

**学生实验报告册**

**第X次实验报告**

|  |  |
| --- | --- |
| 学年学期： | 2024 -2025 学年 🞎春🗹秋学期 |
| 课程名称： | 操作系统 |
| 学生学院： | 国际学院 |
| 专业班级： | 34082201 |
| 学生学号： | 2022214961 |
| 学生姓名： | 周明宇 |
| 联系电话： | 13329148059 |

**重庆邮电大学教务处制**

|  |  |  |  |
| --- | --- | --- | --- |
| **课程名称** | **计算机网络** | **课程编号** | **A2130350** |
| **实验地点** | **综合实验楼C410/C411** | | |
| **实验名称** | **Operating Systems Project2** | | |
| 1. **实验目的** 2. 掌握多线程编程：通过实现多个线程（charA、charE、charI、charO、charU、digit、writer）协作处理数据，理解线程同步与通信机制（如信号量、互斥锁）。 3. 理解进程间通信：通过创建子进程（ServerDecoder和ServerEncoder）和线程间消息队列，学习进程与线程的分工协作。 4. 实践Socket编程：实现基于TCP的客户端-服务器通信模型，掌握网络数据传输的基本流程。 5. 强化资源管理：学习动态内存分配、文件操作及线程/进程的创建与销毁，避免资源泄漏。 6. **实验内容**   **①实验内容简述**  本次实验的内容是开发一个客户端/服务器应用程序，使用Linux TCP套接字和C编程语言，处理字符串数据并实现多线程任务协作。  **②服务器端要求**  服务器端将接受来自客户端的多个请求，每个请求都将启动两个子进程：   1. **ServerDecoder**进程：从套接字读取数据并创建7个线程。每个线程的功能如下：    * **charA**线程：将小写字母a替换为大写字母A，并将处理后的数据传递给charE线程。    * **charE**线程：将小写字母e替换为大写字母E，并将处理后的数据传递给charI线程。    * **charI**线程：将小写字母i替换为大写字母I，并将处理后的数据传递给charO线程。    * **charO**线程：将小写字母o替换为大写字母O，并将处理后的数据传递给charU线程。    * **charU**线程：将小写字母u替换为大写字母U，并将处理后的数据传递给digit线程。    * **digit**线程：计算文本中的所有数字之和，并将结果附加到处理后的文本中，然后将修改后的数据传递给writer线程。    * **writer**线程：将最终处理的数据传递给serverEncoder进程。 2. **ServerEncoder**进程：将最终的数据发送回客户端。   **③客户端要求**  客户端将创建两个进程：  • **clientEncoder**进程：打开输入文件（如intext.txt），将数据写入套接字。  • **clientDecode**进程：从套接字读取数据并将解码后的数据写入文件（如result.txt）。  **④队列模块**  实现一个队列模块，存储字符字符串，并实现以下特性：  • 队列的大小为10。  • 当输入数据结束时，线程应当终止。  **⑤输入文件**   1. **实验步骤及方案**   **①流程图**  图1系统流程图  **②数据流图**  图2数据流图  **③项目总体框架**  图3框架图  **④代码细节**  **宏定义与头文件**  #include <arpa/inet.h>  #include <ctype.h>  #include <pthread.h>  #include <semaphore.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/socket.h>  #include <unistd.h>  #define **MAX\_QUEUE\_SIZE** 10  #define **BUFFER\_SIZE** 1024   * 引入必要的标准库和系统库（socket通信、字符串处理、多线程、信号量等）。 * 定义两个宏：MAX\_QUEUE\_SIZE：每个消息队列最多保存10条消息。BUFFER\_SIZE：接收数据缓冲区大小。   **消息队列结构体**  typedef struct {      char\* data[MAX\_QUEUE\_SIZE];      int head;      int tail;      pthread\_mutex\_t mutex;      sem\_t empty;      sem\_t full;  } MessageQueue;   * 环形队列，用于线程间通信（生产者-消费者模型）。 * data[]：保存字符串消息。 * head/tail：队头/队尾索引。 * mutex：互斥锁，保护数据并发访问。 * empty/full：信号量，控制资源可用性。   **queue\_init**  void **queue\_init**(queue\_t \*queue) {      queue->front = NULL;      queue->rear = NULL;  **pthread\_mutex\_init**(&queue->mutex, NULL);  **pthread\_cond\_init**(&queue->cond, NULL);  }  初始化一个空队列   * front和rear都设为NULL，表示队列为空。 * 初始化互斥锁mutex用于线程间同步。 * 初始化条件变量cond用于在无元素可读时阻塞消费者线程。   **queue\_push**  void **queue\_push**(queue\_t \*queue, const char \*msg) {      node\_t \*new\_node = **malloc**(sizeof(node\_t));      new\_node->msg = **strdup**(msg);  // 深拷贝字符串，防止外部改变      new\_node->next = NULL;  **pthread\_mutex\_lock**(&queue->mutex);      if (queue->rear == NULL) {          queue->front = new\_node;          queue->rear = new\_node;      } else {          queue->rear->next = new\_node;          queue->rear = new\_node;      }  **pthread\_cond\_signal**(&queue->cond);  // 通知等待的消费者  **pthread\_mutex\_unlock**(&queue->mutex);  }  将一条消息加入队列尾部   * 使用strdup深拷贝消息，避免指针悬挂。 * 使用互斥锁保护对队列的并发访问。 * 如果队列为空，则新节点为front和rear。 * 否则更新rear->next并将rear指向新节点。 * 最后pthread\_cond\_signal唤醒等待的消费者线程。   **queue\_pop**  char \***queue\_pop**(queue\_t \*queue) {  **pthread\_mutex\_lock**(&queue->mutex);      while (queue->front == NULL) {  **pthread\_cond\_wait**(&queue->cond, &queue->mutex);      }      node\_t \*temp = queue->front;      char \*msg = temp->msg;      queue->front = temp->next;      if (queue->front == NULL) {          queue->rear = NULL;      }  **free**(temp);  **pthread\_mutex\_unlock**(&queue->mutex);      return msg;  }  从队列头部取出一条消息（阻塞等待机制）   * 进入临界区后，若队列为空，则调用pthread\_cond\_wait等待。 * 一旦有消息，将其取出，并更新front。 * 如果取出的是最后一个元素，设置rear=NULL。 * 返回取出的消息（注意：msg是调用者要free的）。   **queue\_is\_empty**  int **queue\_is\_empty**(queue\_t \*queue) {  **pthread\_mutex\_lock**(&queue->mutex);      int empty = (queue->front == NULL);  **pthread\_mutex\_unlock**(&queue->mutex);      return empty;  }  判断队列是否为空   * 使用互斥锁保证线程安全。 * 可用于主线程判断所有队列是否处理完毕，以关闭服务。   **queue\_destroy**  void **queue\_destroy**(queue\_t \*queue) {  **pthread\_mutex\_lock**(&queue->mutex);      node\_t \*current = queue->front;      while (current != NULL) {          node\_t \*temp = current;          current = current->next;  **free**(temp->msg);  **free**(temp);      }  **pthread\_mutex\_unlock**(&queue->mutex);  **pthread\_mutex\_destroy**(&queue->mutex);  **pthread\_cond\_destroy**(&queue->cond);  }  释放队列中所有节点内存，并销毁锁和条件变量   * 清空链表中的所有节点。 * 分别销毁mutex和cond，避免内存泄漏。   **线程参数结构**  typedef struct {      MessageQueue\* in\_queue;      MessageQueue\* out\_queue;  } ThreadData;  表示：该线程从in\_queue中取数据，处理后放入out\_queue。  **charX\_thread一类函数**  void \***charA\_thread**(void \*arg) {      thread\_arg\_t \*args = (thread\_arg\_t \*)arg;      const char \*input = args->input;      int count = 0;      for (int i = 0; input[i] != '\0'; i++) {          if (input[i] == 'A' || input[i] == 'a') {              count++;          }      }      args->charA\_count = count;  **pthread\_exit**(NULL);  }  扫描字符串，统计其中大小写的X字符的数量，或者执行替换/过滤等操作   * 通过arg传入字符串； * 使用args->charA\_count共享统计结果； * 无需加锁（如果每个线程只操作自己独占的数据）； * 最终通过pthread\_join()在主线程收集结果。   **digit\_thread**  void \***digit\_thread**(void \*arg) {      thread\_arg\_t \*args = (thread\_arg\_t \*)arg;      const char \*input = args->input;      int digit\_count = 0;      char digits\_only[1024] = {0};      int pos = 0;      for (int i = 0; input[i] != '\0'; i++) {          if (**isdigit**((unsigned char)input[i])) {              digits\_only[pos++] = input[i];              digit\_count++;          }      }      digits\_only[pos] = '\0';  **strcpy**(args->digits, digits\_only);      args->digit\_count = digit\_count;  **pthread\_exit**(NULL);  }  识别字符串中的数字字符；可能统计数量、将其提取出来，或做数值运算。   * 使用标准库函数isdigit()识别数字； * 使用一个临时字符串缓冲区保存数字字符； * 最终将结果拷贝回共享结构体中的args->digits； * 字符缓冲区大小要预留足够空间防止溢出。   **writer\_thread**  void \***writer\_thread**(void \*arg) {      thread\_arg\_t \*args = (thread\_arg\_t \*)arg;      FILE \*fp = **fopen**("output.txt", "w");      if (!fp) {  **perror**("fopen");  **pthread\_exit**(NULL);      }  **fprintf**(fp, "Number of 'A' or 'a': %d\n", args->charA\_count);  **fprintf**(fp, "Digits found: %s\n", args->digits);  **fprintf**(fp, "Digit count: %d\n", args->digit\_count);  **fclose**(fp);  **pthread\_exit**(NULL);  }  收集其它线程处理结果；将处理结果写入文件或标准输出。   * 使用fopen()打开输出文件； * 从结构体中读取其他线程填入的数据； * 使用fprintf()写入结果； * 错误处理不能省略。   **handle\_client**  /\* 处理客户端连接的函数 \*/  void **handle\_client**(int sockfd) {      // 初始化所有队列      MessageQueue queue\_a, queue\_e, queue\_i, queue\_o, queue\_u, queue\_digit, queue\_writer;  **queue\_init**(&queue\_a);  **queue\_init**(&queue\_e);  **queue\_init**(&queue\_i);  **queue\_init**(&queue\_o);  **queue\_init**(&queue\_u);  **queue\_init**(&queue\_digit);  **queue\_init**(&queue\_writer);        // 创建线程参数      ThreadData data\_a = {&queue\_a, &queue\_e};      ThreadData data\_e = {&queue\_e, &queue\_i};      ThreadData data\_i = {&queue\_i, &queue\_o};      ThreadData data\_o = {&queue\_o, &queue\_u};      ThreadData data\_u = {&queue\_u, &queue\_digit};      ThreadData data\_digit = {&queue\_digit, &queue\_writer};      ThreadData data\_writer = {&queue\_writer, (MessageQueue\*)&sockfd}; // 复用传递sockfd        // 创建所有线程      pthread\_t tid\_a, tid\_e, tid\_i, tid\_o, tid\_u, tid\_digit, tid\_writer;  **pthread\_create**(&tid\_a, NULL, charA\_thread, &data\_a);  **pthread\_create**(&tid\_e, NULL, charE\_thread, &data\_e);  **pthread\_create**(&tid\_i, NULL, charI\_thread, &data\_i);  **pthread\_create**(&tid\_o, NULL, charO\_thread, &data\_o);  **pthread\_create**(&tid\_u, NULL, charU\_thread, &data\_u);  **pthread\_create**(&tid\_digit, NULL, digit\_thread, &data\_digit);  **pthread\_create**(&tid\_writer, NULL, writer\_thread, &data\_writer);        // 从socket读取数据并送入处理管道      char buffer[BUFFER\_SIZE];      int n;      while ((n = **read**(sockfd, buffer, BUFFER\_SIZE-1)) > 0) {          buffer[n] = '\0';          char\* msg = **strdup**(buffer);  **queue\_push**(&queue\_a, msg);      }        // 发送终止信号  **queue\_push**(&queue\_a, NULL);        // 等待所有线程完成  **pthread\_join**(tid\_a, NULL);  **pthread\_join**(tid\_e, NULL);  **pthread\_join**(tid\_i, NULL);  **pthread\_join**(tid\_o, NULL);  **pthread\_join**(tid\_u, NULL);  **pthread\_join**(tid\_digit, NULL);  **pthread\_join**(tid\_writer, NULL);    **close**(sockfd);  }  **1.初始化所有队列**   * 使用 queue\_init 函数初始化多个队列（queue\_a,queue\_e,queue\_i,queue\_o, queue\_u,queue\_digit,queue\_writer），这些队列用于存放不同类型的字符数据。 * 队列按顺序依次处理字符数据，最终由writer\_thread将结果写入文件或标准输出。   **2.创建线程**   * 为每种字符类型创建一个线程。每个线程都会处理来自前一个线程队列的数据，并将处理结果推送到下一个队列。 * 每个线程传入一个ThreadData结构体，包含输入队列和输出队列的指针。   **3.从客户端读取数据**   * 使用read函数从客户端读取数据并存入缓冲区buffer。 * 读取到的数据会被转换为字符串，并通过queue\_push函数推送到queue\_a队列，供第一个字符处理线程处理。   **4.发送终止信号**   * 当客户端发送的数据全部处理完后，通过queue\_push向queue\_a推送一个 NULL 指针，作为结束信号，告知所有线程停止处理。   **5.等待所有线程完成**   * 使用pthread\_join函数等待所有线程的结束，确保所有的处理完成后再关闭连接。   **6.关闭客户端连接**   * 调用close(sockfd)关闭客户端连接，释放资源。   **服务器端Main**  int **main**(int argc, char\* argv[]) {      if (argc < 2) {  **fprintf**(stderr, "Usage: %s <port>\n", argv[0]);  **exit**(1);      }        int sockfd = **socket**(AF\_INET, SOCK\_STREAM, 0);      if (sockfd < 0) {  **perror**("socket");  **exit**(1);      }        struct sockaddr\_in serv\_addr;  **memset**(&serv\_addr, 0, sizeof(serv\_addr));      serv\_addr.sin\_family = AF\_INET;      serv\_addr.sin\_addr.s\_addr = INADDR\_ANY;      serv\_addr.sin\_port = **htons**(**atoi**(argv[1]));        