

**学生实验报告册**

|  |  |
| --- | --- |
| 学年学期： | 2024 -2025 学年 🞎春🗹秋学期 |
| 课程名称： | 计算机网络 |
| 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. 理解Linux进程管理机制，掌握fork()、exec()系列函数的使用。 3. 掌握进程间通信（IPC）机制，特别是管道（pipe）的使用。 4. 实现一个简单的Shell程序，能够解析并执行包含管道（|）和输出重定向（>）的命令。 5. 熟悉Linux文件I/O操作，实现输出重定向功能。 6. **实验内容**   **实验任务简述**  用C语言编写一个简易Shell程序myShell.c，在Linux环境下运行后，显示提示符：  MyShPrompt >  用户可以输入形如：  command1 | command2 | command3 | command4 > file.txt  的命令行，程序需完成如下功能：  **支持功能：**   * 最多支持 **4 个命令** 用 | 管道连接。 * 最终命令的输出可通过 > 重定向至文件。 * 每个命令可以包含参数（如 wc -l）。 * 自动解析命令行、创建子进程、设置管道与重定向，执行命令。   **示例命令**  MyShPrompt> cat country.txt city.txt | egrep 'g' | sort | more > countryCitygSorted.txt  MyShPrompt> cat country.txt city.txt | egrep 'g' | sort | wc -l > countryCitygCount.txt  **技术要求：**   * 使用fork()创建子进程 * 用pipe()实现命令间通信 * 用exec()执行命令 * 用dup2()重定向输入输出   **输入文件：**   * country.txt和city.txt   **输出文件：**   * countryCitygSorted.txt * countryCitygCount.txt   图0 系统整体流程图  **二、实验步骤及方案**  **① 创建encDec.h和encDec.c**  封装所有和文本处理、管道通信相关的功能，为producerConsumerChild与 consumerProducerParent进程提供统一调用接口。  相关功能如下  **writeToPipe(int fd, const char \*buffer, int size)**   * 向指定管道 fd 写入 buffer 中的 size 字节数据。 * 写入失败会打印错误信息。   int **writeToPipe**(int fd, const char \*buffer, int size)  {      int bytesWritten = **write**(fd, buffer, size);      if (bytesWritten == -1)      {  **perror**("write failed");      }      return bytesWritten;  }  **readFromPipe(int fd, char \*buffer, int size)**   * 从管道 fd 中读取最多 size-1 字节并保留字符串结尾 \0。 * 返回读取字节数，并打印错误信息（若出错）。   int **readFromPipe**(int fd, char \*buffer, int size)  {      int bytesRead = **read**(fd, buffer, size - 1);      if (bytesRead > 0)      {          buffer[bytesRead] = '\0';      }      else if (bytesRead == -1)      {  **perror**("read failed");      }      return bytesRead;  }  **countChar(const char \*text)**   * 返回文本总字符数（不包括 \0 结尾符）。   int **countChar**(const char \*text)  {      return **strlen**(text);  }  **countWords(const char \*text)**   * 返回文本中单词数量。 * 连续空格、换行符、Tab 都被当作分隔符。   int **countWords**(const char \*text)  {      int count = 0;      while (\*text)      {          while (\*text && **isspace**((unsigned char)\*text))              text++; // 跳过空格          if (\*text)              count++; // 找到一个新单词          while (\*text && !**isspace**((unsigned char)\*text))              text++; // 跳过当前单词      }      return count;  }  **countLines(const char \*text)**   * 统计文本中的行数（基于 \n）。 * 空文本返回 0，非空则至少算作 1 行。   // 计算文本中的行数  int **countLines**(const char \*text)  {      int count = 0;      const char \*p = text;      while (\*p)      {          if (\*p == '\n')  // 检测换行符              count++;          p++;      }      // 如果文本非空，且最后一个字符不是 '\n'，说明还有一行      if (p != text && p[-1] != '\n')          count++;      return count;  // 返回行数  }  **toLowerCase(char \*text)**   * 将输入字符串中所有大写字符转为小写。 * 传入空指针将不会执行任何操作。   