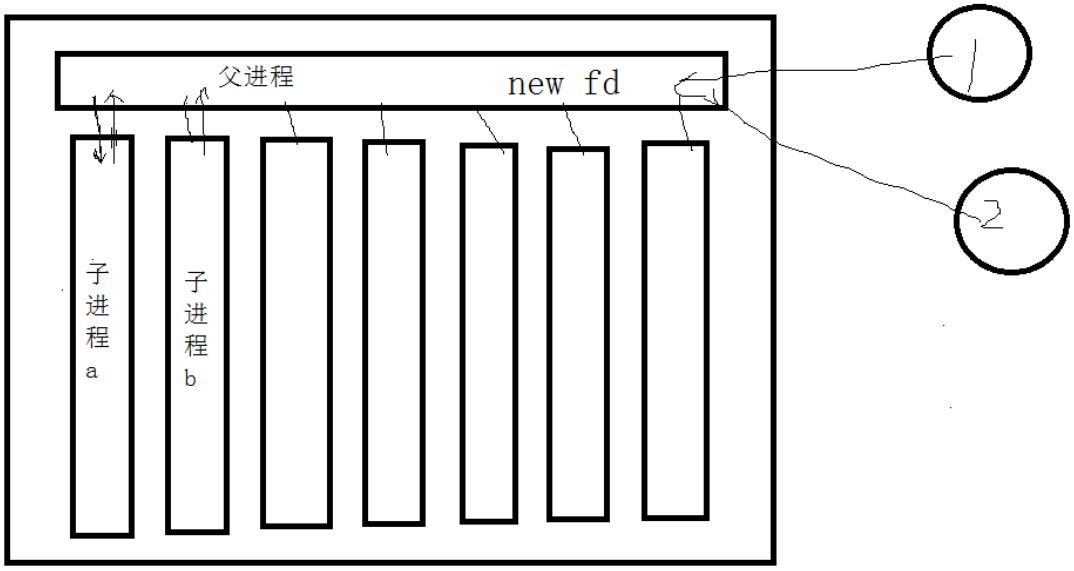
1、进程池

目的：实现多个客户端同时下载文件
流程：客户端连接服务器，连接成功后传输文件，传输完毕
服务器断开连接

1.1 进程池工作流程



如上图所示，首先我们通过父进程创建了很多个子进程，每个子进程与父进程直接都有一条全双工的管道，父进程是我们的代理，当 1 号客户端请求连接下载文件时，父进程接收到请求，产生 new_fd，并把 new_fd 发送给非忙碌的子进程 a，由子进程 a 将文件传输给 1 号客户端。这时 2 号客户端请求下载文件，父进程接收请求得到 new_fd,由于这时子进程 a 忙碌，所以将 new_fd 发送给子进程 b，由子进程 b 负责给 2 号客户端下载文件。

1.2 主要数据结构

父进程管理子进程所使用的数据结构

```
typedef struct{
        pid_t pid;//子进程的 pid
        int fd;//管道的一端
        short busy;//代表子进程是否忙碌，0 代表非忙碌，1 代表忙碌
}process_data;
```

创建多少个子进程，我们就用多少个对应的结构体管理子进程。

1.3 进程池代码编写流程

第一步：

make_child 函数初始化子进程

循环创建子进程，并初始化父进程的子进程管理结构体数组 parr，通过 socket_pair 将 socket 描述符一端放入数组

第二步

子进程流程，目前让子进程死循环，接收任务，给客户端发文件，然后通知父进程完成任务，退出机制暂时先不考虑

```
While（1）
{
    Recv_fd 等待父进程发送任务
    Hand_request 发送文件数据
    Write 向父进程发送完成任务
}
```

第三步：

父进程 epoll 监控 fd_listen 描述符。

父进程 epoll 监控 parr 结构体数组的 socket 描述符，结构体数组中的描述符是每一个子进程的管道对端，通过监控这个，当子进程通过 write 向我们写通知时，我们就知道子进程非忙碌了。

第四步：

While 1 启动 epoll_wait，等待是否有客户端连接

有客户端连接后，accept 获得描述符，循环找到非忙碌的子进程，并发送给子进程，标记对应子进程忙碌。

当子进程完成任务后，父进程一旦监控 socket 描述符可读，代表子进程非忙碌，然后标记子进程非忙碌。

1.4 进程间传递文件描述符（难点）

第一步，初始化 socketpair 类型描述符

```
int fds[2];
socketpair(AF_LOCAL,SOCK_STREAM,0,fds);
```

第二步：sendmsg 发送描述符

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

使用的 sockfd 即 sockpair 初始化的描述符 fds[1];

