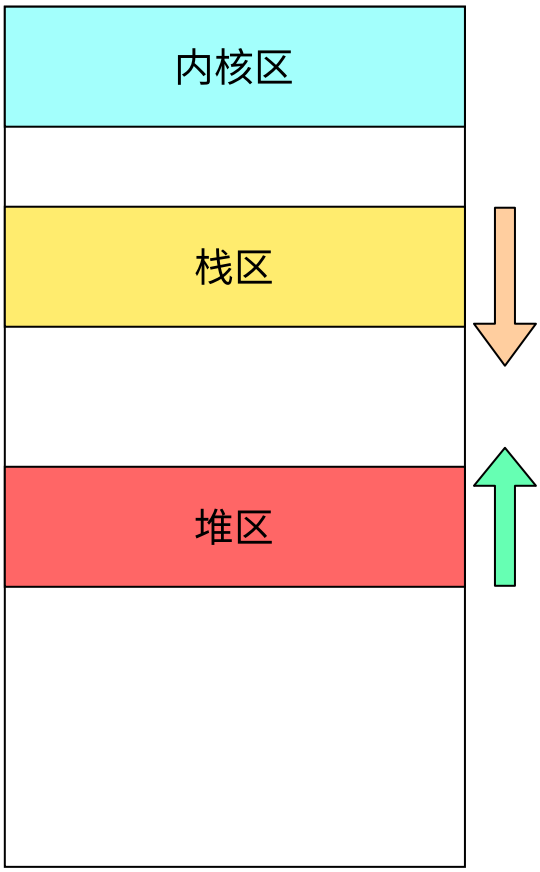
IPC进程通信 #2022.9.8

进程通信方式

- 套接字 socket
  - 网络套接字
  - 本地套接字
- 管道
  - Pipe 匿名
  - Fifo 有名
- 信号 signal
- 信号量 semget
- 共享内存 shmget mmap
- 消息队列 msgget

内核态

- 用户态和内核态
- 32位 4G



基本IO函数

- Open 打开文件
  - 成功：非负整数 文件描述符
  - 失败：-1 打开失败
- Read 读入数据
  - 读到文件结束返回0
  - 读失败返回-1
  - 读成功返回读到的字节数
- Write 写入数据
- 文件描述符：标识文件

Read使用说明

```
ssize_t read(int fd, void*buf, size_t count)
参数说明:
    fd: 是文件描述符，从command line获取数据时，为0
    buf: 为读出数据的缓冲区;
    count: 为每次读取的字节数（是请求读取的字节数，读上来的数据保存在缓冲区buf中，同时文件
```

Write使用说明

```
ssize_t write(int fd, const void*buf, size_t count);

参数说明:
    fd: 是文件描述符（输出到command line，就是1）
    buf: 通常是一个字符串，需要写入的字符串
    count: 是每次写入的字节数
```

代码实现

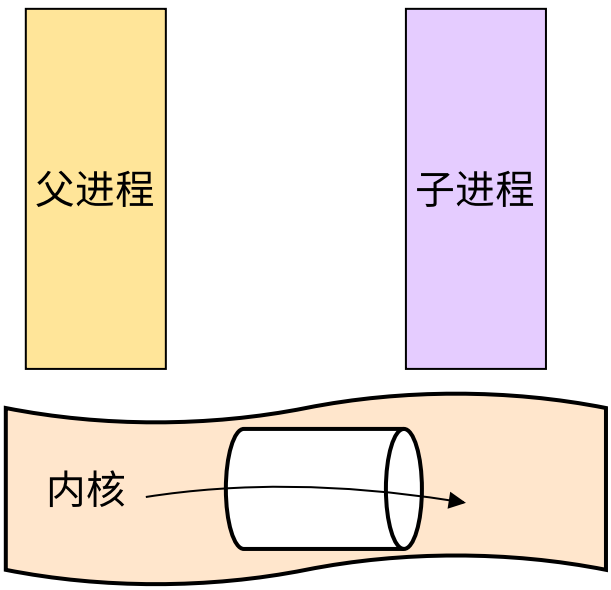
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int fd = open("hello.txt", O_RDONLY);
    int fx = open("newhello.txt", O_CREAT | O_WRONLY, 0644);
    char buf[1024];
    int n;
    while (n = read(fd, buf, sizeof(buf)))
    {
        int ret = write(fx, buf, n);
        if (ret == -1)
        {
            perror("write error");
            exit(1);
        }
        printf("write bytes: %d\n", n);
    }

