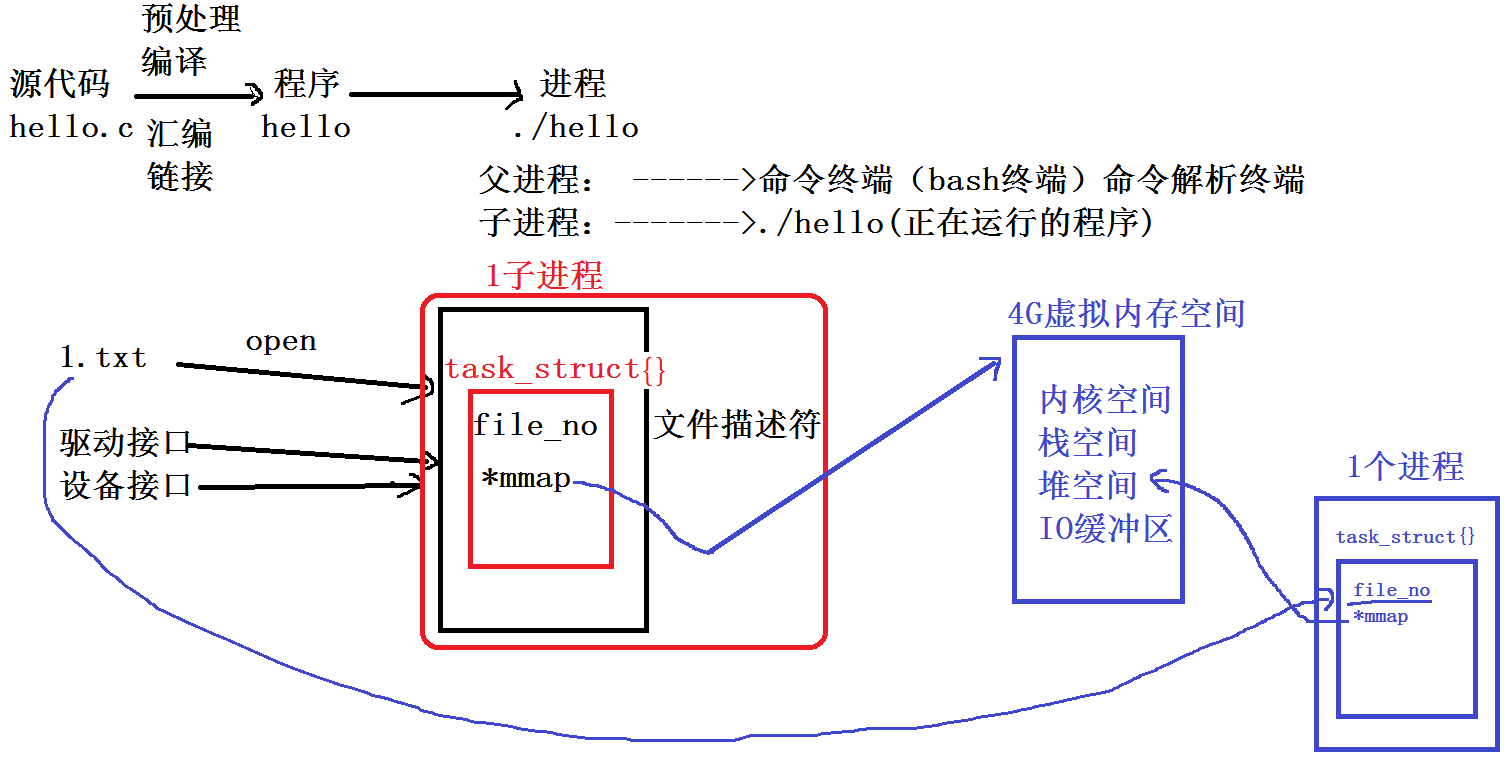
|  |  |  |
| --- | --- | --- |
| 程序 | 一堆待执行的数据 gcc hello.c -o hello 产生可执行文件hello | hello |
| 进程 | 只有程序加载到cpu，占用系统资源,根据代码执行相关的操作 | ./hello |

## 进程资源

可执行文件hello------>执行./hello---------》开启进程

命令获取程序的执行内容:readelf 文件名 -S

进程控制块PCB



以下的几个区域的数据直接从程序拷贝到内存当中

.Init 初始化

.Bss 和.data 数据段

.Rodata 常量段

1. 进程分配

程序执行的时候，除了发生分配空间以外，系统帮你配置任务结构体

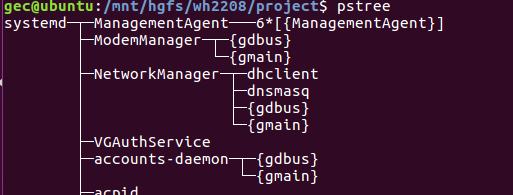
|  |  |  |
| --- | --- | --- |
| 比较  对象 | 数据获取 | 作用 |
| 目录 | 读取一个目录项的时候，返回一个目录结构体 | 这个结构体用于描述目录项  类型 大小 索引号 偏移量 文件名 |
| 进程 | 开启一个进程之后，配置一个任务结构体 | 这个结构体用于描述进程项  进程ID 文件 信号 父进程 ... |

如何查找结构体？

查找命令：grep -r “struct task\_struct {” /usr/src



1. 关于进程的命令
2. 查看这个linux系统进程的命令 pstree

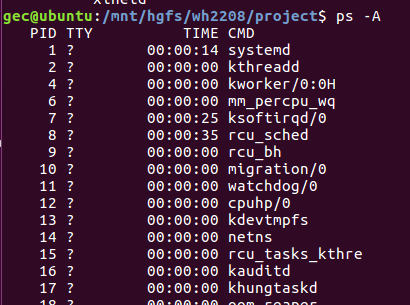


当前的所有进程的祖先进程是systemd，不同的内核进程名不一样

1. 查看当前操作的进程的属性

为了分辨进程,系统给每个进程提供单独的进程号，进程号取值是 1~65535

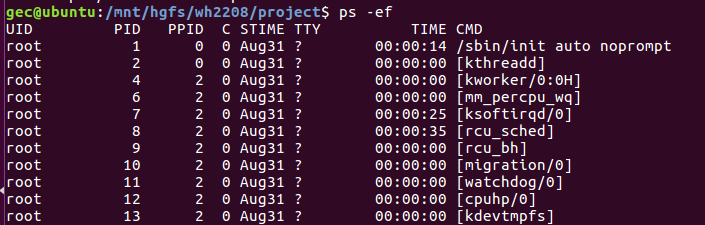
ps -A 查看所有进程的进程号



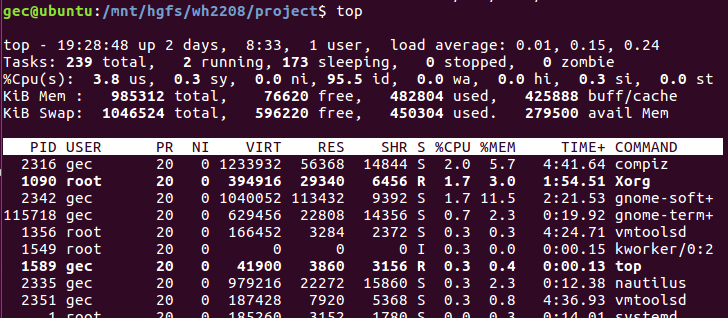
ps -a 只查看当前进程的进程号



1. 查看进程的PID号静态查找 （ps -ef）



1. 查看进程的PID号动态查找 （top） 退出q



具体的解释：F:\WH2208\07 系统编程\系统编程资料\进程参数.txt

## 进程状态

就绪态 TASK\_RUNNING 等待cpu资源

运行态 TASK\_RUNNING 占用cpu资源 5毫秒到10毫秒

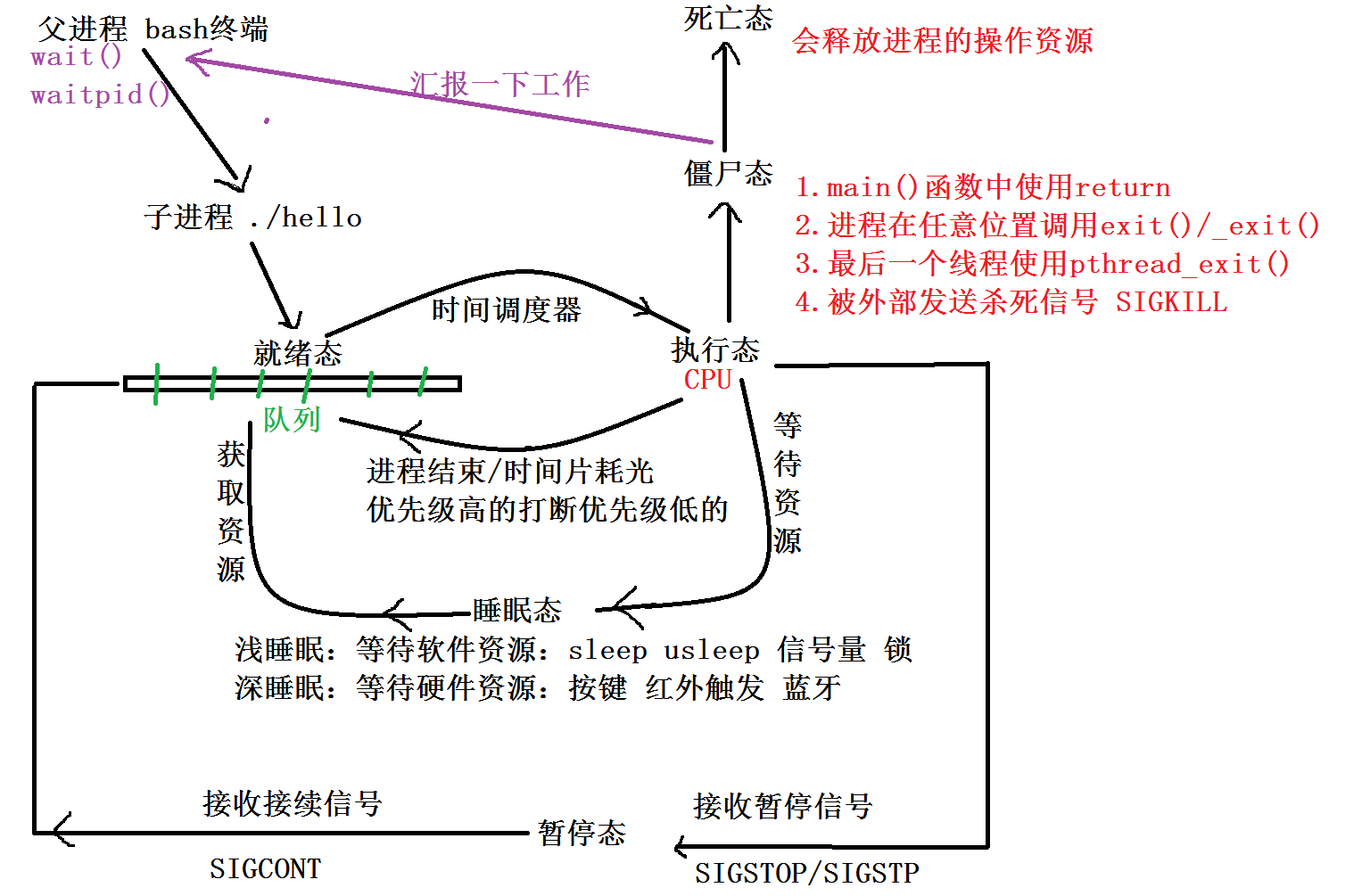
暂停态 TASK\_STOPPEO 接收到暂停信号

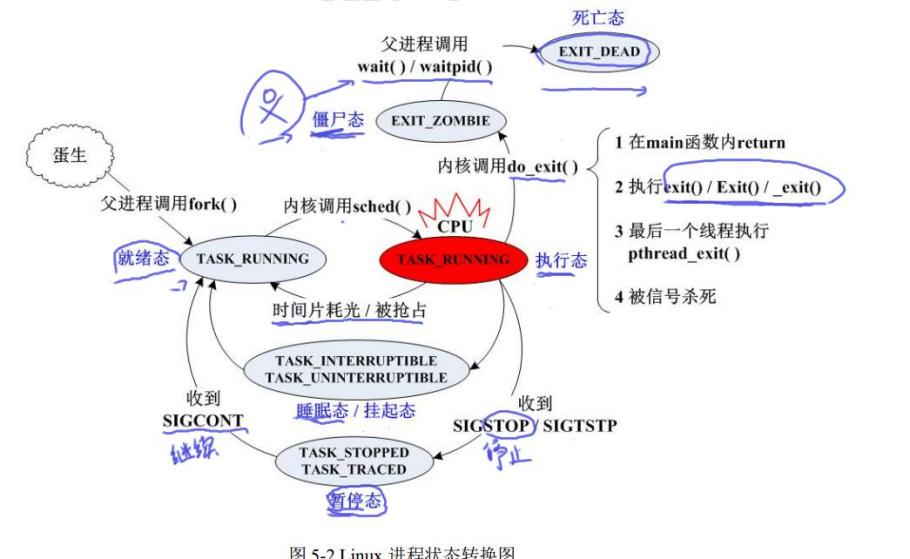
睡眠态 TASK\_INTERRUPTIBL 响应信号 浅度睡眠

TASK\_UNINTERRUPTIBL 不响应信号 深度睡眠

僵尸态 TASK\_ZOMBIE 进程退出的时候，首先变成是僵尸态，资源占用cpu资源

死亡态 DEAD 进程退出时候，资源回收完成，直接结束进程



书上配图

## 进程的操作API接口



1. 创建一个子进程 fork

|  |
| --- |
| NAME 创建一个字子进程  fork - create a child process  SYNOPSIS  #include <unistd.h>头文件  //函数原型  pid\_t fork(void); 无需参数  返回值：  返回值 >0 返回父进程  返回值 ==0 返回创建的子进程  返回值 -1 创建子进程失败 |