void **toLowerCase**(char \*text)  {      if (text == **NULL**)          return; // 避免空指针错误      while (\*text)      {          \*text = **tolower**((unsigned char)\*text);          text++;      }  }   每个函数**独立**、**清晰**、**可重用**，将所有与 pipe 和文本操作相关的功能被良好封装。错误处理基本健全，为主程序减少出错风险。  **② 实现核心服务程序、**  **countCharService.c**   * 统计输入字符串的字符总数（包括空格、标点符号和换行符）   #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h"  int **main**(int argc, char \*argv[])  {      if (argc != 2)      {  **fprintf**(**stderr**, "Usage: %s <string>\n", argv[0]);          return 1;      }      int charCount = **countChar**(argv[1]);  **printf**("Character count: %d\n", charCount);      return 0;  }  **countWordService**   * 统计输入字符串中的单词数（以空白字符为分隔）   #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h"  int **main**(int argc, char \*argv[])  {      if (argc != 2)      {  **fprintf**(**stderr**, "Usage: %s <string>\n", argv[0]);          return 1;      }      int wordCount = **countWords**(argv[1]);  **printf**("Word count: %d\n", wordCount);      return 0;  }  **countLineService.c**   * 计算文件中的行数中   #include <stdio.h>  #include "encDec.h"  #include <stdlib.h>  #include <ctype.h>  int **main**(int argc, char \*argv[])  {      if (argc != 2)      {  **fprintf**(**stderr**, "Usage: %s <string>\n", argv[0]);          return 1;      }      int lineCount = **countLines**(argv[1]);  **printf**("Line count: %d\n", lineCount);      return 0;  }  **toLowerCaseService.c**   * 将输入字符串全部转为小写   #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h"  int **main**(int argc, char \*argv[])  {      if (argc != 2)      {  **fprintf**(**stderr**, "Usage: %s <string>\n", argv[0]);          return 1;      }  **toLowerCase**(argv[1]);  **printf**("Lowercase string: %s\n", argv[1]);      return 0;  }  **readFromPipeService.c**   * 从管道中读取文件   #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <ctype.h>  #include <fcntl.h>  #include "encDec.h"  int **main**(int argc, char \*argv[])  {      if (argc != 2)      {  **fprintf**(**stderr**, "Usage: %s <pipe\_fd>\n", argv[0]);          return 1;      }      int fd = **atoi**(argv[1]);      char buffer[1024];      int bytesRead = **readFromPipe**(fd, buffer, sizeof(buffer));      if (bytesRead > 0)      {  **printf**("Read from pipe: %s\n", buffer);      }      else      {  **fprintf**(**stderr**, "Error reading from pipe\n");      }      return 0;  }  **writeToPipeService.c**   * 将内容写入管道   #include <stdio.h>  #include <unistd.h>  #include <fcntl.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h"  int **main**(int argc, char \*argv[])  {      if (argc != 3)      {  **fprintf**(**stderr**, "Usage: %s <pipe\_fd> <string>\n", argv[0]);          return 1;      }      int fd = **atoi**(argv[1]);      const char \*text = argv[2];      int bytesWritten = **writeToPipe**(fd, text, **strlen**(text));      if (bytesWritten == -1)      {  **fprintf**(**stderr**, "Error writing to pipe\n");      }      else      {  **printf**("Written to pipe: %s\n", text);      }      return 0;  }  **③ 实现主程序**  程序通过父子进程合作，利用管道进行数据传输。父进程负责读取源文件并传递给子进程，子进程进行文件内容的处理和统计，生成两个输出文件并将其路径发送回父进程。