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

注意事项

- 文件打开的模式
  - 只读
  - 只写
  - 读写
  - 阻塞

匿名管道 pipe

- 父子进程通信
- Fork 创建子进程
- 一种经典的IPC方式



- 血缘关系间的进程通信，本质是继承了文件描述符。
- 管道本质是一个伪文件，实质上是内核缓冲区。
- 管道的原理：管道实质为内核使用环形队列机制，借助内核缓冲区4K实现 单向流动
- 可以设置为非阻塞管道

管道的使用

  - 通常结合fork使用

```
1 int pipe(int fd[2]);
```

a. 成功：返回0

b. 失败：返回-1，设置errno

代码实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(){
    int fd[2]; //0为读(子) 1为写(父)
    pid_t pid;
    int ret = pipe(fd);
    if (ret == -1){
        perror("pipe error!");
        exit(1);
    }
    pid = fork();
    if (pid == -1){
        perror("fork error!");
        exit(1);
    }else if (pid == 0){
        //子进程 读0
        close(fd[1]);
        char buf[1024];
        printf("child = %d\n", getpid());
        ret = read(fd[0], buf, sizeof(buf));
        write(STDOUT_FILENO, buf, ret);
    }else if (pid > 0) {
        //父进程 写1
        sleep(1);
        close(fd[0]);
        printf("parent = %d\n", getpid());
        write(fd[1], "hello pipe\n", 11);
        wait(NULL); // 回收子进程，避免成为僵尸进程
    }
    printf("finished! %d\n", getpid());
    return 0;
}
```

管道的优缺点

- 优点：常用，简单。
- 缺点：
  - 半双工模式，数据单方向流动
  - 只能用于父子或兄弟进程。
  - 缓冲区大小有限，默认是4K
  - 管道没有名称
  - 数据不能重复读

非阻塞管道

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/fcntl.h>
int main()
{
    int fd[2]; //0为读 1为写
    pid_t pid;
    int ret = pipe(fd);
    if (ret == -1)
    {
        perror("pipe error!");
        exit(1);
    }
    pid = fork();
    if (pid == -1)
    {
        perror("fork error!");
        exit(1);
    }
    else if (pid == 0)
    {
        //子进程 读
        close(fd[1]);
        //非阻塞管道 没有数据直接返回
```

```
32          int flags = fcntl(fd[0], F_GETFL);
33          flags |= O_NONBLOCK;
34          fcntl(fd[0], F_SETFL, flags);
35
36          char buf[1024];
37      tryagain:
38          ret = read(fd[0], buf, sizeof(buf));
39          if (ret == -1)
40          {
41              if (errno == EAGAIN)
42              { //如果数据没到达 会返回EAGAIN
43                  write(STDOUT_FILENO, "try agian!\n", 11);
44                  sleep(1);
45                  goto tryagain;
46              }
47              else
48              {
49                  perror("read error");
50                  exit(1);
51              }
52          }
53          write(STDOUT_FILENO, buf, ret);
54          close(fd[0]);
55      }
56      else if (pid > 0)
57      {
58          //父进程 写
59          sleep(5);
60          close(fd[0]);
61          write(fd[1], "hello pipe\n", 11);
62          wait(NULL);
63          close(fd[1]);
64      }
65      return 0;
66  }
67
```

有名管道 fifo

- 非血缘关系的进程间通信
  - 提供了一个路径名与管道关联
  - 以文件形式存在于文件系统中

代码实现

```
int mkfifo (const char * pathname, mode_t mode)
参数说明:
    pathname:  路径名
    mode: 打开模式
```