1)定义结构体　struct msghdr msg;
Sendmsg 关键是初始化 msghdr 结构体
struct msghdr {
　　void　　*msg_name;　　/* optional address */　没用
　　socklen_t　　msg_namelen;　　/* size of address */ 没用
　　struct iovec *msg_iov;　　/* scatter/gather array */ 没用
　　size_t　　msg_iovlen;　　/* # elements in msg_iov */ 没用
　　void　　*msg_control;　　/* ancillary data, see below */ 关键，即下面的 cmsghdr 结构体地址
　　size_t　　msg_controllen; /* ancillary data buffer len */　cmsghdr 结构体的长度
　　int　　msg_flags;　　/* flags (unused) */ 没用
　};

iovec 必须赋值
Cmsg 构造结构体 cmsghdr
man cmsg
得到如下信息：
struct cmsghdr {
　　socklen_t cmsg_len;　　/* data byte count, including header */
　　int　　cmsg_level;　/* originating protocol */
　　int　　cmsg_type;　　/* protocol-specific type */
　　/* followed by unsigned char cmsg_data[]; */
　};

首先定义　struct cmsghdr *cmsg 指针
cmsg_len 中存取 cmsghdr 结构体的长度，通过 CMSG_LEN 进行计算，我们传递的 fd 的大小为整型四个字节，所以

Int len = CMSG_LEN(sizeof(int));
然后为结构体申请空间：
cmsg = (struct cmsghdr *)calloc(1,len);
Cmsg->cmsg_len = len;
cmsg->cmsg_level =SOL_SOCKET;
Cmsg->cmsg_type = SCM_RIGHTS;
int *fdptr;
fdptr= (int *) CMSG_DATA(cmsg);
*fdptr = fd;
最后就可以通过 sendmsg 来发送文件描述符

第三步：
Recvmsg 接收文件描述符，接收的　msghdr 结构体初始化和 sendmsg 几乎完全一致，区别如下：

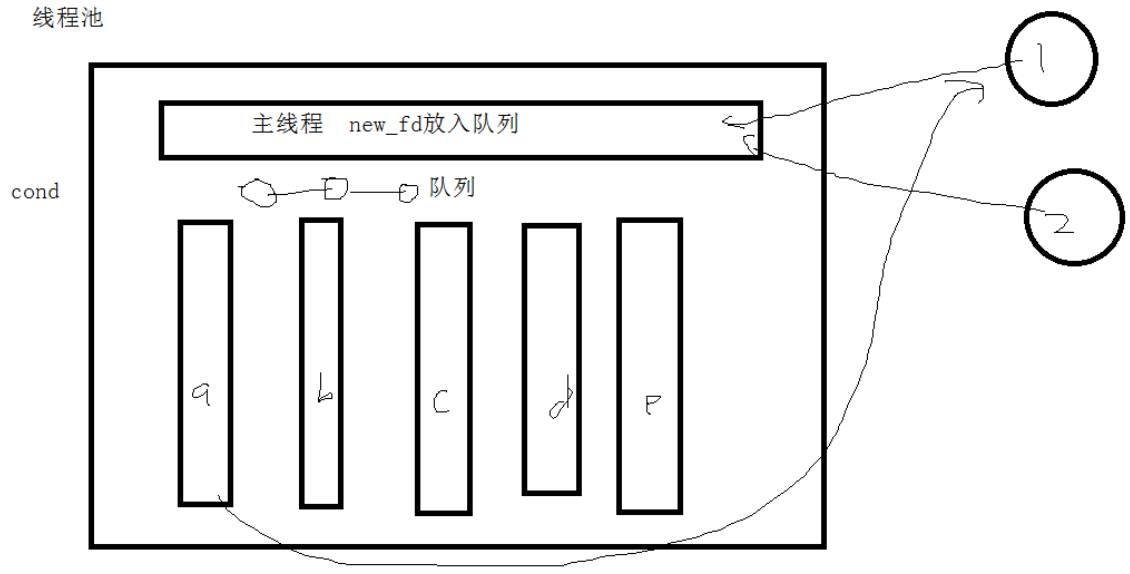
*fd = *fdptr;

1.5 子进程采用变长结构体发送文件（难点）

由于实际我们发送的文件中可能是字符串，可能是音频，可能是视频，所以发送时，对方要知道多少数据，我们必须采用控制数据，这就是我们的应用层协议设计，这里叫其小火车，每次火车头 data_len，记录火车 buf 中到底装载了多少数据发到对端。

```
typedef struct{
    int data_len;//控制数据，火车头，记录火车装载内容长度
    char buf[1000];//火车车厢
}train;
```

2、线程池



服务器创建子线程，子线程去队列里取任务，由于一开始队列为空，所以所有的子线程都睡觉，睡在条件变量 cond。客户端连接，主线程 accept 后将 new_fd 放入队列，放之前需要加锁，放入队列后，执行 signal，这样就会唤醒一个子线程，子线程被唤醒后，就去队列里拿 new_fd，拿到后，给客户端发送文件。

