1、Linux开发环境搭建

虚拟机或云服务器搭建Linux系统，使用NAT模式

虚拟机安装ssh：

```
sudo apt install openssh-server
```

安装ssh服务端的时候可能会出问题，因为ubuntu中默认安装了openssh-client，可以尝试执行以下命令：

```
sudo apt-get remove openssh-client
sudo apt-get install openssh-client openssh-server
```

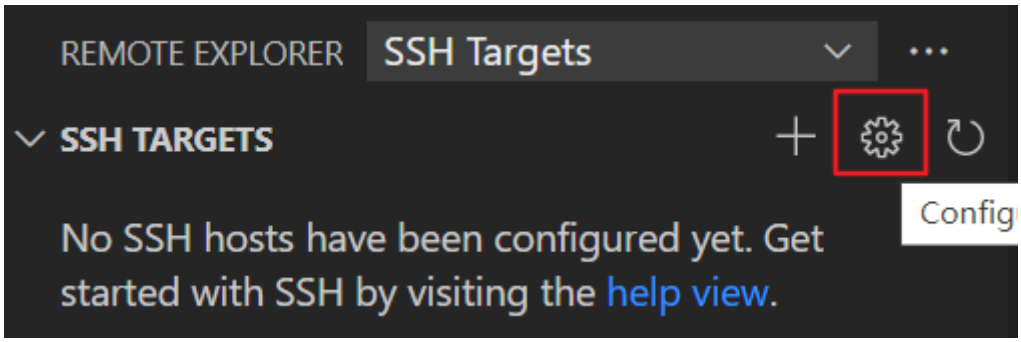
开启ssh服务：

```
service sshd start
```

Visual Studio Code开发配置

本机使用 Visual Studio Code 进行开发，安装插件 Remote Development，或者Remote SSH，使得可以远程连接到服务器。

安装了 Remote Development 后，切换，将 REMOTE EXPLORER 选项改为 SSH Targets，然后设置并配置 config 文件，路径选第一个就行。

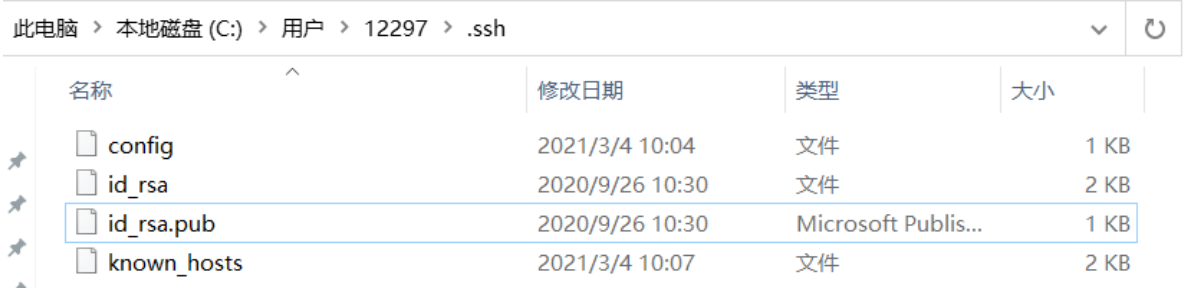


配置文件中Host自己取个名字，HostName为IP地址，User是远程登录名，IdentityFile是id_rsa文件的位置。

避免每次打开文件夹都输入密码：

本机命令行输入：ssh-keygen -t rsa

会在用户目录下生成一个 `.ssh` 文件夹，将公钥文件 `id_rsa.pub` 发送给虚拟机



在虚拟机中也生成公钥和私钥：ssh-keygen -t rsa

也会在默认目录下生成 `.ssh` 文件夹

```
lhx@lhx-virtual-machine:~$ cd .ssh
lhx@lhx-virtual-machine:~/.ssh$ ls
id_rsa  id_rsa.pub
lhx@lhx-virtual-machine:~/.ssh$
```

创建一个文件名叫：authorized_keys，将本机公钥文件内容复制到该文件中。

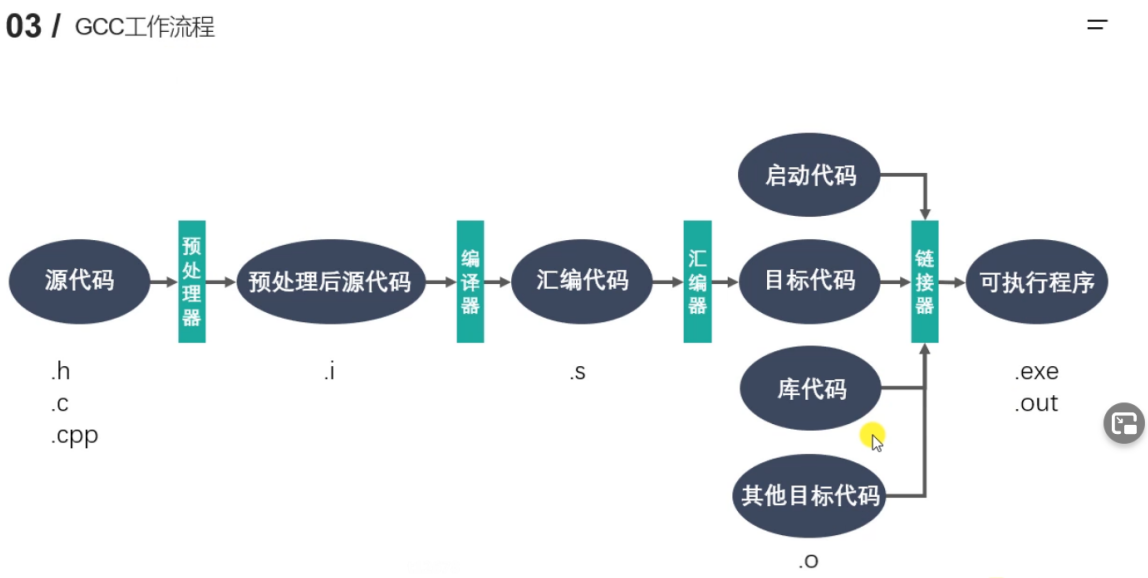
```
chmod 600 authorized_keys
chmod 700 ~/.ssh
service sshd restart
```

然后再进行远程连接

2、GCC、GDB、Makefile、GDB

2.1 gcc

gcc的工作流程




```
gcc test.c -E -o test.i #预处理
gcc test.i -S -o test.s #编译
gcc test.s -c -o test.o #汇编
gcc test.o -o test.out #链接
```

gcc的一些其它选项

```
-include file #相当于#include <file>
-g #在编译时生成调试信息，GNU 调试器可利用该信息
-On #n取值范围0-3，表示优化级别，越大优化级别越高，默认为1
-w #不生成任何警告信息
-c #只编译并生成目标文件，只激活预处理,编译,和汇编,也就是他只把程序做成obj文件
-E #只激活预处理
-S #只激活预处理和编译，就是指把文件编译成为汇编代码
-D #编译时给文件指定宏
-Wall #生成所有警告
-l #程序编译时指定使用的库
-L #编译时的搜索库的路径
```

```
-fpic #生成与位置无关的代码
-shared #生成共享目标文件
-I #指定目录搜索文件，即include目录
```

2.2 静态库的制作和使用

库文件有两种：静态库和动态库。

静态库在程序的链接阶段被复制到程序中；动态库在程序运行时由系统动态加载到内存中供程序调用。

命名规则：

- Linux：libxxx.a
  - lib：前缀（固定）
  - xxx：库的名字
  - .a：后缀（固定）
- Windows：libxxx.lib

制作：

1. 用 gcc 获得 `.o` 文件
2. 将 `.o` 文件使用 ar（archive）工具打包：

```
ar rcs libxxx.a xxx.o xxx.o
#r: 将文件插入备存文件中，即将.o文件插入到库文件中
#c: 建立备存文件，即库文件名
#s: 索引，为.o文件创建索引
```

例子

先生成 `.o` 文件

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$ ls
add.c  div.c  head.h  mult.c  sub.c
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$ gcc -c add.c div.c mult.c sub.c
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$ ls
add.c  add.o  div.c  div.o  head.h  mult.c  mult.o  sub.c  sub.o
```

生成静态库文件

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$ ar rcs libcalc.a add.o div.o mult.o sub.o
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$ ls
add.c  add.o  div.c  div.o  head.h  libcalc.a  mult.c  mult.o  sub.c  sub.o
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$
```

将库文件拷贝到项目目录下

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$ cp libcalc.a ~/桌面/Linux-Server/lesson/library/lib/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/calc$ cd ../library/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ tree
.
├── include
│   └── head.h
├── lib
│   └── libcalc.a
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c

3 directories, 7 files
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$
```

编译 `main.c` 文件，`-I` 指定了 `.h` 文件的目录，`-l` 指定库名，`-L` 指定静态库的路径

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ gcc main.c -o app -I ./include/ -l calc -L ./lib/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$
```

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ ls
app  include  lib  main.c  src
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ ./app
a+b=24
a-b=-4
a*b=140
a/b=0.714286
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$
```

完整流程：

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ tree
.
├── include
│   └── head.h
├── lib
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c

3 directories, 6 files
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ cd src
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library/src$ gcc -c add.c div.c mult.c sub.c -I ../include/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library/src$ ar rcs libcalc.a add.o div.o mult.o sub.o
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library/src$ ls
add.c  add.o  div.c  div.o  libcalc.a  mult.c  mult.o  sub.c  sub.o
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library/src$ cp libcalc.a ../lib
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library/src$ cd ..
```

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ gcc main.c -o app -I ./include/ -l calc -L ./lib/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ tree
.
├── app
├── include
│   └── head.h
├── lib
│   └── libcalc.a
├── main.c
└── src
    ├── add.c
    ├── add.o
    ├── div.c
    ├── div.o
    ├── libcalc.a
    ├── mult.c
    ├── mult.o
    ├── sub.c
    └── sub.o

3 directories, 13 files
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$ ./app
a+b=24
a-b=-4
a*b=140
a/b=0.714286
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson/library$
```

2.3 动态库的制作和使用

命名规则：

- Linux：libxxx.so
  - lib：前缀（固定）
  - xxx：库的名字
  - .so：后缀（固定）
- Windows：libxxx.dll

制作

1. gcc 得到 `.o` 文件，得到和位置无关的代码

```
gcc -c -fpic a.c b.c
```

2. gcc 得到动态库

```
gcc -shared a.o b.o -o libcalc.so
```

例子

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so$ tree
.
├── calc
│   ├── add.c
│   ├── div.c
│   ├── head.h
│   ├── main.c
│   ├── mult.c
│   └── sub.c
└── library
    ├── app
    ├── include
    │   └── head.h
    ├── lib
    │   └── libcalc.a
    ├── main.c
    └── src
        ├── add.c
        ├── div.c
        ├── mult.c
        └── sub.c
```

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/calc$ ls
add.c  div.c  head.h  main.c  mult.c  sub.c
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/calc$ gcc -c -fpic add.c div.c sub.c mult.c
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/calc$ gcc -shared add.o div.o mult.o sub.o -o libcalc.so
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/calc$ ls
add.c  add.o  div.c  div.o  head.h  libcalc.so  main.c  mult.c  mult.o  sub.c  sub.o
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/calc$ cp libcalc.so ../library/lib/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/calc$ cd ../library/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ tree
.
├── include
│   └── head.h
├── lib
│   ├── libcalc.a
│   └── libcalc.so
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mult.c
    └── sub.c

3 directories, 8 files
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ rm lib/libcalc.a
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ gcc main.c -o app -I include/ -l calc -L lib/
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ ls
app  include  lib  main.c  src
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ ./app
./app: error while loading shared libraries: libcalc.so: cannot open shared object file: No such file or directory
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$
```

动态库加载失败原因

工作原理:

- 静态库：gcc 进行链接时，会把静态库中代码打包到可执行程序中。
- 动态库：gcc 进行链接时，动态库的代码不会被打包到可执行程序中。
- 程序启动之后，动态库会被动态加载到内存中，通过 ldd（list dynamic dependencies）命令检查动态库依赖关系。
- 如何定位共享库文件：
    - 当系统加载可执行代码时，能够知道其所依赖的库的名字，但是还需要知道绝对路径。此时就需要系统的动态载入器来获取该绝对路径。对于 elf 格式的可执行程序，是由 `ld-linux.so` 来完成的，它先后搜索 elf 文件的 **DT_RPATH段->环境变量LD_LIBRARY_PATH->/etc/ld.so.cache文件列表->/lib/ 或 /usr/lib** 目录找到库文件后将其载入内存。

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ ./app
./app: error while loading shared libraries: libcalc.so: cannot open shared object file: No such file or directory
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ ldd app
        linux-vdso.so.1 (0x00007ffc7014a000)
        libcalc.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1933998000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1933f8b000)
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$
```

解决动态库加载失败问题

第一种方法：终端配置环境变量

该方法配置的环境变量只是临时的，只对当前终端有效。

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/lhx/桌面/Linux-Server/lesson-so/library/lib
```

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library/lib$ pwd
/home/lhx/桌面/Linux-Server/lesson-so/library/lib
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library/lib$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/lhx/桌面/Linux-Server/
lesson-so/library/lib
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library/lib$ echo $LD_LIBRARY_PATH
:/home/lhx/桌面/Linux-Server/lesson-so/library/lib
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library/lib$ ldd app
ldd: ./app: 没有那个文件或目录
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library/lib$ cd ..
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ ldd app
        linux-vdso.so.1 (0x00007fffac54c000)
        libcalc.so => /home/lhx/桌面/Linux-Server/lesson-so/library/lib/libcalc.so (0x00007faebb0fa000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007faebad09000)
        /lib64/ld-linux-x86-64.so.2 (0x00007faebb4fe000)
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$ ./app
a+b=24
a-b=-4
a*b=140
a/b=0.714286
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-so/library$
```

第二种方法：用户级别配置或系统级别配置

用户级别配置：

修改 `home/用户名` 目录下的 `.bashrc` 文件，在最后一行添加环境变量

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/lhx/桌面/Linux-Server/lesson-so/library/lib
```

保存后使该文件生效：`. .bashrc` 或 `source .bashrc`

系统级别配置

```
sudo vim /etc/profile
```

在最后一行写入同样的内容，最后使之生效 `source /etc/profile`

第三种方法：配置/etc/ld.so.cache

间接修改文件

```
sudo vim /etc/ld.so.conf
```

将路径加入该文件中，然后执行 `sudo ldconfig`

2.4 静态库和动态库的优缺点

静态库：

- 优点：
  - 静态库被打包到应用程序中加载速度块
  - 发布程序无需提供静态库，移植方便
- 缺点：
  - 消耗系统资源，浪费内存
  - 更新、部署、发布麻烦

动态库：

- 优点：
  - 可以实现进程间资源共享
  - 更新、部署、发布简单

    - 可以控制何时加载动态库
- 缺点：
    - 加载速度比静态库慢
    - 发布程序需要提供依赖的动态库

2.5 Makefile

文件命名：makefile 或 Makefile

Makefile规则：

- ```
  目标 ...: 依赖 ...
  #注意下一条语句前有Tab，Tab代表此行为命令行
      命令(shell命令)
      ...
  ```

- 目标：最终要生成的文件（伪目标除外）
- 依赖：生成目标所需要的文件或目标
- 命令：通过执行命令对依赖操作生成目标（命令前必须 Tab 缩进）
- **Makefile中其它规则一般都是为第一条规则服务的。**如果其它规则与第一条规则没有关系，则其它规则不会被执行，如果想执行，需要手动指定。

2.5.1 Makefile编写示例

```
vim Makefile
```

如下makefile语句表示：目标 `app` 的依赖有 `add.c sub.c div.c mult.c main.c` ，下一句则是生成目标的命令，终端执行make命令就生成了目标文件。

```
app:add.c sub.c div.c mult.c main.c
    gcc sub.c add.c mult.c div.c main.c -o app
```

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ ls
add.c  div.c  head.h  main.c  Makefile  mult.c  sub.c
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ make
gcc sub.c add.c mult.c div.c main.c -o app
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ ls
add.c  app  div.c  head.h  main.c  Makefile  mult.c  sub.c
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ cat Makefile
app:add.c sub.c div.c mult.c main.c
        gcc sub.c add.c mult.c div.c main.c -o app
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ ./app
a+b=24
a-b=-4
a*b=140
a/b=0.714286
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$
```

2.5.2 Makefile工作原理

命令在执行之前，需要先检查规则中的依赖是否存在：

- 如果存在，执行命令
- 如果不存在，向下检查其它规则，检查有没有一个规则是用来生成这个依赖的，如果找到了，则执行该规则的命令。

检测更新，在执行规则中的命令时，会比较目标和依赖文件的时间：

- 如果依赖时间比目标时间晚，需要重新生成目标。
- 如果依赖比目标时间早，目标不需要更新，对应规则中的命令不需要被执行。
```

```
app:add.o sub.o div.o mult.o main.o
    gcc add.o sub.o div.o mult.o main.o -o app
sub.o:sub.c
    gcc -c sub.c -o sub.o
add.o:add.c
    gcc -c add.c -o add.o
div.o:div.c
    gcc -c div.c -o div.o
mult.o:mult.c
    gcc -c mult.c -o mult.o
main.o:main.c
    gcc -c main.c -o main.o
```

2.5.3 Makefile变量、模式匹配、函数

- 自定义变量

  ```
  变量名=变量值
  ```

- 预定义变量

  ```
  AR：归档维护程序的名称，默认值为ar
  CC：C编译器的名称，默认值为cc
  CXX：C++编译器的名称，默认值为g++
  $@：目标的完整名称
  $<：第一个依赖文件的名称
  $^：所有的依赖文件
  ```

- 获取变量的值

  ```
  $(变量名)
  ```

- 模式匹配

  ```
  %：通配符，匹配一个字符串
  ```

- 函数：`$(wildcard PATTERN...)`，wildcard是函数名，PATTERN代表参数
  - 功能：获取指定目录下指定类型的文件列表
  - 参数：PATTERN指的是某个或多个目录下的对应的某种类型的文件，如果有多个目录，一般使用空格间隔。
  - 返回：得到的若干个文件的文件列表，文件名之间使用空格间隔。
  - 示例：

    ```
    $(wildcard *.c ./sub/*.c)
    返回值：a.c b.c c.c d.c
    ```

- 函数：`$(patsubst <pattern>,<replacement>,<text>)`
  - 功能：查找 <text> 中的单词（单词以空格、Tab、回车、换行分隔）是否符合模式 <pattern>，如果匹配，则以 <replacement>替换。

- <pattern> 可以包括通配符%，表示任意长度的字符串。如果 <replacement> 中也包含%，那么，<replacement> 中的这个%将是 <pattern> 中的那个%所代表的字符串。
- 饭hi：函数返回被替换过后的字符串
- 示例

  ```
  (passubst %.c, %.o x.c bar.c)
  返回值: x.o bar.o
  ```

```
#获取当前目录下的所有.c文件
src=$(wildcard ./*.c)
#将.c替换成.o，因为我们用.o文件生成目标文件
objs=$(patsubst %.c, %.o, $(src))
target=app
#目标和依赖
target:$(objs)
    $(CC) $(objs) -o $(target)
%.o:%.c
    $(CC) -c $< -o $@
#伪目标
.PHONY:clean
clean:
    rm $(objs) -f
```

```
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ ls
add.c  div.c  head.h  main.c  Makefile  Makefile2  Makefile3  mult.c  sub.c
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ make
cc -c mult.c -o mult.o
cc -c main.c -o main.o
cc -c add.c -o add.o
cc -c div.c -o div.o
cc -c sub.c -o sub.o
cc  ./mult.o  ./main.o  ./add.o  ./div.o  ./sub.o -o app
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ ls
add.c  add.o  app  div.c  div.o  head.h  main.c  main.o  Makefile  Makefile2  Makefile3  mult.c  mult.o  sub.c  sub.o
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ make clean
rm  ./add.o  ./mult.o  ./main.o  ./div.o  ./sub.o -f
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$ ./app
a+b=24
a-b=-4
a*b=140
a/b=0.714286
lhx@lhx-virtual-machine:~/桌面/Linux-Server/lesson-makefile$
```

2.6 GDB

通常，在为调试而编译时，需要关闭编译器的优化选项 `-O`，并打开调试选项 `-g` 。另外，`-Wall` 在尽量不影响程序行为的情况下选择打开所有warning，也可以发现许多问题，避免一些不必要的 bug。

`-g` 的作用是向可执行程序中加入源代码信息