- 创建管道，等待读入数据。

```c
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#define P_FIFO "/tmp/p_fifo"
int main(int argc, char** argv){
    char cache[100];
    int fd;
    memset(cache,0, sizeof(cache));    /*初始化内存*/
    if(access(P_FIFO,F_OK)==0){  /*管道文件存在*/
        execlp("rm","-f", P_FIFO, NULL);  /*删掉*/
        printf("access.\n");
    }
    if(mkfifo(P_FIFO, 0777) < 0){
        printf("createnamed pipe failed.\n");
    }
    fd = open(P_FIFO, O_RDONLY|O_NONBLOCK); /*非阻塞方式打开，只读*/
    while(1){
        memset(cache,0, sizeof(cache));
        if((read(fd,cache, 100)) == 0 ){ /*没有读到数据*/
            printf("nodata:\n");
        }
        else{
            printf("getdata:%s\n", cache); /*读到数据，将其打印*/
        }
        sleep(1);/*休眠1s*/
    }
    close(fd);
    return 0;
}
```

- 写入数据

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

```c
#include <fcntl.h>
#include <unistd.h>
#define P_FIFO "/tmp/p_fifo"
int main(int argc, char **argv){
    int fd;
    if(argc< 2){
        printf("please input the write data.\n");
    }
    fd = open(P_FIFO, O_WRONLY|O_NONBLOCK); /*非阻塞方式*/
    if(fd == -1) {
        printf("open failed!\n");
        return 0;
    }else{
        printf("open success!");
    }
    write(fd, argv[1], 100); /*将argv[1]写道fd里面去*/
    close(fd);
    return 0;
}
```

- 先启动读，再启动写。

有名管道的特点

- 使互不相关的两个进程间通信
- 两个进程将管道当做普通文件进行读写操作

内存映射 mmap

> Why learn：有什么用?
>
> How use：怎么用?

```c
#include<sys/mman.h>
void*mmap(void*start,size_t length,int prot,int flags,int fd,off_t offset);
int munmap(void*start,size_t length);
```

❑PROT_READ，内存段可读。

❑PROT_WRITE，内存段可写。

❑PROT_EXEC，内存段可执行。

❑PROT_NONE，内存段不能被访问。

表 6-1　mmap 的 flags 参数的常用值及其含义

常 用 值	含　义
MAP_SHARED	在进程间共享这段内存。对该内存段的修改将反映到被映射的文件中。它提供了进程间共享内存的 POSIX 方法
MAP_PRIVATE	内存段为调用进程所私有。对该内存段的修改不会反映到被映射的文件中
MAP_ANONYMOUS	这段内存不是从文件映射而来的。其内容被初始化为全 0。这种情况下，mmap 函数的最后两个参数将被忽略
MAP_FIXED	内存段必须位于 start 参数指定的地址处。start 必须是内存页面大小（4096 字节）的整数倍
MAP_HUGETLB	按照“大内存页面”来分配内存空间。“大内存页面”的大小可通过 /proc/meminfo 文件来查看

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

#define GiB *(1024LL * 1024 * 1024)
int main() {
    void *p = mmap(NULL, 3 GiB, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIV
    if(p == MAP_FAILED) {
        printf("mmap failed!\n");
    }else{
        printf("mmap success!\n");
    }

    *(int *)((u_int8_t*)p + 1 GiB) = 114;
    *(int *)((u_int8_t*)p + 2 GiB) = 115;

    printf("read = %d\n", *(int *)((u_int8_t*)p + 2 GiB));
    return 0;
}
```

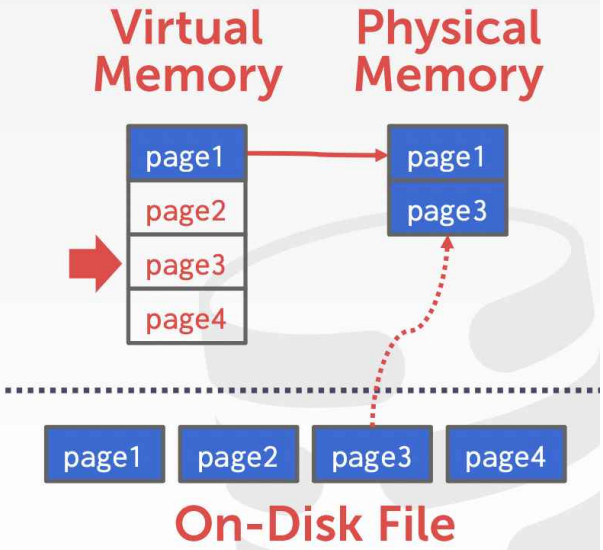
- 数据库系统不建议使用mmap，具体可参考：
  - 你确定你想用 MMAP 实现数据库么?

- CMU15-445 storage engine I

WHY NOT USE THE OS?

The DBMS can use memory mapping (**mmap**) to store the contents of a file into the address space of a program.

The OS is responsible for moving the pages of the file in and out of memory, so the DBMS doesn't need to worry about it.



WHY NOT USE THE OS?

What if we allow multiple threads to access the **mmap** files to hide page fault stalls?

This works good enough for read-only access.
It is complicated when there are multiple writers…

WHY NOT USE THE OS?

There are some solutions to this problem:

→ **madvise**: Tell the OS how you expect to read certain pages.
→ **mlock**: Tell the OS that memory ranges cannot be paged out.
→ **msync**: Tell the OS to flush memory ranges out to disk.

WHY NOT USE THE OS?

DBMS (almost) always wants to control things itself and can do a better job than the OS.

→ Flushing dirty pages to disk in the correct order.
→ Specialized prefetching.
→ Buffer replacement policy.
→ Thread/process scheduling.

The OS is **not** your friend.