分析：

|  |
| --- |
| int main()  {  printf("helloworkd\n");  fork();//调用创建子进程  //子进程和父进程会同时执行这条语句  printf("===helllo===\n");  return 0;  }  结果：    此时：父进程先执行，子进程后执行，由于父进程遇到return 0,所以父进程结束退出  子进程的打印出现在命令行，即父进程没有等待子进程的退出，导致子进程变成了孤儿进程  解决方案：父进程执行结束之后，等待子进程的执行汇报，然后再退出父进程，那么可以同时死亡。  pid\_t son=fork();//调用创建子进程  //子进程和父进程会同时执行这条语句  if(son > 0)//父进程  {  printf("===helllo===父进程\n");  printf("son=%d\n",son);  //延时1秒钟  sleep(1);  }  if(son == 0)//子进程  {  printf("===helllo===子进程\n");  } |

总结：

1. fork函数之后，父子进程并发执行
2. 父子进程执行，需要等待子进程的执行，否则子进程会进入僵尸态
3. Pid号没有负数，fork函数一个调用，两次返回结果

创建子进程 vfork

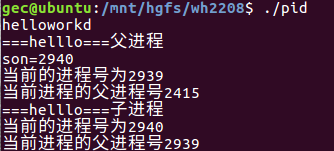
|  |
| --- |
| NAME 创建子进程并且阻塞父进程  vfork - create a child process and block parent  SYNOPSIS  #include <sys/types.h>  #include <unistd.h>  函数原型：  pid\_t vfork(void);//无参数据 |

区别：

1. vfork创建的子进程一定优先父进程执行
2. vfork创建的子进程和父进程共享数据

1. 查看自身的pid号和查看父进程的pid号

|  |
| --- |
| NAME 获取进程的pid号  getpid, getppid - get process identification  SYNOPSIS  #include <sys/types.h>  #include <unistd.h>  函数原型：  pid\_t getpid(void);//查看当前进程的pid号  pid\_t getppid(void);//查看当前进程的父进程pid号 |



注意：fork函数大于0的返回值，实际上就是当前进程创建子进程的pid号

1. 进程资源的退出操作

进程通过return或者exit进行退出操作，返回数值的取值范围是0~255

进程退出

return 退出进程前刷新IO缓冲区

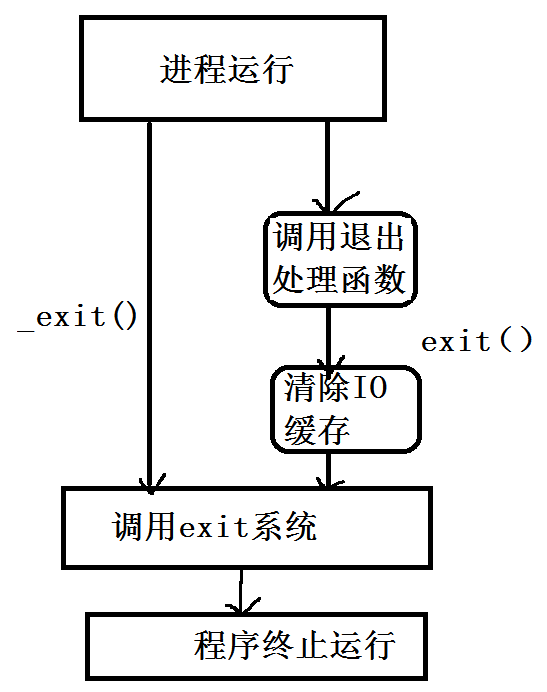
exit() 退出进程前刷新IO缓冲区

\_exit() 退出时不会刷新IO缓冲区

信号杀死 退出时候不会刷新IO缓冲区

由于在linux的标准函数库，有一种被称为“缓冲I/O (buffered I/O)”的操作，

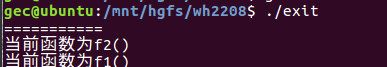
其特征就是对应每一个打开的文件，在内存当中都有一块缓冲区



退出函数

|  |
| --- |
| #include <stdlib.h>  /清空缓冲区，然后再退出  void exit(int status);//status退出的状态值  #include <unistd.h>  //不清空缓冲区，直接退出  void \_exit(int status);//status退出的状态值    #include <stdlib.h>  //注册一个函数，只有进程结束并且刷新缓冲区才会被调用  //正常情况不会直接调用，atexit所注册的函数，是栈的形式保存在缓冲区中  int atexit(void (\*function)(void)); |

exit调用情况



\_exit()调用情况



练习： 尝试创建孙子进程和儿子的兄弟进程，打印对应的pid号

1. 等待子进程的结束 wait

|  |
| --- |
| #include <sys/types.h>  #include <sys/wait.h>  函数原型：  pid\_t wait(int \*status);  参数：监听子进程退出的状态，如果不关注退出值的获取，那么直接写为NULL  返回值：  成功：退出的子进程的pid  失败：返回-1  状态值的获取：  WIFEXITED(status) 判断子进程是否正常退出，如果是返回为真  WEXITSTATUS(status) 判断子进程正常退出的值，解析子进程的退出值  WIFSIGNALED(status) 判断子进程是否为信号终止，如果是返回为真  WTERMSIG(status) 判断如果是信号终止，那么解析信号响应的编号 |



1. 等待子进程的结束 waitpid

|  |
| --- |
| #include <sys/types.h>  #include <sys/wait.h>  函数原型：  pid\_t waitpid(pid\_t pid, int \*status, int options);  函数的参数：  pid\_t pid, 子进程的pid号  < -1 等待这个负数的绝对值的pid的进程对象  -1 等待任何一个子进程对象 相当于wait waitpid(-1, &status, 0);  0 等待进程组当中的任何一个子进程  > 0 等待一个指定的子进程对象  int \*status, 存放的是子进程的退出值，如果不需要获取状态直接写为NULL  int options 子进程的操作方式  WNOHANG 监听子进程的退出状态，是一个非阻塞状态，  无论子进程是否退出那么都会立即返回。  0 监听子进程的退出，阻塞到子进程退出为止才结束 |

注意：

1. 子进程继承父进程大部分的资源，局部变量是各自使用
2. 子进程继承的过程中，不会继承父进程的id号
3. 子进程和父进程之间的空间都是独立的，相互不干扰并行执行
4. 父进程需要回收子进程的资源， 运行态 占用cpu资源 运行程序的中代码

僵尸态 占用cpu资源 不运行程序代码

死亡态 不占用cpu资源

每次退出子进程的时候，需要指定父进程等待，直到子进程退出，不等待退出系统代为回收

## exec函数簇

简单来说，exec负责是在子进程中执行其他的任务功能，相当于其他的任务进程直接替换子进程资源

|  |
| --- |
| NAME 执行一个文件  execl, execlp, execle, execv, execvp, execvpe - execute a file  SYNOPSIS  #include <unistd.h>  全局变量，外部声明，在系统当中定义用于存放env这命令当中的环境参数  extern char \*\*environ;  函数原型：  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[]);  在exec函数簇中，后缀l、v、p、e指定函数具有某种操作    const char \*path 执行文件的路径，比如说操作ls -l 参数应该是 “/bin/ls”  const char \*file 执行文件的对象，比如说操作ls -l 参数应该是 “ls”  const char \*arg 执行程序所需要的参数列表，以NULL作为结束标志 “ls”,”-l”,NULL  char \*const argv[] 执行程序所需要的参数列表，将参数列表打包成数组，以NULL作为结束标志  char \*const envp[] 用于修改换环境，变量进行存储数据  返回值：  失败： 返回-1 |

练习：使用execlp和execvp进行操作ls -l

# 进程的运行方式

进程运行：

方式1：前台进程 ./main ctrl+c结束操作

方式2：后台进程 ./mian & ctrl+c不会结束操作

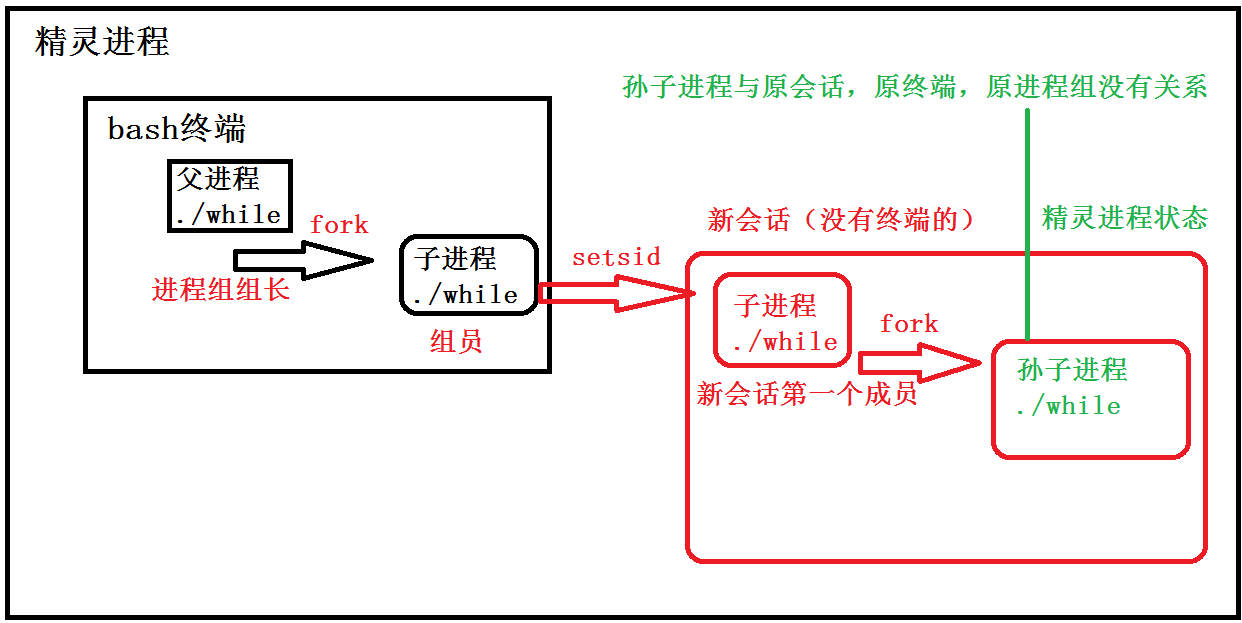
方式3：精灵进程(守护进程) ,一般是从操作系统启动的时候开始创建，专门用于记录当前的操作系统的状态

分析： 精灵进程只能通过系统的信号杀死和系统关机才可以关闭精灵进程，剩下的方式无效

前后台进程在关闭终端的时候，两种进程都会被关闭

精灵进程作用：

系统日志的写入等守护工作的完成，默默实现不影响前台程序的执行



如何创建一个精灵进程？

1. 忽略SIGHUP信号-----SIGHUP是终端关闭的信号，防止因终端关闭导致进程结束 signal(sig, SIG\_IGN);
2. 创建子进程，父进程退出----父进程是bash终端的第一个进程，是组长
3. 子进程创建新的会话-------创建的新会话，子进程是新会话第一个成员 setsid()
4. 创建子进程，父进程退出----当前父进程是前一个的子进程，与原终端有关系，需要结束它
5. 子进程创建新的会话组-----新的子进程称为新的会话组的成员setpgrp()
6. 屏蔽文件的操作权限----为了精灵进程不受文件影响
7. 关闭所有的文件描述符----把进程的所有被打开的文件全部关闭
8. 修改精灵进程的运行路径到根目录---防止目录被误删导致进程结束

|  |
| --- |
| //精灵进程的实现  int daemon\_init(void)  {  pid\_t a;  int maxfd,i;  //1.忽略SIGHUP信号signal(sig, SIG\_IGN);  signal(SIGHUP,SIG\_IGN);    //2.创建子进程，父进程退出  a=fork();  if(a > 0)//父进程  exit(0);    //3.子进程创建新的会话 setsid()  setsid();  //4.创建子进程，父进程退出  a=fork();//孙子进程  if(a > 0)//父进程  exit(0);    //5.子进程创建新的会话组setpgrp()  setpgrp();    //6.屏蔽文件的操作权限  umask(0);  //7.关闭所有的文件描述符  maxfd=sysconf(\_SC\_OPEN\_MAX);//获取最大的描述符  for(i=0;i<maxfd;i++)  {  close(i);//关闭文件描述符  }    //8.修改精灵进程的运行路径到根目录  chdir("/");//选择工作目录  } |

精灵进程控制系统日志发送数据

系统日志文件：

日志目录：/var/log目录下面

查看数据：cat syslog

关于系统日志的操作，需要函数的支持

打开系统日志 openlog

|  |
| --- |
| NAME 发送消息给系统日志  closelog, openlog, syslog- send messages to the system logger  SYNOPSIS  #include <syslog.h>  函数原型：  void openlog(const char \*ident, int option, int facility);  函数参数：  const char \*ident, 写入信息的前缀  int option, 写入进程的pid  LOG\_PID 将进程的pid写入  LOG\_CONS 提供系统日志的吸入  int facility 标志位 打开syslog，守护进程  LOG\_DAEMON 精灵进程标志 |

访问系统日志 syslog

|  |
| --- |
| void syslog(int priority, const char \*format, ...);  函数参数：  int priority, 写入的标志 LOG\_DEBUG  const char \*format 写入的内容格式 |

关闭系统日志 closelog

|  |
| --- |
| void closelog(void); |

# 管道通信

1. 什么是管道通信？其机制如何

管道分为有名管道和无名管道，管道是一种特殊文件，进程通过将数据写入管道，另外一个进程从管道中读取数据的方式进行通信

1. 什么是无名管道？其机制如何?