```
gcc -g -Wall program.c -o program
```

2.6.1 GDB基本命令

- 启动：gdb 可执行程序
- 退出：quit
- 给程序设置参数/获取设置的参数

  ```
  set args 10 20
  show args
  ```

- 查看当前文件代码

```
list/l (默认位置显示)
list/l 行号 (显示以该行号为中心的范围)
list/l 函数名 (显示以该函数名为中心的范围)
```

- 查看非当前文件代码

```
list/l 文件名:行号
list/l 文件名:函数名
```

- 设置显示的行数

```
show lsit/listsize
set list/listsize 行数
```

2.6.2 断点操作

- 设置断点

```
b/break 行号
b/break 函数名
b/break 文件名:行号
b/break 文件名:函数
```

- 查看断点

```
i/info b/break
```

- 删除断点

```
d/del/delete 断点编号
```

- 设置断点无效

```
dis/disable 断点编号
```

- 设置断点生效

```
ena/enabel 断点编号
```

- 设置条件断点（一般用在循环的位置）

```
b/break 10 if i==5
```

2.6.3 调试命令

- 运行GDB程序

```
start (程序停在第一行)
run (遇到断点才停)
```

- 继续运行，到下一个断点停

```
c/continue
```

- 向下执行一行代码（不会进入函数体）

```
n/next
```

- 变量操作

```
p/print 变量名 (打印变量值)
ptype 变量名 (打印变量类型)
```

- 向下单步调试（遇到函数进入函数体）

```
s/step
finish (跳出函数体，函数内不能有断点)
```

- 自动变量操作

```
display num (自动打印指定变量的值)
i/info display
undisplay 编号 (删除自动变量)
```

- 其它

```
set var 变量名=变量值
untill (跳出循环，循环内不能有断点，当前循环要执行完，否则无法跳出)
```

3、Linux系统函数

3.1 fcntl

fcntl针对文件描述符进行控制

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ...);
//fd: 表示需要操作的文件描述符
//cmd: 表示对文件描述符进行什么操作
```

cmd参数：

```
//F_DUPFD: 复制文件描述符，复制的是第一个参数fd，得到一个新的文件描述符
int ret=fcntl(fd,F_DUPFD);
//F_GETFL: 获取指定的文件描述符文件状态flag
//F_SETFL: 设置文件描述符的文件状态
    //必选项: O_RDONLY, O_WRONLY, O_RDWR
    //可选项: O_APPEND,O_NONBLOCK
```

```
#include <unistd.h>
#include <fcntl.h>
```

```c
#include <stdio.h>
#include <string.h>
int main(){
    /*
    复制文件描述符
    int fd=open("1.txt",O_RDONLY);
    int ret=fcntl(fd,F_DUPFD);
    */

    int fd=open("1.txt",O_RDWR);
    if(fd==-1){
        perror("open");
        return -1;
    }
    int flag=fcntl(fd,F_GETFL);
    flag |= O_APPEND;

    int ret=fcntl(fd,F_SETFL,flag);
    char* str="world";
    write(fd,str,strlen(str));
    close(fd);
    return 0;
}
```

3.2 open

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

//flag: 对文件的操作权限设置和其它设置
    //必须的一个选项: O_RDONLY, O_WRONLY, O_RDWR
    //可选项: O_CLOEXEC, O_CREAT, O_DIRECTORY,
        //   O_EXCL, O_NOCTTY, O_NOFOLLOW, O_TMPFILE, O_TRUNC
//mode: 八进制的数，表示用户对创建出的新的文件的操作权限，如chmod后跟的数字，例: 0775
    //最终结果: mode&~umask      umask可以在终端输入查看
//返回一个文件描述符，错误返回-1
```

3.3 read、write

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
//从fd向buf读取最多count个字节
//buf: 读取的数据存放的地方
//count: 读取数据的大小

ssize_t write(int fd, const void *buf, size_t count);
//从buf开始的缓冲区写count字节到fd的文件中
```

3.4 lseek

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
//文件指针偏移，从whence偏移offset，作用和C的fseek一样
//返回值：文件指针的位置
//whence：SEEK_SET，SEEK_CUR，SEEK_END
```

3.5 stat、lstat

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *pathname, struct stat *statbuf);
//获取一个文件的相关信息
//pathname：文件路径
//statbuf：传出参数，用于保存获取到的文件的信息

int lstat(const char *pathname, struct stat *statbuf);
//获取软连接文件的信息
```

```
struct stat
{
    dev_t       st_dev;     /* 文件所在设备的ID*/
    ino_t       st_ino;     /* 节点号*/
    mode_t      st_mode;    /* 文件类型和存储权限*/
    nlink_t     st_nlink;   /* 连接到该文件的硬连接数*/
    uid_t       st_uid;     /* 用户ID*/
    gid_t       st_gid;     /* 组ID*/
    dev_t       st_rdev;    /* 设备文件的设备编号*/
    off_t       st_size;    /* 文件大小，字节为单位*/
    blksize_t   st_blksize; /* 系统块的大小*/
    blkcnt_t    st_blocks;  /* 文件所占块数*/
    time_t      st_atime;   /* 最后异常访问时间*/
    time_t      st_mtime;   /* 最后一次修改时间*/
    time_t      st_ctime;   /* 最后一次改变时间(属性)*/
};
```

3.6 文件属性操作函数

```
#include <unistd.h>
int access(const char *pathname, int mode);
//判断某个文件是否有某个权限，或者判断文件是否存在
//mode:
    //R_OK：是否有读权限
    //W_OK：是否有写权限
    //X_OK：是否有执行权限
    //F_OK：文件是否存在
```

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
//修改文件权限
//mode：需要修改的权限值，8进制数
```

```c
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
//更改文件的所有者和组
```

```c
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
//截取或扩展文件的尺寸至指定的大小
```

3.7 目录操作函数

```c
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
//创建目录
```

```c
#include <unistd.h>
int rmdir(const char *pathname);
//删除目录，只能删除空目录
```

```c
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
//更改文件的名称或位置
```

```c
#include <unistd.h>
int chdir(const char *path);
//更改进程的工作目录
```

```c
#include <unistd.h>
char *getcwd(char *buf, size_t size);
//获取当前工作目录，保存到buf中
```

3.8 目录遍历函数

```c
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
//打开目录
```

```c
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
//读取目录
//dirp是opendir返回的结果
//返回值是读取到的文件的信息

struct dirent
{
   long d_ino; /* inode number 索引节点号 */
   off_t d_off; /* offset to this dirent 目录文件开头到该目录进入点的偏移 */
   unsigned short d_reclen; /* length of this d_name 文件名长 */
   unsigned char d_type; /* the type of d_name 文件类型 */
```

```
    char d_name [256]; /* file name (null-terminated) 文件名，最长255字符 */
}
```

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
//关闭目录
```

3.9 dup、dup2

```
#include <unistd.h>
int dup(int oldfd);
//复制文件描述符，返回新的文件描述符


int dup2(int oldfd, int newfd);
//重定向文件描述符，返回新的文件描述符，返回值和newfd相同
```

4、进程

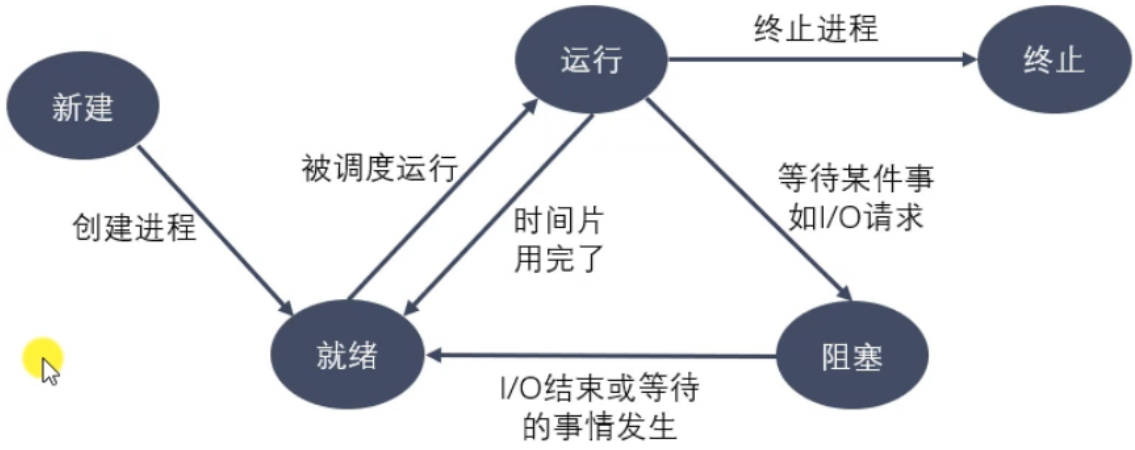
进程组是一个或多个进程的集合

4.1 进程的状态

三态模型：就绪态，运行态，阻塞态。

五态模型：新建态，就绪态，运行态，阻塞态，终止态。

1. 就绪态：进程具备运行条件，等待系统分配处理器以便运行。在一个系统中处于就绪状态的进程可能有多个，通常将他们排成一个队列，称为就绪队列。
2. 运行态：进程占有处理器正在运行。
3. 阻塞态：又称为等待或睡眠状态，指进程不具备运行条件，正在等待某个事件的完成。
4. 新建态：进程刚被创建时的状态，尚未进入就绪队列。
5. 终止态：进程完成任务到达正常结束点，或出现无法克服的错误而异常终止，或被操作系统及有终止权的进程所终止时所处的状态。进入终止态的进程以后不再执行，但依然保留在操作系统中等待善后，一旦其它进程完成了对终止态进程的信息抽取之后，操作系统将删除该进程。



4.2 进程相关的命令

查看进程：

```
ps aux 或 ajx
a: 显示终端上的所有进程，包括其它用户的进程
u: 显示进程的详细信息
x: 显示没有控制终端的进程
j: 列出与作业控制相关的信息
```

STAT参数

```
D 不可中断
R 正在运行，或在队列中的进程
S 处于休眠状态
T 停止或被追踪
Z 僵尸进程
W 进入内存交换
X 死掉的进程
< 高优先级
N 低优先级
s 包含子进程
+ 位于前台的进程组
```

实时显示进程信息

```
top
-d 可以指定显示信息更新的时间间隔
使用top后可以按照以下对结果进程排序:
    M 根据内存使用量
    P 根据CPU占有率
    T 根据进程运行时间长短
    U 根据用户名筛选
    K 输入指定的PID杀死进程
```

杀死进程

```
kill [-signal] pid
kill -l
kill -SIGKILL 进程ID
kill -9 进程ID                  强制杀死进程
killall name                    根据进程名杀死进程
```

4.3 进程创建 fork

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
//创建子进程
//返回值:
    //fork会返回两次，一次在父进程中，一次在子进程中
    //成功: 子进程返回0，父进程返回子进程ID
    //失败: 返回-1，父进程中返回-1表示创建子进程失败
```

Linux的 fork() 是通过写时拷贝实现。

写时拷贝是一种可以推迟甚至避免拷贝数据的技术，内核此时并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间。只用在需要写入的时候才会复制地址空间，从而是各个进程拥有各自的地址空间。**即资源只有在写入时才会复制，在此之前，只有以只读方式共享内存。**

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
    pid_t pid = fork();
    if(pid > 0){
        printf("返回的子进程pid: %d\n",pid);
        printf("父进程, pid=%d, ppid=%d\n",getpid(),getppid());
    }
    else if(pid == 0){
        printf("子进程, pid=%d, ppid=%d\n",getpid(),getppid());
    }
    for(int i=0;i<3;i++){
        printf("i:%d, pid=%d\n",i,getpid());
    }
    return 0;
}
```

4.4 GDB多进程调试

设置调试父进程还是子进程：set follow-fork-mode [parent | child]

设置调试模式：set detach-on-fork [on | off]

- 默认为 on，表示调试当前进程的时候，其它的进程继续执行；如果为 off，表示调试当前进程的时候，其它进程被 GDB 挂起。

查看调试的进程：info inferiors

切换当前调试的进程：inferior id

使进程脱离 GDB 调试： detach inferiors id

4.5 exec 函数族

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来**取代调用进程的内容**，即在调用进程内部执行一个可执行文件。

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
//path: 需要执行的文件的路径
//arg: 执行可执行文件的参数列表，第一个参数为文件名，参数最后需要以NULL结束


int execlp(const char *file, const char *arg);
//会到环境变量中查找指定的可执行文件，如果找到了就执行，找不到就执行不成功
//file: 需要执行的可执行文件名
```

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t pid=fork();
```

```
    if(pid > 0){
        printf("父进程, pid=%d\n",getpid());
    }
    else if(pid == 0){
        execl("a.out","a.out",NULL);
        printf("子进程, pid=%d\n",getpid());
    }
    return 0;
}
```

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t pid=fork();
    if(pid > 0){
        printf("父进程, pid=%d\n",getpid());
    }
    else if(pid == 0){
        execlp("ps","ps","aux",NULL);
        printf("子进程, pid=%d\n",getpid());
    }
    return 0;
}
```

4.6 wait、waitpid

4.6.1 进程回收

在每个进程退出时，内核释放该进程所有的资源，但会保留一定的信息，这些信息主要指进程控制块PCB的信息。

父进程可以通过调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。

wait和waitpid的功能一样，区别在于wait会阻塞，waitpid可以设置不阻塞，waitpid还可以指定等待哪个子进程结束。

一次wait或waitpid调用只能清理一个子进程，**清理多个子进程应使用循环。**

4.6.2 wait

调用wait函数的进程会被阻塞，直到它的一个子进程退出或者收到一个不能被忽略的信号时才被唤醒。**如果没有子进程或子进程都结束了，函数立刻返回-1，否则返回被回收的子进程的id。**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus);
//wstatus: 进程退出时的状态信息
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main(){
    pid_t pid;
    for(int i=0;i<5;i++){
```

```
        pid=fork();
        if(pid == 0) break;
    }
    if(pid > 0){
        while(1){
            printf("父进程, pid=%d\n",getpid());
            int ret = wait(NULL);
            if(ret == -1) break;
            printf("子进程死亡, pid=%d\n",ret);
            sleep(1);
        }


    }
    else if(pid == 0){
        while(1){
            printf("子进程, pid=%d\n",getpid());
            sleep(2);
        }

    }

    return 0;
}
```

4.6.3 waitpid

回收指定进程号的子进程，可以设置非阻塞

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
//pid = 0 : 回收当前进程组的所有子进程
//pid = -1 : 回收所有的子进程
//pid < -1 : 回收某个进程组的组id的绝对值，回收指定进程组的子进程
//options: 设置阻塞或非阻塞
    //0: 阻塞
    //WNOHANG: 非阻塞
//返回值:
    //>0: 返回子进程的id
    //=0: options=WNOHANG, 表示还有子进程
    //=-1: 错误，或没有子进程
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main(){
    pid_t pid;
    for(int i=0;i<5;i++){
        pid=fork();
        if(pid == 0) break;
    }
    if(pid > 0){
        while(1){
            printf("父进程, pid=%d\n",getpid());
```

```
            int ret = waitpid(-1,NULL,WNOHANG);
            if(ret == -1) break;
            printf("子进程死亡, pid=%d\n",ret);
            sleep(1);
        }


    }
    else if(pid == 0){
        while(1){
            printf("子进程, pid=%d\n",getpid());
            sleep(10);
        }

    }
    return 0;
}
```

4.7 匿名管道

创建匿名管道

```
#include <unistd.h>
int pipe(int pipefd[2]);
//pipefd[2]: 是一个传出参数
    //pipefd[0]是管道的读端，pipefd[1]是管道的写端
//返回值: 成功返回0，失败返回-1
```

查看管道缓冲区大小：ulimit -a

查看管道缓冲区大小函数

```
#include <unistd.h>
long fpathconf(int fd,int name);
```

4.7.1 例子：父子进程通过匿名管道通信

注意：实际中父子进程不要同时读写数据，否则会造成数据不对；同时读写可以加sleep()保持数据同步。

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    //在fork之前创建管道
    int pipefd[2];
    int ret = pipe(pipefd);
    if(ret == -1){
        perror("pipe");
        exit(0);
    }
    pid_t pid = fork();
```

```
    if(pid > 0){
        char buf[256] = {0};
        printf("父进程: pid=%d\n",getpid());
        close(pipefd[1]);
        while(1){
            read(pipefd[0], buf, sizeof(buf));
            printf("父进程接收数据: %s, pid=%d\n", buf, getpid());
            bzero(buf,sizeof(buf));
            //char *str = "the data from parent";
            //write(pipefd[1], str, strlen(str));
        }
    }
    else if(pid == 0){
        //char buf[256] = {0};
        printf("子进程: pid=%d\n",getpid());
        close(pipefd[0]);
        while(1){
            char *str = "the data from child";
            write(pipefd[1], str, strlen(str));
            sleep(1);
            //read(pipefd[0], buf, sizeof(buf));
            //printf("子进程接收数据: %s, pid=%d\n", buf, getpid());
            //bzero(buf,sizeof(buf));
        }
    }
    return 0;
}
```

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <wait.h>
int main()
{
    int fd[2];
    int ret = pipe(fd);
    if(ret == -1){
        perror("pipe");
        exit(0);
    }

    pid_t pid = fork();
    if(pid > 0){
        close(fd[1]);
        char buf[1024] = {0};
        int len = -1;
        while(len=read(fd[0],buf,sizeof(buf)-1)>0){
            printf("%s",buf);
            memset(buf,0,sizeof(buf));
        }
        wait(NULL);
    }
    else if(pid == 0){
        close(fd[0]);
        //重定向标准输出到管道写端
```

```
        dup2(fd[1], STDOUT_FILENO);
        execlp("ps","ps","aux",NULL);
        perror("execlp");
        exit(0);
    }
    else{
        perror("fork");
        exit(0);
    }
    return 0;
}
```

4.7.2 管道的读写特点

使用管道时，需要注意以下几种特殊的情况（假设都是阻塞I/O操作）

1. 所有的指向管道写端的文件描述符都关闭了（管道写端引用计数为0），有进程从管道的读端读数据，那么管道中剩余的数据被读取以后，再次read会返回0，就像读到文件末尾一样。
2. 如果有指向管道写端的文件描述符没有关闭（管道的写端引用计数大于0），而持有管道写端的进程也没有往管道中写数据，这个时候有进程从管道中读取数据，那么管道中剩余的数据被读取后，再次read会阻塞，直到管道中有数据可以读了才读取数据并返回。
3. 如果所有指向管道读端的文件描述符都关闭了（管道的读端引用计数为0），这个时候有进程向管道中写数据，那么该进程会收到一个信号SIGPIPE, 通常会导致进程异常终止。
4. 如果有指向管道读端的文件描述符没有关闭（管道的读端引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道中写数据，那么在管道被写满的时候再次write会阻塞，直到管道中有空位置才能再次写入数据并返回。

总结:

- 读管道:
  - 管道中有数据，read返回实际读到的字节数。
  - 管道中无数据:
    - 写端被全部关闭，read返回0（相当于读到文件的末尾）
    - 写端没有完全关闭，read阻塞等待
- 写管道:
  - 管道读端全部被关闭，进程异常终止（进程收到SIGPIPE信号）
  - 管道读端没有全部关闭:
    - 管道已满，write阻塞
    - 管道没有满，write将数据写入，并返回实际写入的字节数

4.7.3 例子：管道设置非阻塞

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
/*
    设置管道非阻塞
    int flags = fcntl(fd[0], F_GETFL);  // 获取原来的flag
    flags |= O_NONBLOCK;            // 修改flag的值
    fcntl(fd[0], F_SETFL, flags);   // 设置新的flag
```

```c
*/
int main() {

    // 在fork之前创建管道
    int pipefd[2];
    int ret = pipe(pipefd);
    if(ret == -1) {
        perror("pipe");
        exit(0);
    }

    // 创建子进程
    pid_t pid = fork();
    if(pid > 0) {
        // 父进程
        printf("i am parent process, pid : %d\n", getpid());

        // 关闭写端
        close(pipefd[1]);

        // 从管道的读取端读取数据
        char buf[1024] = {0};

        int flags = fcntl(pipefd[0], F_GETFL);  // 获取原来的flag
        flags |= O_NONBLOCK;            // 修改flag的值
        fcntl(pipefd[0], F_SETFL, flags);   // 设置新的flag

        while(1) {
            int len = read(pipefd[0], buf, sizeof(buf));
            printf("len : %d\n", len);
            printf("parent recv : %s, pid : %d\n", buf, getpid());
            memset(buf, 0, 1024);
            sleep(1);
        }

    } else if(pid == 0){
        // 子进程
        printf("i am child process, pid : %d\n", getpid());
        // 关闭读端
        close(pipefd[0]);
        char buf[1024] = {0};
        while(1) {
            // 向管道中写入数据
            char * str = "hello,i am child";
            write(pipefd[1], str, strlen(str));
            sleep(5);
        }

    }
    return 0;
}
```