```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/mman.h>
int main(){
```

```
    //把磁盘文件放入共享内存，这样可以使用指针访问磁盘文件 mytest.txt必须有内容。
    int fd = open("mytest.txt", O_RDWR | O_CREAT, 0644);
    if (fd < 0)
    {
        perror("open error!");
        exit(1);
    }
    //申请共享映射
    void *p = mmap(NULL, 100, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED)
    {
        perror("mmap error!");
        exit(1);
    }
    strcpy((char *)p, "abc");
    int ret = munmap(p, 100); //释放共享映射
    if (ret == -1)
    {
        perror("munmap error!");
        exit(1);
    }
    close(fd);
    return 0;
}

```

优点

- 对文件的读取操作跨过了页缓存，减少了数据的拷贝次数，用内存读写取代I/O读写，提高了文件读取效率。减少read、write系统调用。
- 提供进程间共享内存及相互通信的方式。
- 各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉。msync强制刷新

缺点

- 文件如果很小，比如是小于4k的，比如60bytes，由于在内存当中的组织都是按页组织的，将文件调入到内存当中是一个页4k，这样其他的4096-60=4036 bytes的内存空间就会浪费掉了。
- 创建、销毁、缺页造成的开销很大。mmap2
- 参考文档

文件控制 fcntl

- 设置阻塞与非阻塞

- 获得或设置已经打开的文件的属性

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

int setnonblocking(int fd) {
    int old_option = fcntl(fd, F_GETFL);
    int new_option = old_option | O_NONBLOCK;
    fcntl(fd, F_SETFL, new_option);
    return old_option;
}
int main() {
    int fd = open("jun.txt", O_CREAT | O_RDWR);
    if(fd < 0) {
        printf("error!");
        _exit(0);
    }
    int old_ = setnonblocking(fd);
    int new_ = fcntl(fd, F_GETFL);
    printf("old = %d, new = %d\n", old_, new_);
    return 0;
}
```

文件拓展 lseek()

- 获取文件大小（字节）
- 移动文件指针
- 文件拓展

```c
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
int main() {
    // 没有则创建这个文件
    int fd = open("aa.txt", O_CREAT | O_RDWR);

    // 获取文件大小
    int file_size = lseek(fd, 0, SEEK_END);
    printf("file size = %d\n", file_size);

    // 拓展文件大小 2000字节
    int total_size = lseek(fd, 2000, SEEK_END);
```

```
    printf("total size = %d\n", total_size);

    // 写入数据
    write(fd, "a", 1);
    if(res == 0) {
        printf("fsync success!\n");
    }
    return 0;
}
```

大纲

- exec
- dup2
- pipe

exec函数族

- 作用：在某一个进程中运行另一个可执行文件

```
int main(){
    printf("start...\n");
    execlp("ls", "-a", "-l",NULL);
    printf("never run except wrong\n");
}
```

```
//参考Stanford CS110
static int my_system(const char *command) {
    pid_t pid = fork();
    if (pid == 0) {
        char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
        execvp(arguments[0], arguments);
        printf("Failed to invoke /bin/sh to execute the supplied command.");
        exit(0);
    }
    // 父进程
    int status;
    waitpid(pid, &status, 0);
    // 没有学习过如何根据进程退出状态，判断信息
    return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
}
```