无名管道只能作用于亲缘关系的进程之间的通信，例如：父子进程，无名管道其实是一个数组，在这个数组中有两个端口

一个读端，一个写端，进程如果想写入数据，就往数组的写端进行写入，如果想要读取就从读端读取

1. 无名管道的使用：

首先申请一个数组，一个由两个int型成员组成的数组：int fd[2] ->这里的数组还不是无名管道，只是个数组

使用函数初始化这个数组：pipe，初始化后，这个数组的两个成员就是读端和写端了

|  |
| --- |
| 头文件  #include <unistd.h>  函数原型  int pipe(int pipefd[2]);  参数：  int pipefd[2] 一个有两个成员的int型数组  返回值：  成功：0  失败：-1  初始化后，数组就变成了无名管道  pipefd[0] -> 读端  pipefd[1] -> 写端 |

思考：初始化以后的无名管道里面的两个成员是否会占用文件描述符

会，初始化无名管道后，会占用两个文件描述符

也就是说无名管道的读端和写端都是文件描述符

也就是说可以系统IO的函数来操作它

无名管道的实际应用：

|  |
| --- |
| int pipefd[2];  pipe(pipefd);  pid\_t x = fork();  if(x > 0)  {  char buf[50] = "helloworld";  write(pipefd[1],buf,strlen(buf));  wait(NULL);  }  if(x == 0)  {  char buf[50];  bzero(buf,sizeof(buf));  read(pipefd[0],buf,sizeof(buf));  printf("read is %s\n",buf);  exit(0);  } |

练习：使用无名管道进行父子进程之间的通信，父进程发送给子进程接收，当发送了quit就退出

1. 有名管道

无名管道是一个数组，而有名管道是一个文件，一个管道文件，在linux下只要知道了这个文件的路径

理论来说，linux下的所有进程都可以看到这个文件，所以有名管道的作用范围就是linux下任意两个进程

有名管道的使用：

首先，创建一个有名管道

方法1：使用命令：mkfifo

方法2：使用函数：mkfifo

|  |
| --- |
| 头文件  #include <sys/types.h>  #include <sys/stat.h>  函数原型  int mkfifo(const char \*pathname, mode\_t mode);  参数  const char \*pathname：要创建的管道文件的路径名  mode\_t mode： 八进制权限值，例如0777  返回值  成功：0  失败：-1  注意：windows下是没有管道这种文件的，所以在共享目录下是创建不了管道文件的，最好将路径名设置为纯linux路径 |

创建好以后这个文件就可以被打开了，然后通过往管道中写入数据和从管道中读取数据来进行通信

实际应用：

|  |
| --- |
| #define FIFO "/home/gec/myfifo/fifo1"  int main(void)  {  //判断管道是否存在  if(access(FIFO,F\_OK))  {  //在指定路径下创建管道文件  int fifofd = mkfifo(FIFO,0777);  if(-1 == fifofd)  {  perror("make fifo error\n");  return -1;  }  }  //打开管道文件  int fd = open(FIFO,O\_RDWR);  if(-1 == fd)  {  perror("open fifo error\n");  return -1;  }  //创建子进程  pid\_t x = fork();  char buf[50];  if(x > 0)  {  while(1)  {  bzero(buf,sizeof(buf));  scanf("%s",buf);  write(fd,buf,strlen(buf));  if(!strncmp(buf,"quit",4))  break;  }  wait(NULL);  }  if(x == 0)  {  while(1)  {  bzero(buf,sizeof(buf));  read(fd,buf,sizeof(buf));  printf("read is %s\n",buf);  if(!strncmp(buf,"quit",4))  exit(0);  }  }    return 0;  } |

练习：实现两个非亲缘关系进程之间的通信

|  |
| --- |
| 读端  char buf[50];  while(1)  {  bzero(buf,sizeof(buf));  read(fd,buf,sizeof(buf));  printf("recv is %s",buf);  if(!strncmp(buf,"quit",4))  break;  } |
| 写端  char buf[50];  while(1)  {  bzero(buf,sizeof(buf));  printf("please input:");  fgets(buf,sizeof(buf),stdin);  write(fd,buf,strlen(buf));  if(!strncmp(buf,"quit",4))  break;  } |

思考：父进程往管道写入了30字节的数据，子进程读取了20字节，剩下10字节会消失吗？

不会，管道是根据读取的字节数来清空的，读多少，清多少

练习：通过管道实现两个程序之间相互通信

client可以给server发送信息，也可以接受来自server的消息，server也一样

管道不限，进程数最多父子进程

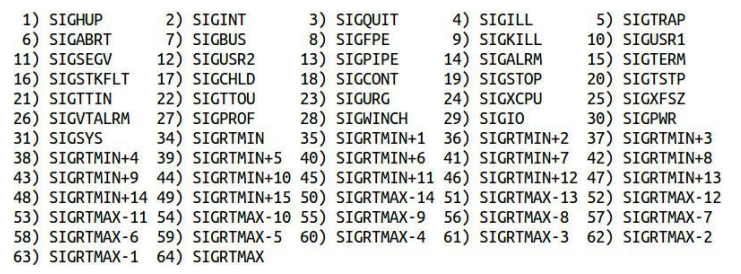
思考：如果想要只用一条管道实现两个进程之间的相互通信，最重要的就是知道数据是由谁发出的

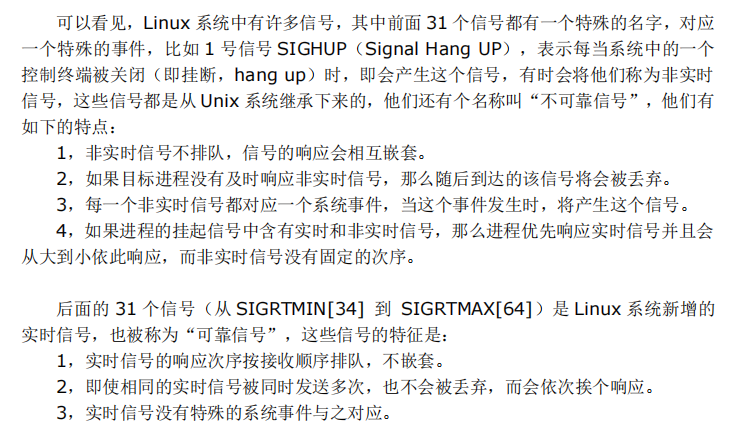
可以使用信号来进行辅助

# 信号

1. 了解信号

在linux下有哪些信号？使用命令kill -l来查看





我们主要围绕非实时信号来进行操作，有哪些比较常用：

2) SIGINT 中断信号，一般在终端输入”ctrl + c”

9) SIGKILL 停止信号，进程彻底停止

10) SIGUSR1 无动作信号

12) SIGUSR2 无动作信号，留个用户自定义的信号

18) SIGCONT 继续信号，可以让暂停态的进程继续运行

19) SIGSTOP 暂停信号，可以让进程进入暂停态

前面的数字是该信号的值，后面的是信号名，信号名其实就是信号值的宏定义

所以使用信号值和信号名是一样的效果

2、使用信号

在linux下，这些信号都是由谁发送？系统

用户如果发送信号？

方法1：使用kill指令

kill + 信号值/信号名 + 目标进程PID号

例如：

ps -ef 查看进程PID

kill -SIGKILL 4397 向进程号为4397的进程发送停止信号

kill -19 4398 向进程号为4398的进程发送暂停信号

除了kill还有一个指令也可以发送信号：killall

killall +信号值/名 + 进程名称

缺点就是会将所有同名进程都发送同样的信号

killall -9 server 向所有名为server的进程发送9号信号

方法2：使用函数

3、信号相关函数接口

1）发送信号的函数：kill

|  |
| --- |
| 头文件  #include <sys/types.h>  #include <signal.h>  函数原型  int kill(pid\_t pid, int sig);  参数  pid\_t pid：目标进程的PID号  int sig： 要发送的信号值/名  返回值  成功：0  失败：1 |

发送非实时信号，一般收到信号后会去执行信号的动作

但不是所有信号都有动作，例如SIGUSR1和SIGUSR2就是由用户自定义动作的

如果没有给信号自定义一套动作，收到信号后，进程不知道该怎么做，就会报提示并退出进程

2）捕捉信号函数：signal

|  |
| --- |
| 头文件  #include <signal.h>  函数原型  typedef void (\*sighandler\_t)(int);  //给void (\*)(int)这种类型的指针取别名为sighandler\_t  sighandler\_t signal(int signum, sighandler\_t handler);  参数  int signum： 要捕捉的信号值/名  sighandler\_t handler： 信号处理函数的地址  返回值  成功：信号处理函数的地址  失败：SIG\_ERR |

signal函数可以声明收到指定信号后，做什么动作(信号处理函数)

信号处理函数：一个void型返回值，可以接收一个int型参数的函数

当进程收到指定信号后，就会去执行信号处理函数

signal函数不会阻塞等待信号到来，只是类似声明的作用,所以一般放在代码前面

signal设置自定义动作后，接收到信号后将会去执行自定义动作

信号处理函数的形参接收的是信号值

signal不会捕捉未声明的信号，例如signal里面填的要捕捉信号为sig1就不会捕捉sig2

注意：SIGKILL和SIGSTOP这两个信号是无法被捕捉和忽略的

练习：子进程给父进程发送一个信号，父进程收到后也会给子进程发送一个信号

1. 信号等待函数：pause

|  |
| --- |
| NAME  pause - wait for signal  SYNOPSIS  #include <unistd.h>  int pause(void); |

函数功能：挂起进程，直到收到一个信号为止

例如：

signal(SIGUSR1,func);

pause();//在信号到来之前会一直挂起进程等待

1. 给自己信号

|  |
| --- |
| NAME  raise - send a signal to the caller  SYNOPSIS  #include <signal.h>  int raise(int sig);  参数：要给自己发送的信号值/名 |

练习：结合信号的知识，使用一条管道实现两个陌生进程之间的相互通信

答案：07系统编程/day5/ans1.c和ans2.c

4、关于signal的第二参数

signal捕捉到信号后，一般有三种情形

第一种：自定义动作，收到信号后，去信号处理函数中做对应的事情

第二种：忽略行为，SIG\_IGN，收到信号后，会忽略它，收到当没收到一样，也不会响应

第三种：默认动作，SIG\_DFL，收到信号后，去做信号本来的动作

实际应用：

|  |
| --- |
| void func(int sig)  {  printf("recv signal is %d\n",sig);  }  int main(void)  {  //signal(SIGINT,func);//自定义动作  //signal(SIGINT,SIG\_IGN);//忽略行为  signal(SIGUSR1,SIG\_DFL);//默认动作  pause();  return 0;  } |