4.8 有名管道

通过命令创建有名管道：mkfifo name

通过函数创建有名管道

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

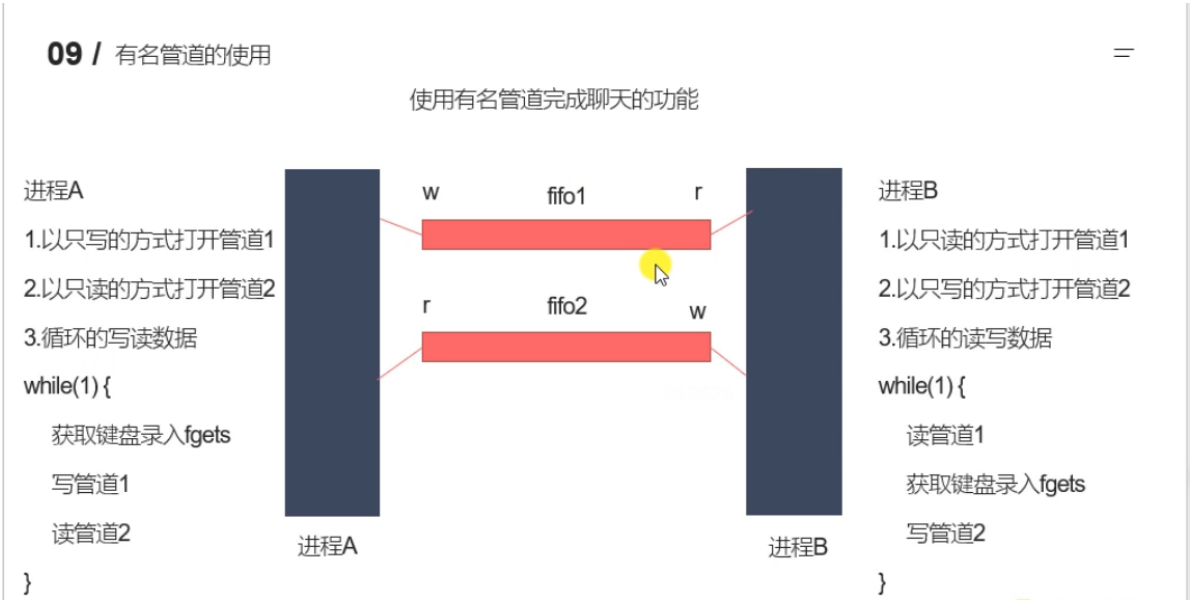
一旦使用了 mkfifo 创建了一个 FIFO ， 就可以使用 open 打开。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int ret = mkfifo("fifo1",0664);
    if(ret == -1){
        perror("mkfifo");
        exit(0);
    }
    return 0;
}
```

注意:

- 一个为只读而打开一个管道的进程会阻塞，直到另一个进程为只写打开管道；
- 一个为只写而打开一个管道的进程会阻塞，直到另一个进程为只读打开管道；

4.8.1 例子：有名管道实现简单聊天功能




```
//chatA.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
int main()
{
    char *str1="testA",*str2="testB";
```

```
    int ret = access(str1,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
        if(mkfifo(str1,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access(str2,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
        if(mkfifo(str2,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }
    int fdw = open(str1,O_WRONLY);
    if(fdw != -1)
        printf("打开%s管道成功\n",str1);

    int fdr = open(str2,O_RDONLY);
    if(fdr != -1)
        printf("打开%s管道成功\n",str2);
    char buf[1024] = {0};
    while (1){
        char str[512] = {0};
        fgets(str,sizeof(str),stdin);
        if(write(fdw,str,strlen(str)) == -1){
            perror("write");
            exit(0);
        }
        if(read(fdr,buf,sizeof(buf)) == -1){
            perror("read");
            exit(0);
        }
        printf("A recv data: %s\n",buf);
        memset(buf,0,sizeof(buf));
    }

    return 0;
}
```

```
//chatB.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
int main()
{
    char *str1="testA",*str2="testB";
    int ret = access(str1,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
```

```
        if(mkfifo(str1,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access(str2,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
        if(mkfifo(str2,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }
    int fdr = open(str1,O_RDONLY);
    if(fdr != -1)
        printf("打开%s管道成功\n",str1);

    int fdw = open(str2,O_WRONLY);
    if(fdw != -1)
        printf("打开%s管道成功\n",str2);
    char buf[1024] = {0};
    while (1){
        if(read(fdr,buf,sizeof(buf)) == -1){
            perror("read");
            exit(0);
        }
        printf("B recv data: %s\n",buf);
        char str[512] = {0};
        fgets(str,sizeof(str),stdin);
        if(write(fdw,str,strlen(str)) == -1){
            perror("write");
            exit(0);
        }
        memset(buf,0,sizeof(buf));
    }

    return 0;
}
```

注意，由于 write 和 read 的阻塞原因，当一个进程连续发送数据后，只有当另一个进程发送了数据才可以将后面的数据发送出去，即以上指实现了发一次，读一次的功能。

解决办法：创建子进程，父进程用来发数据，子进程用来收数据。

```
//chatA.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <wait.h>
int main()
{
    char *str1="testA",*str2="testB";
```

```
    int ret = access(str1,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
        if(mkfifo(str1,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access(str2,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
        if(mkfifo(str2,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }
    int fdw = open(str1,O_WRONLY);
    if(fdw != -1)
        printf("打开%s管道成功\n",str1);

    int fdr = open(str2,O_RDONLY);
    if(fdr != -1)
        printf("打开%s管道成功\n",str2);

    pid_t pid = fork();
    if(pid > 0){
        wait(NULL);
        while(1){
            char str[512] = {0};
            fgets(str,sizeof(str),stdin);
            if(write(fdw,str,strlen(str)) == -1){
                perror("write");
                exit(0);
            }
        }
    }
    else if(pid == 0){
        char buf[1024] = {0};
        int len=-1;
        while (1){
            len = read(fdr,buf,sizeof(buf));
            if(len == -1){
                perror("read");
                exit(0);
            }
            else if(len == 0) break;
            printf("A recv data: %s\n",buf);
            memset(buf,0,sizeof(buf));
        }
        exit(0);
    }

    return 0;
}
```

```
//chatB.c
```

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <wait.h>
int main()
{
    char *str1="testA",*str2="testB";
    int ret = access(str1,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
        if(mkfifo(str1,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access(str2,F_OK);
    if(ret == -1){
        printf("管道不存在，创建管道\n");
        if(mkfifo(str2,0664) == -1){
            perror("mkfifo");
            exit(0);
        }
    }
    int fdr = open(str1,O_RDONLY);
    if(fdr != -1)
        printf("打开%s管道成功\n",str1);

    int fdw = open(str2,O_WRONLY);
    if(fdw != -1)
        printf("打开%s管道成功\n",str2);

    pid_t pid = fork();
    if(pid > 0){
        wait(NULL);
        while(1){
            char str[512] = {0};
            fgets(str,sizeof(str),stdin);
            if(write(fdw,str,strlen(str)) == -1){
                perror("write");
                exit(0);
            }
        }
    }
    else if(pid == 0){
        char buf[1024] = {0};
        int len=-1;
        while (1){
            len = read(fdr,buf,sizeof(buf));
            if(len == -1){
                perror("read");
                exit(0);
            }
            else if(len == 0) break;
```

```
            printf("B recv data: %s\n",buf);
            memset(buf,0,sizeof(buf));
        }
        exit(0);
    }

    return 0;
}
```

4.9 内存映射

内存映射是将磁盘文件的数据映射到内存，用户通过修改内存就能修改磁盘文件。

```
#include <sys/mman.h>
void *mmap(void *addr,size_t length,int prot,int flags,int fd,off_t offset);
//将一个文件或设备的数据映射到内存中
//返回值: 创建的内存的地址
//void* addr: NULL, 由内核指定
//length: 要映射的数据的长度，不能为0，建议使用文件的长度。最终为分页的整数倍
//prot: 对申请的内存映射区的的操作权限（读、写、执行、无）
//flags:
    //MAP_SHARED: 映射区的数据会自动和磁盘文件进行同步，进程间通信必须设置这个选项
    //MAP_PRIVATE: 不同步，内存映射区的数据改变，对原来的文件不影响，会创建一个新文件
//fd: 通过open一个磁盘文件得到，该文件权限不能和prot冲突
//offset: 偏移量，一般不用，必须指定的是4k的整数倍，0表示不偏移


int munmap(void *arrd,size_t length);
//arrd: 要释放的内存的首地址
//length: 要释放的内存的大小
```

```
/*
    使用内存映射实现进程间通信:
    1.有关系的进程（父子进程）
        - 还没有子进程的时候
            - 通过唯一的父进程，先创建内存映射区
        - 有了内存映射区以后，创建子进程
        - 父子进程共享创建的内存映射区

    2.没有关系的进程间通信
        - 准备一个大小不是0的磁盘文件
        - 进程1 通过磁盘文件创建内存映射区
            - 得到一个操作这块内存的指针
        - 进程2 通过磁盘文件创建内存映射区
            - 得到一个操作这块内存的指针
        - 使用内存映射区通信

    注意：内存映射区通信，是非阻塞。
*/

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

```c
#include <stdlib.h>
#include <wait.h>

// 作业:使用内存映射实现没有关系的进程间的通信。
int main() {

    // 1.打开一个文件
    int fd = open("test.txt", O_RDWR);
    int size = lseek(fd, 0, SEEK_END);  // 获取文件的大小

    // 2.创建内存映射区
    void *ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(ptr == MAP_FAILED) {
        perror("mmap");
        exit(0);
    }

    // 3.创建子进程
    pid_t pid = fork();
    if(pid > 0) {
        wait(NULL);
        // 父进程
        char buf[64];
        strcpy(buf, (char *)ptr);
        printf("read data : %s\n", buf);

    }else if(pid == 0){
        // 子进程
        strcpy((char *)ptr, "nihao a, son!!!");
    }

    // 关闭内存映射区
    munmap(ptr, size);

    return 0;
}
```

4.9.1 内存映射的注意事项

1. 如果对mmap的返回值(ptr)做++操作(ptr++), munmap是否能够成功?

```c
void * ptr = mmap(...);
ptr++;  可以对其进行++操作
munmap(ptr, len);   // 错误,要保存地址
```

2. 如果open时O_RDONLY, mmap时prot参数指定PROT_READ | PROT_WRITE会怎样?
    - 错误，返回MAP_FAILED
    - open()函数中的权限建议和prot参数的权限保持一致。
3. 如果文件偏移量为1000会怎样?

- 偏移量必须是4K的整数倍，返回MAP_FAILED

4. mmap什么情况下会调用失败?
    - 第二个参数：length = 0
    - 第三个参数：prot
    - 只指定了写权限

- prot PROT_READ | PROT_WRITE
- 第5个参数fd 通过open函数时指定的 O_RDONLY / O_WRONLY

5. 可以open的时候O_CREAT一个新文件来创建映射区吗?
    - 可以的，但是创建的文件的大小如果为0的话，肯定不行
    - 可以对新的文件进行扩展，如
        - lseek()
        - truncate()
6. mmap后关闭文件描述符，对mmap映射有没有影响?

```
int fd = open("XXX");
mmap(,,,,fd,0);
close(fd);
```

    - 映射区还存在，创建映射区的fd被关闭，没有任何影响。
7. 对ptr越界操作会怎样?

```
void * ptr = mmap(NULL, 100,,,,,);
4K
```

    - 越界操作操作的是非法的内存 -> 段错误

5、信号

5.1 kill、raise、abort

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
//给某个进程发送某个信号
//pid:
    //>0: 将信号发送给指定的进程
    //=0: 将信号发送给当前的进程组
    //=-1: 将信号发送给每一个有权限接收这个信号的进程
    //<-1: 这个pid=某个进程组的ID取反
//sig: 发送的信号的编号或宏
```

```
#include <signal.h>
int raise(int sig);
//给当前进程发送信号
```

```
#include <stdlib.h>
void abort(void);
//发送SIGABRT信号给当前进程，杀死当前进程
```

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
int main()
{
    pid_t pid=fork();
```

```
    if(pid == 0){
        int i=0;
        for(i=0;i<5;i++){
            printf("子进程\n");
            sleep(5);
        }
    }
    else if(pid > 0){
        printf("父进程\n");
        sleep(2);
        printf("kill 子进程\n");
        kill(pid,SIGINT);
    }
    return 0;
}
```

5.2 定时函数：alarm、setitimer

5.2.1 alarm

SIGALRM：默认终止当前进程，一个进程有且只有一个定时器。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
//设置定时器。函数调用开始倒计时，当倒计时为0时，函数会给当前的进程发送一个信号：SIGALARM
//如果参数为0，表示定时器无效
//返回值：
    //之前有定时器，返回0
    //之前没有定时器，返回之前的定时器剩余时间
```

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    int seconds = alarm(5);
    printf("seconds=%d\n",seconds); //0
    sleep(2);
    seconds = alarm(2);
    printf("seconds=%d\n",seconds); //3，2秒后程序终止
    while(1){}
    return 0;
}
```

5.2.2 setitimer

```
int setitimer(int which, const struct itimerval *new_value, struct itimerval
*old_value);
//
//设置定时器,可以替代alarm函数，可以实现周期性定时
//which: 定时器以什么时间计时
    //ITIMER_REAL: 系统真实时间，时间到达，发送SIGALRM
    //ITIMER_VIRTUAL: 用户态花费的时间，时间到达，发送SIGVTALRM
    //ITIMER_PROF: 以该进程在用户态和内核态下所消耗的时间来计算，发送SIGPROF
//new_value: 设置定时器属性
//old_value: 记录上一次的定时的时间参数，一般不使用，指定为NULL
```

setitimer方法调用成功后，延时it_value后触发一次SIGALRM信号，以后每隔it_interval触发一次SIGALRM信号。

settimer工作机制是，先对it_value倒计时，当it_value为零时触发信号。然后重置为it_interval。继续对it_value倒计时。一直这样循环下去。

假如it_value为0是不会触发信号的，所以要能触发信号，it_value得大于0；假设it_interval为零，仅仅会延时。不会定时（也就是说仅仅会触发一次信号)。

```
struct itimerval{
    struct timeval it_interval; //间隔时间
    struct timeval it_value;    //延迟多长时间执行定时器
}
struct timeval{
    time_t tv_sec;
    suseconds_t tv_usec;
}
```

```
#include <sys/time.h>
#include <stdio.h>
int main()
{
    struct itimerval new_value;
    //间隔2s
    new_value.it_interval.tv_sec=2;
    new_value.it_interval.tv_usec=0;
    //定时3s
    new_value.it_value.tv_sec=3;
    new_value.it_value.tv_sec=0;
    setitimer(ITIMER_REAL,&new_value,NULL);
}
```

5.3 信号捕捉函数

5.3.1 signal

```c
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
//功能: 设置某个信号的捕捉行为
//signum: 要捕捉的信号
//handler: 捕捉到信号要如何处理，这是个回调函数
    //SIG_IGN: 忽略信号          SIG_DFL: 使用信号默认的行为
//成功返回上一次注册的信号处理函数的地址。第一次调用返回NULL
//失败返回SIG_ERR，设置错误号
//SIGKILL SIGSTOP不能被捕捉，不能被忽略
```

```c
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void myalarm(int num) {
    printf("捕捉到了信号的编号是：%d\n", num);
    printf("xxxxxxx\n");
}

// 过3秒以后，每隔2秒钟定时一次
int main() {

    // 注册信号捕捉
    // signal(SIGALRM, SIG_IGN);
    // signal(SIGALRM, SIG_DFL);
    // void (*sighandler_t)(int); 函数指针，int类型的参数表示捕捉到的信号的值。
    signal(SIGALRM, myalarm);

    struct itimerval new_value;

    // 设置间隔的时间
    new_value.it_interval.tv_sec = 2;
    new_value.it_interval.tv_usec = 0;

    // 设置延迟的时间,3秒之后开始第一次定时
    new_value.it_value.tv_sec = 3;
    new_value.it_value.tv_usec = 0;

    int ret = setitimer(ITIMER_REAL, &new_value, NULL); // 非阻塞的
    printf("定时器开始了...\n");

    if(ret == -1) {
        perror("setitimer");
        exit(0);
    }

    getchar();

    return 0;
}
```

5.3.2 sigaction

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact);
//检查或者改变信号的处理。信号捕捉
//signum: 需要捕捉的信号的编号或者宏值
//act: 捕捉到信号之后的处理动作
//oldact: 上一次对信号捕捉相关的设置，一般不使用，传递NULL
```

```
struct sigaction {
    // 函数指针，指向的函数就是信号捕捉到之后的处理函数
    void     (*sa_handler)(int);
    // 不常用
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    // 临时阻塞信号集，在信号捕捉函数执行过程中，临时阻塞某些信号。
    sigset_t   sa_mask;
    // 使用哪一个信号处理对捕捉到的信号进行处理
    // 这个值可以是0，表示使用sa_handler,也可以是SA_SIGINFO表示使用sa_sigaction
    int        sa_flags;
    // 被废弃掉了
    void     (*sa_restorer)(void);
};
```

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void myalarm(int num) {
    printf("捕捉到了信号的编号是：%d\n", num);
    printf("xxxxxxx\n");
}

// 过3秒以后，每隔2秒钟定时一次
int main() {

    struct sigaction act;
    act.sa_flags = 0;
    act.sa_handler = myalarm;
    sigemptyset(&act.sa_mask);  // 清空临时阻塞信号集

    // 注册信号捕捉
    sigaction(SIGALRM, &act, NULL);

    struct itimerval new_value;

    // 设置间隔的时间
    new_value.it_interval.tv_sec = 2;
    new_value.it_interval.tv_usec = 0;

    // 设置延迟的时间,3秒之后开始第一次定时
    new_value.it_value.tv_sec = 3;
    new_value.it_value.tv_usec = 0;
```

```
    int ret = setitimer(ITIMER_REAL, &new_value, NULL); // 非阻塞的
    printf("定时器开始了...\n");

    if(ret == -1) {
        perror("setitimer");
        exit(0);
    }

    // getchar();
    while(1);

    return 0;
}
```

5.4 信号集函数

1. 用户通过键盘 Ctrl + C, 产生2号信号SIGINT (信号被创建)
2. 信号产生但是没有被处理 （未决）
    - 在内核中将所有的没有被处理的信号存储在一个集合中 （未决信号集）
    - SIGINT信号状态被存储在第二个标志位上
        - 这个标志位的值为0， 说明信号不是未决状态
        - 这个标志位的值为1， 说明信号处于未决状态
3. 这个未决状态的信号，需要被处理，处理之前需要和另一个信号集（阻塞信号集），进行比较

- 阻塞信号集默认不阻塞任何的信号
- 如果想要阻塞某些信号需要用户调用系统的API

4. 在处理的时候和阻塞信号集中的标志位进行查询，看是不是对该信号设置阻塞了
    - 如果没有阻塞，这个信号就被处理
    - 如果阻塞了，这个信号就继续处于未决状态，直到阻塞解除，这个信号就被处理

5.4.1 处理用户自定义信号集