重定向dup2

- 文件描述符的复制或者重定向

```c
int main() {
    int fd = open("pp.txt", O_CREAT | O_RDWR, 0744);
    if(fd == -1) {
        printf("open failed!\n");
        exit(1);
    }
    dup2(fd, STDOUT_FILENO); // 原来访问STDOUT_FILENO的，现在变成了访问fd
    // 操作标准输出
    write(STDOUT_FILENO, "bai ge", 6);
    return 0;
}
```

```c
void test(){
    // 字符串数组
    char *buf = "a.txt:b.txt";
    char *buf2 = ":";
    int i = 0;
    int len = strlen(buf2);
    char p[1024];
    for(i = 0; i < len && buf2[i] != ':'; i++) {
        p[i] += buf2[i];
    }
    char q[1024];
    i++;
    int j = 0;
    while(i < len) {
        q[j++] = buf2[i++];
    }
    p[i] = '\0';
    q[j] = '\0';
    puts(p);
    puts(q);
    int pin[2];
    char temp[1024];
    if(strcmp(p, "") == 0) {
        printf("input empty!\n");
        pipe(pin);

        pid_t pid = fork();
        if(pid > 0) {
```

```
29              printf("I am parent, pid = %d\n", getpid());
30              close(pin[0]);
31              int fd = open("a.txt", O_CREAT | O_RDWR);
32              if(fd == -1) {
33                  printf("open failed!\n");
34              }
35              dup2(fd, STDIN_FILENO); // fd重定向为标准输入
36              char pp[1024];
37              memset(pp, 0, sizeof pp);
38              read(STDIN_FILENO, pp, sizeof(pp));
39              write(pin[1], pp, strlen(pp));
40              close(pin[1]);
41              wait(NULL);
42              printf("parent end!\n");
43          }else{
44              printf("I am child, pid = %d\n", getpid());
45              close(pin[1]); // 关闭写
46              char tt[1024];
47              memset(tt, 0, sizeof tt);
48              size_t n = read(pin[0], tt, sizeof(tt));
49              if(n == -1) {
50                  printf("read failed\n");
51                  exit(0);
52              }
53              printf("read n = %d, data = %s\n", n, tt);
54          }
55      }
56  }
```

信号

定义

- 一种开销很小的通信机制
- 处理方式
  a. 捕获
  b. 终止
  c. 忽略

信号量

定义

- PV互斥操作

- 常用API
  - sem_init
    - 参数1
    - 参数2:
      - 非0，进程间，PTHREAD_PROCESS_SHARED
      - 0，线程间，PTHREAD_PROCESS_PRIVATE
    - 参数3
  - sem_destory

C++20 信号量

> Posix和SYSTEM V
>
> Wait 获取信号量 value--
>
> Post 释放信号量 value++

```cpp
#include <iostream>
#include <thread>
#include <semaphore>

std::counting_semaphore<32> sem(6);
//std::binary_semaphore bsem(0);


void producer(){
    sem.release(1);
    std::cout<<"do something..."<<sem.max()<<std::endl;
    sem.acquire();
    std::cout<<"do something..2\n";
    if(sem.try_acquire())   std::cout<<"do something..3\n";
}


int main()
{
    std::thread t1(producer);
    t1.join();
    return 0;
}
```

Perf

- 性能查看

Strace

- 查看系统调用的路径

第六章

> 第六章 高级I/O函数

- 创建文件描述符
  - pipe
  - dup/dup2
- 读写数据的函数
  - readv/writev
  - sendfile
  - mmap/munmap
  - splice
  - tee
- 控制I/O行为和属性的函数
  - fcntl

本地套接字

定义

- 使用套接字除了可以实现网络间不同主机间的通信外，还可以实现同一主机的不同进程间的通信，且建立的通信是双向的通信。
- **它的地址是它所在的文件系统的路径名，创建之后套接字就和路径名绑定在一起。**
- AF_UNIX