if (**bind**(sockfd, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0) {  **perror**("bind");  **exit**(1);      }    **listen**(sockfd, 5);  **printf**("Server listening on port %s\n", argv[1]);        while (1) {          struct sockaddr\_in cli\_addr;          socklen\_t clilen = sizeof(cli\_addr);          int newsockfd = **accept**(sockfd, (struct sockaddr\*)&cli\_addr, &clilen);          if (newsockfd < 0) {  **perror**("accept");              continue;          }    **printf**("New client connected\n");  **handle\_client**(newsockfd);      }    **close**(sockfd);      return 0;  }  1.**检查命令行参数**   * main函数首先检查命令行参数，确保传入了端口号。 * 如果没有提供端口号，程序会输出使用说明并退出。   **2.创建服务器 socket**   * 使用socket函数创建一个新的TCP socket，sockfd保存该socket描述符。   **3.绑定服务器地址**   * 使用bind函数将创建的socket与指定的端口绑定。服务器地址 (serv\_addr) 使用INADDR\_ANY，表示接受来自任何 IP 地址的连接。   4.**监听连接请求**   * 使用listen函数将服务器socket设置为监听状态，等待客户端连接请求。5 表示最大等待队列大小。   **5.接受客户端连接**   * 在主循环中，accept函数被调用来接收客户端的连接。每当一个新的客户端连接时，accept返回一个新的socket描述符newsockfd。   **6.处理客户端请求**   * 对于每个客户端连接，调用handle\_client函数来处理连接。 * handle\_client函数会负责所有的队列初始化、线程创建以及数据读取与处理。   **7.关闭服务器 socket**   * 当程序退出时，使用close关闭服务器的监听socket。   **客户端程序实现细节**  int **main**(int argc, char\* argv[]) {      if (argc < 5) {  **fprintf**(stderr, "Usage: %s <hostname> <port> <input\_file> <output\_file>\n", argv[0]);  **exit**(1);      }      const char\* hostname = argv[1];      int port = **atoi**(argv[2]);      const char\* input\_file = argv[3];      const char\* output\_file = argv[4];      // 创建 socket      int sockfd = **socket**(AF\_INET, SOCK\_STREAM, 0);      if (sockfd < 0) {  **perror**("socket");  **exit**(1);      }      struct hostent\* server = **gethostbyname**(hostname);      if (server == NULL) {  **fprintf**(stderr, "No such host\n");  **exit**(1);      }      struct sockaddr\_in serv\_addr;  **memset**(&serv\_addr, 0, sizeof(serv\_addr));      serv\_addr.sin\_family = AF\_INET;  **memcpy**(&serv\_addr.sin\_addr.s\_addr, server->h\_addr, server->h\_length);      serv\_addr.sin\_port = **htons**(port);      if (**connect**(sockfd, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0) {  **perror**("connect");  **exit**(1);      }      // 创建子进程      pid\_t pid = **fork**();      if (pid < 0) {  **perror**("fork");  **exit**(1);      }      if (pid == 0) {          // 子进程 - clientEncoder: 读文件 -> socket          FILE\* infile = **fopen**(input\_file, "r");          if (!infile) {  **perror**("fopen input\_file");  **close**(sockfd);  **exit**(1);          }          char buffer[BUFFER\_SIZE];          size\_t n;          while ((n = **fread**(buffer, 1, BUFFER\_SIZE, infile)) > 0) {              if (**write**(sockfd, buffer, n) < 0) {  **perror**("write to socket");                  break;              }          }  **fclose**(infile);  **shutdown**(sockfd, SHUT\_WR); // 关闭写通道，通知对方EOF  **exit**(0);      } else {          // 父进程 - clientDecoder: socket -> 写文件          FILE\* outfile = **fopen**(output\_file, "w");          if (!outfile) {  **perror**("fopen output\_file");  **close**(sockfd);  **exit**(1);          }          char buffer[BUFFER\_SIZE];          ssize\_t n;          while ((n = **read**(sockfd, buffer, BUFFER\_SIZE)) > 0) {              if (**fwrite**(buffer, 1, n, outfile) < (size\_t)n) {  **perror**("fwrite");                  break;              }          }  **fclose**(outfile);  **close**(sockfd);  **wait**(NULL); // 等待子进程结束      }      return 0;  }  **1. 