```
#include <signal.h>
int sigemptyset(sigset_t *set);
//功能: 清空信号集中的数据,将信号集中的所有的标志位置为0
//set: 传出参数，需要操作的信号集
//返回值: 成功返回0， 失败返回-1
```

```
#include <signal.h>
int sigfillset(sigset_t *set);
//将信号集中的所有的标志位置为1
//set: 传出参数，需要操作的信号集
//返回值：成功返回0， 失败返回-1
```

```
#include <signal.h>
int sigaddset(sigset_t *set, int signum);
//设置信号集中的某一个信号对应的标志位为1，表示阻塞这个信号
//set: 传出参数，需要操作的信号集
//signum: 需要设置阻塞的那个信号
//返回值: 成功返回0， 失败返回-1
```

```c
#include <signal.h>
int sigdelset(sigset_t *set, int signum);
//设置信号集中的某一个信号对应的标志位为0，表示不阻塞这个信号
//set: 传出参数，需要操作的信号集
//signum: 需要设置不阻塞的那个信号
//返回值: 成功返回0， 失败返回-1
```

```c
#include <signal.h>
int sigismember(const sigset_t *set, int signum);
//判断某个信号是否阻塞
//set: 需要操作的信号集
//signum: 需要判断的那个信号
//返回值: 1: signum被阻塞, 0: signum不阻塞, -1: 失败
```

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

int main() {

    // 创建一个信号集
    sigset_t set;

    // 清空信号集的内容
    sigemptyset(&set);

    // 判断 SIGINT 是否在信号集 set 里
    int ret = sigismember(&set, SIGINT);
    if(ret == 0) {
        printf("SIGINT 不阻塞\n");
    } else if(ret == 1) {
        printf("SIGINT 阻塞\n");
    }

    // 添加几个信号到信号集中
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);

    // 判断SIGINT是否在信号集中
    ret = sigismember(&set, SIGINT);
    if(ret == 0) {
        printf("SIGINT 不阻塞\n");
    } else if(ret == 1) {
        printf("SIGINT 阻塞\n");
    }

    // 判断SIGQUIT是否在信号集中
    ret = sigismember(&set, SIGQUIT);
    if(ret == 0) {
        printf("SIGQUIT 不阻塞\n");
    } else if(ret == 1) {
        printf("SIGQUIT 阻塞\n");
    }

    // 从信号集中删除一个信号
    sigdelset(&set, SIGQUIT);
```

```
    // 判断SIGQUIT是否在信号集中
    ret = sigismember(&set, SIGQUIT);
    if(ret == 0) {
        printf("SIGQUIT 不阻塞\n");
    } else if(ret == 1) {
        printf("SIGQUIT 阻塞\n");
    }

    return 0;
}
```

5.4.2 sigprocmask、sigpending

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
//将自定义信号集中的数据设置到内核中
//how: 如何对内核阻塞信号集进行处理
    //SIG_BLOCK: 将用户设置的阻塞信号集添加到内核中，内核中原来的数据不变
    //SIG_UNBLOCK: 根据用户设置的数据，对内核中的数据进行解除阻塞
    //SIG_SETMASK:覆盖内核中原来的值
//set: 已经初始化好的用户自定义的信号集
//oldset: 保存设置之前的内核中的阻塞信号集的状态，可以是 NULL
```

```
#include <signal.h>
int sigpending(sigset_t *set);
//功能: 获取内核中的未决信号集
//set: 传出参数，保存的是内核中的未决信号集中的信息。
```

```
// 编写一个程序，把所有的常规信号（1-31）的未决状态打印到屏幕
// 设置某些信号是阻塞的，通过键盘产生这些信号

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    // 设置2、3号信号阻塞
    sigset_t set;
    sigemptyset(&set);
    // 将2号和3号信号添加到信号集中
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);

    // 修改内核中的阻塞信号集
    sigprocmask(SIG_BLOCK, &set, NULL);

    int num = 0;
    while(1) {
        num++;
        // 获取当前的未决信号集的数据
        sigset_t pendingset;
        sigemptyset(&pendingset);
```

```
        sigpending(&pendingset);

        // 遍历前32位
        for(int i = 1; i <= 31; i++) {
            if(sigismember(&pendingset, i) == 1) {
                printf("1");
            }else if(sigismember(&pendingset, i) == 0) {
                printf("0");
            }else {
                perror("sigismember");
                exit(0);
            }
        }

        printf("\n");
        sleep(1);
        if(num == 10) {
            // 解除阻塞
            sigprocmask(SIG_UNBLOCK, &set, NULL);
        }
    }
    return 0;
}
```

5.5 SIGCHLD信号

SGICHLD信号产生的条件

- 子进程终止时
- 子进程接收到 SIGSTOP 信号停止时
- 子进程处在停止态，接受到 SIGCONT 后唤醒时

以上三种条件会给父进程发送 SIGCHLD 信号，父进程默认会忽略该信号。

5.5.1 使用 SIGCHLD 解决僵尸进程

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>
#include <sys/wait.h>

void myFun(int num) {
    printf("捕捉到的信号 : %d\n", num);
    // 回收子进程PCB的资源
    // 没有while(1) 只会回收一次子进程
    while(1) {
       int ret = waitpid(-1, NULL, WNOHANG);
       if(ret > 0) {
           printf("child die , pid = %d\n", ret);
       } else if(ret == 0) {
           // 说明还有子进程
           break;
       } else if(ret == -1) {
           // 没有子进程
           break;
```

```
        }
    }
}

int main() {

    // 提前设置好阻塞信号集，阻塞SIGCHLD，因为有可能子进程很快结束，父进程还没有注册完信号捕捉
    // 如果不进行阻塞，小概率会发生段错误
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigprocmask(SIG_BLOCK, &set, NULL);

    // 创建一些子进程
    pid_t pid;
    for(int i = 0; i < 20; i++) {
        pid = fork();
        if(pid == 0) {
            break;
        }
    }

    if(pid > 0) {
        // 父进程

        // 捕捉子进程死亡时发送的SIGCHLD信号
        struct sigaction act;
        act.sa_flags = 0;
        act.sa_handler = myFun;
        sigemptyset(&act.sa_mask);
        sigaction(SIGCHLD, &act, NULL);

        // 注册完信号捕捉以后，解除阻塞-------注意这里
        sigprocmask(SIG_UNBLOCK, &set, NULL);

        while(1) {
            printf("parent process pid : %d\n", getpid());
            sleep(2);
        }
    } else if( pid == 0) {
        // 子进程
        printf("child process pid : %d\n", getpid());
    }

    return 0;
}
```

6、共享内存

共享内存运行两个或多个进程共享物理内存的同一块区域（通常称为段）。由于一个共享内存会称为一个进程用户空间的一部分，因此这种IPC机制无需内核介入。

共享内存使用步骤：

1. 调用 shmget() 创建一个新共享内存段或取得一个既有共享内存段的标识符。这个调用将返回后续调用中需要用到的共享内存标识符。
2. 使用 shmat() 来附上共享内存段，即使该段成为调用进程的虚拟内存的一部分。

3. 此刻在程序中可以像对待其它可用内存那样对待这个共享内存段。为引用这块共享内存，程序需要使用由 shmat() 调用返回的 addr 值，它是一个指向进程的虚拟地址空间中该共享内存段的起点的指针。
4. 调用 shmdt() 来分离共享内存段。在这个调用后，进程就无法再引用这块共享内存。这一步可选，并且在进程终止时会自动完成这一步。
5. 调用 shmctl() 删除共享内存段。只有当当前所有附加内存段的进程都与之分离后内存段才会销毁。只有一个进程需要执行这一步。

6.1 共享内存相关函数

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
//创建一个新的共享内存段，或者获取一个既有的共享内存段的标识，新创建的内存段中的数据都会被初始化为0
//key: key_t类型是一个整形，通过这个找到或者创建一个共享内存，一般使用16进制表示，非0值
//size: 共享内存的大小，最终会为分页大小的整数倍
//shmflg: 需要与IPC对象权限进行|运算
    //IPC_CREAT: 内核中不存在和key相等的共享内存，则创建共享内存
                //存在则返回此共享内存的标识符
    //IPC_EXCL: 和IPC_CREAT一起使用，即IPC_CREAT|IPC_EXCL
        //如果内核中不存在键值 与key相等的共享内存，则新建一个共享内存
        //如果存在这样的共享内存则报错
//返回值: 成功返回共享内存的标识符，失败返回-1，设置errno
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
//和当前的进程进行关联
//shmid: 共享内存的标识（ID）,由shmget返回值获取
//shmaddr: 申请的共享内存的起始地址，指定NULL，内核指定
//shmflg: 对共享内存的操作，SHM_RDONLY为只读模式，其他为读写模式
//返回值: 成功返回共享内存的首地址。  失败-1
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
//解除当前进程和共享内存的关联
//shmaddr: 共享内存的首地址
//返回值: 成功 0，  失败 -1
```

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
//对共享内存进行操作
//shmid: 共享内存的ID
//cmd: 要做的操作
    //IPC_STAT: 得到共享内存的状态，把共享内存的shmid_ds结构复制到buf中
    //IPC_SET: 改变共享内存的状态，把buf中的内容复制到共享内存的shmid_ds结构内
    //IPC_PMID: 删除这片共享内存
//buf: 共享内存管理结构体，用来设置或者获取的共享内存的属性信息
```

```c
#include <sys/ipc.h>
#include <sys/shm.h>
key_t ftok(const char *pathname, int proj_id);
//根据指定的路径名，和int值，生成一个共享内存的key
//proj_id: 系统调用只会使用其中的1个字节,范围: 0-255  一般指定一个字符 'a'
```

6.2 例子：共享内存操作

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {

    // 1.创建一个共享内存
    int shmid = shmget(100, 4096, IPC_CREAT|0664);
    printf("shmid : %d\n", shmid);

    // 2.和当前进程进行关联
    void * ptr = shmat(shmid, NULL, 0);

    char * str = "helloworld";

    // 3.写数据
    memcpy(ptr, str, strlen(str) + 1);
    //这里如果不暂停，会导致数据还没读就直接删除了共享内存
    printf("按任意键继续\n");
    getchar();

    // 4.解除关联
    shmdt(ptr);

    // 5.删除共享内存
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {

    // 1.获取一个共享内存
    int shmid = shmget(100, 0, IPC_CREAT);
    printf("shmid : %d\n", shmid);

    // 2.和当前进程进行关联
    void * ptr = shmat(shmid, NULL, 0);

    // 3.读数据
    printf("%s\n", (char *)ptr);
```

```
    printf("按任意键继续\n");
    getchar();

    // 4.解除关联
    shmdt(ptr);

    // 5.删除共享内存
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

6.3 共享内存的注意点

共享内存和内存映射的区别

1. 共享内存可以直接创建，内存映射需要磁盘文件（匿名映射除外）。
2. 共享内存效果更高。
3. 对于共享内存，所有的进程操作的是同一块共享内存；对于内存映射，每个进程在自己的虚拟地址空间中有一个独立的内存。
4. 数据安全
    1. 进程突然退出：共享内存还存在，内存映射区消失。
    2. 运行进程的电脑死机，宕机了：数据存在在共享内存中，没有了；内存映射区的数据 ，由于磁盘文件中的数据还在，所以内存映射区的数据还存在。
5. 生命周期
    1. 共享内存：进程退出，共享内存还在，标记删除（所有的关联的进程数为0才会删除共享内存），或者关机。如果一个进程退出，会自动和共享内存进行取消关联。
    2. 内存映射区：进程退出，内存映射区销毁

操作系统如何知道一块共享内存被多少个进程关联：共享内存维护了一个结构体struct shmid_ds 这个结构体中有一个成员 shm_nattch，shm_nattach 记录了关联的进程个数。

可不可以对共享内存进行多次删除 shmctl：可以，因为 shmctl 标记删除共享内存，不是直接删除，当和共享内存关联的进程数为0的时候，就真正被删除。如果一个进程和共享内存取消关联，那么这个进程就不能继续操作这个共享内存。也不能进行关联。

7、守护进程

守护进程是Linux中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以 d 结尾的名字。

特征：

- 生命周期很长，守护进程会在系统启动的时候被创建并一直运行直至系统被关闭。
- 它在后台运行并且不拥有控制终端。没有控制终端确保了内核永远不会为守护进程自动生成任何控制信号以及终端相关的信号。

守护进程创建步骤：

1. 执行一个 fork()，之后父进程退出，子进程继续执行
2. 子进程调用 setstd() 开启一个新会话
3. 清除进程的umask确保当守护进程创建文件和目录时所拥有的权限
4. 修改进程的当前工作目录，通常改为根目录

5. 关闭守护进程从父进程继承来的所有打开的文件描述符
6. 关闭了文件描述符0、1、2后，守护进程通常会打开 /dec/null 并使用 dup2() 使所有这些描述符指向这个设备。
7. 核心业务逻辑

7.1 进程组

- 进程组和会话在进程之间形成了一种两级层次关系：**进程组是一组相关进程的集合，会话是一组相关进程组的集合。**
- 进行组由一个或多个共享同一进程组标识符（PGID）的进程组成。一个进程组拥有一个进程组的首进程，该首进程是创建该组的进程，其进程 ID 为该进程组的 ID，新进程会继承其父进程所属的进程组 ID。
- 进程组拥有一个生命周期，其开始时间为首进程创建组的时刻，结束时间为最后一个成员进程退出组的时刻。一个进程可能会因为终止而退出进程组，也可能会因为加入了另外一个进程组而退出进程组。进程组首进程无需是最后一个离开进程组的成员。

7.2 会话

- 会话是一组进程组的集合。会话首进程是创建该新会话的进程，其进程 ID 会成为会话 ID。新进程会继承其父进程的会话 ID。
- 一个会话中的所有进程共享单个控制终端。控制终端会在会话首进程首次打开一个终端设备时被建立。一个终端最多可能会成为一个会话的控制终端。
- 在任一时刻，会话中的其中一个进程组会成为终端的前台进程组，其他进程组会成为后台进程组。只有前台进程组中的进程才能从控制终端中读取输入。当用户在控制终端中输入终端字符生成信号后，该信号会被发送到前台进程组中的所有成员。
- 当控制终端的连接建立起来之后，会话首进程会成为该终端的控制进程。

7.3 例子

```c
/*
    写一个守护进程，每隔2s获取一下系统时间，将这个时间写入到磁盘文件中。
*/

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/time.h>
#include <signal.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

void work(int num) {
    // 捕捉到信号之后，获取系统时间，写入磁盘文件
    time_t tm = time(NULL);
    struct tm * loc = localtime(&tm);
    // char buf[1024];

    // sprintf(buf, "%d-%d-%d %d:%d:%d\n",loc->tm_year,loc->tm_mon
    // ,loc->tm_mday, loc->tm_hour, loc->tm_min, loc->tm_sec);

    // printf("%s\n", buf);
```

```
    char * str = asctime(loc);
    int fd = open("time.txt", O_RDWR | O_CREAT | O_APPEND, 0664);
    write(fd ,str, strlen(str));
    close(fd);
}

int main() {

    // 1.创建子进程，退出父进程
    pid_t pid = fork();

    if(pid > 0) {
        exit(0);
    }

    // 2.将子进程重新创建一个会话
    setsid();

    // 3.设置掩码
    umask(022);

    // 4.更改工作目录
    chdir("/");

    // 5. 关闭、重定向文件描述符
    int fd = open("/dev/null", O_RDWR);
    dup2(fd, STDIN_FILENO);
    dup2(fd, STDOUT_FILENO);
    dup2(fd, STDERR_FILENO);

    // 6.业务逻辑

    // 捕捉定时信号
    struct sigaction act;
    act.sa_flags = 0;
    act.sa_handler = work;
    sigemptyset(&act.sa_mask);
    sigaction(SIGALRM, &act, NULL);

    struct itimerval val;
    val.it_value.tv_sec = 2;
    val.it_value.tv_usec = 0;
    val.it_interval.tv_sec = 2;
    val.it_interval.tv_usec = 0;

    // 创建定时器
    setitimer(ITIMER_REAL, &val, NULL);

    // 不让进程结束
    while(1) {
        sleep(10);
    }

    return 0;
}
```

8、线程

进程是资源分配的最小单位，线程是 CPU 调度的最小单位。

查看指定进程的线程号：ps -Lf pid

Linux中使用 `pthread.h` 后，编译要加上 `-pthread`，因为是非标准库。

8.1 线程创建与终止

8.1.1 pthread_create

```
#include <pthread.h>
int pthread_create(pthread *thread, const pthread_attr_t *attr, void *
(*start_routine) (void *), void *arg);
//创建一个子线程
//thread: 传出参数，线程成功创建后，线程ID会存储在thread中
//attr: 设置线程的属性，一般使用默认值NULL
//start_routine: 函数指针，这个函数是子线程需要处理的逻辑代码
//arg: 给函数指针传参
//成功返回0，失败返回错误号，错误信息通过strerror获得
```

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
//注意这里为 void* 类型函数
void *sayhello(void *arg)
{
    printf("子线程%d: say hello\n", *(int*)arg);
}
int main()
{
    pthread_t tid;
    int num=1;
    int ret = pthread_create(&tid, NULL, sayhello, (void*)&num);
    if(ret != 0){
        char* errstr = strerror(ret);
        printf("error: %s\n",errstr);
    }
    printf("主线程!!!\n");
    //这里还没有使用join
    sleep(1);
    return 0;
}
```

8.1.2 pthread_exit、pthread_self、pthread_equal

```c
#include <pthread.h>
void pthread_exit(void *retval);
//终止一个线程，在哪个线程中调用，就表示终止哪个进程
//retval: 需要传递一个指针，作为一个返回值，可以在pthread_join()中

pthread_t pthread_self(void);
//获取当前线程的线程ID

int pthread_equal(pthread_t t1, pthread_t t2);
//判断两个线程ID是否相等
```

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
void *sayhello(void *arg)
{
    printf("子线程ID: %ld\n", pthread_self());
    pthread_exit(NULL);
}
int main()
{
    pthread_t tid;

    int ret = pthread_create(&tid, NULL, sayhello, NULL);
    if(ret != 0){
        char* errstr = strerror(ret);
        printf("error: %s\n",errstr);
    }
    printf("tid: %ld\n", tid);
    printf("主线程ID: %ld\n", pthread_self());
    pthread_exit(NULL);
    return 0;
}
```

8.2 pthread_join

```c
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
//和一个已经终止的线程连接，回收子线程的资源。阻塞函数，调用一次只能回收一个子线程
//thread: 需要回收的子线程的ID
//retval: 接收子线程退出时的返回值, 一般用NULL, 注意变量的生存周期
```

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>

int num=10;
void *sayhello(void *arg)
{
    printf("子线程ID: %ld\n", pthread_self());
    printf("子线程结束了\n");
    //注意这里是全局变量
    pthread_exit((void*)&num);
}
```

```c
int main()
{
    pthread_t tid;

    int ret = pthread_create(&tid, NULL, sayhello, NULL);
    if(ret != 0){
        char* errstr = strerror(ret);
        printf("error: %s\n",errstr);
    }
    printf("主线程ID: %ld\n", pthread_self());

    int *thread_re=NULL;
    pthread_join(tid, (void**)&thread_re);
    printf("主线程接收到子线程的返回值: %d\n", *thread_re);
    printf("主线程之前调用了pthread_join，现在主线程结束\n");
    return 0;
}
```

8.3 线程分离和取消

8.3.1 pthreaed_detach

分离线程，有时候主线程终止了，被分离的线程可以在后台运行，被分离的线程终止后时会自动释放资源返回给操作系统。

注意：一个线程已经被 detach 了，就不能再次 detach；一个线程被 join 了，也不能 detach