这是一个经典的生成者，消费者模型，我们的主线程是生成者，子线程是消费者。

1.初始化线程池

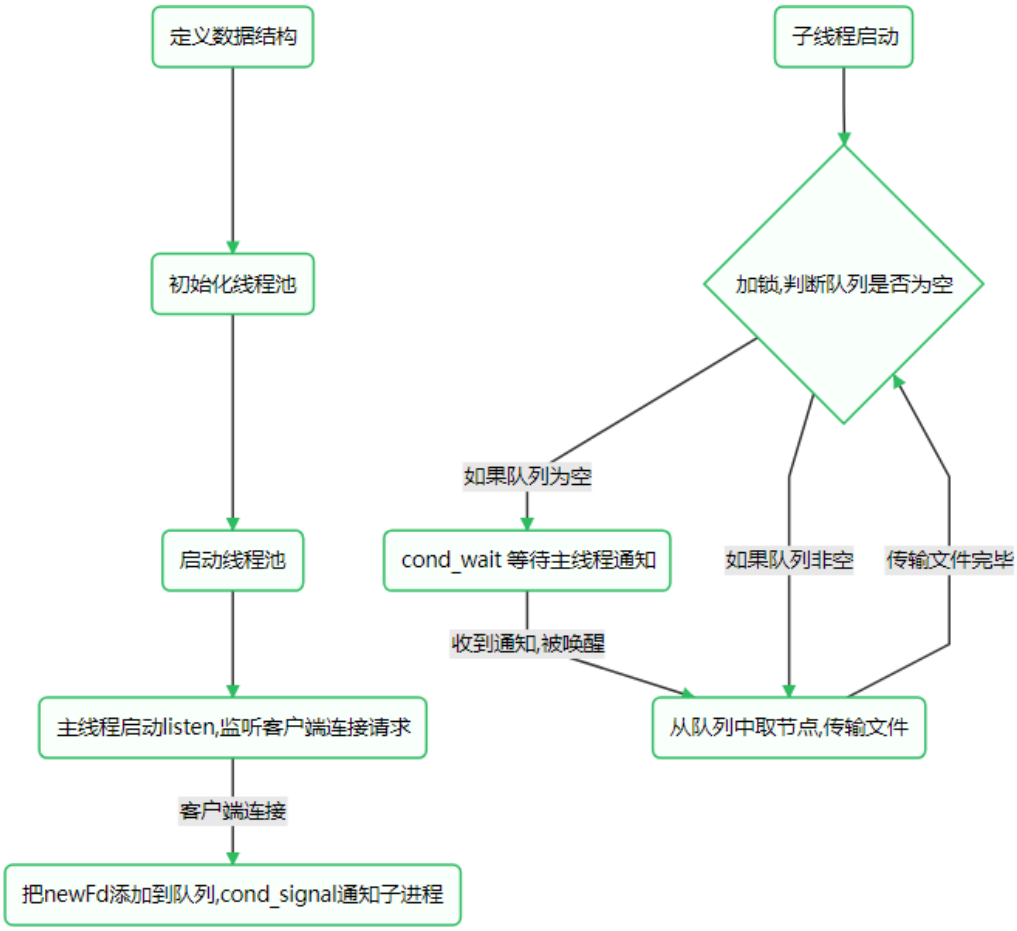
初始化队列，队列头，队列尾初始化，队列能力初始化（队列长度），队列锁
初始化线程池条件变量
给子线程赋入口函数

为线程池的子线程的线程 ID 申请空间
线程池是否启动标志初始化为 0
2.启动线程池

循环启动子线程，设置线程池启动标志设置为 1

3.主线程启动 listen，接收客户端请求

加锁，然后将接收到的 rs_fd 放入到 que 队列中，解锁，pthread_cond_signal 等待在对应条件上的子线程



队列使用的结构体及操作

```
#ifndef __WORK_QUE_H__
#define __WORK_QUE_H__
#include "head.h"
typedef struct tag_node
{
        int nd_sockfd ;
```

```
        struct tag_node* nd_next ;
}node_t, *pnode_t;//元素结构体，存储实际 client fd
typedef struct tag_que
{
                pnode_t que_head, que_tail;
                int que_capacity;
                int que_size;
                pthread_mutex_t que_mutex ;
}que_t, *pque_t;//描述队列的结构体
void factory_que_init(pque_t pq, int capacity);
void factory_que_set(pque_t pq, pnode_t pnew);
void factory_que_get(pque_t pq, pnode_t* p);
void factory_que_destroy(pque_t pq);
int factory_que_full(pque_t pq);
int factory_que_empty(pque_t pq);
#endif

typedef struct{
    Que_t que;
    pthread_cond_t cond;
    pthread_t *pthid;
    int threadNum;
    int startFlag;
}Factory_t,*pFactory_t;

int factoryInit(pFactory_t,int,int);
int factoryStart(pFactory_t);
```