命令行参数解析**   * 程序通过命令行参数获取以下信息：   + hostname：服务器的主机名或 IP 地址。   + port：服务器的端口号。   + input\_file：要发送的输入文件路径。   + output\_file：接收服务器响应并保存到的输出文件路径。 * 如果命令行参数不足，则提示使用方法并退出程序。   **2. 创建和连接 socket**   * 使用socket()函数创建一个 TCP socket。 * 使用gethostbyname()获取服务器主机名对应的IP地址。 * 设置服务器的地址结构sockaddr\_in，包括：   + sin\_family：地址族，指定为 IPv4。   + sin\_addr.s\_addr：通过gethostbyname获取的服务器IP地址。   + sin\_port：通过htons()转换端口号。 * 使用connect()建立与服务器的连接。   **3. 创建子进程**   * 使用fork()创建一个子进程，该进程负责将输入文件数据发送给服务器。 * 父进程负责接收服务器返回的数据，并将其写入到输出文件。   **4. 子进程 - 读取文件并发送数据**   * 子进程打开输入文件input\_file，并读取其中的数据。 * 使用write()系统调用将数据逐块发送到服务器。 * 每次读取 BUFFER\_SIZE 大小的数据块，通过write()写入到已连接的服务器 socket中。 * 一旦文件读取完毕，调用 shutdown(sockfd, SHUT\_WR) 关闭socket的写通道，表示发送完毕并通知服务器EOF（End of File）。   **5. 父进程 - 从服务器接收数据并写入文件**   * 父进程打开输出文件output\_file。 * 使用read()系统调用从socket中读取服务器返回的数据。 * 逐块写入到输出文件中。 * 如果写入文件时发生错误，会调用perror()输出错误信息。 * 通过wait(NULL) 等待子进程结束。   **6. 关闭资源**   * 子进程完成文件传输后退出。 * 父进程完成文件写入后，关闭文件并退出。  1. **结果及分析**   **①编译**  图4编译示意图  **②运行**  首先运行服务器端，命令行指定端口  图5服务器运行  再运行客户端，命令行参数指定端口和输入文件路径以及输出路径  图6客户端运行  此时服务器端会提示建立了连接  图7服务器端反馈  **③结果**  输出文件内容如下  图8 output.txt结果  程序成功地将所有小写的元音字符转换为大写，并且在末尾添加了文本中所有数字之和   1. **心得体会**   通过本次实验，我对**Socket编程、多线程通信以及进程协作机制**有了更加深入的理解和实践体会。实验要求实现一个多线程的数据处理系统，并结合客户端-服务器架构，这不仅考察了我对TCP套接字通信的掌握程度，也锻炼了我对多线程编程与同步机制的应用能力。  在服务器端，七个线程依次处理数据的不同阶段，通过**线程间共享队列**实现流水线式的数据流转，这种分工协作的设计让我更直观地理解了线程并发处理的优势和挑战。特别是在处理共享资源时，合理使用信号量（semaphore）或互斥锁（mutex）对队列进行同步，确保数据一致性，是整个系统稳定运行的关键。  此外，我也体会到了**进程和线程之间的职责划分**。服务器通过创建两个子进程解耦了读写任务，每个进程再通过线程完成具体的字符替换、数字统计等功能，实现了功能模块的清晰划分。客户端同样采用双进程结构来实现输入输出的并行操作，这让我对多进程编程的使用场景有了更深的认识。  虽然在开发过程中遇到了一些困难，如**线程间通信的阻塞问题、Socket数据边界处理、线程同步等**，但在不断调试和查阅资料的过程中，我收获了宝贵的编程经验，也进一步提高了自己分析问题和解决问题的能力。  总之，这次实验不仅加深了我对操作系统中进程、线程、同步机制的理解，也增强了我编写复杂C语言程序的综合能力。相信这次实践将为我今后进行更高层次的系统开发打下坚实基础。 | | | |
| **代码附录**  Server.c  #include <arpa/inet.h>  #include <ctype.h>  #include <pthread.h>  #include <semaphore.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/socket.h>  #include <unistd.h>  #define **MAX\_QUEUE\_SIZE** 10  #define **BUFFER\_SIZE** 1024  /\* 消息队列结构 \*/  typedef struct {      char\* data[MAX\_QUEUE\_SIZE];      int head;      int tail;      pthread\_mutex\_t mutex;      sem\_t empty;      sem\_t full;  } MessageQueue;  /\* 初始化队列 \*/  void **queue\_init**(MessageQueue\* q) {      q->head = 0;      q->tail = 0;  **pthread\_mutex\_init**(&q->mutex, NULL);  **sem\_init**(&q->empty, 0, MAX\_QUEUE\_SIZE);  **sem\_init**(&q->full, 0, 0);  }  /\* 向队列添加消息 \*/  void **queue\_push**(MessageQueue\* q, char\* message) {  **sem\_wait**(&q->empty);  **pthread\_mutex\_lock**(&q->mutex);        q->data[q->tail] = message;      q->tail = (q->tail + 1) % MAX\_QUEUE\_SIZE;    **pthread\_mutex\_unlock**(&q->mutex);  **sem\_post**(&q->full);  }  /\* 从队列获取消息 \*/  char\* **queue\_pop**(MessageQueue\* q) {  **sem\_wait**(&q->full);  **pthread\_mutex\_lock**(&q->mutex);        char\* message = q->data[q->head];      q->head = (q->head + 