```c
#include <pthread.h>
int pthread_detach(pthread_t thread);
//成功返回0，失败返回错误号
```

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>

void *sayhello(void *arg)
{
    printf("子线程ID: %ld\n", pthread_self());
    printf("子线程结束了\n");
    pthread_exit(NULL);
}
int main()
{
    pthread_t tid;

    int ret = pthread_create(&tid, NULL, sayhello, NULL);
    if(ret != 0){
        char* errstr = strerror(ret);
        printf("error: %s\n",errstr);
    }
    printf("主线程ID: %ld\n", pthread_self());

    pthread_detach(tid);
    printf("主线程结束\n");
    return 0;
}
```

8.3.2 pthread_cancel

取消某个线程的运行，但不是立马终止，而是当子线程执行到一个取消点，线程才会终止。

取消点：系统规定好的一些系统调用。

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>

void *sayhello(void *arg)
{
    printf("子线程ID: %ld\n", pthread_self());
    for(int i=0;i<5;i++){
        printf("子线程: %d\n",i);
    }
    printf("子线程结束了\n");
    pthread_exit(NULL);
}
int main()
{
    pthread_t tid;

    int ret = pthread_create(&tid, NULL, sayhello, NULL);
    if(ret != 0){
        char* errstr = strerror(ret);
        printf("error: %s\n",errstr);
    }
    pthread_cancel(tid);

    printf("主线程ID: %ld\n", pthread_self());
    printf("主线程结束\n");
    return 0;
}
```

8.4 线程属性

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
//初始化线程属性

int pthread_attr_destory(pthread_attr_t *attr);
//释放线程属性资源

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
//获取线程分离的状态属性

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
//设置线程分离的状态属性
//detachstate: PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE
```

```
#include <pthread.h>
```

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


void *sayhello(void *arg)
{
    printf("子线程ID: %ld\n", pthread_self());
    printf("子线程结束了\n");
    pthread_exit(NULL);
}
int main()
{
    //创建属性变量
    pthread_attr_t attr;
    //初始化属性
    pthread_attr_init(&attr);
    //设置属性
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

    pthread_t tid;
    //这里第二个参数用了attr
    int ret = pthread_create(&tid, &attr, sayhello, NULL);
    if(ret != 0){
        char* errstr = strerror(ret);
        printf("error: %s\n",errstr);
    }
    printf("主线程ID: %ld\n", pthread_self());
    printf("主线程结束\n");
    //释放属性资源
    pthread_attr_destroy(&attr);
    return 0;
}
```

8.5 互斥锁

互斥锁有两种状态：锁定和未锁定。任何时候，至多只有一个线程可以锁定该互斥锁。试图对已经锁定的某一互斥锁再次加锁，将可能阻塞线程或者报错失败，具体取决于加锁时使用的方法。

一旦线程锁定互斥量，随即成为该互斥量的所有者，只有所有者才能给互斥量解锁。

如果多个线程试图访问执行同一块代码，这时只有持有锁的线程能够执行，其它线程将被阻塞。

```c
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr);
//mutex: 需要初始化的互斥量
//attr: 互斥量相关属性，一般用NULL
//返回值: 成功返回0，失败返回一个错误号

int pthread_mutex_destroy(pthread_mutex_t *mutex);
//释放互斥量的资源
//返回值: 成功返回0，失败返回一个错误号

int phtread_mutex_lock(pthread_mutex_t *mutex);
//加锁，阻塞
//返回值: 成功返回0，失败返回一个错误号

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
//尝试加锁，加锁失败直接返回，不会阻塞

int pthread_mutex_unlock(pthread_mutex_t *mutex);
//解锁
//返回值：成功返回0，失败返回一个错误号
```

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>

int tickets = 1000;
pthread_mutex_t mutex;

void *sellticket(void *arg)
{

    while(1){
        pthread_mutex_lock(&mutex);
        if(tickets > 0){
            printf("子线程%d: 正在卖第%d张票\n", *(int*)arg, tickets);
            tickets--;
        }
        else{
            pthread_mutex_unlock(&mutex);
            break;
        }
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}
int main()
{
    pthread_mutex_init(&mutex,NULL);

    pthread_t tid1, tid2, tid3;
    int num1 = 1,num2 = 2,num3 = 3;
    pthread_create(&tid1, NULL, sellticket, (void*)&num1);
    pthread_create(&tid2, NULL, sellticket, (void*)&num2);
    pthread_create(&tid3, NULL, sellticket, (void*)&num3);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    pthread_mutex_destroy(&mutex);
    printf("主线程结束\n");

    return 0;
}
```

8.6 死锁

必要条件:

1. 互斥：一个资源只能被一个线程使用，其它线程请求该资源时，只能等待，直到资源使用完毕后释放。
2. 请求和保持：线程已经占有至少一个资源，但又请求新资源，而这个新资源被其它线程占用，自己占用的资源却保持不放。
3. 不可抢占：任何一个资源在没被该线程释放之前，任何其它线程都无法对它剥夺占用。
4. 循环等待：当发生死锁时，所等待的线程必定会形成一个环路，造成永久阻塞。

解决死锁的方法：

- 一个线程已经获取一个锁时，就不要获取第二个。
- 持有锁时，避免调用用户提供的代码。
- 按固定顺序获取锁。
- 使用层次锁，如果一个锁被底层持有，就不允许再上锁。

死锁的场景

- 忘记释放锁
- 重复加锁
- 多线程多锁，抢占资源

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

// 创建2个互斥量
pthread_mutex_t mutex1, mutex2;

void * workA(void * arg) {

    pthread_mutex_lock(&mutex1);
    sleep(1);
    pthread_mutex_lock(&mutex2);

    printf("workA....\n");

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}


void * workB(void * arg) {
    pthread_mutex_lock(&mutex2);
    sleep(1);
    pthread_mutex_lock(&mutex1);

    printf("workB....\n");

    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);

    return NULL;
}

int main() {

    // 初始化互斥量
```

```
    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);

    // 创建2个子线程
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, workA, NULL);
    pthread_create(&tid2, NULL, workB, NULL);

    // 回收子线程资源
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    // 释放互斥量资源
    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}
```

8.7 读写锁

当有一个线程已经持有锁时，互斥锁将所有试图进入临界区的线程都阻塞住。但是考虑一种情形，当前持有互斥锁的线程只是要读共享资源，而同时有其它几个线程也想读取这个共享资源，但是由于互斥锁的排他性，所有其它线程都无法获取锁，也就无法访问共享资源，但是实际上多个线程同时读访问共享资源并不会导致问题。

读写锁就是为了解决这种问题。读写锁允许多个线程读数据，但只允许一个线程写数据。

读写锁的特点：

- 如果有其它线程读数据，允许执行读操作，但不允许写操作。
- 如果有其它线程写数据，则其它线程都不允许读、写操作。
- 写是独占的，写的优先级高。

```
int pthread_rwlock_init(pthread_rwlock_t *restruct rwlock, const
pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destory(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

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

// 创建一个共享数据
int num = 1;

pthread_rwlock_t rwlock;

void * writeNum(void * arg) {
    while(1) {
        pthread_rwlock_wrlock(&rwlock);
        num++;
```

```
        printf("++write, tid : %ld, num : %d\n", pthread_self(), num);
        pthread_rwlock_unlock(&rwlock);
        usleep(100);
    }
    return NULL;
}

void * readNum(void * arg) {
    while(1) {
        pthread_rwlock_rdlock(&rwlock);
        printf("===read, tid : %ld, num : %d\n", pthread_self(), num);
        pthread_rwlock_unlock(&rwlock);
        usleep(100);
    }
    return NULL;
}

int main() {
    pthread_rwlock_init(&rwlock, NULL);

    // 创建3个写线程，5个读线程
    pthread_t wtids[3], rtids[5];
    for(int i = 0; i < 3; i++) {
        pthread_create(&wtids[i], NULL, writeNum, NULL);
    }

    for(int i = 0; i < 5; i++) {
        pthread_create(&rtids[i], NULL, readNum, NULL);
    }

    // 设置线程分离
    for(int i = 0; i < 3; i++) {
       pthread_detach(wtids[i]);
    }

    for(int i = 0; i < 5; i++) {
         pthread_detach(rtids[i]);
    }
    pthread_exit(NULL);
    pthread_rwlock_destroy(&rwlock);

    return 0;
}
```

8.8 条件变量

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。

- 如果条件为假，线程通常会基于条件变量阻塞，并以原子方式释放等待条件变化的互斥锁。
- 如果另一个线程更改了条件，该线程可能会向相关的条件变量发出信号，从而唤醒一个或多个等待的线程，并再次获取锁。

通常条件变量和互斥锁同时使用，对条件的测试是在互斥锁的保护下进行的。

```
int pthread_cond_init(pthread_cond_t *restrict cond,const pthread_condattr_t
*restrict attr);
```

```
//成功返回0
int pthread_cond_destroy(pthread_cond_t *cond);
//成功返回0

int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict
mutex);
//等待，阻塞函数
//成功返回0
//wait虽然是阻塞的，但在其阻塞时会对互斥锁解锁；当不阻塞，继续向下执行，会重新加锁

int pthread_cond_timewait(pthread_cond_t *restrict cond,pthread_mutex_t
*restrict mutex,const struct timespec *restrict abstime);
//等待多长时间，调用了这个函数，线程会阻塞，直到指定的时间结束
//成功返回0

int pthread_cond_signal(pthread_cond_t *cond);
//唤醒一个或多个等待的线程
//成功返回0

int pthread_cond_broadcast(pthread_cond_t *cond);
//唤醒所有等待的线程
//成功返回0
```

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

// 创建一个互斥量
pthread_mutex_t mutex;
// 创建条件变量
pthread_cond_t cond;

struct Node{
    int num;
    struct Node *next;
};

// 头结点
struct Node * head = NULL;

void * producer(void * arg) {

    // 不断的创建新的节点，添加到链表中
    while(1) {
        pthread_mutex_lock(&mutex);
        struct Node * newNode = (struct Node *)malloc(sizeof(struct Node));
        newNode->next = head;
        head = newNode;
        newNode->num = rand() % 1000;
        printf("add node, num : %d, tid : %ld\n", newNode->num, pthread_self());

        // 只要生产了一个，就通知消费者消费
        pthread_cond_signal(&cond);

        pthread_mutex_unlock(&mutex);
        usleep(100);
```

```
    }

    return NULL;
}

void * customer(void * arg) {

    while(1) {
        pthread_mutex_lock(&mutex);
        // 保存头结点的指针
        struct Node * tmp = head;
        // 判断是否有数据
        if(head != NULL) {
            // 有数据
            head = head->next;
            printf("del node, num : %d, tid : %ld\n", tmp->num, pthread_self());
            free(tmp);
            pthread_mutex_unlock(&mutex);
            usleep(100);
        } else {
            // 没有数据，需要等待
            // 当这个函数调用阻塞的时候，会对互斥锁进行解锁，当不阻塞的，继续向下执行，会重新加锁。
            pthread_cond_wait(&cond, &mutex);
            pthread_mutex_unlock(&mutex);
        }
    }
    return  NULL;
}

int main() {

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    // 创建5个生产者线程，和5个消费者线程
    pthread_t ptids[5], ctids[5];

    for(int i = 0; i < 5; i++) {
        pthread_create(&ptids[i], NULL, producer, NULL);
        pthread_create(&ctids[i], NULL, customer, NULL);
    }

    for(int i = 0; i < 5; i++) {
        pthread_detach(ptids[i]);
        pthread_detach(ctids[i]);
    }

    while(1) {
        sleep(10);
    }

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

    pthread_exit(NULL);

    return 0;
```

```
}
```

8.9 信号量

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
//sem: 信号量变量的地址
//pshared: 0用在线程，非0用在进程
//value: 信号量中的值
//成功返回0；失败返回 -1，并设置errno

int sem_destroy(sem_t *sem);
//成功返回0；失败返回 -1，并设置errno

int sem_wait(sem_t *sem);
//锁上一个信号量，调用一次对信号量的值-1，如果为0，就阻塞
//成功返回0；失败返回 -1，并设置errno

int sem_trywait(sem_t *sem);
//成功返回0；失败返回 -1，并设置errno

int sem_timewait(sem_t *sem, const struct timespec *abs_timeout);
//成功返回0；失败返回 -1，并设置errno

int sem_post(sem_t *sem);
//对信号量解锁，调用一次对信号量的值+1
//成功返回0；失败返回 -1，并设置errno

int sem_getvalue(sem_t *sem,int *sval);
//成功返回0；失败返回 -1，并设置errno
```

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>

// 创建一个互斥量
pthread_mutex_t mutex;
// 创建两个信号量
sem_t psem;
sem_t csem;

struct Node{
    int num;
    struct Node *next;
};

// 头结点
struct Node * head = NULL;

void * producer(void * arg) {

    // 不断的创建新的节点，添加到链表中
```

```
    while(1) {
        sem_wait(&psem);
        pthread_mutex_lock(&mutex);
        struct Node * newNode = (struct Node *)malloc(sizeof(struct Node));
        newNode->next = head;
        head = newNode;
        newNode->num = rand() % 1000;
        printf("add node, num : %d, tid : %ld\n", newNode->num, pthread_self());
        pthread_mutex_unlock(&mutex);
        sem_post(&csem);
    }

    return NULL;
}

void * customer(void * arg) {

    while(1) {
        sem_wait(&csem);
        pthread_mutex_lock(&mutex);
        // 保存头结点的指针
        struct Node * tmp = head;
        head = head->next;
        printf("del node, num : %d, tid : %ld\n", tmp->num, pthread_self());
        free(tmp);
        pthread_mutex_unlock(&mutex);
        sem_post(&psem);

    }
    return  NULL;
}

int main() {

    pthread_mutex_init(&mutex, NULL);
    sem_init(&psem, 0, 8);
    sem_init(&csem, 0, 0);

    // 创建5个生产者线程，和5个消费者线程
    pthread_t ptids[5], ctids[5];

    for(int i = 0; i < 5; i++) {
        pthread_create(&ptids[i], NULL, producer, NULL);
        pthread_create(&ctids[i], NULL, customer, NULL);
    }

    for(int i = 0; i < 5; i++) {
        pthread_detach(ptids[i]);
        pthread_detach(ctids[i]);
    }

    while(1) {
        sleep(10);
    }

    pthread_mutex_destroy(&mutex);

    pthread_exit(NULL);
```

```
    return 0;
}
```

9、网络编程

9.1 MAC地址、IP地址、端口

MAC地址长度为48位（6个字节），通常表示为 12 个 16 进制数

9.1.1 IP地址类别

IP地址类别有A~E类，其中D、E类为特殊地址。

类别	最大网络数	IP地址范围	单个网段最大主机数	私有IP地址范围
A	126(2^7-2)	1.0.0.0-126.255.255.255	16777214	10.0.0.0-10.255.255.255
B	16384(2^14)	128.0.0.1-191.255.255.254	65534	172.16.0.0-172.31.255.255
C	2097152(2^21)	192.0.0.1-223.255.255.254	254	192.168.0.0-192.168.255.255

A类IP地址

在A类IP地址的四段号码中，第一段号码为网络号码，剩下三段号码为本地计算机的号码。即A类IP地址由1字节的网络地址和3字节主机地址组成，网络地址的最高位必须为 0。

A类地址适用于具有大量主机（直接个人用户）而局域网络个数较少的大型网络。A类地址每个网络可容纳主机数达1600多万台。

A类IP地址的可使用范围为：从 1.0.0.1 到 126.255.255.254

A类IP地址的子网掩码为 255.0.0.0

B类IP地址

在B类IP地址的四段号码中，前两段号码为网络号码，剩下两段号码为本地计算机的号码。即B类IP地址由2字节的网络地址和2字节主机地址组成，网络地址的最高位必须为 10。

B类网络地址适用于中等规模的网络，有16384个网络，每个网络所能容纳的计算机为65534多台。

B类IP地址的子网掩码为 255.255.0.0

C类IP地址

在C类IP地址的四段号码中，前三段号码为网络号码，剩下一段号码为本地计算机的号码。即C类IP地址由3字节的网络地址和1字节主机地址组成，网络地址的最高位必须为 110。

C类网络地址适用于小规模的局域网，每个网络最多只能包含254台计算机。

C类IP地址的子网掩码为 255.255.255.0

D类IP地址

D类IP地址也被称为多播地址（组播地址），在以太网中，多播地址命名了一组应该在这个网络中应用接收到一个分组的站点。多播地址的最高位必须是 1110，范围从 224.0.0.0-239.255.255.255

9.1.2 端口

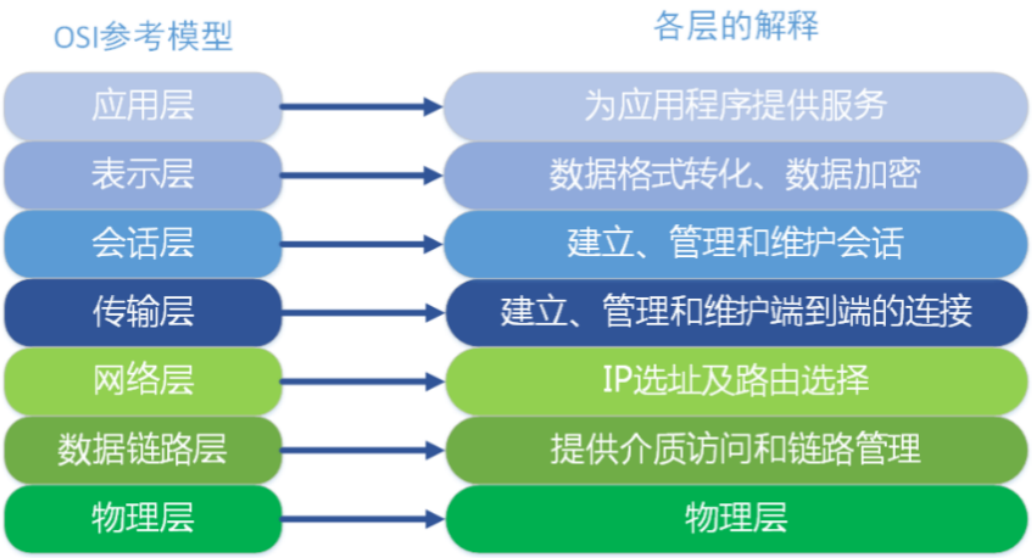
知名端口：也叫常用端口，范围从0~1023。

注册端口：范围从 1024~49151

动态端口/私有端口：范围从 49152~65535

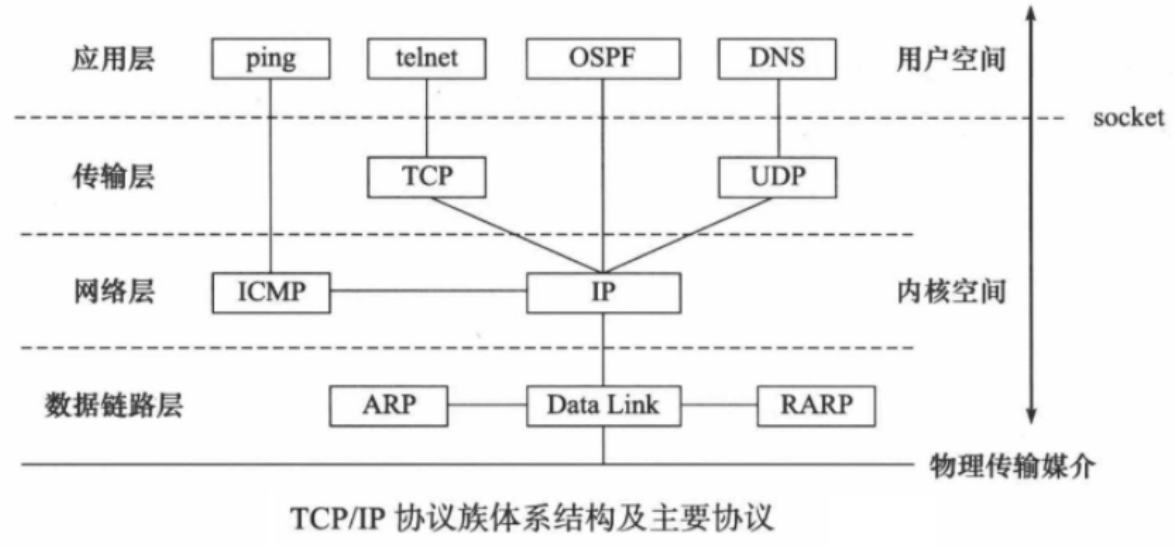
9.2 网络模型

9.2.1 OSI七层模型



1. 物理层：定义物理设备标准。主要作用是**传输比特流**，这一层的数据叫比特。
2. 数据链路层：建立逻辑连接、进行硬件地址寻址、差错校验等功能。**将比特组合成字节进而组合成帧，用MAC地址访问介质。**
3. 网络层：进行**逻辑地址寻址**，实现不同网络之间的路径选择。主要功能是利用数据链路层所提供的相邻节点间的无差错数据传输功能，通过路由选择和中继功能，实现两个系统之间的连接。
4. 传输层：定义传输数据的协议和端口号，以及流控和差错校验。主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常把这一层的数据叫做**段。**
5. 会话层：建立、管理、终止会话。通过端口号建立数据传输的通路。
6. 表示层：数据的表示、安全、压缩。加密解密，压缩解压缩等。
7. 应用层：网络服务与最终用户的一个接口