自定义动作：改变了信号原本的动作，使进程收到信号后可以去做我们想做的事情

忽略行为：会让信号始终无法到达进程，可以通过这种方式来屏蔽一些信号

在进程运行期间不会收到某些信号的打扰，但是SIGSTOP和SIGKILL无法被屏蔽

默认动作：一般信号如果没有声明自定义动作就是去执行默认动作，但如果声明过自定义动作后

想要取消掉自定义动作，可以使用SIG\_DFL来重置它的处理方式

1. 信号集

什么是信号集？

信号的集合，每一个成员都是一个信号，通过将信号添加到一个信号集合中

再设置阻塞状态给信号集，那么整个信号集的信号都会变成阻塞状态

信号阻塞和信号忽略的区别

信号响应：收到信号后，做一个动作->自定义/默认动作

信号忽略：收到信号后，直接将信号丢弃

信号阻塞：进程如果阻塞了某个信号，那么收到信号后，不会马上响应，

而是要等到信号解除阻塞后才响应，在阻塞时，这个信号虽然没有被响应

但也没有被丢弃，而是放在一个挂起队列上

信号集处理函数

1. 定义信号集

sigset\_t set;

1. 处理函数

|  |
| --- |
| 头文件：  #include <signal.h>  处理函数：  //清空信号集并初始化  int sigemptyset(sigset\_t \*set);  //将所有信号添加到信号集  int sigfillset(sigset\_t \*set);  //将单个信号添加到集合中  int sigaddset(sigset\_t \*set, int signum);  //将单个信号从集合中删除  int sigdelset(sigset\_t \*set, int signum);  //判断某个信号是否还在集合中  int sigismember(const sigset\_t \*set, int signum);  参数：  sigset\_t \*set：信号集的地址  int signum：要添加/删除的信号值  返回值：  前4个函数：  成功：0  失败：-1  sigismember的返回值：  在集合中：1  不在集合：0  失败：-1 |

练习：写一个程序，将所有信号添加到信号集，再将某个信号从集合中剔除

判断那些信号不在集合中，不在就将这个信号值打印出来

|  |
| --- |
| //定义信号集  sigset\_t set;  //清空信号集  sigemptyset(&set);  //将所有信号添加到集合中  sigfillset(&set);  //将几个信号从集合中剔除  sigdelset(&set,12);  sigdelset(&set,10);  sigdelset(&set,55);  sigdelset(&set,43);  //判断哪个信号不在集合  int i;  for(i=1;i<65;i++)  {  if(i == 32 || i == 33)  {  continue;  }  //如果不在集合返回0,逻辑取反就为真  if(!sigismember(&set,i))  printf("signal %d is delete\n",i);  } |

1. 信号集设置阻塞函数

|  |
| --- |
| 函数原型：  int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);  参数：  int how： 要给集合数值怎样的属性  SIG\_BLOCK 阻塞属性  SIG\_UNBLOCK 非阻塞属性  const sigset\_t \*set： 要设置属性的集合的地址  sigset\_t \*oldset： 保留之前状态的集合,不关心就填NULL  返回值：  成功：0  失败：-1 |

练习：父进程将SIGUSR1设置为阻塞，持续10s，在这10s内由子进程给父进程发送该信号

如果父进程马上响应：信号响应

如果父进程10s后响应：信号阻塞

如果父进程一直不响应：信号忽略

|  |
| --- |
| pid\_t x = fork();  if(x > 0)  {  signal(SIGUSR1,func); //声明好信号处理函数  sigset\_t set; //定义信号开集  sigemptyset(&set); //清空信号集  sigaddset(&set,SIGUSR1);//将信号添加到集合  sigprocmask(SIG\_BLOCK,&set,NULL);//设置集合阻塞  for(int i=0;i<10;i++)  {  printf("i=%d\n",i);  sleep(1);  }  sigprocmask(SIG\_UNBLOCK,&set,NULL);  printf("SIGUSR1 is unblock\n");  wait(NULL);  }  if(x == 0)  {  sleep(3);  kill(getppid(),SIGUSR1);  printf("child send SIGUSR1 to parent\n");  exit(0);  } |

1. 关于阻塞的思考

思考1：验证在进程的信号挂起队列上，没有相同的信号，即相同信号要丢弃

|  |
| --- |
| 父进程是持续10s的阻塞  if(x == 0)  {  sleep(1);  kill(getppid(),SIGUSR1);  printf("child send SIGUSR1 to parent\n");  sleep(1);  kill(getppid(),SIGUSR1);  printf("child send SIGUSR1 to parent\n");  sleep(1);  kill(getppid(),SIGUSR1);  printf("child send SIGUSR1 to parent\n");  exit(0);  } |

结果：在父进程对SIGUSR1阻塞期间子进程给父进程连续发送了3次信号

但解除阻塞后，只响应了一次，所以说挂起队列上没有相同信号

在挂起队列上每一个信号只有一个属于它的位置，当里面已经挂起了一个

再来一个信号就没有位置来挂，只能丢弃

思考2：验证阻塞属性是否会被子进程继承，设父进程对某一个信号阻塞

在这种状态下产生的子进程是否也对该信号阻塞

会继承

|  |
| --- |
| int main(void)  {  signal(SIGUSR1,func); //声明好信号处理函数  sigset\_t set; //定义信号开集  sigemptyset(&set); //清空信号集  sigaddset(&set,SIGUSR1);//将信号添加到集合  sigprocmask(SIG\_BLOCK,&set,NULL);//设置集合阻塞  pid\_t x = fork();  if(x > 0)  {  printf("parent pid =%d\n",getpid());  for(int i=0;i<20;i++)  {  printf("i=%d\n",i);  sleep(1);  }  sigprocmask(SIG\_UNBLOCK,&set,NULL);  printf("parent is unblock\n");  wait(NULL);  }  if(x == 0)  {  printf("child pid =%d\n",getpid());  for(int i=0;i<30;i++)  {  printf("j=%d\n",i);  sleep(1);  }  sigprocmask(SIG\_UNBLOCK,&set,NULL);  printf("child is unblock\n");  exit(0);  }    return 0;  } |

# 五、IPC对象

1、什么是IPC对象？

在linux下，IPC对象指的就是消息队列、共享内存、信号量，如果使用户要使用IPC对象进行通信

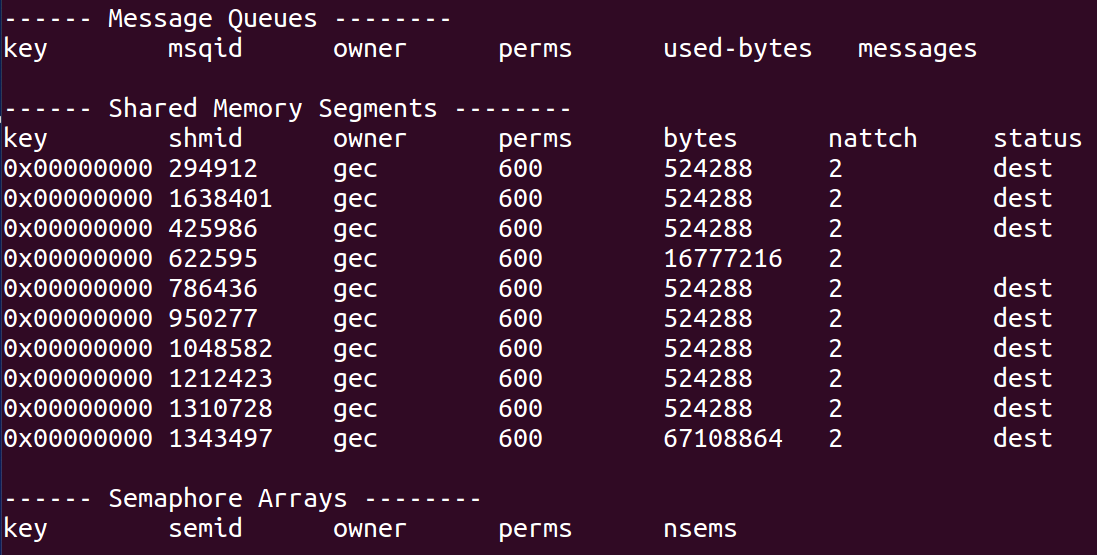
就要为IPC对象申请对应的资源：key值和ID号，IPC对象的key值都是唯一的，ID可变

key值类似文件的路径，ID类似文件描述符

每打开一个IPC对象就会获取一个对象的ID号，通过ID号来操作IPC对象

1. 如何查看linux系统下的所有IPC对象

查看IPC对象的指令：ipcs -a



Message Queues：消息队列

Shared Memory Segments：共享内存

Semaphore Arrays：信号量

ipcs -a：查看所有IPC对象

ipcs -q：只查看消息队列

ipcs -m：只查看共享内存

ipcs -s： 只查看信号量

1. 删除IPC对象的指令：

ipcrm -IPC对象(大小写)字母 对象ID/key值

例如：

ipcrm -q 10086 //删除ID号为10086的消息队列

ipcrm -Q 0x10010 //删除key值为0x10010的消息队列

总结：大写字母就是通过key值删除，小写字母就是通过ID删除

对应字母可以参照上面查看IPC对象的字母

1. 申请key值

所有IPC对象申请key值的方法都是统一的，但申请ID号的方式不一样

|  |
| --- |
| 头文件：  #include <sys/types.h>  #include <sys/ipc.h>  函数原型：  key\_t ftok(const char \*pathname, int proj\_id);  参数：  const char \*pathname：合法的路径名  int proj\_id： 一个整数  只要传入的路径名和整数一模一样，那么哪怕是在两个不同进程之间，也能得到相同的key值，然后就可以通过相同的key值得到一个相同的IPC对象的ID号  返回值：  成功：key值  失败：-1 |

练习：在两个进程之间都申请一次key值，要求得到相同的key值

|  |
| --- |
| pid\_t x = fork();  if(x > 0)  {  key\_t key = ftok(".",10);  printf("key0=%#x\n",key);  wait(NULL);  }  if(x == 0)  {  key\_t key1 = ftok(".",10);  key\_t key2 = ftok(".",20);  printf("key1=%#x\n",key1);  printf("key2=%#x\n",key2);  exit(0);  } |

只要申请key值时的路径名和数组一样，就可以确保在不同进程中也能找到同一IPC对象

# 六、消息队列

1、消息队列的机制

进程往消息队列中写入数据：”类型”+”正文”

进程从消息队列中读取数据，只需要提供”类型”就可以获取指定的数据

只要key值和id号一样，消息队列的作用范围就是linux下任意两个进程

1. 管道通信和消息队列的区别

管道通信：只要管道中有数据，就必须读取出来，不然会阻塞后面的数据

要读取先输入的数据，才能读取后输入的数据，先进先出，操作使用IO函数

消息队列：可以读取指定类型的数据，操作使用独有的函数

1. 消息队列的通信方式：
2. 申请key值
3. 根据key值来申请消息队列的ID号

|  |
| --- |
| 头文件  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  函数原型  int msgget(key\_t key, int msgflg);  参数：  key\_t key：key值  int msgflg：  IPC\_CREAT：不存在就创建  如果同时包含了IPC\_CREAT和IPC\_EXCL,并且对象的确存在，将会报错  IPC\_CREAT|0666创建时要给权限  返回值：  成功：消息队列ID号  失败：-1 |

练习：在两个不同进程之间获取同一个消息队列的ID号

|  |
| --- |
| pid\_t x = fork();  if(x > 0)  {  key\_t key = ftok(".",10);  int msgid = msgget(key,IPC\_CREAT|0666);  printf("key=%#x,msgid=%d\n",key,msgid);  wait(NULL);  }  if(x == 0)  {  key\_t key = ftok(".",10);  int msgid = msgget(key,IPC\_CREAT|0666);  printf("key=%#x,msgid=%d\n",key,msgid);  exit(0);  }  结果：  gec@ubuntu:/mnt/hgfs/WH2208/07 系统编程/day5$ ./msgget  key=0xa300071,msgid=32768  key=0xa300071,msgid=32768  gec@ubuntu:/mnt/hgfs/WH2208/07 系统编程/day5$ ipcs -q  ------ Message Queues --------  key msqid owner perms used-bytes messages  0x0a300071 32768 gec 666 0 0 |

可以看到，没有就会创建对象，有了就直接获取它的ID号

1. 往消息队列中写入数据

|  |
| --- |
| 函数原型：  int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);  参数：  int msqid： 消息队列的ID号  const void \*msgp：存放要发送数据的地址，一个结构体的地址  size\_t msgsz： 正文的长度  int msgflg： 属性，一般填0  发送结构体相关：  The msgp argument is a pointer to a caller-defined structure of the following general form:  这个参数必须是一个用户自定义结构体，且必须包含以下内容：  struct msgbuf {  long mtype; /\* 数据类型，就是一个大于0的数 \*/  char mtext[长度]; /\* 正文，真正要发送的数据，长度自己觉得 \*/  };  返回值：  成功：0  失败：-1 |

1. 从消息队列中获取数据

|  |
| --- |
| 函数原型：  ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg);  参数：  int msqid： 消息队列ID号  void \*msgp： 存放获取到的数据的结构体的地址  size\_t msgsz： 存放正文的数组的长度  long msgtyp： 想要获取的数据的类型  int msgflg： 属性，一般填0  返回值：  成功：实际获取到的字节数  失败：-1 |