1) % MAX\_QUEUE\_SIZE;    **pthread\_mutex\_unlock**(&q->mutex);  **sem\_post**(&q->empty);        return message;  }  /\* 线程参数结构 \*/  typedef struct {      MessageQueue\* in\_queue;      MessageQueue\* out\_queue;  } ThreadData;  /\* charA线程 - 转换a为A \*/  void\* **charA\_thread**(void\* arg) {      ThreadData\* data = (ThreadData\*)arg;      char\* message;        while (1) {          message = **queue\_pop**(data->in\_queue);          if (message == NULL) {  **queue\_push**(data->out\_queue, NULL);              break;          }            for (int i = 0; message[i]; i++) {              if (message[i] == 'a') message[i] = 'A';          }    **queue\_push**(data->out\_queue, message);      }      return NULL;  }  /\* charE线程 - 转换e为E \*/  void\* **charE\_thread**(void\* arg) {      ThreadData\* data = (ThreadData\*)arg;      char\* message;        while (1) {          message = **queue\_pop**(data->in\_queue);          if (message == NULL) {  **queue\_push**(data->out\_queue, NULL);              break;          }            for (int i = 0; message[i]; i++) {              if (message[i] == 'e') message[i] = 'E';          }    **queue\_push**(data->out\_queue, message);      }      return NULL;  }  /\* charI线程 - 转换i为I \*/  void\* **charI\_thread**(void\* arg) {      ThreadData\* data = (ThreadData\*)arg;      char\* message;        while (1) {          message = **queue\_pop**(data->in\_queue);          if (message == NULL) {  **queue\_push**(data->out\_queue, NULL);              break;          }            for (int i = 0; message[i]; i++) {              if (message[i] == 'i') message[i] = 'I';          }    **queue\_push**(data->out\_queue, message);      }      return NULL;  }  /\* charO线程 - 转换o为O \*/  void\* **charO\_thread**(void\* arg) {      ThreadData\* data = (ThreadData\*)arg;      char\* message;        while (1) {          message = **queue\_pop**(data->in\_queue);          if (message == NULL) {  **queue\_push**(data->out\_queue, NULL);              break;          }            for (int i = 0; message[i]; i++) {              if (message[i] == 'o') message[i] = 'O';          }    **queue\_push**(data->out\_queue, message);      }      return NULL;  }  /\* charU线程 - 转换u为U \*/  void\* **charU\_thread**(void\* arg) {      ThreadData\* data = (ThreadData\*)arg;      char\* message;        while (1) {          message = **queue\_pop**(data->in\_queue);          if (message == NULL) {  **queue\_push**(data->out\_queue, NULL);              break;          }            for (int i = 0; message[i]; i++) {              if (message[i] == 'u') message[i] = 'U';          }    **queue\_push**(data->out\_queue, message);      }      return NULL;  }  /\* digit线程 - 统计数字并附加结果 \*/  void\* **digit\_thread**(void\* arg) {      ThreadData\* data = (ThreadData\*)arg;      char\* message;      int total\_sum = 0;        while (1) {          message = **queue\_pop**(data->in\_queue);          if (message == NULL) {              // 发送终止信号前附加最终总和              char\* final\_msg = **malloc**(100);  **queue\_push**(data->out\_queue, final\_msg);  **queue\_push**(data->out\_queue, NULL);              break;          }            // 计算数字总和          for (int i = 