9.2.2 TCP/IP四层模型



TCP/IP 协议族体系结构及主要协议

9.3 协议

9.3.1 常见协议

应用层常见协议：FTP(文件传输协议)、HTTP(超文本传输)、NFS(网络文件系统)

传输层常见协议：TCP(传输控制协议)、UDP(用户数据包协议)

网络层常见协议：IP(因特网互联协议)、ICMP(因特网控制报文协议)、IGMP(因特网组管理协议)

网络接口层常见协议：ARP(地址解析协议)

9.3.2 UDP、TCP、IP头部结构

0								15 16	31
16位源端口号									16位目的端口号
32位序号									
32位确认号									
4位 头部长度	6位保留		URG	ACK	PSH	RST	SYN	FIN	16位窗口大小
16位校验和									16位紧急指针
选项，最多40字节									

TCP 头部结构

0			15 16	31
4位 版本号	4位头部 长度	8位服务类型 （TOS）	16位总长度（字节数）	
16位标识			3位 标志	13位片偏移
8位生存时间 （TTL）		8位协议	16位头部校验和	
32位源端IP地址				
32位目的端IP地址				
选项，最多40字节				

IPv4 头部结构

9.4 字节序转换函数

```
h  - host, 主机, 主机字节序
to - 转换成什么
n  - network 网络字节序
s  - short
l  - long
```

```
#include <arpa/inet.h>
//转换端口
uint16_t htons(uint16_t hostshort);
uint16_t ntohs(uint16_t netshort);

//转换IP
uint32_t htonl(uint32_t hostlong);
uint32_t ntohl(uint32_t netlong);
```

9.5 socket套接字

9.5.1 通用套接字

一般不用

```
#include <bits/socket.h>
//typedef unsigned short in sa_family_t;
struct sockaddr{
    sa_family_t sa_family;
    char sa_data[14];
};
//由于sa_data只有14个字节，因此该socket套接字无法存储IPv6地址
```

sa_family_t 称为地址族类型，地址族类型通常与协议族类型对应。

协议族	地址族	描述
PF_UNIX	AF_UNIX	UNIX本地协议族
PF_INET	AF_INET	TCP/IPv4协议族
PF_INET6	AF_INET6	TCP/IPv6协议族

```
#include <bits/socket.h>
//#define __ss_aligntype    unsigned long int
struct sockaddr_storage
{
    sa_family_t sa_family;
    char __ss_padding[128-sizeof(__ss_align)];
    __ss_aligntype __ss_align;
};
```

9.5.2 专用 socket 地址

```
#include <netinet/in.h>
//IPv4
struct in_addr{
    in_addr_t s_addr; //32位IPv4地址，无符号整数
};

struct sockaddr_in{
    sa_family_t sin_family; //AF_INET
    in_port_t sin_port;
    struct in_addr sin_addr;
```

```
    unsigned char sin_zero[8];
};

//IPv6
struct in6_addr{
    unit8_t s6_addr[16];
};

struct sockaddr_in6{
    sa_family_t sin6_family; //AF_INET6
    in_port_t sin6_port;
    uint32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t sin6_scope_id;
};

typedef unsigned short uint16_t;
typedef unsigned int   uint32_t;
typedef uint16_t in_port_t;
typedef uint32_t in_addr_t;
```

所有专用 socket 地址类型的变量在实际使用时都需要转化为通用 socket 地址类型 sockaddr（强制转换即可），因为所有 socket 编程接口使用的地址参数类型都是 sockaddr

9.6 IP地址转换函数

```
#include <arpa/inet.h>
typedef uint32_t in_addr_t;
in_addr_t inet_addr(const char* strptr);
//点分10进制IPv4地址转网络字节序

int inet_aton(const char* cp, struct in_addr* inp);
//IPv4字符串转网络字节序，结果存于inp指向的地址结构
//返回值：0表示成功，1表示非法

char* inet_ntoa(struct in_addr in);
//网络字节序转字符串，函数内部用一个静态变量存储结果，返回值指向该静态内存
//因此inet_ntoa不可重入
```

以下同时适用于IPv4和IPv6

```
#include <arpa/inet.h>
//p：点分10进制的IP字符串，n：表示网络字节序
int inet_pton(int af, const char* src, void* dst);
//IP字符串转网络字节序，存于dst指向的内存
//af：地址族：AF_INET，AF_INET6

const char* inet_ntop(int af, const void* src, char* dst, socklen_t cnt);
//将src网络字节序转成IP字符串，存于dst中
//size：dst的大小，数组的容量
//返回转换后的字符串
```

9.7 套接字函数

9.7.1 socket

```
#include <arpa/inet.h>
int socket(int family,int type,int protocol);
//family: 协议族: AF_INET, AF_INET6, AF_UNIX
//type: 套接字类型: SOCK_STREAM, SOCK_DGRAM
//protocol: 一般写0
//返回值: 成功返回文件描述符，失败返回-1
client=socket(AF_INET, SOCK_STREAM, 0) ;
```

9.7.2 bind

```
#include <arpa/inet.h>
int bind(int sockfd,const struct sodckaddr *myaddr,socklen_t addrlen);
//sockfd: 通过socket获得的文件描述符
//myaddr：需要绑定的socket地址，使用时会强制转换，因为创建套接字时使用的结构体是sockaddr_in
//addrlen: myaddr的大小
```

9.7.3 listen

内核为任何一个给定的监听套接字维护两个队列：

- **未完成连接队列**：每个这样的SYN分节对应其中一项：已由某个客户发出并到达服务器，而服务器正在等待完成相应的TCP三次握手过程。这些套接字处于 SYN_RCVD 状态。
- **已完成连接队列**：每个已完成TCP三次握手过程的客户对应其中一项。这些套接字处于ESTABLISHED状态。

```
#include <arpa/inet.h>
int listen(int sockfd,int backlog);
//sockfd: 通过socket获得的文件描述符
//backlog: 未连接和已连接的和的最大值
//返回值: 成功返回0，失败返回-1
```

9.7.4 accept

从 listen 监听队列的**已完成连接队列**中接受一个连接。

```
#include <arpa/inet.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
//cliaddr: 连接成功的客户端的信息
//addrlen: 第二个参数cliaddr对应的内存大小
//返回值: 成功返回用于通信的文件描述符，失败返回-1
```

9.7.5 connect

```
#include <arpa/inet.h>
int connect(int sockfd,const struct sockaddr *servaddr,socklen_t addrlen);
//servaddr: 客户端要连接的服务器的地址信息
//addrlen: servaddr的大小
//返回值: 成功返回0，失败返回-1
```

9.8 简单TCP客户端与服务端实现

9.8.1 客户端

```
#include <string.h>
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
int main()
{
    char str[256] = "hello";
    const int PORT = 11255;
    const char *IP = "127.0.0.1";
    int sockfd;
    int Len=0;
    char buff[256];

    struct sockaddr_in serverAddr;
    if((sockfd=socket(AF_INET,SOCK_STREAM,0)) == -1)
    {
        perror("socket\n");
        return -1;
    }

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr(IP);

    if(connect(sockfd,(struct sockaddr *)&serverAddr,sizeof(serverAddr)) == -1)
    {
        perror("connect\n");
        return -1;
    }

    int k = 10;
    while(k--){
        if(send(sockfd,str,256,0) == -1){
            perror("send\n");
            return -1;
        }
        Len=recv(sockfd,buff,256,0);
        if(Len == -1){
            perror("recv\n");
            return -1;
        }
        else if(Len == 0){
            printf("server closed\n");
            break;
        }
        printf("recvData: %s\n",buff);

    }
    close(sockfd);
    return 0;
}
```

9.8.2 服务端

```
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 11255
#define IP "127.0.0.1"
#define BACKLOG 5
#define MAXDATASIZE 512

int main()
{
    int sock,new_sock;
    struct sockaddr_in serverAddr;

    if((sock = socket(AF_INET,SOCK_STREAM,0)) == -1){
        perror("socket\n");
        return 0;
    }
    memset(&serverAddr,0,sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr(IP);

    if(bind(sock,(struct sockaddr *)&serverAddr,sizeof(serverAddr)) == -1){
        perror("bind\n");
        return 0;
    }

    if(listen(sock,BACKLOG) == -1){
        perror("listen\n");
        return 0;
    }

    struct sockaddr_in clientAddr;
    socklen_t len=sizeof(clientAddr);
    if((new_sock=accept(sock,(struct sockaddr *)&clientAddr,&len)) == -1)
    {
        perror("accept\n");
        return 0;
    }
    char cliIP[16];
    inet_ntop(AF_INET,&clientAddr.sin_addr,cliIP,16);
    unsigned short cliPort = ntohs(clientAddr.sin_port);
    printf("get connection with %s:%d\n",cliIP,cliPort);
    char recvData[MAXDATASIZE];

    while (1){
        memset(recvData,0,MAXDATASIZE);
        int recvLength = recv(new_sock,recvData,MAXDATASIZE,0);

        if(recvLength == 0){
            printf("client closed!\n");
            close(new_sock);
```

```
            break;
        }
        else if(recvLength > 0){
            printf("recvData: %s\n",recvData);
        }
        else{
            perror("recv");
            exit(0);
        }

        const char *str = "All Data has been recv!";
        if(send(new_sock,str,strlen(str),0) == -1){
            perror("send\n");
            close(new_sock);
            break;
        }
    }
    close(sock);
    return 0;
}
```

9.9 并发服务器实现

9.9.1 多进程实现并发

服务端

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <wait.h>
#include <errno.h>
void reChild(int arg)
{
    while(1){
        int ret = waitpid(-1,NULL,WNOHANG);
        if(ret == -1){
            //所有的子进程都被回收了
            break;
        }
        else if(ret == 0){
            //说明还有子进程活着，不需要回收
            break;
        }
        else if(ret > 0){
            printf("子进程 %d 被回收了\n",ret);
        }
    }
}
int main()
{
    //注册信号捕捉函数，回收子进程资源
    struct sigaction act;
```

```
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    act.sa_handler = reChild;
    sigaction(SIGCHLD,&act,NULL);

    //创建套接字
    int fd;
    struct sockaddr_in saddr;
    if((fd=socket(AF_INET,SOCK_STREAM,0)) == -1){
        perror("socket");
        exit(-1);
    }
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(6789);
    inet_pton(AF_INET, "127.0.0.1", &saddr.sin_addr);

    if(bind(fd,(struct sockaddr*)&saddr,sizeof(saddr)) == -1){
        perror("bind");
        exit(-1);
    }

    if(listen(fd,128) == -1){
        perror("listen");
        exit(-1);
    }

    //循环等待客户端连接
    while(1){
        int clifd;
        struct sockaddr_in cliaddr;
        socklen_t len = sizeof(cliaddr);

        //accpet: Interrupted system call
        clifd=accept(fd,(struct sockaddr*)&cliaddr,&len);
        if(clifd == -1){
            if(errno == EINTR){
                continue;
            }
            perror("accpet");
            exit(-1);
        }

        pid_t pid = fork();

        if(pid == 0){
            char cliIP[16];
            inet_ntop(AF_INET,&cliaddr.sin_addr,cliIP,sizeof(cliIP));
            unsigned short cliPort = ntohs(cliaddr.sin_port);
            printf("connect with %s:%d\n",cliIP,cliPort);

            char buff[1024] = {0};
            while(1){
                int r_len = recv(clifd,buff,sizeof(buff),0);
                if(r_len > 0){
                    printf("server recv from %d: %s\n",cliPort,buff);
                }
                else if(r_len == 0){
                    printf("client closed\n");
```

```
                    break;
                }
                else{
                    //recv: Connection reset by peer
                    perror("recv");
                }
                send(clifd,buff,strlen(buff)+1,0);
            }
            close(clifd);
            exit(0);
        }
    }
    close(fd);
    return 0;
}
```

客户端

```
#include <string.h>
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
int main()
{

    const int PORT = 6789;
    const char *IP = "127.0.0.1";
    int sockfd;

    struct sockaddr_in serverAddr;
    if((sockfd=socket(AF_INET,SOCK_STREAM,0)) == -1)
    {
        perror("socket");
        return -1;
    }

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr(IP);

    if(connect(sockfd,(struct sockaddr *)&serverAddr,sizeof(serverAddr)) == -1)
    {
        perror("connect");
        return -1;
    }
    char buff[1024];

    int Len=0;
    int i = 0;
    while(1){
        sprintf(buff,"data: %d\n",i++);
        if(send(sockfd,buff,strlen(buff)+1,0) == -1){
            perror("send");
            return -1;
        }
```

```
        Len=recv(sockfd,buff,sizeof(buff),0);
        if(Len == -1){
            perror("recv");
            return -1;
        }
        else if(Len == 0){
            printf("server closed\n");
            break;
        }
        else
            printf("recvData: %s\n",buff);
        sleep(1);
    }
    close(sockfd);
    return 0;
}
```

注意:

- `accpet: Interrupted system call`：因为使用了信号来回收子进程，当在终端上对正在运行的客户端执行 Ctrl+C，会产生一个中断信号，这时中断会被服务端程序捕捉到，在一个有效的连接到达前会传递给 accept 函数（The system call was interrupted by a signal that was caught before a valid connection arrived），accept 抛出错误 EINTR，导致程序再也无法进行连接。因此需要判断 accpet 的错误信息。
- `recv: Connection reset by peer`：当在终端上对正在运行的客户端执行 Ctrl+C，这时客户端程序结束，但客户端没有关闭 socket 连接，这时服务端再进行读写操作会抛出该异常。

9.9.2 多线程实现并发

服务端

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>

struct sockInfo{
    int fd;                     //客户端文件描述符
    struct sockaddr_in addr;    //客户端信息
    pthread_t tid;              //子线程tid
};

//限定客户端做大连接数
struct sockInfo sockInfos[128];

void* thread_child(void* arg)
{
    struct sockInfo *info = (struct sockInfo*)arg;
    char cliIP[16];
    inet_ntop(AF_INET,&info->addr.sin_addr.s_addr,cliIP,sizeof(cliIP));
    unsigned short cliPort = ntohs(info->addr.sin_port);
    printf("connect with %s:%d\n",cliIP,cliPort);
```

```
    char buff[1024] = {0};
    while(1){
        int len = read(info->fd, &buff, sizeof(buff));

        if(len == -1) {
            perror("read");
            exit(-1);
        }else if(len > 0) {
            printf("recv client : %s\n", buff);
        } else if(len == 0) {
            printf("client closed....\n");
            break;
        }
        write(info->fd, buff, strlen(buff) + 1);

    }

    close(info->fd);
    return NULL;
}
int main()
{
    //创建套接字
    int fd = socket(AF_INET,SOCK_STREAM,0);
    if(fd == -1){
        perror("socket");
        exit(-1);
    }
    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(9999);
    saddr.sin_addr.s_addr = INADDR_ANY;

    if(bind(fd,(struct sockaddr*)&saddr,sizeof(saddr)) == -1){
        perror("bind");
        exit(-1);
    }

    if(listen(fd,128) == -1){
        perror("listen");
        exit(-1);
    }

    //对sockInfos进行初始化
    int num = sizeof(sockInfos) / sizeof(sockInfos[0]);
    for(int i=0;i<num;i++){
        //必须先置零，放后面会把fd和tid都变为0，使得有效了，即导致所有文件描述符都连接了
        bzero(&sockInfos[i],sizeof(sockInfos[i]));
        sockInfos[i].fd = -1;    //-1表示该文件描述符无效
        sockInfos[i].tid = -1;  //-1表示线程号无效

    }

    while(1){
        int clifd;
        struct sockaddr_in cliaddr;
        socklen_t len = sizeof(cliaddr);
```

```
        clifd=accept(fd,(struct sockaddr*)&cliaddr,&len);
        if(clifd == -1){
            perror("accpet");
            exit(-1);
        }

        struct sockInfo *info;
        for(int i=0;i<num;i++){
            //寻找还未连接的文件描述符
            if(sockInfos[i].fd == -1){
                info = &sockInfos[i];
                break;
            }
            //有可能文件描述符已用完，需要等待可用的文件描述符
            if(i == num-1){
                sleep(1);
                //执行完这一句，i还会自增，仍然需要从0开始遍历
                i = -1;
            }
        }

        info->fd = clifd;
        memcpy(&info->addr,&cliaddr,len);
        pthread_create(&info->tid,NULL,thread_child,info);
        //只能使用pthread_detach，join会阻塞，导致无法接收到下一个连接
        pthread_detach(info->tid);

    }
    close(fd);
    return 0;
}
```

客户端

```
// TCP通信的客户端
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

int main() {

    // 1.创建套接字
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if(fd == -1) {
        perror("socket");
        exit(-1);
    }

    // 2.连接服务器端
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &serveraddr.sin_addr.s_addr);
    serveraddr.sin_port = htons(9999);
    int ret = connect(fd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
```

```
    if(ret == -1) {
        perror("connect");
        exit(-1);
    }

    // 3. 通信
    char recvBuf[1024];
    int i = 0;
    while(1) {

        sprintf(recvBuf, "data : %d\n", i++);

        // 给服务器端发送数据
        write(fd, recvBuf, strlen(recvBuf)+1);

        int len = read(fd, recvBuf, sizeof(recvBuf));
        if(len == -1) {
            perror("read");
            exit(-1);
        } else if(len > 0) {
            printf("recv server : %s\n", recvBuf);
        } else if(len == 0) {
            // 表示服务器端断开连接
            printf("server closed...");
            break;
        }

        sleep(1);
    }

    // 关闭连接
    close(fd);

    return 0;
}
```

9.10 半连接与端口复用

close函数

```
#include <unistd.h>
int close(int sockfd);
```

close系统调用并非是立即关闭一个连接，而是将 sockfd 的引用计数减1，只有当引用计数为0，才关闭连接。

shutdown函数

shutdown可以不管引用计数就激发TCP的正常连接终止序列

```
#include <sys/socket.h>
int shutdowm(int sockfd, int howto);
```

howto的参数：

- SHUT_RD(0): 关闭sockfd的读。任何在当前套接字接收缓冲区的数据将被丢弃。
- SHUT_WR(1): 关闭sockfd的写
- SHUT_RDWR(2): 关闭sockfd的读写