实际应用：

|  |
| --- |
| struct msgbuf{  long mtype;  char mtext[50];  };  int main(void)  {  key\_t key = ftok(".",10);  int msgid = msgget(key,IPC\_CREAT|0666);  struct msgbuf buf;  pid\_t x = fork();  if(x > 0)  {  buf.mtype = 1;  while(1)  {  bzero(&buf,sizeof(buf));  scanf("%s",,buf.mtext);  msgsnd(msgid,&buf,sizeof(buf.mtext),0);  if(!strncmp(buf.mtext,"quit",4))  break;  }  wait(NULL);  }  if(x == 0)  {  long type = 1;  while(1)  {  bzero(&buf,sizeof(buf));  msgrcv(msgid,&buf,sizeof(buf.mtext),type,0);  printf("recv is %s\n",buf.mtext);  if(!strncmp(buf.mtext,"quit",4))  break;  }  exit(0);  }    return 0;  } |

练习：使用消息队列实现两个程序之间的相互通信

1. 销毁消息队列

|  |
| --- |
| 函数原型  int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);  参数  int msqid：消息队列ID号  int cmd：  IPC\_STAT 从内核复制一封关于消息队列的信息到buf中  IPC\_RMID 通过ID号删除消息队列  struct msqid\_ds \*buf：选择删除的话，这里填NULL就行 |

如果要删除消息队列：

|  |
| --- |
| int msgid = msgget(ftok(".",10),IPC\_CREAT|0666);//创建消息队列  printf("msgid=%d\n",msgid);  msgctl(msgid,IPC\_RMID,NULL);//销毁消息队列 |

# 七、共享内存

共享内存也属于IPC对象，也要申请key值

1. 共享内存的机制：

任意两个进程，通过申请key值，ID号，共享内存就可以得到一片内存空间

两个进程就可以将数据写入共享内存，并读取里面的数据

因此，共享内存是最高效的进程间通信方式

1. 相关函数
2. 申请key值：ftok
3. 获取共享内存ID(创建共享内存)

|  |
| --- |
| 头文件：  #include <sys/ipc.h>  #include <sys/shm.h>  函数原型：  int shmget(key\_t key, size\_t size, int shmflg);  参数：  key\_t key：key值  size\_t size：共享内存的大小，最好是1024的倍数，最少也要是偶数  int shmflg：IPC\_CREAT|0666  返回值：  成功：共享内存ID  失败：-1 |

1. 销毁共享内存：

|  |
| --- |
| int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);  参数：  int shmid：共享内存ID号  int cmd：IPC\_RMID，删除共享内存  struct shmid\_ds \*buf：选择删除就填NULL |

1. 根据ID申请共享内存使用区域(映射共享内存)

|  |
| --- |
| 函数原型：  void \*shmat(int shmid, const void \*shmaddr, int shmflg);  参数：  int shmid：共享内存ID号  const void \*shmaddr：起始地址，填NULL让系统分配  int shmflg：属性，填0默认属性  返回值：  成功：共享内存中的地址  失败：(void \*)-1 |

1. 撤销共享内存映射

|  |
| --- |
| 函数原型：  int shmdt(const void \*shmaddr);  参数：要撤销的共享内存映射的地址  返回值：  成功：0  失败：-1 |

1. 对共享内存的读写操作函数

标准输入输出函数，就和以前使用堆内存类似

练习：在两个进程之间申请共享内存映射地址，看是否是同一块内存地址

地址不同，但一个进程写入的数据可以在另一个进程中打印出来

映射的共享内存地址会适应每个进程所在的虚拟内存，所以打印出来的地址不同

但其实都是指向的同一块共享内存地址

思考：往共享内存中写入数据，是清空式写入，还是覆盖式写入

|  |
| --- |
| scanf("%s",str);  printf("str=%s\n",str);  scanf("%s",str);  for(int i=0;i<10;i++)  {  printf("%c--",str[i]);  } |
| 结果：  helloworld 先输入10字节  str=helloworld  apple 再输入5字节  a--p--p--l--e----o--r--l--d-- |

是覆盖式写入，只不过因为’\0’的存在导致旧数据无法打印

练习：使用共享内存实现两个不同进程之间的互相通信

|  |
| --- |
| char \*str = NULL;  void print\_data(int sig)  {  printf("recv is %s\n",str);  }  int main(void)  {  signal(SIGUSR1,print\_data);  printf("myid is %d\n",getpid());  key\_t key = ftok(".",10);  int shmid = shmget(key,1024,IPC\_CREAT|0666);  str = (char \*)shmat(shmid,NULL,0);  int pid;  scanf("%d",&pid);  while(1)  {  scanf("%s",str);  kill(pid,SIGUSR1);  }  shmdt(str);  shmctl(shmid,IPC\_RMID,NULL);  return 0;  } |

实际上使用信号处理函数来分离读取与写入是不太合适的

能被多个进程同时访问的资源被称为共享资源，也叫临界资源

访问临界资源其实是需要申请资源，访问完毕后需要释放申请的资源

这种申请与释放的操作就属于信号量的P/V操作。

所以最适合用来控制共享内存的方式其实是信号量。

# 信号量

1. 信号量属于IPC对象，和之前讲过的信号不是同一个东西

它不用于进程之间的通信，而是作用于进程之间的通信

1. 信号量的相关函数
2. 申请key值：ftok
3. 获取信号量ID(创建信号量)

|  |
| --- |
| 头文件：  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  函数原型：  int semget(key\_t key, int nsems, int semflg);  参数：  key\_t key：key值  int nsems：信号量元素个数，例如1个成员就填1  int semflg：IPC\_CREAT|0666  返回值：  成功：信号量ID  失败：-1 |

1. 初始化信号量成员/删除信号量

|  |
| --- |
| 函数原型：  int semctl(int semid, int semnum, int cmd, ...);  参数：  int semid： 信号量ID号  int semnum：信号量成员的下标,从0开始数  int cmd： 操作方式  IPC\_RMID 根据ID号删除信号量  SETVAL 设置信号量成员的初始值  ...：如果选择做初始化，这里就填初始值，否则就不用填  返回值：  成功：0  失败：-1  初始化：semctl(semid,0,SETVAL,0);//初始化信号量第一个成员的值为0  删除：semctl(semid,0,IPC\_RMID); |

1. 对信号量进行P/V操作

P操作：申请资源，-1操作

V操作：释放资源，+1操作

|  |
| --- |
| 函数原型：  int semop(int semid, struct sembuf \*sops, size\_t nsops);  参数：  int semid： 信号量ID  struct sembuf \*sops：决定P/V操作的结构体的地址  该结构体包含以下成员：  unsigned short sem\_num; /\* 要操作的信号量成员的下标 \*/  short sem\_op; /\* P/V操作：大于0->V操作，小于0->P操作，等于0->无操作\*/  short sem\_flg; /\* 属性，填0为默认属性 \*/  size\_t nsops：操作信号量的个数  返回值：  成功：0  失败：-1 |

实际应用：

使用信号量来解决共享内存的数据践踏问题

信号量定义一个成员，初始化值为0

输入端在输入后对信号量成员做V操作，信号量值=1

输出端在输出前对信号量成员做P操作，信号量值=0

printf不会阻塞，马上进入下一次循环，又要对信号量做P操作

信号量值=0时，做P操作会阻塞，就要等到信号量值>0时才能做P操作

输入端输入数据后，做一次V操作，信号量值+1，大于0，输出端又能继续往下走

输入端：

|  |
| --- |
| int main(void)  {  //申请key值，创建共享内存和信号量  key\_t key = ftok(".",10);  int shmid = shmget(key,1024,IPC\_CREAT|0666);  int semid = semget(key,1,IPC\_CREAT|0666);  //映射共享内存地址，初始化信号量成员的初始值为0  char \*str = (char \*)shmat(shmid,NULL,0);  semctl(semid,0,SETVAL,0);  //定义一个对信号量做V操作的结构体变量  struct sembuf data;  data.sem\_num = 0;//要操作的信号量成员的下标  data.sem\_op = 1;//大于0做V操作，+1  data.sem\_flg = 0;//属性，填0默认属性    //对共享内存做输入，每输入一次就做一次V操作  while(1)  {  printf("please input:");  scanf("%s",str);//输入时会阻塞  semop(semid,&data,1);//输入后进行一次V操作  if(!strncmp(str,"quit",4))  break;  }  //撤销共享内存的映射  shmdt(str);  return 0;  } |

输出端：

|  |
| --- |
| int main(void)  {  //申请key值，创建共享内存和信号量  key\_t key = ftok(".",10);  int shmid = shmget(key,1024,IPC\_CREAT|0666);  int semid = semget(key,1,IPC\_CREAT|0666);  //映射共享内存地址，初始化信号量成员的初始值为0  char \*str = (char \*)shmat(shmid,NULL,0);  semctl(semid,0,SETVAL,0);  //定义一个对信号量做P操作的结构体变量  struct sembuf data;  data.sem\_num = 0;//要操作的信号量成员的下标  data.sem\_op = -1;//小于于0做P操作，-1  data.sem\_flg = 0;//属性，填0默认属性    //对共享内存做输出，每输出一次就做一次P操作  while(1)  {  semop(semid,&data,1);//输出之前做一次P操作,如果信号量值为0，就会阻塞  printf("recv is %s\n",str);  if(!strncmp(str,"quit",4))  break;  }  //撤销共享内存的映射、删除共享内存、删除信号量  shmdt(str);  shmctl(shmid,IPC\_RMID,NULL);  semctl(semid,0,IPC\_RMID);  return 0;  } |

进程间通信方式：

无名管道：

一个数组，初始化以后里面的两个成员就是两个文件描述符，分别代表读端和写端

通过往写端写入数据，从读端读取数据进行通信，只作用于亲缘关系进程

有名管道：

一个管道文件，通过打开文件往里面写入数据和读取数据进行通信

先写入的数据先被读取，当管道里面没有数据时读取会阻塞，作用范围是linux下任意两个进程

信号：

通过进程PID号来向进程发送信号，进程捕捉到信号后作出对应的动作：默认、自定义、忽略、阻塞

信号一般用来控制进程

消息队列：

一种IPC对象，以”类型”+”正文”的形式将数据写入队列中，进程可以从消息队列中读取指定”类型”的数据

和有名管道一样，里面的数据被读取后会消失，每被读取的数据会留在消息队列中  
共享内存：

一种IPC对象，最高效的通信方式，通过申请一片内存使任意进程可以使用这片内存来进行数据交互

因为的共享资源/临界资源，访问时最好使用信号量来控制访问，否则容易出现数据践踏问题

信号量：

一种IPC对象，可以申请多个成员，对成员初始化赋值后，可以对成员进行P操作让它的值-1

也可以对成员进行V操作让它的值+1，当信号量成员的值为0时，继续P操作就会阻塞

练习：使用任意方式实现两个不同进程之间的相互通信，并且可以将通信记录保存到一个文件中

# 九、线程

1、进程和线程的区别

进程：linux下最小的资源单位

线程：linux下最小的调度单位

假设linux是一个社会，进程就人，线程就是人身上的一条手

2、线程相关事项：

所有线程函数都是封装在一个线程库里面，无法查看其源码，头文件：#include <pthread.h>

在所有线程函数的man手册说明中都有这么一段话：Compile and link with -pthread.