父进程展示文件统计信息并执行 diff 命令进行对比。  **1管道的创建**  首先，程序创建了两个管道：pipe1 和 pipe2。管道用于父进程和子进程之间进行进程间通信（IPC）。  int pipe1[2], pipe2[2];  if (**pipe**(pipe1) == -1 || **pipe**(pipe2) == -1)  {  **perror**("pipe creation failed");  **exit**(EXIT\_FAILURE);  }   * pipe1 用于将父进程的文本数据传送到子进程。 * pipe2 用于将子进程的结果（如文件路径信息）传回父进程。   每个管道由两个文件描述符组成：   * 读端（pipe1[READ\_END] 和 pipe2[READ\_END]）。 * 写端（pipe1[WRITE\_END] 和 pipe2[WRITE\_END]）。   **2. 创建子进程**  使用 fork() 创建一个子进程，并通过判断 pid 的值来区分父进程和子进程的行为。  pid\_t pid = **fork**();  if (pid == -1)  {  **perror**("fork failed");  **exit**(EXIT\_FAILURE);  }   * 如果 fork() 返回 0，说明是子进程；如果返回正值，是父进程。 * 如果 fork() 返回负值，表示创建子进程失败。   **3. 子进程的操作**  子进程的操作主要有以下几部分：  **3.1 关闭不需要的管道端**  子进程只需要读取 pipe1 和写入 pipe2，因此需要关闭不相关的管道端：  **close**(pipe1[WRITE\_END]);  **close**(pipe2[READ\_END]);  **3.2 从管道读取数据**  子进程通过 read(pipe1[READ\_END], buffer, sizeof(buffer) - 1) 从父进程传来的管道中读取数据，读取的是 editSource.txt 文件的内容。  char buffer[4096];  int bytesRead = **read**(pipe1[READ\_END], buffer, sizeof(buffer) - 1);  if (bytesRead <= 0)  {  **perror**("read from pipe1 failed");  **exit**(EXIT\_FAILURE);  }  buffer[bytesRead] = '\0';  如果读取失败（例如文件为空或管道错误），程序会终止。否则，读取到的数据保存在 buffer 中。  **3.3 执行外部服务程序**  子进程通过 execvp() 调用执行四个外部服务程序，用于对文件内容进行统计（字符数、单词数、行数）和转换为小写。  char \*countCharArgs[] = {"./countCharService", buffer, NULL};  char \*countWordArgs[] = {"./countWordService", buffer, NULL};  char \*countLineArgs[] = {"./countLineService", buffer, NULL};  char \*toLowerArgs[] = {"./toLowerCaseService", buffer, NULL};  **executeService**("./countCharService", countCharArgs);  **executeService**("./countWordService", countWordArgs);  **executeService**("./countLineService", countLineArgs);  **executeService**("./toLowerCaseService", toLowerArgs);   * countCharService：统计字符数。 * countWordService：统计单词数。 * countLineService：统计行数。 * toLowerCaseService：将文本转为小写。   这些服务都使用 execvp() 来调用，执行时将 buffer 内容传递给这些服务的命令行参数。  **3.4 创建 noUpper.txt 和 theCount.txt**  在执行统计操作后，子进程生成两个文件：   1. noUpper.txt：存储将文本转换为小写后的内容。 2. theCount.txt：存储统计结果，包括字符数、单词数和行数。   FILE \*noUpperFile = **fopen**("noUpper.txt", "w");  if (!noUpperFile)  {  **perror**("failed to create noUpper.txt");  **exit**(EXIT\_FAILURE);  }  **fprintf**(noUpperFile, "%s", buffer);  **fclose**(noUpperFile);  FILE \*countFile = **fopen**("theCount.txt", "w");  if (!countFile)  {  **perror**("failed to create theCount.txt");  **exit**(EXIT\_FAILURE);  }  **fprintf**(countFile, "Number of characters: %d\n", **strlen**(buffer));  **fprintf**(countFile, "Number of words: %d\n", **countWords**(buffer));  **fprintf**(countFile, "Number of lines: %d\n", **countLines**(buffer));  **fclose**(countFile);   * 使用 fopen 创建文件，如果创建失败，程序终止。 * 使用 fprintf 将统计信息写入 theCount.txt，并将转换为小写的文本写入 noUpper.txt。   **3.5 获取文件的绝对路径**  为了将文件路径传回父进程，子进程需要获取文件的绝对路径。  char noUpperPath[PATH\_MAX];  char countPath[PATH\_MAX];  **realpath**("noUpper.txt", noUpperPath);  **realpath**("theCount.txt", countPath);  realpath() 获取文件的绝对路径，将路径存入 noUpperPath 和 countPath。  **3.6 通过管道传递文件路径信息**  子进程将 theCount.txt 和 noUpper.txt 的路径信息通过管道 pipe2 传递给父进程。  char pathInfo[PATH\_MAX \* 2 + 2];  **snprintf**(pathInfo, sizeof(pathInfo), "%s\n%s", countPath, noUpperPath);  **write**(pipe2[WRITE\_END], pathInfo, **strlen**(pathInfo));  **3.7 子进程退出**  子进程完成所有任务后，关闭管道端并退出。  **close**(pipe1[READ\_END]);  **close**(pipe2[WRITE\_END]);  **exit**(EXIT\_SUCCESS);  **4. 父进程的操作**  父进程的任务是：   1. 从 editSource.txt 文件读取内容并通过管道发送给子进程。 2. 从 pipe2 中读取子进程传回的文件路径信息。 3. 显示 theCount.txt 的内容。 4. 执行 diff 命令比较 editSource.txt 和 noUpper.txt。   **4.1 读取 editSource.txt 文件内容**  父进程打开 editSource.txt 文件并读取其内容：  FILE \*sourceFile = **fopen**("editSource.txt", "r");  if (!sourceFile)  {  **perror**("failed to open editSource.txt");  **exit**(EXIT\_FAILURE);  }  **fseek**(sourceFile, 0, SEEK\_END);  long fileSize = **ftell**(sourceFile);  **fseek**(sourceFile, 0, SEEK\_SET);  char \*fileContent = **malloc**(fileSize + 1);  if (!fileContent)  {  **perror**("memory allocation failed");  **exit**(EXIT\_FAILURE);  }  **fread**(fileContent, 1, fileSize, sourceFile);  fileContent[fileSize] = '\0';  **fclose**(sourceFile);  父进程首先读取文件的大小，然后分配内存存储文件内容，最后将内容写入管道 pipe1。  **4.2 从管道读取文件路径信息**  父进程从 pipe2 中读取子进程发送的文件路径信息，解析出 theCount.txt 和 noUpper.txt 的路径。  char pathInfo[PATH\_MAX \* 2 + 2];  int bytesRead = **read**(pipe2[READ\_END], pathInfo, sizeof(pathInfo) - 1);  if (bytesRead <= 0)  {  **perror**("read from pipe2 failed");  **exit**(EXIT\_FAILURE);  }  pathInfo[bytesRead] = '\0';  char \*countPath = **strtok**(pathInfo, "\n");  char \*noUpperPath = **strtok**(NULL, "\n");  **4.3 显示 theCount.txt 内容**  父进程打开 theCount.txt 文件并打印其内容。  FILE \*countFile = **fopen**(countPath, "r");  if (countFile)  {      char line[256];  **printf**("Contents of theCount.txt:\n");      while (**fgets**(line, sizeof(line), countFile))      {  **printf**("%s", line);      }  **fclose**(countFile);  }  **4.4 执行 diff 命令**  父进程执行diff命令，比较editSource.txt和noUpper.txt，显示两者之间的差异。  **printf**("\nResult of diff command:\n");  char diffCommand[PATH\_MAX \* 2 + 20];  **snprintf**(diffCommand, sizeof(diffCommand), "diff editSource.txt noUpper.txt");  **system**(diffCommand);  **4.5 父进程退出**  父进程关闭管道并等待子进程结束。  **close**(pipe1[WRITE\_END]);  **close**(pipe2[READ\_END]);  **wait**(NULL);  1  **三．结果及分析**  图1文件所在位置  **① 编译程序**  图2编译程序  **② 运行程序**  图3运行程序  1  **③ 结果**  图4 显示字符数，词数，行数  以及转换为小写的文本内容  使用wc指令验证结果是否正确  图5 统计文件中字符数  图6 统计文件行数  与程序输出结果一致  图7 diff指令输出的结果  **四、心得体会**  本次实验让我系统地掌握了进程间通信的基本机制，尤其是通过 pipe 管道实现父子进程之间的全双工通信。通过 fork() 创建子进程，并结合 exec() 系列函数执行不同的处理任务，加深了我对进程控制与资源分配的理解。  实验中构建的生产者-消费者模型，使我清晰地认识到进程功能划分与协同工作的重要性。父进程与子进程通过两个管道分别进行信息传输和任务响应，有效体现了并发编程中的协作思想。  此外，文本统计与大小写转换的功能实现，也让我熟悉了文件读写与字符串处理在系统层级下的具体操作。封装模块函数至 encDec.h，并统一通过系统调用访问，进一步增强了我对模块化编程的认识和实际应用能力。  总体而言，该实验有效地将操作系统的理论知识与实际编程相结合，提升了我对系统调用、进程通信、文件操作等内容的综合掌握水平，为后续深入学习操作系统原理和系统编程打下了坚实的基础。 | | | |