0; message[i]; i++) {              if (**isdigit**(message[i])) {                  total\_sum += message[i] - '0';              }          }            // 附加当前总和到消息          char\* new\_msg = **malloc**(**strlen**(message) + 50);  **sprintf**(new\_msg, "%s\nCurrent digit sum: %d", message, total\_sum);  **free**(message);    **queue\_push**(data->out\_queue, new\_msg);          total\_sum=0;      }      return NULL;  }  /\* writer线程 - 将结果写入socket \*/  void\* **writer\_thread**(void\* arg) {      ThreadData\* data = (ThreadData\*)arg;      int sockfd = \*(int\*)data->out\_queue; // 复用out\_queue传递socket fd      char\* message;        while (1) {          message = **queue\_pop**(data->in\_queue);          if (message == NULL) break;    **write**(sockfd, message, **strlen**(message));  **write**(sockfd, "\n", 1);  **free**(message);      }      return NULL;  }  /\* 处理客户端连接的函数 \*/  void **handle\_client**(int sockfd) {      // 初始化所有队列      MessageQueue queue\_a, queue\_e, queue\_i, queue\_o, queue\_u, queue\_digit, queue\_writer;  **queue\_init**(&queue\_a);  **queue\_init**(&queue\_e);  **queue\_init**(&queue\_i);  **queue\_init**(&queue\_o);  **queue\_init**(&queue\_u);  **queue\_init**(&queue\_digit);  **queue\_init**(&queue\_writer);        // 创建线程参数      ThreadData data\_a = {&queue\_a, &queue\_e};      ThreadData data\_e = {&queue\_e, &queue\_i};      ThreadData data\_i = {&queue\_i, &queue\_o};      ThreadData data\_o = {&queue\_o, &queue\_u};      ThreadData data\_u = {&queue\_u, &queue\_digit};      ThreadData data\_digit = {&queue\_digit, &queue\_writer};      ThreadData data\_writer = {&queue\_writer, (MessageQueue\*)&sockfd}; // 复用传递sockfd        // 创建所有线程      pthread\_t tid\_a, tid\_e, tid\_i, tid\_o, tid\_u, tid\_digit, tid\_writer;  **pthread\_create**(&tid\_a, NULL, charA\_thread, &data\_a);  **pthread\_create**(&tid\_e, NULL, charE\_thread, &data\_e);  **pthread\_create**(&tid\_i, NULL, charI\_thread, &data\_i);  **pthread\_create**(&tid\_o, NULL, charO\_thread, &data\_o);  **pthread\_create**(&tid\_u, NULL, charU\_thread, &data\_u);  **pthread\_create**(&tid\_digit, NULL, digit\_thread, &data\_digit);  **pthread\_create**(&tid\_writer, NULL, writer\_thread, &data\_writer);        // 从socket读取数据并送入处理管道      char buffer[BUFFER\_SIZE];      int n;      while ((n = **read**(sockfd, buffer, BUFFER\_SIZE-1)) > 0) {          buffer[n] = '\0';          char\* msg = **strdup**(buffer);  **queue\_push**(&queue\_a, msg);      }        // 发送终止信号  **queue\_push**(&queue\_a, NULL);        // 等待所有线程完成  **pthread\_join**(tid\_a, NULL);  **pthread\_join**(tid\_e, NULL);  **pthread\_join**(tid\_i, NULL);  **pthread\_join**(tid\_o, NULL);  **pthread\_join**(tid\_u, NULL);  **pthread\_join**(tid\_digit, NULL);  **pthread\_join**(tid\_writer, NULL);    **close**(sockfd);  }  int **main**(int argc, char\* argv[]) {      if (argc < 2) {  **fprintf**(stderr, "Usage: %s <port>\n", argv[0]);  **exit**(1);      }        int sockfd = **socket**(AF\_INET, SOCK\_STREAM, 0);      if (sockfd < 0) {  **perror**("socket");  **exit**(1);      }        struct sockaddr\_in serv\_addr;  **memset**(&serv\_addr, 0, sizeof(serv\_addr));      serv\_addr.sin\_family = AF\_INET;      