编译使用了线程函数的代码时，需要连接线程库，-lpthread

线程函数大多数都是以pthread\_开头

1. 线程的基本函数接口

1）创建一条线程

|  |
| --- |
| 函数原型：  int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg);  参数：  pthread\_t \*thread： 存放线程号的变量的地址，变量类型：pthread\_t  const pthread\_attr\_t \*attr：属性变量，填NULL普通属性  void \*(\*start\_routine) (void \*)：线程执行函数，要让线程区做的事，一个返回值类型为void \*型，可以接收void \*型参数的一个函数的地址  void \*arg：传递给线程执行函数的参数  返回值：  成功：0  失败：错误值，非0值 |

子线程的运行时间和主线程的运行时间对比

主线程运行时间长，子线程运行完后不会影响主线程的运行时间

主线程运行时间短，主线程运行完后，子线程也会结束，因为子线程的资源由主线程分配，

主线程结束就代表着进程结束，进程结束要回收资源，因此子线程资源被回收旧无法继续运行

思考：线程执行函数和调用函数有什么区别？

线程：子线程和主线程同步运行，互不干扰

调用：要等到函数调用完毕以后，主线程才能接着往下运行

2）接合线程

|  |
| --- |
| 函数原型：  int pthread\_join(pthread\_t thread, void \*\*retval);  参数：  pthread\_t thread：存放线程号的变量  void \*\*retval：存放子线程退出值的指针，不关心就填NULL  返回值：  成功：0  失败：错误值，非0值 |

接合函数会让主线程阻塞等待线程的退出，类似进程间的waitpid

3）线程的退出

|  |
| --- |
| 函数原型：  void pthread\_exit(void \*retval);  参数：  void \*retval：退出变量的地址，这个退出值不能是函数里的地址，最好的全局变量 |

实际应用：

|  |
| --- |
| int exit\_data = 10;  void \*func(void \*arg)  {  char \*str = (char \*)arg;  printf("str=%s\n",str);  pthread\_exit((void \*)&exit\_data);  }  int main(void)  {  char \*str = "helloworld";  pthread\_t tid;  pthread\_create(&tid,NULL,func,(void \*)str);  void \*p = NULL;  pthread\_join(tid,&p);  printf("exit data = %d\n",\*(int \*)p);  return 0;  } |

练习：创建两条线程，一条负责发送，一条负责接收，实现两条线程之间单方向发送消息

|  |
| --- |
| int flag = 0; //标志位  void \*send\_data(void \*arg)  {  char \*str = (char \*)arg;  while(1)  {  scanf("%s",str);  flag++;//输入了数据就让标志位为真  if(!strncmp(str,"quit",4))  break;  }  pthread\_exit(NULL);  }  void \*recv\_data(void \*arg)  {  char \*str = (char \*)arg;  while(1)  {  while(!flag);//如果标志位为0就循环阻塞  printf("recv is %s\n",str);  if(!strncmp(str,"quit",4))  break;  flag = 0;//打印以后，重新让标志位为假，等待下一次输入  }  pthread\_exit(NULL);  }  int main(void)  {  char \*str = (char \*)malloc(1024);  pthread\_t tid1,tid2;  pthread\_create(&tid1,NULL,send\_data,(void \*)str);  pthread\_create(&tid2,NULL,recv\_data,(void \*)str);    pthread\_join(tid1,NULL);  pthread\_join(tid2,NULL);  free(str);  return 0;  } |

思考：同时又多条线程在运行，每条线程都做好了线程捕捉声明，当向进程发送信号时

是怎样的响应效果

4、线程的分离属性

分离属性：有了分离属性的线程，不需要其他线程的接合，会自己回收资源

哪怕设置了分离属性，当主线程结束时，线程还是要一起结束

设置分离属性有两种方式：

第一种：通过pthread\_create创建线程时添加一个具有分离属性的变量

第二种：在线程创建后，在线程内部设置分离属性

第一种方式：

1）定义一个属性变量

pthread\_attr\_t attr;

1. 初始化属性变量

int pthread\_attr\_init(pthread\_attr\_t \*attr);

参数：为初始化的属性变量的地址

1. 设置分离属性给变量

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);

参数：

pthread\_attr\_t \*attr：属性变量的地址

int detachstate：要设置的属性

PTHREAD\_CREATE\_DETACHED 分离属性

PTHREAD\_CREATE\_JOINABLE 不分离属性，可以接合

4）将具有分离属性的变量添加到线程创建过程中

pthread\_create(&tid,&attr,func,NULL);

5）销毁属性变量

int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

参数：属性变量的地址

练习1：使用第一种方式设置一个具有分离属性的线程，验证当具有分离属性线程退出时，主线程无法接合它

|  |
| --- |
| void \*func(void \*arg)  {  printf("i'm thread\n");  sleep(2);  printf("thread is exit\n");//这句话没有打印，因为join不会阻塞等待分离线程  pthread\_exit(NULL);  }  int main(void)  {  pthread\_t tid; //线程号  pthread\_attr\_t attr; //属性变量  pthread\_attr\_init(&attr); //初始化属性变量  pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_DETACHED);//设置属性变量为分离属性  pthread\_create(&tid,&attr,func,NULL);//创建时使用具有分离属性的变量  sleep(1);//让线程先运行起来  int rvl = pthread\_join(tid,NULL);  if(rvl != 0)  printf("return number = %d\n",rvl);  return 0;  } |

结果：线程只打印了第一句话就结束了，因为join没有阻塞等待它结束就往下运行了

主线程退出，分离属性线程也要退出

第二种方法：

将线程设置为分离属性：

int pthread\_detach(pthread\_t thread);

参数：线程的线程号

这种方法可以在创建线程后将线程设置为分离属性，也可以在线程函数内中途把线程设置为分离属性

在线程内获取线程本身的线程号

pthread\_t pthread\_self(void);

返回值：当前线程的线程号

结合起来就是：

pthread\_detach(pthread\_self());

练习2：在线程内部将自己设置为分离属性，再去验证主线程无法接合

|  |
| --- |
| void \*func2(void \*arg)  {  printf("i'm thread\n");  sleep(1);  pthread\_detach(pthread\_self());//将本线程设置为分离属性  printf("thread is detached\n");  sleep(2);  printf("thread is exit\n");//这句话没有打印，因为join不会阻塞等待分离线程  pthread\_exit(NULL);  }  int main(void)  {  pthread\_t tid; //线程号  pthread\_create(&tid,NULL,func2,NULL);//创建时使用具有分离属性的变量  sleep(2);  int rvl = pthread\_join(tid,NULL);  if(rvl != 0)  printf("return number = %d\n",rvl);  return 0;  } |

结果：如果在join之前没有延时，接合会返回0表示成功

如果加了延时，保证线程内设置分离属性在接合之前，就会返回非0表示接合失败

就好比离家出走，先走了，家人再来接你就接不到，但如果家人先接到你了，再想离家出走就走不了了

5、线程的优先级

静态优先级：

取值范围：0 ~ 99

优先级：数字越大，优先级越高

动态优先级：

取值范围：-20 ~ 19

优先级：数字越小，优先级越高

关于静态优先级：

静态优先级为0的线程，被称之为非实时线程，其他的就属于实时线程

线程的三种调度策略：

FIFO：先进先出，针对实时线程

轮询：每一条线程会给一个固定的CPU访问时间片

非实时线程调度：静态优先级为0时，就看动态优先级

相关函数：

1）决定是否继承主线程调度策略

|  |
| --- |
| 函数原型：  int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr, int inheritsched);  参数：  pthread\_attr\_t \*attr：属性变量的地址  int inheritsched：是否继承主线程调度策略  PTHREAD\_INHERIT\_SCHED 继承之前的调度策略  PTHREAD\_EXPLICIT\_SCHED 不继承之前的调度策略 |

1. 设置调度策略

|  |
| --- |
| 函数原型：  int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy);  参数：  pthread\_attr\_t \*attr：属性变量的地址  int policy：策略选择  SCHED\_FIFO 先进先出，静态优先级1 ~ 99  SCHED\_RR 轮询时间片，静态优先级1 ~ 99  SCHED\_OTHER 非实时调度，动态优先级 |

1. 静态优先级设置

|  |
| --- |
| 函数原型：  int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, const struct sched\_param \*param);  参数：  pthread\_attr\_t \*attr：属性变量的地址  const struct sched\_param \*param：设置优先级的结构体变量的地址  该结构体类型如下：  struct sched\_param {  int sched\_priority; /\* 静态优先级数值 \*/  }; |

注意：重新配置了优先级后，要运行时，记得使用管理员身份运行

实际应用：

|  |
| --- |
| void \*func(void \*arg)  {  char \*str = (char \*)arg;  while(1)  {  printf("%s",str);  }  pthread\_exit(NULL);  }  int main(void)  {  //定义属性变量  pthread\_attr\_t attr1,attr2;  //初始化属性变量  pthread\_attr\_init(&attr1);  pthread\_attr\_init(&attr2);  //不继承之前的调度策略  pthread\_attr\_setinheritsched(&attr1,PTHREAD\_EXPLICIT\_SCHED);  pthread\_attr\_setinheritsched(&attr2,PTHREAD\_EXPLICIT\_SCHED);  //选择新的调度策略->轮询  pthread\_attr\_setschedpolicy(&attr1,SCHED\_RR);  pthread\_attr\_setschedpolicy(&attr2,SCHED\_RR);  //配置静态优先级结构体  struct sched\_param param1 = {90};  struct sched\_param param2 = {80};  //设置静态优先级  pthread\_attr\_setschedparam(&attr1,&param1);  pthread\_attr\_setschedparam(&attr2,&param2);  //创建两条线程  pthread\_t tid1,tid2;  pthread\_create(&tid1,&attr1,func,(void \*)"A");  pthread\_create(&tid2,&attr2,func,(void \*)"B");  //接合两条线程  pthread\_join(tid1,NULL);  pthread\_join(tid2,NULL);  return 0;  } |

结果：

优先级高的线程一直抢占着资源，优先级低的线程抢不到资源

当优先级一样时：轮着来，一个线程挂起后另一个线程就接着使用资源，依次轮询

1. 动态优先级设置

在静态优先级为0时，在线程内部设置动态优先级

当静态优先级为0时，线程为非实时线程，只能使用非实时线程调度：SCHED\_OTHER

|  |
| --- |
| 头文件：  #include <unistd.h>  函数原型：  int nice(int incr);  参数：  动态优先级值 |

实际应用：

|  |
| --- |
| void \*func(void \*arg)  {  char \*str = (char \*)arg;  if(str[0] == 'A')  {  nice(10);//优先级较低  while(1)  {  printf("A");  }  }  if(str[0] == 'B')  {  nice(-20);//优先级最高  while(1)  {  printf("B");  }  }  pthread\_exit(NULL);  }  //在main函数里选择非实时调度，并将静态优先级设置为0 |

结果：

优先级高的一直抢占资源，优先级低的抢不到资源

另一种动态优先级设置方式：

|  |
| --- |
| void \*func1(void \*arg)  {  char \*str = (char \*)arg;  nice(-20);//优先级最高  while(1)  {  printf("%s\n",str);  sleep(1);  }  pthread\_exit(NULL);  }  void \*func2(void \*arg)  {  char \*str = (char \*)arg;  nice(19);//优先级最低  while(1)  {  printf("%s\n",str);  sleep(1);  }  pthread\_exit(NULL);  }  void \*func3(void \*arg)  {  char \*str = (char \*)arg;  nice(5);//优先级偏中间  while(1)  {  printf("%s\n",str);  sleep(1);  }  pthread\_exit(NULL);  }  int main(void)  {  pthread\_t tid1,tid2,tid3;  pthread\_create(&tid1,NULL,func1,(void \*)"thread 1");  pthread\_create(&tid2,NULL,func2,(void \*)"thread 2");  pthread\_create(&tid3,NULL,func3,(void \*)"thread 3");  pthread\_join(tid1,NULL);  pthread\_join(tid2,NULL);  pthread\_join(tid3,NULL);  return 0;  } |

结果：

一开始按照创建顺序运行，运行一段时间后，逐渐变成：132和线程中设置的优先级顺序一致

6、线程的取消状态与类型

一般主线程是不会去处理任务，而是控制子线程的状态：接合、取消

相关函数接口：

1. 取消线程的函数

|  |
| --- |
| 函数原型：  int pthread\_cancel(pthread\_t thread);  参数：  pthread\_t thread：要取消的线程的线程号 |

1. 线程响应取消状态

如果线程在执行某项比较重要的事情，不能被临时取消，

可以通过设置线程的取消状态来决定是否响应取消请求

相关函数：

|  |
| --- |
| 函数原型：  int pthread\_setcancelstate(int state, int \*oldstate);  参数：  int state：状态  PTHREAD\_CANCEL\_ENABLE 使能，能响应取消请求  PTHREAD\_CANCEL\_DISABLE 不使能，不响应取消请求  int \*oldstate：保留之前状态，不关心就填NULL  注意：  If a cancellation request is received, it is blocked until cancelability is enabled.  如果在不使能期间，收到取消请求，这个请求不会被丢弃，而是会等待线程可以响应为止 |

练习：验证线程如果不响应取消请求，取消请求是会等待而不是丢弃

1. 线程响应取消类型

取消的状态决定了是否响应取消请求

取消的类型决定了何时响应取消请求

相关函数：

|  |
| --- |
| 函数原型：  int pthread\_setcanceltype(int type, int \*oldtype);  参数：  int type：类型  PTHREAD\_CANCEL\_DEFERRED  ----延时响应，直到遇到一个取消点函数才会取消延时响应，是线程的默认类型  PTHREAD\_CANCEL\_ASYNCHRONOUS  ----立即响应  int \*oldtype：保留之前的类型，不关心就填NULL  取消点函数：  在第7本man手册中查看：pthreads，找到Cancellation points章节，可以看到一些常用的函数：  printf、fputs、puts、sleep、scanf.......这些函数都是取消点函数  只要线程调用了以上函数，就会从延时响应取消变请求为马上响应取消请求 |