| **代码附录**  encDec.h  #ifndef **ENCDEC\_H**  #define **ENCDEC\_H**  #include <unistd.h>    // 用于 pipe 相关函数声明  #include <sys/types.h> // 用于 size\_t 等类型  /\*\*   \* 向管道写入数据   \* @param fd 管道文件描述符   \* @param buffer 要写入的数据缓冲区   \* @param size 要写入的字节数   \* @return 成功返回写入的字节数，失败返回-1   \*/  int **writeToPipe**(int fd, const char \*buffer, int size);  /\*\*   \* 从管道读取数据   \* @param fd 管道文件描述符   \* @param buffer 存储读取数据的缓冲区   \* @param size 缓冲区大小   \* @return 成功返回读取的字节数，失败返回-1   \*/  int **readFromPipe**(int fd, char \*buffer, int size);  /\*\*   \* 统计字符数   \* @param text 要统计的文本   \* @return 字符数量（包含结束符）   \*/  int **countChar**(const char \*text);  /\*\*   \* 统计单词数   \* @param text 要统计的文本   \* @return 单词数量（以空白字符分隔）   \*/  int **countWords**(const char \*text);  /\*\*   \* 统计行数   \* @param text 要统计的文本   \* @return 行数（以换行符分隔）   \*/  int **countLines**(const char \*text);  /\*\*   \* 将字符串转换为小写   \* @param text 要转换的字符串（会被原地修改）   \*/  void **toLowerCase**(char \*text);  #endif // ENCDEC\_H  encDec.c  #include "encDec.h"  #include <unistd.h>  #include <string.h>  #include <ctype.h>  #include <stdio.h>  #include <errno.h>  #include <stdlib.h>  // 向管道写入数据  int **writeToPipe**(int fd, const char \*buffer, int size)  {      int bytesWritten = **write**(fd, buffer, size);  // 写入数据到管道      if (bytesWritten == -1)      {  **perror**("write failed");  // 写入失败时输出错误信息      }      return bytesWritten;  // 返回实际写入的字节数  }  // 从管道读取数据  int **readFromPipe**(int fd, char \*buffer, int size)  {      int bytesRead = **read**(fd, buffer, size - 1);  // 从管道读取数据      if (bytesRead > 0)      {          buffer[bytesRead] = '\0';  // 确保字符串以'\0'结尾      }      else if (bytesRead == -1)      {  **perror**("read failed");  // 读取失败时输出错误信息      }      return bytesRead;  // 返回实际读取的字节数  }  // 计算文本中的字符数  int **countChar**(const char \*text)  {      return **strlen**(text);  // 返回字符串的长度  }  // 计算文本中的单词数  int **countWords**(const char \*text)  {      int count = 0;      while (\*text)      {          while (\*text && **isspace**((unsigned char)\*text))  // 跳过空格              text++;          if (\*text)              count++;  // 找到一个新单词          while (\*text && !**isspace**((unsigned char)\*text))  // 跳过当前单词              text++;      }      return count;  // 返回单词数  }  // 计算文本中的行数  int **countLines**(const char \*text)  {      int count = 0;      const char \*p = text;      while (\*p)      {          if (\*p == '\n')  // 检测换行符              count++;          p++;      }      // 如果文本非空，且最后一个字符不是 '\n'，说明还有一行      if (p != text && p[-1] != '\n')          count++;      return count;  // 返回行数  }  // 将文本转换为小写  void **toLowerCase**(char \*text)  {      if (text == NULL)          return;  // 避免空指针错误      while (\*text)      {          \*text = **tolower**((unsigned char)\*text);  // 转换为小写字母          text++;      }  }  countCharService.c  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h"  int **main**(int argc, char \*argv[])  {      // 检查命令行参数数量，确保只有一个参数传入      if (argc != 2)      {  **fprintf**(stderr, "Usage: %s <string>\n", argv[0]); // 提示正确的用法          return 1; // 如果参数不对，则返回错误代码 1      }      // 调用 countChar 