serv\_addr.sin\_addr.s\_addr = INADDR\_ANY;      serv\_addr.sin\_port = **htons**(**atoi**(argv[1]));        if (**bind**(sockfd, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0) {  **perror**("bind");  **exit**(1);      }    **listen**(sockfd, 5);  **printf**("Server listening on port %s\n", argv[1]);        while (1) {          struct sockaddr\_in cli\_addr;          socklen\_t clilen = sizeof(cli\_addr);          int newsockfd = **accept**(sockfd, (struct sockaddr\*)&cli\_addr, &clilen);          if (newsockfd < 0) {  **perror**("accept");              continue;          }    **printf**("New client connected\n");  **handle\_client**(newsockfd);      }    **close**(sockfd);      return 0;  }  client.c  #include <netdb.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/socket.h>  #include <sys/types.h>  #include <unistd.h>  #include <fcntl.h>  #include <sys/wait.h>  #define **BUFFER\_SIZE** 1024  int **main**(int argc, char\* argv[]) {      if (argc < 5) {  **fprintf**(stderr, "Usage: %s <hostname> <port> <input\_file> <output\_file>\n", argv[0]);  **exit**(1);      }      const char\* hostname = argv[1];      int port = **atoi**(argv[2]);      const char\* input\_file = argv[3];      const char\* output\_file = argv[4];      // 创建 socket      int sockfd = **socket**(AF\_INET, SOCK\_STREAM, 0);      if (sockfd < 0) {  **perror**("socket");  **exit**(1);      }      struct hostent\* server = **gethostbyname**(hostname);      if (server == NULL) {  **fprintf**(stderr, "No such host\n");  **exit**(1);      }      struct sockaddr\_in serv\_addr;  **memset**(&serv\_addr, 0, sizeof(serv\_addr));      serv\_addr.sin\_family = AF\_INET;  **memcpy**(&serv\_addr.sin\_addr.s\_addr, server->h\_addr, server->h\_length);      serv\_addr.sin\_port = **htons**(port);      if (**connect**(sockfd, (struct sockaddr\*)&serv\_addr, sizeof(serv\_addr)) < 0) {  **perror**("connect");  **exit**(1);      }      // 创建子进程      pid\_t pid = **fork**();      if (pid < 0) {  **perror**("fork");  **exit**(1);      }      if (pid == 0) {          // 子进程 - clientEncoder: 读文件 -> socket          FILE\* infile = **fopen**(input\_file, "r");          if (!infile) {  **perror**("fopen input\_file");  **close**(sockfd);  **exit**(1);          }          char buffer[BUFFER\_SIZE];          size\_t n;          while ((n = **fread**(buffer, 1, BUFFER\_SIZE, infile)) > 0) {              if (**write**(sockfd, buffer, n) < 0) {  **perror**("write to socket");                  break;              }          }  **fclose**(infile);  **shutdown**(sockfd, SHUT\_WR); // 关闭写通道，通知对方EOF  **exit**(0);      } else {          // 父进程 - clientDecoder: socket -> 写文件          FILE\* outfile = **fopen**(output\_file, "w");          if (!outfile) {  **perror**("fopen output\_file");  **close**(sockfd);  **exit**(1);          }          char buffer[BUFFER\_SIZE];          ssize\_t n;          while ((n = **read**(sockfd, buffer, BUFFER\_SIZE)) > 0) {              if (**fwrite**(buffer, 1, n, outfile) < (size\_t)n) {  **perror**("fwrite");                  break;              }          }  **fclose**(outfile);  **close**(sockfd);  **wait**(NULL); // 等待子进程结束      }      return 0;  } | | | |