练习：验证线程的默认属性是延时响应，而不是马上响应

|  |
| --- |
| void \*func(void \*arg)  {  //pthread\_setcanceltype(PTHREAD\_CANCEL\_ASYNCHRONOUS,NULL);//将类型设置为马上响应  //如果线程设置为马上响应,不用等到取消点函数就会马上响应取消请求而线程结束  //不涉及取消点函数的代码  int i,j;  for(i=0;i<100000;i++)  {  for(j=0;j<50000;j++)  {  //用两层无意义循环来代替延时函数,可以延时8秒左右  }  }  //调用取消点函数  printf("thread\n");  //如果上面的打印了下面的没有，就说明在打印上面这句话时，线程就马上响应取消请求了  printf("exit\n");  } |

结果：

主线程在1秒后发送的取消请求要等到8秒后才响应，证明了线程的默认属性为延时响应

如果在线程函数一开始就设置类型为马上响应，不用等到取消点函数就会马上响应取消请求而线程结束

思考：取消点函数是一次性的函数永久性的？

一次性，只有遇到取消点函数才会马上响应取消请求

7、线程的取消例程函数

什么是线程取消例程函数？

当线程收到取消请求时，先不要马上响应取消请求，而是去执行一个例程函数，执行完这个函数再响应取消请求

为什么要设置取消例程函数？

为了防止线程带着一些公共资源退出线程，导致其他线程无法继续使用公共资源，例如：互斥锁

如何实现？

1）在线程函数内部声明例程函数

|  |
| --- |
| 函数原型：  void pthread\_cleanup\_push(void (\*routine)(void \*), void \*arg);  参数：  void (\*routine)(void \*)：取消例程函数的地址，一个void型返回值，可以接收一个void \*型参数的函数  void \*arg：传递给线程取消例程函数的参数 |

回顾已经学过的例程函数

信号响应函数 void func(int sig)

线程执行函数 void \* func(void \*arg)

线程取消例程函数 void func(void \*arg)

1. 移除线程取消例程函数声明

只要声明了取消例程函数，在线程中就必须有一个移除这个声明的函数，不管线程是否会受到取消请求

收到请求：执行例程函数，线程取消，移除就不执行

没受到请求：不执行例程函数，就要移除这个声明

|  |
| --- |
| 函数原型：  void pthread\_cleanup\_pop(int execute);  参数：  int execute： 值  0 执行时，直接移除例程函数的声明  非0 执行时，会先执行一遍取消例程函数，再移除声明 |

注意：上面两个函数必须成对使用

总结：

设置了分离属性的线程会自己回收资源，不用主线程来接合

设置分离属性有两种方式，一种是在创建线程时使用带分离属性的变量加入创建过程，一种是在线程内部进行

线程的优先级分为静态优先级和动态优先级，静态优先级取值范围在0~99，动态：-20 ~ 19

静态优先级数字越大，优先级越高，动态优先级数字越小，优先级越高

静态优先级为0的线程被称为非实时线程，非实时线程主要看动态优先级

线程的三种调度策略为：队列FIFO、轮询时间片、非实时调度

主线程可以通过发送取消请求来取消一个正在工作的线程，关于取消的响应，线程有两种状态和两种类型

两种状态：响应取消和不响应取消，线程默认的是响应

两种类型：延时响应和马上响应，线程默认的是延时响应

线程默认为延时响应取消请求，只不过因为取消点函数的存在，导致线程调用了这类函数就会马上响应取消

线程可以通过声明取消例程函数来确保线程被临时取消时会先跳转到取消例程函数中将一些公关资源释放后再退出

线程取消例程函数必须和移除声明函数一起使用

# 十、线程的同步互斥

什么是同步互斥？为什么要处理同步互斥？

同步互斥就是使线程在处理任务时有先后顺序，为了防止线程出现资源抢占的问题

处理同步互斥有哪些方法？

信号量 进程

有名信号量 进程

无名信号量 线程

互斥锁 线程

读写锁 线程

注意：有名信号量和信号量不是只能作用于进程之间，而是最好作用于进程之间，因为用在线程上完全没有必要

作用于线程的几种方法就只能作用于线程，因为他们都只存在于代码的上下文之间

1. 有名信号量

有名信号量和信号量非常像，但信号量的值只能是0或1，而有名信号量的值可以达到0到正无穷

相关函数接口：

1. 创建并打开一个有名信号量

|  |
| --- |
| 头文件：  #include <fcntl.h>  #include <sys/stat.h>  #include <semaphore.h>  函数原型：  sem\_t \*sem\_open(const char \*name, int oflag);  sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value);  参数：  const char \*name：有名信号量的名字，要求必须以’/’开头  int oflag：O\_CREAT，不存在就创建，存在就打开  mode\_t mode：八进制权限值，例如0666  unsigned int value：信号量初始值  返回值：  成功：有名信号量的地址  失败：SEM\_FAILED |

1. 有名信号量的P操作

|  |
| --- |
| 函数原型：  int sem\_wait(sem\_t \*sem);  参数：有名信号量的地址  P操作：资源数-1  该函数会对有名信号量进行一次P操作，如果信号量的值减到为0继续减就会阻塞 |

1. 有名信号量的V操作

|  |
| --- |
| 函数原型：  int sem\_post(sem\_t \*sem);  参数：有名信号量的地址  V操作：资源数+1，一定可以加1，不会阻塞 |

1. 关闭和删除有名信号量

|  |
| --- |
| 关闭有名信号量  int sem\_close(sem\_t \*sem);  删除有名信号量  int sem\_unlink(const char \*name); |

注意：使用有名信号量编译时也要链接线程库

1. 无名信号量

一般作用域线程之间的同步互斥，由于是无名信号量，所以没有名字，也就不难使用sem\_open来断开

无名信号量就是一个变量，一个sem\_t类型的变量

1. 定义一个无名信号量

sem\_t sem;

1. 初始化无名信号量

|  |
| --- |
| 函数原型  int sem\_init(sem\_t \*sem, int pshared, unsigned int value);  参数：  sem\_t \*sem：无名信号量的地址  int pshared：  0 作用于同一进程下的线程之间  非0 作用于进程之间  unsigned int value：无名信号量的初始值 |

1. P/V操作：和有名信号量同一个函数

注意：有名信号量是一个指针，无名信量是一个变量

1. 销毁无名信号量

int sem\_destroy(sem\_t \*sem);

参数：无名信号量的地址

练习：在进程中创建5条线程，每一条线程都执行同一个任务

任务：将helloworld这个字符串每隔1s打印一次

不加同步互斥：

hhhhheeeeellllllllllooooowwwwwooooorrrrrlllllddddd 耗时10s

加了同步互斥：

helloworldhelloworldhelloworldhelloworldhelloworld 耗时50s

|  |
| --- |
| sem\_t sem;  void \*func(void \*arg)  {  sem\_wait(&sem);//P操作  char \*str = "helloworld";  for(int i=0;i<10;i++)  {  fprintf(stderr,"%c",str[i]);  sleep(1);  }  sem\_post(&sem);//V操作  pthread\_exit(NULL);  }  int main(void)  {  sem\_init(&sem,0,1);//初始化信号量  pthread\_t tid[5];  int i;  for(i=0;i<5;i++)  pthread\_create(&tid[i],NULL,func,NULL);  for(i=0;i<5;i++)  pthread\_join(tid[i],NULL);  sem\_destroy(&sem);//销毁信号量  return 0;  } |

加了同步互斥后，每次都只有一条线程工作

1. 互斥锁

什么是互斥锁？特点如何？

互斥锁就是专门用于处理线程互斥的一种方式，他有两种状态：上锁状态和解锁状态

特点：当互斥锁处于上锁状态时，其他线程继续上锁就会被阻塞，直到上锁线程解开锁

其他线程才能解除阻塞状态，成功上锁。而在解锁状态下继续解锁是不会造成阻塞

互斥锁的使用场景

当我们访问一些临界资源，防止多个线程同时访问，我们可以使用互斥锁，

在线程访问临界资源之前先上锁再访问，访问完以后再解锁

注意：

互斥锁是不会阻塞没想要上锁的线程，只有当线程函数里调用了上锁函数，才会去看锁的状态

如果锁被上锁了，就阻塞等待，没有上锁就把它锁住

相关函数接口：

1. 定义一个互斥锁变量

pthread\_mutex\_t mutex;

1. 初始化互斥锁变量

互斥锁的初始化有两种方式：动态初始化和静态初始化

|  |
| --- |
| 动态初始化：使用函数  int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr);  参数：  pthread\_mutex\_t \*mutex：互斥锁变量的地址  const pthread\_mutexattr\_t \*mutexattr：属性，普通属性填NULL  静态初始化：使用宏定义  pthread\_mutex\_t fastmutex = PTHREAD\_MUTEX\_INITIALIZER;  一般在函数内定义的变量是使用动态初始化  在函数外定义的变量使用静态初始化 |

1. 上锁/解锁

|  |
| --- |
| 上锁的函数：  int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);  解锁的函数：  int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);  参数：互斥锁变量的地址 |

1. 销毁互斥锁

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

|  |
| --- |
| char buf[256];  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;//静态初始化互斥锁  void \*send\_data(void \*arg)  {  while(1)  {  pthread\_mutex\_lock(&mutex);//上锁  scanf("%s",buf);  pthread\_mutex\_unlock(&mutex);//解锁  if(!strncmp(buf,"quit",4))  break;  usleep(100000);//用时间差避免和另一个线程抢着上锁  }  }  void \*recv\_data(void \*arg)  {  while(1)  {  pthread\_mutex\_lock(&mutex);  printf("recv is %s\n",buf);  pthread\_mutex\_unlock(&mutex);  if(!strncmp(buf,"quit",4))  break;  usleep(200000);  }  }  int main(void)  {  pthread\_t tid1,tid2;  pthread\_create(&tid1,NULL,recv\_data,NULL);  pthread\_create(&tid2,NULL,send\_data,NULL);  pthread\_join(tid1,NULL);  pthread\_join(tid2,NULL);  pthread\_mutex\_destroy(&mutex);  return 0;  } |

思考：有一把互斥锁，线程1在往共享资源写入之前上锁了，线程2想要在读取之前上锁

因为线程1先上锁，线程2阻塞了，这时如果线程3在线程1解锁之前先解锁了，会造成什么后果？

结果：

同步互斥的机制混乱了，在线程3强制解锁后，线程2成功上锁，读取完数据后又马上解锁，接着上锁、解锁

这时如果线程1写入完毕要解锁了，线程2可能正在上锁期间，被解锁，线程抢回锁，但又会别线程2抢回去

除非线程1可以卡在线程2解锁后上锁前这段时间内解锁，才可能回到正常情况

1. 读写锁

读写锁和互斥锁的区别

互斥锁：无论是读取共享资源还是修改共享资源都需要进行上锁，在上锁期间其他线程不能上锁

读写锁：访问资源(读书)同时上读锁，修改资源(写作业)不能同时上写锁，否则就会阻塞

既有读锁又有写锁的就被称为读写锁

相关函数：

1. 定义读写锁变量

pthread\_rwlock\_t rwlock;

1. 初始化读写锁

和互斥锁一样也有静态初始化和动态初始化

|  |
| --- |
| 动态初始化：  int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock, const pthread\_rwlockattr\_t \*restrict attr);  参数：  pthread\_rwlock\_t \*restrict rwlock：读写锁变量的地址  const pthread\_rwlockattr\_t \*restrict attr：属性，填NULL默认属性  静态初始化：  pthread\_rwlock\_t rwlock = PTHREAD\_RWLOCK\_INITIALIZER; |

1. 上锁(读锁/写锁)

|  |
| --- |
| 上读锁：  int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);  上写锁：  int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);  参数：读写锁变量的地址 |

1. 解锁

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);

参数：读写锁变量的地址

1. 销毁读写锁

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

参数：读写锁变量的地址

练习：设有共享资源int a 创建四条线程

两条线程负责打印a的值 ->读锁

thread1/3s thread2/5s ->共经过5s

两条线程负责修改a的值 ->写锁

thread3/4s thread4/6s ->共经过10s

结果：

上了读锁：可以继续上读锁，但不能上写锁，上写锁的线程会阻塞，直到所有读锁被解开才能上写锁

上了写锁：不可以再上读锁或写锁，必须等到这个写锁被解开后才能接着上锁

1. 条件变量

什么是条件变量？

线程因某一个条件/情况不成立下，进入一个变量中等待，这个存放线程的变量就是条件变量

在进入条件变量时，会自动给线程解锁，离开条件变量时，会自动给线程上锁

因此条件变量总是放在上锁到解锁区间之内

相关函数接口：

1. 定义一个条件变量

pthread\_cond\_t cond

1. 初始化条件变量

|  |
| --- |
| 动态初始化：  int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr);  参数：  pthread\_cond\_t \*cond：条件变量的地址  pthread\_condattr\_t \*cond\_attr：属性，填NULL默认属性  静态初始化：  pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER; |

1. 将线程添加到条件变量

|  |
| --- |
| 函数原型：  int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);  参数：  pthread\_cond\_t \*cond：条件变量的地址  pthread\_mutex\_t \*mutex：互斥锁的地址 |