函数来计算传入字符串的字符数      int charCount = **countChar**(argv[1]);      // 输出字符数  **printf**("Character count: %d\n", charCount);      return 0; // 正常结束程序  }  countLineService.c  #include <stdio.h>  #include "encDec.h" // 包含自定义的头文件，定义了countLines函数等  #include <stdlib.h>  #include <ctype.h>  int **main**(int argc, char \*argv[])  {      // 检查命令行参数数量，确保只有一个参数传入      if (argc != 2)      {  **fprintf**(stderr, "Usage: %s <string>\n", argv[0]); // 提示正确的用法          return 1; // 如果参数不对，则返回错误代码 1      }      // 调用 countLines 函数来计算传入字符串中的行数      int lineCount = **countLines**(argv[1]);      // 输出行数  **printf**("Line count: %d\n", lineCount);      return 0; // 正常结束程序  }  countWordService.c  #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h" // 引入自定义头文件，包含countWords函数的声明  int **main**(int argc, char \*argv[])  {      // 检查命令行参数数量，确保用户提供了一个字符串参数      if (argc != 2)      {  **fprintf**(stderr, "Usage: %s <string>\n", argv[0]); // 提示用户正确使用方式          return 1; // 如果参数错误，则返回错误代码 1      }      // 调用 countWords 函数计算传入字符串中的单词数      int wordCount = **countWords**(argv[1]);      // 输出单词数  **printf**("Word count: %d\n", wordCount);      return 0; // 正常结束程序  }  readFromPipeService.c  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <ctype.h>  #include <fcntl.h>  #include "encDec.h" // 引入自定义头文件，包含readFromPipe函数的声明  int **main**(int argc, char \*argv[])  {      // 检查命令行参数数量，确保用户提供了管道文件描述符      if (argc != 2)      {  **fprintf**(stderr, "Usage: %s <pipe\_fd>\n", argv[0]); // 提示用户正确使用方式          return 1; // 如果参数错误，则返回错误代码 1      }      // 将命令行参数转为整数，表示管道文件描述符      int fd = **atoi**(argv[1]);      char buffer[1024]; // 定义一个缓冲区用来存放从管道读取的数据      // 调用 readFromPipe 函数读取管道中的数据      int bytesRead = **readFromPipe**(fd, buffer, sizeof(buffer));      if (bytesRead > 0)      {  **printf**("Read from pipe: %s\n", buffer); // 输出从管道读取的内容      }      else      {  **fprintf**(stderr, "Error reading from pipe\n"); // 如果读取失败，输出错误信息      }      return 0; // 正常结束程序  }  toLowerCaseService.c  #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h" // 引入自定义头文件，包含toLowerCase函数的声明  int **main**(int argc, char \*argv[])  {      // 检查命令行参数数量，确保用户提供了要转换的小写字符串      if (argc != 2)      {  **fprintf**(stderr, "Usage: %s <string>\n", argv[0]); // 如果参数错误，打印使用提示          return 1; // 返回错误代码 1      }      // 调用 toLowerCase 函数将输入字符串转换为小写  **toLowerCase**(argv[1]);      // 打印转换后的小写字符串  **printf**("Lowercase string: %s\n", argv[1]);      return 0; // 正常结束程序  }  writeToPipeService.c  #include <stdio.h>  #include <unistd.h>  #include <fcntl.h>  #include <stdlib.h>  #include <ctype.h>  #include "encDec.h"  // 引入自定义头文件，包含 writeToPipe 函数的声明  #include <string.h>  int **main**(int argc, char \*argv[])  {      // 检查命令行参数数量，确保用户提供了管道文件描述符和要写入的数据      if (argc != 3)      {  **fprintf**(stderr, "Usage: %s <pipe\_fd> <string>\n", argv[0]); // 如果参数错误，打印使用提示          return 1; // 返回错误代码 1      }      // 将管道文件描述符从字符串转换为整数      int fd = **atoi**(argv[1]);      // 获取要写入管道的文本数据      const char \*text = argv[2];      // 调用 writeToPipe 函数将文本写入管道      int bytesWritten = **writeToPipe**(fd, text, **strlen**(text));      if (bytesWritten == -1)      {          // 如果写入失败，打印错误信息  **fprintf**(stderr, "Error writing to pipe\n");      }      else      {          // 如果写入成功，打印已写入的数据  **printf**("Written to pipe: %s\n", text);      }      return 0; // 正常结束程序  }  Main.c  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/wait.