代码

- 服务端

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
```

```c
#include <strings.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define UNIXSTR_PATH "/tmp/unix.str"
#define LISTENQ 5
#define BUFFER_SIZE 256

int main(void) {
    int listenfd, connfd;
    socklen_t len;
    struct sockaddr_un servaddr, cliaddr;

    if(-1 == (listenfd = socket(AF_LOCAL, SOCK_STREAM, 0)))
    {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    unlink(UNIXSTR_PATH);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, UNIXSTR_PATH);
    if(-1 == bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr)))
    {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    listen(listenfd, LISTENQ);

    len = sizeof(cliaddr);

    if(-1 == (connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &len)))
    {
        perror("accept");
        exit(EXIT_FAILURE);
    }

    char buf[BUFFER_SIZE];

    while(1)
    {
        bzero(buf, sizeof(buf));
        if(read(connfd, buf, BUFFER_SIZE) == 0) break;
```

```
        printf("Receive: %s", buf);
    }

    close(listenfd);
    close(connfd);
    unlink(UNIXSTR_PATH);

    return 0;
}

```

- 客户端

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define UNIXSTR_PATH "/tmp/unix.str"
#define LISTENQ 5
#define BUFFER_SIZE 256

int main(void)
{
    int sockfd;
    struct sockaddr_un servaddr;

    sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, UNIXSTR_PATH);

    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    char buf[BUFFER_SIZE];

    while(1)
    {
        bzero(buf, sizeof(BUFFER_SIZE));
        printf(">> ");
        if(fgets(buf, BUFFER_SIZE, stdin) == NULL)
```

```
        {
            break;
        }
        write(sockfd, buf, strlen(buf));
    }

    close(sockfd);

    return 0;
}

```

- 优缺点

  a. 效率高：和其他进程间通信方式相比，Unix 本地套接字使用方便，效率也高 。因为它不需要经过网络协议栈、不需要打包拆包、不需要计算校验和、不需要维护序号和应答等、只是将应用层数据从一个进程拷贝到另一个进程。常用于前后台进程通信，比如 X Window。另外，Unix 本地套接字可用于传递文件描述符、传递用户凭证等场景。

UDP

定义

- send和recv仅用于TCP
- UDP使用sendto和recvfrom

代码

- 服务端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define SERVER_PORT 8888
#define BUFF_LEN 1024

void handle_udp_msg(int fd)
{
    char buf[BUFF_LEN];  //接收缓冲区，1024字节
    socklen_t len;
    int count;
    struct sockaddr_in clent_addr;  //clent_addr用于记录发送方的地址信息
```

```
16      while(1)
17      {
18          memset(buf, 0, BUFF_LEN);
19          len = sizeof(clent_addr);
20          count = recvfrom(fd, buf, BUFF_LEN, 0, (struct sockaddr*)&clent_addr, &l
21          if(count == -1)
22          {
23              printf("recieve data fail!\n");
24              return;
25          }
26          printf("client:%s\n",buf);  //打印client发过来的信息
27          memset(buf, 0, BUFF_LEN);
28          sprintf(buf, "I have recieved %d bytes data!\n", count);  //回复client
29          printf("server:%s\n",buf);  //打印自己发送的信息给
30          sendto(fd, buf, BUFF_LEN, 0, (struct sockaddr*)&clent_addr, len);  //发送
31
32      }
33  }
34
35  /*
36      server:
37              socket-->bind-->recvfrom-->sendto-->close
38  */
39
40  int main(int argc, char* argv[])
41  {
42      int server_fd, ret;
43      struct sockaddr_in ser_addr;
44
45      server_fd = socket(AF_INET, SOCK_DGRAM, 0); //AF_INET:IPV4;SOCK_DGRAM:UDP
46      if(server_fd < 0)
47      {
48          printf("create socket fail!\n");
49          return -1;
50      }
51
52      memset(&ser_addr, 0, sizeof(ser_addr));
53      ser_addr.sin_family = AF_INET;
54      ser_addr.sin_addr.s_addr = htonl(INADDR_ANY); //IP地址，需要进行网络序转换，INA
55      ser_addr.sin_port = htons(SERVER_PORT);  //端口号，需要网络序转换
56
57      ret = bind(server_fd, (struct sockaddr*)&ser_addr, sizeof(ser_addr));
58      if(ret < 0)
59      {
60          printf("socket bind fail!\n");
61          return -1;
62      }
```