1. 唤醒条件变量中的线程

|  |
| --- |
| 广播：全部唤醒  int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);  单播：随机唤醒一个  int pthread\_cond\_signal(pthread\_cond\_t \*cond);  参数：条件变量的地址 |

1. 销毁条件变量

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

参数：条件变量的地址

练习：有5个小孩，每个小孩的任务就是从银行卡中去1000块钱出来，设银行卡一开始有2000块钱

|  |
| --- |
| int money = 2000;  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;  void \*func(void \*arg)  {  int i = \*(int \*)arg+1;  pthread\_mutex\_lock(&mutex);//上锁  while(money < 1000)//如果余额不足1000  {  printf("money too less thread%d is sleep\n",i);  pthread\_cond\_wait(&cond,&mutex);//添加进条件变量中睡觉  }  money -= 1000;//取钱  printf("thread%d get 1000￥\n",i);  pthread\_mutex\_unlock(&mutex);//解锁  pthread\_exit(NULL);//走人  }  int main(void)  {  pthread\_t tid[5];  int i;  for(i=0;i<5;i++)  {  pthread\_create(&tid[i],NULL,func,(void \*)&i);  usleep(10000);  }  //第一次打钱  sleep(5);  pthread\_mutex\_lock(&mutex);  money += 2000;  printf("first add 2000 money\n");  pthread\_mutex\_unlock(&mutex);  pthread\_cond\_signal(&cond);//随机唤醒一个  //第二次打钱  sleep(5);  pthread\_mutex\_lock(&mutex);  money += 2000;  printf("secand add 2000 money\n");  pthread\_mutex\_unlock(&mutex);  pthread\_cond\_broadcast(&cond);//全部唤醒  //接合所有线程  for(i=0;i<5;i++)  pthread\_join(tid[i],NULL);  //销毁互斥锁和条件变量  pthread\_mutex\_destroy(&mutex);  pthread\_cond\_destroy(&cond);  return 0;  } |

运行结果：

thread1 get 1000￥

thread2 get 1000￥ //线程1和线程2把钱全部取走了

money too less thread3 is sleep

money too less thread4 is sleep

money too less thread5 is sleep//其他三条线程只能进入条件变量中睡觉

first add 2000 money //主线程第一次打钱，+2000，然后随机唤醒一条线程

thread3 get 1000￥ //线程3被唤醒，取走1000块钱

secand add 2000 money //第二次打钱，+2000，然后将条件变量中睡眠的线程全部唤醒

thread4 get 1000￥

thread5 get 1000￥ //剩下两条线程醒来后取钱走人

总结：

有名信号量：

通过一个名字来创建的，得到的是一个指针，可以给它初始化一个值，

通过P操作来-1，V操作+1，当减到为0继续减就会阻塞

无名信号量：

定义的一个变量，可以给一个初始值，通过P操作来-1，V操作来+1，当减到为0继续减就会阻塞

互斥锁：

同时只能有一条线程上锁，其他线程在有线程上锁后继续上锁就会阻塞，直到有线程解锁

读写锁：

上了读锁：可以继续上读锁，但不能上写锁，上写锁的线程会阻塞，直到所有读锁被解开才能上写锁

上了写锁：不可以再上读锁或写锁，必须等到这个写锁被解开后才能接着上锁

条件变量：

将空闲线程添加到条件变量中，使其进入睡眠，直到被唤醒，在进入条件变量时会对线程自动解锁

唤醒后再上锁

# 线程池

1. 什么是线程池

线程池就是由多条线程组合起来的集合，就像一家公司，由多名员工组成，当有任务时

员工就会接下任务去处理，没有任务，员工就去睡觉

1. 如何描述一个线程池

线程池其实就是一个结构体，里面包含了线程池所需的基本要素

|  |
| --- |
| //最大的任务数  #define MAX\_WAITING\_TASKS 1000  //最大的线程数据  #define MAX\_ACTIVE\_THREADS 20  //任务链表的节点模型  struct task{  void \*(\*do\_task)(void \*arg);//任务函数  void \*arg;//任务函数的参数  struct task \*next;  };  //声明线程池结构体  typedef struct thread\_pool{  pthread\_mutex\_t lock; //互斥锁  pthread\_cond\_t cond; //条件变量  struct task \*task\_list; //任务链表的头节点  pthread\_t \*tid; //存放tid号的数组  unsigned waiting\_task; //当前任务个数  unsigned active\_threads;//活跃线程数  bool shutdown; //线程池的关闭标志位，true：关闭，false：每关闭  }thread\_pool; |

1. 线程池函数接口：
2. 初始化线程池

函数功能：

初始化一个线程池

参数：

thread\_pool \*pool：线程池的地址

unsigned threads\_number：初始活跃线程数

返回值：

成功：true

失败：false

|  |
| --- |
| bool init\_pool(thread\_pool \*pool,unsigned threads\_number)  {  //初始化互斥锁  pthread\_mutex\_init(&pool->lock,NULL);  //初始化条件变量  pthread\_cond\_init(&pool->cond,NULL);  //初始化关闭标志位为false  pool->shutdown = false;  //给任务队列的头节点申请堆空间  pool->task\_list = (struct task \*)malloc(sizeof(struct task));  //给存放tid号的指针申请堆空间  pool->tids = malloc(sizeof(pthread\_t)\*MAX\_ACTIVE\_THREADS);  //判断两次申请堆内存是否成功  if(pool->task\_list == NULL || pool->tids == NULL)  {  perror("request memory error\n");  return false;  }  //初始化任务队列头节点的指针域  pool->task\_list->next = NULL;  //设置当前的任务个数  pool->waiting\_tasks = 0;  //设置当前的活跃线程数  pool->active\_threads = threads\_number;  //根据活跃线程数据来创建线程  int i,ret;  for(i=0;i<pool->active\_threads;i++)  {  ret = pthread\_create(&pool->tids[i],NULL,routine,(void \*)pool);  if(ret != 0)  {  perror("create thread error\n");  return false;  }  }  //创建线程完毕，返回  return true;  } |

1. 线程池基本例程函数

函数功能：

线程池中的每一条线程的基本任务函数，其功能是去任务队列中获取任务

从任务队列中获取到任务后，去执行该任务，如果没有任务可做就去睡觉

参数：

void \*arg：传入线程池的地址

|  |
| --- |
| //线程取消例程函数,防止线程在上锁期间被取消  void handler(void \*arg)  {  pthread\_mutex\_unlock((pthread\_mutex\_t \*)arg);  }  void \*routine(void \*arg)  {  //获取线程池地址  thread\_pool \*pool = (thread\_pool \*)arg;  struct task \*p = NULL;  //循环从任务队列中获取任务  while(1)  {  pthread\_cleanup\_push(handler,(void \*)&pool->lock);  //上锁  pthread\_mutex\_lock(&pool->lock);  //判断是否有任务可做,并且当前线程池未被关闭  while(pool->waiting\_tasks == 0 && !pool->shutdown)  {  //没有任务就去条件变量中睡觉  pthread\_cond\_wait(&pool->cond,&pool->lock);  }  //判断上面是因为有任务可做还是因为线程池关闭才退出循环  if(pool->shutdown && pool->waiting\_tasks == 0)  {  //线程池关闭了，且任务做完了，就可以解锁走人了  pthread\_mutex\_unlock(&pool->lock);  pthread\_exit(NULL);  }  //不管线程池是否关闭，有任务就要去做  //将任务节点从链表中脱离出来  p = pool->task\_list->next;  pool->task\_list->next = p->next;//保持链表完整  pool->waiting\_tasks--;//等待任务数-1  //解锁  pthread\_mutex\_unlock(&pool->lock);  //删除线程取消例程函数的声明  pthread\_cleanup\_pop(0);  //设置线程不响应取消  pthread\_setcancelstate(PTHREAD\_CANCEL\_DISABLE,NULL);  //执行节点中的任务函数  (p->do\_task)(p->arg);  //做完任务，释放节点内存  free(p);  //设置线程响应取消  pthread\_setcancelstate(PTHREAD\_CANCEL\_ENABLE,NULL);  }  pthread\_exit(NULL);  } |

1. 添加任务的函数

函数功能：

往线程池中投递任务

参数：

thread\_pool \*pool：线程池的地址

void \*(\*do\_task)(void \*arg)：要投递的任务函数

void \*arg：任务函数的参数

返回值：

成功：true

失败：false

|  |
| --- |
| bool add\_task(thread\_pool \*pool,void \*(\*do\_task)(void \*arg),void \*arg)  {  //申请新的任务节点来存放任务函数  struct task \*new\_task = (struct task \*)malloc(sizeof(struct task));  if(new\_task == NULL)  {  perror("request new task node error\n");  return false;  }  //初始化任务节点的数据域和指针域  new\_task->do\_task = do\_task; //函数指针执行任务函数  new\_task->arg = arg; //存放任务函数的参数  new\_task->next = NULL; //后继指针指向NULL  //任务队列属于公共资源，访问时最好上锁  pthread\_mutex\_lock(&pool->lock);  //看任务队列是否满了  if(pool->waiting\_tasks >= MAX\_WAITING\_TASKS)  {  //解锁  pthread\_mutex\_unlock(&pool->lock);  //释放申请的节点内存  free(new\_task);  perror("too many thread is wating\n");  return false;  }  //将任务节点插入到队列中  struct task \*p = pool->task\_list;  for(p; p->next != NULL; p = p->next);  p->next = new\_task;  //当前的任务数+1  pool->waiting\_tasks++;  //添加完毕，解锁  pthread\_mutex\_unlock(&pool->lock);  //随机唤醒一条线程来工作  pthread\_cond\_signal(&pool->cond);  //返回  return true;  } |

1. 往线程池中添加线程

函数功能：

往线程池中添加活跃线程

参数：

thread\_pool \*pool：线程池

unsigned additinoal\_threads：要添加的线程数

返回值：

>0：实际新增的线程数

=0：每添加，也可能是添加数量设置为0

-1：失败

|  |
| --- |
| int add\_thread(thread\_pool \*pool,unsigned additinoal\_threads)  {  //如果要添加的线程数为0，就直接返回  if(additinoal\_threads == 0)  return 0;  //计算添加新线程后的线程总数  unsigned total\_threads = pool->active\_threads + additinoal\_threads;  //添加线程  int i,ret;  int actual\_increment = 0;//用来记录实际添加的线程数  for(i=pool->active\_threads;i<total\_threads&&i<MAX\_ACTIVE\_THREADS;i++)  {  ret = pthread\_create(&pool->tids[i],NULL,routine,(void \*)pool);  if(ret != 0)  {  perror("add thread error\n");  //如果一条线程都创建不了  if(actual\_increment == 0)  return -1;  break;  }  actual\_increment++;  }  //当前线程数增加了  pool->active\_threads += actual\_increment;  //返回实际添加的线程数  return actual\_increment;  } |

1. 删除线程池中的线程

函数功能：

将线程池中的线程删除

参数：

thread\_pool \*pool：线程池

unsigned removeing\_threads：要删除的线程数

>0：要删除的线程数据

=0：直接返回当前的线程数

返回值：

>0：当前线程池中剩余的线程数

-1：失败

|  |
| --- |
| int remove\_thread(thread\_pool \*pool,unsigned removeing\_threads)  {  //如果删除数设置为0，就直接返回当期线程池剩余线程  if(removeing\_threads == 0)  return pool->active\_threads;  //计算当前线程池剩余的线程是否足够删除  int remain\_threads = pool->active\_threads - removeing\_threads;  //线程池最少要保留一条线程  remain\_threads = remain\_threads > 0 ? remain\_threads : 1;  //取消线程池中的线程  int i,ret;  for(i=pool->active\_threads-1;i>remain\_threads-1;i--)  {  ret = pthread\_cancel(pool->tids[i]);  if(ret != 0)  break;  }  //重新统计当前线程池剩余线程数  pool->active\_threads = i+1;  //判断是否有取消失败的线程  if(i != remain\_threads-1)  {  return -1;  }  else  {  //返回实际剩余的线程数  return pool->active\_threads;  }  } |

1. 销毁线程池

函数功能：

销毁线程池

参数：

thread\_pool \*pool：要销毁的线程池

返回值：

成功：true

失败：false

|  |
| --- |
| bool destroy\_pool(thread\_pool \*pool)  {  //将线程池的关闭标准设置为true  pool->shutdown = true;  //唤醒所有线程  pthread\_cond\_broadcast(&pool->cond);  //接合所有线程  int i;  for(i=0;i<pool->active\_threads;i++)  {  errno = pthread\_join(pool->tids[i],NULL);  if(errno != 0)  printf("join tid%d error:%s\n",,i,strerror(errno));  else  printf("[%u] is joined\n",(unsigned )pool->tids[i]);  }  //释放申请过的堆内存  free(pool->task\_list);  free(pool->tids);  free(pool);  return true;  } |