注意

如果有多个进程共享同一个套接字，close每被调用一次，引用计数-1，当引用计数为0时，套接字被释放。

在多进程中，如果一个进程调用了shutdown，将导致其它进程无法用该套接字进行通信，close不会。

setsockopt

端口复用的设置的时机是在服务器绑定端口之前，即 bind 前。

```
#include <sys/socket.h>
#include <sys/types.h>
int setsockopt(int sockfd, int level, int optname,const void *optval, socklen_t
optlen);
//level: 端口复用级别, SOL_SOCKET、IPPROTO_TCP、IPPROTO_IP和IPPROTO_IPV6
//optname: 选项: SO_REUSEADDR  SO_REUSEPORT
//optval: 端口复用的值，1表示可以复用，0表示不能复用
//optlen: optval缓冲区长度
```

端口复用例子

服务端

```
#include <stdio.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);

    if(lfd == -1) {
        perror("socket");
        return -1;
    }

    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons(9999);

    int optval = 1;
    setsockopt(lfd, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof(optval));

    // 绑定
    int ret = bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));
    if(ret == -1) {
```

```
        perror("bind");
        return -1;
    }

    // 监听
    ret = listen(lfd, 8);
    if(ret == -1) {
        perror("listen");
        return -1;
    }

    // 接收客户端连接
    struct sockaddr_in cliaddr;
    socklen_t len = sizeof(cliaddr);
    int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);
    if(cfd == -1) {
        perror("accpet");
        return -1;
    }

    // 获取客户端信息
    char cliIp[16];
    inet_ntop(AF_INET, &cliaddr.sin_addr.s_addr, cliIp, sizeof(cliIp));
    unsigned short cliPort = ntohs(cliaddr.sin_port);

    // 输出客户端的信息
    printf("client's ip is %s, and port is %d\n", cliIp, cliPort );

    // 接收客户端发来的数据
    char recvBuf[1024] = {0};
    while(1) {
        int len = recv(cfd, recvBuf, sizeof(recvBuf), 0);
        if(len == -1) {
            perror("recv");
            return -1;
        } else if(len == 0) {
            printf("客户端已经断开连接...\n");
            break;
        } else if(len > 0) {
            printf("read buf = %s\n", recvBuf);
        }

        // 小写转大写
        for(int i = 0; i < len; ++i) {
            recvBuf[i] = toupper(recvBuf[i]);
        }

        printf("after buf = %s\n", recvBuf);

        // 大写字符串发给客户端
        ret = send(cfd, recvBuf, strlen(recvBuf) + 1, 0);
        if(ret == -1) {
            perror("send");
            return -1;
        }
    }

    close(cfd);
```

```
    close(lfd);

    return 0;
}
```

客户端

```
#include <stdio.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {

    // 创建socket
    int fd = socket(PF_INET, SOCK_STREAM, 0);
    if(fd == -1) {
        perror("socket");
        return -1;
    }

    struct sockaddr_in seraddr;
    inet_pton(AF_INET, "127.0.0.1", &seraddr.sin_addr.s_addr);
    seraddr.sin_family = AF_INET;
    seraddr.sin_port = htons(9999);

    // 连接服务器
    int ret = connect(fd, (struct sockaddr *)&seraddr, sizeof(seraddr));

    if(ret == -1){
        perror("connect");
        return -1;
    }

    while(1) {
        char sendBuf[1024] = {0};
        fgets(sendBuf, sizeof(sendBuf), stdin);

        write(fd, sendBuf, strlen(sendBuf) + 1);

        // 接收
        int len = read(fd, sendBuf, sizeof(sendBuf));
        if(len == -1) {
            perror("read");
            return -1;
        }else if(len > 0) {
            printf("read buf = %s\n", sendBuf);
        } else {
            printf("服务器已经断开连接...\n");
            break;
        }
    }

    close(fd);

    return 0;
```

```
}
```

9.11 IO多路复用

IO多路复用能够使得程序同时监听多个文件描述符，提高程序的性能。

例子中使用的客户端代码是相同的

```
#include <stdio.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {

    // 创建socket
    int fd = socket(PF_INET, SOCK_STREAM, 0);
    if(fd == -1) {
        perror("socket");
        return -1;
    }

    struct sockaddr_in seraddr;
    inet_pton(AF_INET, "127.0.0.1", &seraddr.sin_addr.s_addr);
    seraddr.sin_family = AF_INET;
    seraddr.sin_port = htons(9999);

    // 连接服务器
    int ret = connect(fd, (struct sockaddr *)&seraddr, sizeof(seraddr));

    if(ret == -1){
        perror("connect");
        return -1;
    }

    int num = 0;
    while(1) {
        char sendBuf[1024] = {0};
        // sprintf(sendBuf, "send data %d", num++);
        fgets(sendBuf, sizeof(sendBuf), stdin);

        write(fd, sendBuf, strlen(sendBuf) + 1);

        // 接收
        int len = read(fd, sendBuf, sizeof(sendBuf));
        if(len == -1) {
            perror("read");
            return -1;
        }else if(len > 0) {
            printf("read buf = %s\n", sendBuf);
        } else {
            printf("服务器已经断开连接...\n");
            break;
        }
    }
```

```
    close(fd);

    return 0;
}
```

9.11.1 select

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/select.h>
int select(int nfds,fd_set *readfds,fd_set *writefds,fd_set *exceptfds,struct
timeval *timeout);
//返回值：-1表示失败，>0表示检测的集合中有n个文件描述符发生了变化，=0表示超时时间内没有检测到
```

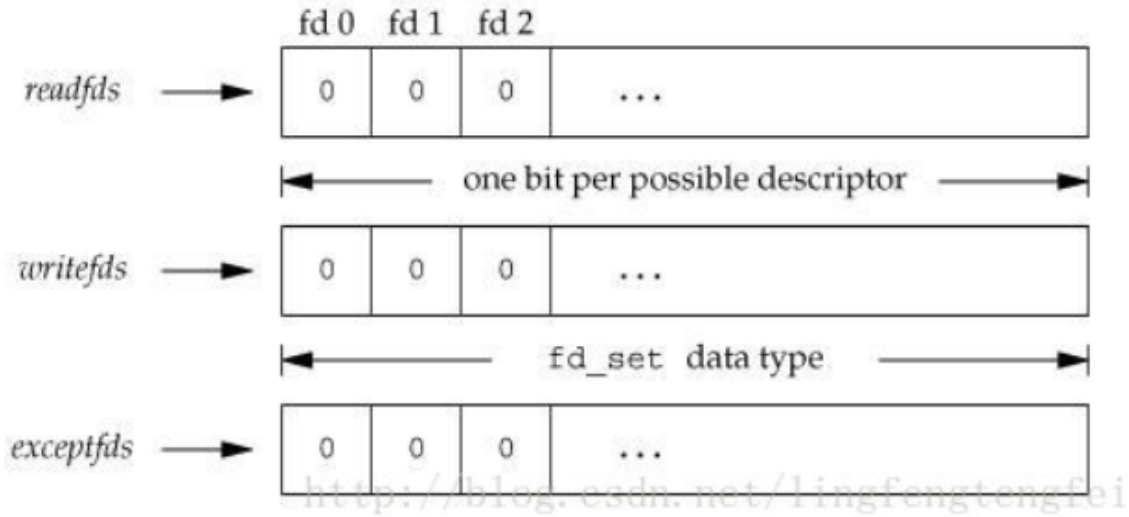
select 监听的文件描述符分3类，分别是可读、可写、异常。调用select会阻塞，直到有描述符就绪或超时，函数才返回。当select函数返回后，可以通过遍历fdset，来找到就绪的描述符。

select能够监视的文件描述符数量存在最大限制，一般为1024，可以修改。

```
struct timeval{
    long tv_sec;
    long tv_usec;
};
```

fd_set类型

fd_set类型变量每一位代表了一个描述符。我们也可以认为它只是一个由很多二进制位构成的数组。如下图所示：



对于 fd_set类型的变量我们所能做的就是声明一个变量，为变量赋一个同种类型变量的值，或者使用以下几个宏来控制它：

```
#include <sys/select.h>
int FD_ZERO(int fd,fd_set *fdset);
int FD_CLR(int fd,fd_set *fdset);
int FD_SET(int fd,fd_set *fdset);
int FD_ISSET(int fd,fd_set *fdset);
```

当声明了一个文件描述符集后，必须用 FD_ZERO 将所有位置置0，之后将我们所感兴趣的描述符所对应的位置位，如：

```
fd_set rset;
int fd;
FD_ZERO(&rset);
FD_SET(fd,&rset);
FD_SET(stdin,&rset);
```

select返回后，可以用 FD_ISSET 测试给定位是否置位。

```
if(FD_ISSET(fd,&rset));
```

每次使用完select后，都要将文件描述符重新置位，因为事件发生后，select会改变文件描述符。

select使用步骤

1. 构造一个关于文件描述符的列表，将要监听的文件描述符添加到该列表中。
2. 调用一个系统函数，监听该列表中的文件描述符，直到这些文件描述符中的一个或者多个进行I/O操作时，该函数返回。
    - 该函数是阻塞的
    - 函数对文件描述符的检测操作是由内核完成的
3. 该函数返回时会告诉进程有哪些文件描述符要进行I/O操作。

缺点：

- 每次需要在用户态和内核态之间拷贝；
- 有事件发生需要遍历文件描述符；
- fds集合不能重用，每次都需要重置。

select服务端通信例子

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>

int main() {

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in saddr;
    saddr.sin_port = htons(9999);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
```

```c
    // 绑定
    bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));

    // 监听
    listen(lfd, 8);

    // 创建一个fd_set的集合，存放的是需要检测的文件描述符
    fd_set rdset, tmp;
    FD_ZERO(&rdset);
    FD_SET(lfd, &rdset);
    int maxfd = lfd;

    while(1) {

        tmp = rdset;

        // 调用select系统函数，让内核帮检测哪些文件描述符有数据
        int ret = select(maxfd + 1, &tmp, NULL, NULL, NULL);
        if(ret == -1) {
            perror("select");
            exit(-1);
        } else if(ret == 0) {
            continue;
        } else if(ret > 0) {
            // 说明检测到了有文件描述符的对应的缓冲区的数据发生了改变
            if(FD_ISSET(lfd, &tmp)) {
                // 表示有新的客户端连接进来了
                struct sockaddr_in cliaddr;
                int len = sizeof(cliaddr);
                int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

                // 将新的文件描述符加入到集合中
                FD_SET(cfd, &rdset);

                // 更新最大的文件描述符
                maxfd = maxfd > cfd ? maxfd : cfd;
            }

            for(int i = lfd + 1; i <= maxfd; i++) {
                if(FD_ISSET(i, &tmp)) {
                    // 说明这个文件描述符对应的客户端发来了数据
                    char buf[1024] = {0};
                    int len = read(i, buf, sizeof(buf));
                    if(len == -1) {
                        perror("read");
                        exit(-1);
                    } else if(len == 0) {
                        printf("client closed...\n");
                        close(i);
                        FD_CLR(i, &rdset);
                    } else if(len > 0) {
                        printf("read buf = %s\n", buf);
                        write(i, buf, strlen(buf) + 1);
                    }
                }
            }

        }
```

```
    }
    close(lfd);
    return 0;
}
```

9.11.2 poll

```
#include <poll.h>
int poll(struct pollfd *fds,unsigned int nfds,int timeout);
//fds: 需要检测的文件描述符集合
//nfds: 第一个参数数组中最后一个有效元素的下标+1
//timeout: 阻塞时长，0不阻塞，-1阻塞，>0阻塞时长
//返回值: -1失败，>0表示检测到集合中有n文件描述符发生变化
```

```
struct pollfd{
    int fd;         //文件描述符
    short events;   //要监视的事件
    short revents;  //实际发生的事件
};
//events: 常用: POLLIN(可读) POLLOUT(可写) POLLERR(错误)
```

poll返回后，也需要轮询pollfd来获取就绪的描述符。

pollfd没有最大数量限制。

由于同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。

poll服务端通信例子

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <poll.h>


int main() {

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in saddr;
    saddr.sin_port = htons(9999);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;

    // 绑定
    bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));

    // 监听
    listen(lfd, 8);

    // 初始化检测的文件描述符数组
    struct pollfd fds[1024];
```

```
    for(int i = 0; i < 1024; i++) {
        fds[i].fd = -1;
        fds[i].events = POLLIN;
    }
    fds[0].fd = lfd;
    int nfds = 0;

    while(1) {

        // 调用poll系统函数，让内核帮检测哪些文件描述符有数据
        int ret = poll(fds, nfds + 1, -1);
        if(ret == -1) {
            perror("poll");
            exit(-1);
        } else if(ret == 0) {
            continue;
        } else if(ret > 0) {
            // 说明检测到了有文件描述符的对应的缓冲区的数据发生了改变
            if(fds[0].revents & POLLIN) {
                // 表示有新的客户端连接进来了
                struct sockaddr_in cliaddr;
                int len = sizeof(cliaddr);
                int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

                // 将新的文件描述符加入到集合中
                for(int i = 1; i < 1024; i++) {
                    if(fds[i].fd == -1) {
                        fds[i].fd = cfd;
                        fds[i].events = POLLIN;
                        break;
                    }
                }

                // 更新最大的文件描述符的索引
                nfds = nfds > cfd ? nfds : cfd;
            }

            for(int i = 1; i <= nfds; i++) {
                if(fds[i].revents & POLLIN) {
                    // 说明这个文件描述符对应的客户端发来了数据
                    char buf[1024] = {0};
                    int len = read(fds[i].fd, buf, sizeof(buf));
                    if(len == -1) {
                        perror("read");
                        exit(-1);
                    } else if(len == 0) {
                        printf("client closed...\n");
                        close(fds[i].fd);
                        fds[i].fd = -1;
                    } else if(len > 0) {
                        printf("read buf = %s\n", buf);
                        write(fds[i].fd, buf, strlen(buf) + 1);
                    }
                }
            }

        }
```

```
    }
    close(lfd);
    return 0;
}
```

9.11.3 epoll

epoll是select和poll的增强版本，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关心的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的拷贝只需一次。

```
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_ctl(int epfd,int op,int fd,struct epoll_event *event);
int epoll_wait(int epfd,struct epoll_event *events,int maxevents,int timeout);
```

epoll_create：创建一个epoll句柄，size用来告诉内核这个监听的数目。创建epoll句柄后会占用一个fd值，在使用完epoll后，必须调用close()关闭，否则会导致fd被耗尽。（size已经无意义，但必须大于0）

- 返回-1表示调用失败
- >0表示创建epoll实例，即返回epfd

epoll_ctl：对指定描述符fd执行op操作。

- epfd：epoll_create()的返回值。
- op：三个宏：`EPOLL_CTL_ADD、EPOLL_CTL_DEL、EPOLL_CTL_MOD`。分别对应添加、删除、修改对fd的监听事件。
- fd：需要监听的文件描述符。
- event：告诉内核需要监听什么事。
    - ```
      struct epoll_event{
          __uint32_t events;  //epoll events
          epoll_data_t data;  //user data variable，用socket返回的文件描述符即可
      };
      /*
      EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
      EPOLLOUT：表示对应的文件描述符可以写；
      EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
      EPOLLERR：表示对应的文件描述符发生错误；
      EPOLLHUP：表示对应的文件描述符被挂断；
      EPOLLET：  将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。
      EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里
      */

      typedef union epoll_data{
          void    *ptr;
          int      fd;
          uint32_t u32;
          uint64_t u64;
      }epoll_data_t;
      ```

epoll_wait：等待epfd上的IO事件，最多返回maxevents个事件。返回需要处理的事件数目，返回0表示超时。
      ```

- events：从内核得到事件的集合。这里是个传出参数
- maxevents：告诉内核这个events有多大，一般取第二个参数结构体数组的大小
- timeout：0不阻塞，-1阻塞，>0表示阻塞的时长
- 返回值：成功返回发生变化的文件描述符的个数，失败返回-1

epoll

工作模式

LT模式（默认，水平触发）：支持阻塞和非阻塞。当epoll_wait检测到描述符事件发生并将此事件通知应用程序，**应用程序可以不立即处理该事件。**下次调用epoll_wait时，会再次响应应用程序并通知此事件。

```
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
int main()
{
    //创建套接字
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if(fd == -1){
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(9999);
    inet_pton(AF_INET, "127.0.0.1", &saddr.sin_addr.s_addr);

    if(bind(fd, (struct sockaddr*)&saddr, sizeof(saddr)) == -1){
        perror("bind");
        exit(-1);
    }

    if(listen(fd, 64) == -1){
        perror("listen");
        exit(-1);
    }

    //创建epoll，并注册监听
    int epfd = epoll_create(100);
    struct epoll_event event;
    event.events = EPOLLIN; //监听读事件
    event.data.fd = fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);

    //创建一个epoll_event数组，用来存储发生的事件
    struct epoll_event events[1024];
    while(1){
        //等待事件，-1表示阻塞
        int ret = epoll_wait(epfd, events, 1024, -1);
        if(ret == -1){
            perror("epoll_wait");
```

```
            exit(-1);
        }

        printf("ret = %d\n", ret);
        for(int i=0; i<ret; i++){
            int curfd = events[i].data.fd;

            //监听的文件描述符上有连接
            if(curfd == fd){
                struct sockaddr_in cliaddr;
                socklen_t c_len = sizeof(cliaddr);
                int clifd = accept(fd, (struct sockaddr*)&cliaddr, &c_len);

                char cliIP[16];
                inet_ntop(AF_INET, &cliaddr.sin_addr.s_addr, cliIP,
sizeof(cliIP));
                int cliPort = ntohs(cliaddr.sin_port);
                printf("connection with %s:%d\n", cliIP, cliPort);

                //对cliaddr进行监听
                event.events = EPOLLIN;
                event.data.fd = clifd;
                //注意这里注册了监听对象为clifd，即监听客户端的通信
                epoll_ctl(epfd, EPOLL_CTL_ADD, clifd, &event);
            }
            else{
                //不需要写事件
                if(events[i].events & EPOLLOUT) continue;

                //收到了客户端的数据
                char buff[1024] = {0};
                int buff_len = sizeof(buff);
                //curfd表示当前触发事件的文件描述符
                int len =recv(curfd, buff, buff_len, 0);
                if(len == -1){
                    if(errno == EAGAIN)
                        printf("data over.....\n");
                    else {
                        perror("recv");
                        exit(-1);
                    }
                }
                else if(len == 0){
                    printf("client closed...\n");
                    //客户端关闭了连接，服务端就要将该连接关闭，epoll不再监听该文件描述符
                    //删除了监听事件，第四个参数为NULL，不需要监听
                    epoll_ctl(epfd, EPOLL_CTL_DEL, curfd, NULL);
                    close(curfd);
                }
                else{
                    //服务端回射数据
                    printf("server recv: %s\n", buff);
                    send(curfd, buff, buff_len, 0);
                }
            }
        }
    }
    close(epfd);
```