```
63
64      handle_udp_msg(server_fd);    //处理接收到的数据
65
66      close(server_fd);
67      return 0;
68  }
```

- 客户端

```
1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <sys/socket.h>
4   #include <netinet/in.h>
5   #include <string.h>
6   #include <unistd.h>
7   #define SERVER_PORT 8888
8   #define BUFF_LEN 512
9   #define SERVER_IP "172.0.5.182"
10
11  void udp_msg_sender(int fd, struct sockaddr* dst)
12  {
13
14      socklen_t len;
15      struct sockaddr_in src;
16      while(1)
17      {
18          char buf[BUFF_LEN] = "TEST UDP MSG!\n";
19          len = sizeof(*dst);
20          printf("client:%s\n",buf);  //打印自己发送的信息
21          sendto(fd, buf, BUFF_LEN, 0, dst, len);
22          memset(buf, 0, BUFF_LEN);
23          recvfrom(fd, buf, BUFF_LEN, 0, (struct sockaddr*)&src, &len);  //接收来自
24          printf("server:%s\n",buf);
25          sleep(1);  //一秒发送一次消息
26      }
27  }
28
29  /*
30      client:
31              socket-->sendto-->revcfrom-->close
32  */
33
34  int main(int argc, char* argv[])
35  {
36      int client_fd;
```

```
37      struct sockaddr_in ser_addr;
38
39      client_fd = socket(AF_INET, SOCK_DGRAM, 0);
40      if(client_fd < 0)
41      {
42          printf("create socket fail!\n");
43          return -1;
44      }
45
46      memset(&ser_addr, 0, sizeof(ser_addr));
47      ser_addr.sin_family = AF_INET;
48      //ser_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
49      ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);  //注意网络序转换
50      ser_addr.sin_port = htons(SERVER_PORT);  //注意网络序转换
51
52      udp_msg_sender(client_fd, (struct sockaddr*)&ser_addr);
53
54      close(client_fd);
55
56      return 0;
57  }
```

生产者消费者

条件变量

- 条件变量不是锁，但是可以造成线程阻塞，通常与互斥锁配合。

中断

- 硬中断
- 软中断

线程

- 进程是资源分配的单位
- 线程是CPU调度的单位

定义

- 线程是进程中的一个执行流程
- TCB：线程控制块。（可以放在用户态，也可以放在内核态，如windows是放在内核态的）
- 在用户空间实现的线程机制，不依赖于操作系统的内核，由一组用户级的线程库函数来完成线程的创建，销毁，同步和调度。自定义线程调度算法，无序切换用户态和内核态。

    - 缺点：
        - 阻塞系统调用，则整个进程在等待。
        - 一个线程除非主动交出CPU使用权，否则它所在的进程中的其他线程将无法运行。
        - 每个线程时间片更少。
- 内核线程（Windows）
    - 用户态和内核态切换开销。
    - 粒度更小。
- 轻量级进程LWP（Solaris/Linux）
    - 是内核支持的用户级线程。
    - 一个进程可以有一个或多个轻量级进程，每个轻量级进程由一个单独的内核线程来支持。
- 独占
    - 寄存器
    - 栈
    - 线程优先级
- 共享
    - code代码段
    - data数据段
    - file文件

优点

- 多线程共享资源
- 并发执行
- 开销更小
    - 创建，销毁，上下文切换，通信可以不依赖于内核。

缺点

- 一个线程崩溃，会导致所属进程的所有线程崩溃。

进程

- 两个角度
    - 资源
    - 运行

应用场景

- 强调性能，用线程。
- 网页线程崩溃，用进程chrome。

上下文切换

- 保存在PCB，汇编实现，需要硬件支持。

时间函数time

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

int main(){
    time_t timer;
    time(&timer);
    printf("current time = %s\n", ctime(&timer));
    return(0);
}
```

参考资料

- 黑马程序员B站课程
- 《Linux内核设计与实现》
- 《Linux高性能服务器编程》游双
- 《后台开发：核心技术与应用实践》 徐晓鑫
- CMU15-445
- 零拷贝技术
- CAS锁机制
- 内存结构图片参考博客
- 英文图片参考博客
- 正月点灯笼Linux命令教程
- 进程互斥文件锁
- 本地套接字
- UDP