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <string.h>  #include <limits.h>  #include <ctype.h>  #define **READ\_END** 0  // 定义管道的读取端  #define **WRITE\_END** 1 // 定义管道的写入端  // 执行服务程序的函数  void **executeService**(const char \*service, char \*const args[])  {      pid\_t pid = **fork**(); // 创建子进程      if (pid == 0) // 如果是子进程      {  **execvp**(service, args); // 执行指定的服务  **perror**("execvp failed"); // 如果execvp调用失败，输出错误信息  **exit**(EXIT\_FAILURE); // 退出子进程      }      else if (pid > 0) // 如果是父进程      {  **wait**(NULL); // 等待子进程结束      }      else      {  **perror**("fork failed"); // 如果fork调用失败，输出错误信息      }  }  int **main**()  {      // 创建两个管道：pipe1用于父进程和子进程之间传递数据，pipe2用于子进程向父进程传递路径信息      int pipe1[2], pipe2[2];      if (**pipe**(pipe1) == -1 || **pipe**(pipe2) == -1) // 创建管道失败      {  **perror**("pipe creation failed");  **exit**(EXIT\_FAILURE);      }      pid\_t pid = **fork**(); // 创建子进程      if (pid == -1) // 如果fork失败      {  **perror**("fork failed");  **exit**(EXIT\_FAILURE);      }      if (pid > 0) // 父进程 - producerConsumerChild      {  **close**(pipe1[READ\_END]); // 父进程关闭pipe1的读取端  **close**(pipe2[WRITE\_END]); // 父进程关闭pipe2的写入端          // 读取editSource.txt文件的内容          FILE \*sourceFile = **fopen**("editSource.txt", "r");          if (!sourceFile) // 文件打开失败          {  **perror**("failed to open editSource.txt");  **exit**(EXIT\_FAILURE);          }          // 获取文件大小  **fseek**(sourceFile, 0, SEEK\_END);          long fileSize = **ftell**(sourceFile);  **fseek**(sourceFile, 0, SEEK\_SET);          // 读取文件内容到内存          char \*fileContent = **malloc**(fileSize + 1);          if (!fileContent) // 内存分配失败          {  **perror**("memory allocation failed");  **exit**(EXIT\_FAILURE);          }  **fread**(fileContent, 1, fileSize, sourceFile);          fileContent[fileSize] = '\0'; // 确保文件内容以'\0'结尾  **fclose**(sourceFile);          // 将文件内容写入管道pipe1  **write**(pipe1[WRITE\_END], fileContent, **strlen**(fileContent));  **free**(fileContent);          // 从pipe2读取路径信息          char pathInfo[PATH\_MAX \* 2 + 2];          int bytesRead = **read**(pipe2[READ\_END], pathInfo, sizeof(pathInfo) - 1);          if (bytesRead <= 0) // 从管道读取失败          {  **perror**("read from pipe2 failed");  **exit**(EXIT\_FAILURE);          }          pathInfo[bytesRead] = '\0';          // 解析路径信息          char \*countPath = **strtok**(pathInfo, "\n");          char \*noUpperPath = **strtok**(NULL, "\n");          // 显示theCount.txt文件的内容          FILE \*countFile = **fopen**(countPath, "r");          if (countFile)          {              