```
    close(fd);
    return 0;
}
```

ET模式（边沿触发）：只支持非阻塞。当epoll_wait检测到描述符事件发生并将此事件通知应用程序，**应用程序必须立即处理该事件。**如果不处理，下次调用epoll_wait时，不会再次响应应用程序并通知此事件。如果用户做了某些操作导致那个文件描述符不再为就绪状态，再调用epoll_wait时会响应应用程序并通知此事件。

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <errno.h>

int main() {

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in saddr;
    saddr.sin_port = htons(9999);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;

    // 绑定
    bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));

    // 监听
    listen(lfd, 8);

    // 调用epoll_create()创建一个epoll实例
    int epfd = epoll_create(100);

    // 将监听的文件描述符相关的检测信息添加到epoll实例中
    struct epoll_event epev;
    epev.events = EPOLLIN;
    epev.data.fd = lfd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &epev);

    struct epoll_event epevs[1024];

    while(1) {

        int ret = epoll_wait(epfd, epevs, 1024, -1);
        if(ret == -1) {
            perror("epoll_wait");
            exit(-1);
        }

        printf("ret = %d\n", ret);
```

```
        for(int i = 0; i < ret; i++) {

            int curfd = epevs[i].data.fd;

            if(curfd == lfd) {
                // 监听的文件描述符有数据达到，有客户端连接
                struct sockaddr_in cliaddr;
                int len = sizeof(cliaddr);
                int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

                // 设置cfd属性非阻塞
                int flag = fcntl(cfd, F_GETFL);
                flag |= O_NONBLOCK;
                fcntl(cfd, F_SETFL, flag);

                epev.events = EPOLLIN | EPOLLET;    // EPOLLET设置边沿触发
                epev.data.fd = cfd;
                epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &epev);
            } else {
                if(epevs[i].events & EPOLLOUT) {
                    continue;
                }

                // 循环读取出所有数据
                char buf[5];
                int len = 0;
                while( (len = read(curfd, buf, sizeof(buf))) > 0) {
                    // 打印数据
                    // printf("recv data : %s\n", buf);
                    write(STDOUT_FILENO, buf, len);
                    write(curfd, buf, len);
                }
                if(len == 0) {
                    printf("client closed....");
                }else if(len == -1) {
                    if(errno == EAGAIN) {
                        printf("data over.....");
                    }else {
                        perror("read");
                        exit(-1);
                    }

                }

            }

        }
    }
    close(lfd);
    close(epfd);
    return 0;
}
```

9.12 UDP

```
#include <sys/types.h>
#include <sys/socket.h>
//UDP没有连接，每次读取数据都要获取发送端的socket地址

ssize_t recvfrom(int sockfd, void *buff, size_t len, int flags, struct sockaddr*
src_addr, socklen_t* addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t len, int flags, const struct
sockaddr* dest_addr, socklen_t addrlen);
```

UDP服务端

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>

int main() {

    // 1.创建一个通信的socket
    int fd = socket(PF_INET, SOCK_DGRAM, 0);

    if(fd == -1) {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(9999);
    addr.sin_addr.s_addr = INADDR_ANY;

    // 2.绑定
    int ret = bind(fd, (struct sockaddr *)&addr, sizeof(addr));
    if(ret == -1) {
        perror("bind");
        exit(-1);
    }

    // 3.通信
    while(1) {
        char recvbuf[128];
        char ipbuf[16];

        struct sockaddr_in cliaddr;
        int len = sizeof(cliaddr);

        // 接收数据
        int num = recvfrom(fd, recvbuf, sizeof(recvbuf), 0, (struct sockaddr
*)&cliaddr, &len);

        printf("client IP : %s, Port : %d\n",
            inet_ntop(AF_INET, &cliaddr.sin_addr.s_addr, ipbuf, sizeof(ipbuf)),
            ntohs(cliaddr.sin_port));
```

```
        printf("client say : %s\n", recvbuf);

        // 发送数据
        sendto(fd, recvbuf, strlen(recvbuf) + 1, 0, (struct sockaddr *)&cliaddr,
sizeof(cliaddr));

    }

    close(fd);
    return 0;
}
```

UDP客户端

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>

int main() {

    // 1.创建一个通信的socket
    int fd = socket(PF_INET, SOCK_DGRAM, 0);

    if(fd == -1) {
        perror("socket");
        exit(-1);
    }

    // 服务器的地址信息
    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(9999);
    inet_pton(AF_INET, "127.0.0.1", &saddr.sin_addr.s_addr);

    int num = 0;
    // 3.通信
    while(1) {

        // 发送数据
        char sendBuf[128];
        sprintf(sendBuf, "hello , i am client %d \n", num++);
        sendto(fd, sendBuf, strlen(sendBuf) + 1, 0, (struct sockaddr *)&saddr,
sizeof(saddr));

        // 接收数据
        int num = recvfrom(fd, sendBuf, sizeof(sendBuf), 0, NULL, NULL);
        printf("server say : %s\n", sendBuf);

        sleep(1);
    }

    close(fd);
    return 0;
}
```

9.12.2 广播

向子网中多台计算机发送消息，并且子网中所有的计算机都可以接收到发送方发送的消息，每个广播消息都包含一个特殊的IP地址，这个IP中子网内主机标记部分的二进制全部为1

广播只能在局域网中使用

客户端需要绑定服务器广播使用的端口，才可以收到广播消息。

可以用 setsockopt 设置广播属性。

9.12.3 组播

单播地址标识单个IP接口，广播地址标识某个子网的所有IP接口，多播地址标识一组IP接口。

多播数据报只由对它感兴趣的接口接收。

多播可以在局域网中使用，也能跨广域网使用。

客户端需要加入到多播组，才能接收到多播的数据。

IP地址	说明
224.0.0.0~224.0.0.255	局部链接多播地址：是为路由协议和其它用途保留的地址，路由器并不转发属于此范围的IP包
224.0.1.0~224.0.1.255	预留多播地址：公用组播地址，可用于Internet；使用前需要申请
224.0.2.0~238.255.255.255	预留多播地址：用户可用组播地址(临时组地址)，全网范围内有效
239.0.0.0~239.255.255.255	本地管理组播地址，可供组织内部使用，类似于私有 IP 地址，不能用于Internet，可限制多播范围

可以通过 setsockopt 设置组播

9.13 本地套接字进程间通信

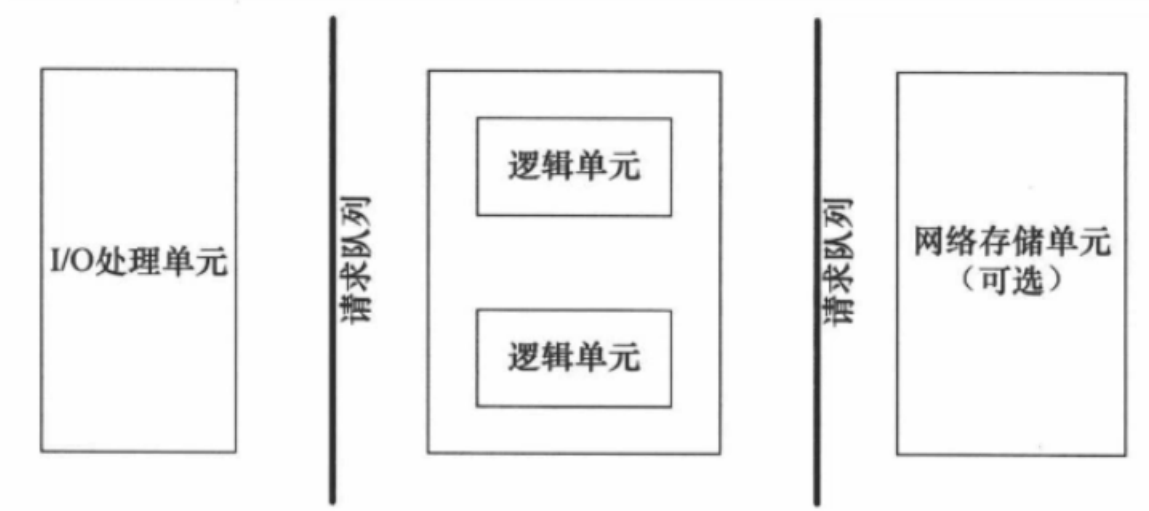
```
// 本地套接字通信的流程 - tcp
// 服务器端 1. 创建监听的套接字
int lfd = socket(AF_UNIX/AF_LOCAL, SOCK_STREAM, 0);
2. 监听的套接字绑定本地的套接字文件 -> server端
    struct sockaddr_un addr;
    // 绑定成功之后，指定的sun_path中的套接字文件会自动生成。
    bind(lfd, addr, len);
3. 监听
    listen(lfd, 100);
4. 等待并接受连接请求
    struct sockaddr_un cliaddr;
    int cfd = accept(lfd, &cliaddr, len);
5. 通信
    接收数据：read/recv
    发送数据：write/send
6. 关闭连接
    close();

// 客户端的流程
1. 创建通信的套接字
    int fd = socket(AF_UNIX/AF_LOCAL, SOCK_STREAM, 0);
```

```
2. 监听的套接字绑定本地的IP 端口
    struct sockaddr_un addr;
    // 绑定成功之后，指定的sun_path中的套接字文件会自动生成。
    bind(lfd, addr, len);
3. 连接服务器
    struct sockaddr_un serveraddr;
    connect(fd, &serveraddr, sizeof(serveraddr));
4. 通信
    接收数据：read/recv
    发送数据：write/send
5. 关闭连接
    close()
```

10、服务器基本框架和事件处理模式

10.1 服务器基本框架



模块	功能
I/O处理单元	处理客户连接，读写网络数据
逻辑单元	业务进程或线程
网络存储单元	数据库、文件或缓存
请求队列	各单元之间的通信方式

I/O处理单元是服务器管理客户连接的模块。它通常要完成：等待并接收新的客户连接，接收客户数据，将服务器响应数据返回给客户端。但数据的收发不一定在I/O处理单元中执行，也可能在逻辑单元中执行，具体在何处执行取决于事件处理模式。

一个逻辑单元通常是一个进程或线程。它分析并处理客户数据，然后将结果传递给I/O处理单元或者直接发送给客户端。服务器通常拥有多个逻辑单元，以实现对多个客户端任务的并发处理。

网络存储单元可以是数据库、缓存和文件，但不是必须的。

请求队列是各单元之间的通信方式的抽象。I/O处理单元接收到客户请求 时，需要以某种方式通知一个逻辑单元来处理该请求。同样，多个逻辑单元同时访问一个存储单元时，也需要采用某种机制来协调处理竞态条件。请求队列通常被实现为池的一部分。

10.2 两种高效的事件处理模式

Linux服务器程序必须处理的三类事件：I/O事件、信号和定时事件。

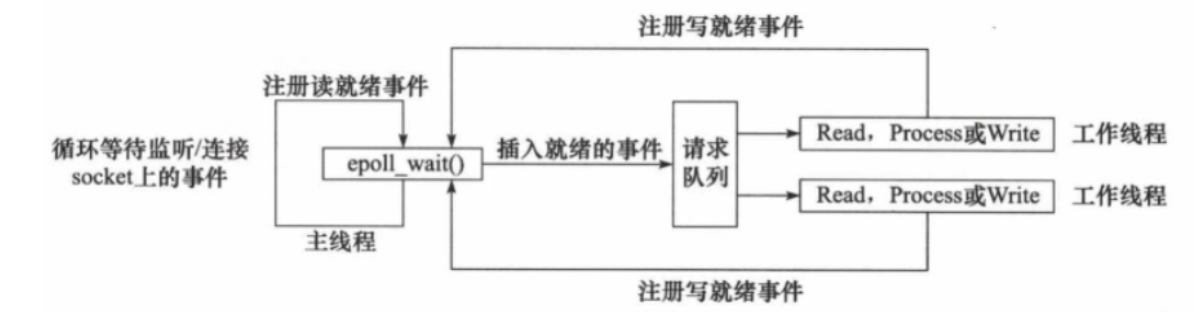
两种高效的事件处理模式：Reactor（用于同步模型）和Proactor（用于异步IO模型）。

10.2.1 Reactor模式

Reactor模式：通常由同步I/O模型实现。它要求主线程（I/O处理单元）只负责监听文件描述符上是否有事件发生，有的话立即将该事件通知工作线程（逻辑单元）。主线程不做任何其它的工作，读写数据、接受新的连接、处理客户请求均在工作线程中完成。

由同步I/O模型（以epoll_wait为例）实现Reactor模式的工作流程：

1. 主线程往epoll内核时间表中注册socket上的读就绪事件；
2. 主线程调用epoll_wait等待socket上有数据可读；
3. 当socket上有数据可读时，epoll_wait通知主线程；主线程将socket可读事件放入请求队列；
4. 睡眠在请求队列上的某个工作线程被唤醒，从socket上读数据，并处理客户请求，然后往内核事表中注册该socket上的写就绪事件；
5. 主线程调用epoll_wait等待socket可写；
6. 当socket可写时，epoll_wait通知主线程，主线程将socket可写事件放入请求队列；
7. 睡眠在请求队列上的某个工作线程被唤醒，它往socket上写入服务器处理客户请求的结果；



10.2.2 Proactor

Proactor模式：使用异步I/O模型（常用aio_read和aio_write），将所有的I/O操作都交给主线程和内核处理，工作线程仅仅负责业务逻辑。

工作流程：

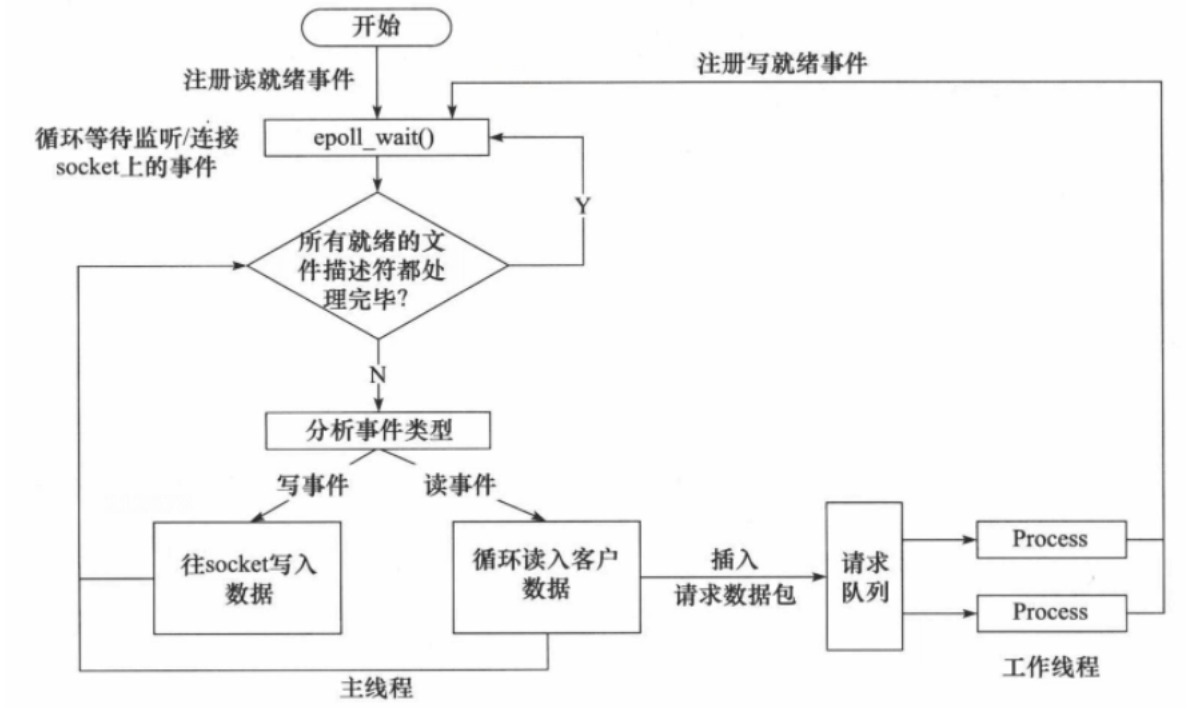
1. 主线程调用aio_read函数向内核注册socket上的读完成事件，并告诉用户读缓冲区的位置，以及读操作完成时如何通知应用程序（一般用信号通知）；
2. 主线程继续处理其它逻辑；
3. 当socket上的数据被读入用户缓冲区后，内核将向应用程序发送一个信号，以通知应用程序数据已经可用；
4. 应用程序选择一个工作线程来处理客户请求。工作线程处理完客户请求之后，调用aio_wait函数向内核注册socket上的写完成事件，同时告诉内核用户写缓冲区的位置，以及操作完成时如何通知应用程序。
5. 主线程继续处理其它逻辑。
6. 当用户缓冲区的数据被写入socket之后，内核将向应用程序发送一个信号，以通知应用程序数据已经发送完毕。
7. 应用程序预先定义好的信号处理函数选择一个工作线程来做后续处理，比如是否关闭socket。

10.2.3 模拟Proactor模式

我们可以使用同步IO模拟出Proactor模式：主线程直接执行数据的读写操作，读写完成之后，主线程向工作队列通知这一“完成事件”。工作线程直接获取读写的结果，之后只是对读写的结果进行逻辑处理。

使用同步IO模型（epoll_wait）模拟出的Proactor模式的工作流程如下：

1. 主线程往epoll内核事件表中注册socket上的读就绪事件。
2. 主线程调用epoll_wait等待socket上有数据可读。
3. 当socket上有数据可读时，epoll_wait通知主线程。主线程从socket循环读取数据，直到没有更多数据可读，然后将读取到的数据封装成一个请求对象并插入到请求队列。
4. 睡眠在请求队列上的某个工作线程被唤醒，它获得请求对象并处理客户请求，然后往epoll内核事件表中注册socket上的写就绪事件。
5. 主线程调用epoll_wait等待socket可写。
6. 当socket可写时，epoll_wait通知主线程。主线程往socket上写入服务器处理客户端请求的结果。



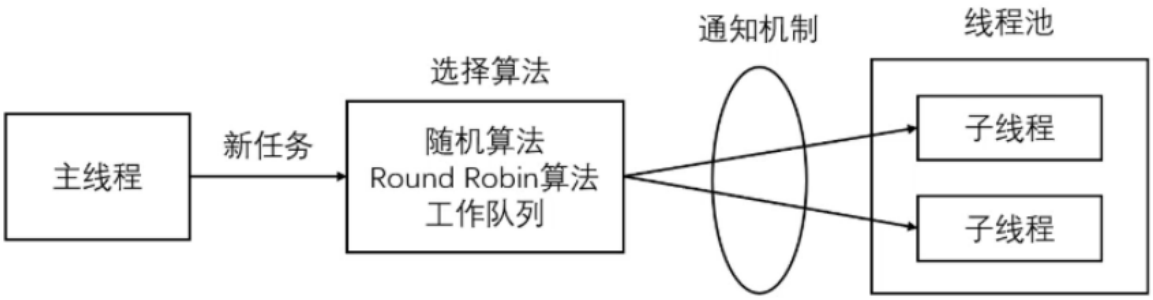
11、进程池和线程池

动态创建子进程（或子线程）来实现并发服务器有如下缺点：

- 动态创建进程（或线程）是比较耗费时间的，这将导致较慢的客户响应。
- 动态创建的子进程（或子线程）通常只用来为一个客户服务，这将导致系统上产生大量的细微进程（或线程）。进程（或线程）间的切换将消耗大量CPU时间。
- 动态创建的子进程是当前进程的完整映像。当前进程必须谨慎地管理其分配的文件描述符和堆内存等系统资源，否则子进程可能复制这些资源，从而使系统的可用资源急剧下降，进而影响服务器的性能。

11.1 线程池概述

- 线程池是由服务器预先创建的一组子线程，线程池中的线程数量应该和CPU数量差不多。
- 线程池中的所有子线程都运行着相同的代码。
- 当有新的任务到来时，主线程将通过某种方式选择线程池中的某一个子线程来为之服务。相比于动态创建子线程，选择一个已经存在的子线程的代价显然要小的多。置于主线程选择哪个子线程来为新任务服务，则有两种方式：
    1. 主线程使用某种算法来主动选择子线程。最简单、最常用的算法是随机算法和Round Robin（轮流选取）算法，当更优秀、更智能的算法将使任务在各个工作进程中更均匀地分配，从而减轻服务器的整体压力。
    2. 主线程和所有子线程通过一个共享的工作队列来同步，子线程都睡眠在该工作队列上。当有新的任务到来时，主线程将任务添加到工作队列中。这将唤醒正在等待任务的子线程，不过只有一个子线程将获得新任务的“接管权”，它可以从工作队列中取出任务并执行之，而其它子线程将继续睡眠在工作队列上。



线程池的特点和好处

- 空间换时间，浪费服务器的硬件资源，换取运行效率。
- 池是一组资源的集合，这组资源在服务器启动之初就被完全创建好并初始化，这称为静态资源。
- 当服务器进入正式运行阶段，开始处理客户端请求的时候，如果它需要相关的资源，可以直接从池中获取，无需动态分配。
- 当服务器处理完一个客户端连接后，可以把相关的资源放回池中，无需执行系统调用释放资源。