char line[256];  **printf**("Contents of theCount.txt:\n");              while (**fgets**(line, sizeof(line), countFile)) // 逐行读取并打印              {  **printf**("%s", line);              }  **fclose**(countFile);          }          // 执行diff命令  **printf**("\nResult of diff command:\n");          char diffCommand[PATH\_MAX \* 2 + 20];  **snprintf**(diffCommand, sizeof(diffCommand), "diff editSource.txt noUpper.txt");  **system**(diffCommand); // 执行diff命令比较两个文件  **close**(pipe1[WRITE\_END]); // 关闭管道  **close**(pipe2[READ\_END]);  **wait**(NULL); // 等待子进程结束      }      else // 子进程 - consumerProducerParent      {  **close**(pipe1[WRITE\_END]); // 子进程关闭pipe1的写入端  **close**(pipe2[READ\_END]);  // 子进程关闭pipe2的读取端          // 从管道pipe1读取父进程传递的文本数据          char buffer[4096];          int bytesRead = **read**(pipe1[READ\_END], buffer, sizeof(buffer) - 1);          if (bytesRead <= 0) // 从管道读取失败          {  **perror**("read from pipe1 failed");  **exit**(EXIT\_FAILURE);          }          buffer[bytesRead] = '\0'; // 确保读取的文本是以'\0'结尾          // 调用各服务程序进行统计          char \*countCharArgs[] = {"./countCharService", buffer, NULL};          char \*countWordArgs[] = {"./countWordService", buffer, NULL};          char \*countLineArgs[] = {"./countLineService", buffer, NULL};          char \*toLowerArgs[] = {"./toLowerCaseService", buffer, NULL};          // 执行统计服务  **executeService**("./countCharService", countCharArgs);  **executeService**("./countWordService", countWordArgs);  **executeService**("./countLineService", countLineArgs);  **executeService**("./toLowerCaseService", toLowerArgs);          // 将文本转换为小写后写入noUpper.txt  **toLowerCase**(buffer);  // 转换为小写          FILE \*noUpperFile = **fopen**("noUpper.txt", "w");          if (!noUpperFile) // 创建文件失败          {  **perror**("failed to create noUpper.txt");  **exit**(EXIT\_FAILURE);          }  **fprintf**(noUpperFile, "%s", buffer); // 将转换后的内容写入文件  **fclose**(noUpperFile);          // 创建theCount.txt并写入统计信息          FILE \*countFile = **fopen**("theCount.txt", "w");          if (!countFile) // 创建文件失败          {  **perror**("failed to create theCount.txt");  **exit**(EXIT\_FAILURE);          }          // 使用修正后的统计逻辑写入文件  **fprintf**(countFile, "Number of characters: %d\n", bytesRead);  // 精确字符数  **fprintf**(countFile, "Number of words: %d\n", **countWords**(buffer));  **fprintf**(countFile, "Number of lines: %d\n", **countLines**(buffer));  **fclose**(countFile);          // 获取文件的绝对路径          char noUpperPath[PATH\_MAX];          char countPath[PATH\_MAX];  **realpath**("noUpper.txt", noUpperPath);  **realpath**("theCount.txt", countPath);          // 将路径信息写入pipe2          char pathInfo[PATH\_MAX \* 2 + 2];  **snprintf**(pathInfo, sizeof(pathInfo), "%s\n%s", countPath, noUpperPath);  **write**(pipe2[WRITE\_END], pathInfo, **strlen**(pathInfo));  **close**(pipe1[READ\_END]); // 关闭管道  **close**(pipe2[WRITE\_END]);  **exit**(EXIT\_SUCCESS); // 子进程结束      }      return 0; // 主程序结束  } | | | |