

**学生作业报告册**

**作业2报告**

|  |  |
| --- | --- |
| **学年学期：** | **20242025 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** | **操作系统** | **课程编号** | **A2130330** |
| **作业名称** | **进程间的单向通信(Assignment2)** | | |
| **一、作业内容**  本次作业要求完成一个基于Linux的C语言程序，利用fork()和pipe()系统调用，实现通过普通文件进行的进程间通信。  作业要求编写两个 C 程序：   1. producer.c    * 创建一个名为 numbers.txt 的文本文件，并写入20个整数；    * 使用无名管道与子进程通信，将该文件路径传递给子进程； 2. consumer.c    * 读取文件中的整数；    * 输出所有偶数；    * 计算并输出所有奇数的总和。   在 producer.c中需使用fork()创建子进程，子进程使用exec()执行consumer.c 程序。父子进程之间通过pipe()管道传递数据，比如文件路径或描述符。    图1 进程间关系  **二、实现步骤**    图2 系统数据流图  **① 实现概述**   1. 在 producer.c中使用creat()或open()函数创建并写入 numbers.txt 文件，文件中包含 20 个随机整数； 2. 使用pipe()创建通信管道； 3. 使用fork()创建子进程； 4. 父进程将文件路径（或信息）通过write()写入管道； 5. 子进程通过read()从管道读取文件路径，并使用execlp()调用consumer程序； 6. consumer.c程序读取 numbers.txt，将偶数输出并计算奇数的总和； 7. 所有进程输出其PID信息以验证父子关系。   **② producer.c实现**  1.创建管道与文件  管道创建：使用 pipe(fd) 创建无名管道，fd[0] 为读端，fd[1] 为写端。  文件操作  int file\_fd = **open**(FILENAME, O\_CREAT | O\_WRONLY | O\_TRUNC, 0644);  • 创建并打开 numbers.txt，权限设置为 0644（用户可读写，其他用户只读）。  • 若文件存在则清空内容（O\_TRUNC）。  2.生成随机数并写入文件  • 随机数生成：  **srand**(**time**(NULL)); // 使用时间作为随机种子  for (int i = 0; i < COUNT; i++)  {  **dprintf**(file\_fd, "%d\n", **rand**() % 100);  }  生成20个0-99的随机整数，通过dprintf写入文件。  3.显示文件描述符与内容  • 文件描述符输出：  **printf**("File numbers.txt fd is: %d\n", file\_fd);  • 文件内容读取：  **lseek**(file\_fd, 0, SEEK\_SET); // 重置文件指针到开头  while ((bytesRead = **read**(file\_fd, buffer, sizeof(buffer))) {  **write**(STDOUT\_FILENO, buffer, bytesRead); // 输出到终端  }  4.创建子进程（fork）  • fork() 返回值处理：  pid\_t pid = **fork**();  if (pid < 0)  {  **perror**("Fork failed");  **exit**(EXIT\_FAILURE);  }  else if (pid > 0)  {      // 父进程逻辑  **close**(fd[0]); // 关闭读端      // 重新打开文件并通过管道发送数据  }  else  {      // 子进程逻辑  **close**(fd[1]);                           // 关闭写端  **dup2**(fd[0], STDIN\_FILENO);              // 重定向标准输入到管道读端  **execlp**("./consumer", "consumer", NULL); // 执行消费者程序  **perror**("execlp failed");  **exit**(EXIT\_FAILURE);  }  父进程（pid > 0）：关闭管道读端 fd[0]，通过管道发送文件内容。  子进程（pid == 0）：关闭管道写端 fd[1]，重定向标准输入到管道读端。  5.父进程发送数据  • 通过管道发送文件内容：  file\_fd = **open**(FILENAME, O\_RDONLY); // 重新以只读模式打开文件  while ((bytesRead = **read**(file\_fd, buffer, sizeof(buffer))) {  **write**(fd[1], buffer, bytesRead); // 将文件内容写入管道  }  **close**(fd[1]);  // 关闭管道写端  • 等待子进程结束：  **wait**(NULL);  // 阻塞等待子进程退出  6.子进程执行消费者程序  • 重定向标准输入：  **dup2**(fd[0], STDIN\_FILENO); // 将管道读端映射到标准输入  **close**(fd[0]);              // 关闭原始管道读端  • 执行消费者程序：  **execlp**("./consumer", "consumer", NULL); // 替换当前进程为 consumer  若execlp失败，输出错误信息并退出  **③ consumer.c实现步骤**  1.从标准输入读取数据  • 循环读取整数：  while (**scanf**("%d", &num) != EOF)  {      if (num % 2 == 0)      {  **printf**("%d ", num); // 输出偶数      }      else      {          sum\_odd += num; // 累加奇数      }  }  通过scanf从管道读端（已重定向到标准输入）读取数据。  2.输出结果  • 格式化输出：  **printf**("\nSum of odd numbers: %d\n", sum\_odd);  **三、分析与结果**  ① 编译程序: 在运行程序之前编译文件，通过指令对源文件进行编译：    ② 运行程序    ③ 结果  文本  AI 生成的内容可能不正确。  **a) 文件描述符信息**   |  |  | | --- | --- | | 文件名 | 文件描述符 (fd) | | numbers.txt | 5 |   **b) 文件内容**  numbers.txt内容如下：  97,15,73,7,17,30,71,75,50,48,43,61,92,24,49,17,75,12,79,22  **c) consumer 程序输出**  (1)偶数列表: 30 50 48 92 24 12 22  (2)奇数的总和：679  **d) 父进程信息**  Parent Process: My pid = 646. I created child pid = 647.  **e) 子进程信息**  Child Process: My pid = 647. My parent pid = 646.  **四、心得体会**  本次作业通过实现基于管道通信的生产者消费者模型，让我对Linux进程间通信机制有了更深入的理解。在完成作业的过程中，我不仅掌握了fork()、pipe()、dup2()等系统调用的具体用法，还深刻体会到多进程协作的程序设计思想。在管道通信的实现中，我最初忽略了文件描述符的关闭时机，导致子进程读取管道时出现阻塞。通过调试发现，父进程必须在写入数据后及时关闭写端，子进程读取完数据后关闭读端，否则管道会因引用计数不为零而无法正常释放。这一问题的解决让我理解了内核管理文件描述符的机制，以及进程间资源共享的注意事项。  在本次作业中，数据同步问题给我留下了深刻印象。父进程通过wait()等待子进程结束，确保了数据处理的时序正确性。这让我认识到，在多进程环境中，必须谨慎设计同步逻辑，避免出现竞态条件或数据不一致的情况。同时，使用dup2()重定向标准输入的设计，让我体会到Linux“一切皆文件”哲学的实际应用——管道作为特殊的文件描述符，可以与标准输入输出无缝衔接。  另外错误处理的实践让我受益匪浅。通过为每个系统调用添加返回值检查（如pipe()、fork()的异常处理），我养成了编写健壮代码的习惯。特别是在execl()调用失败时输出错误信息的设计，帮助我快速定位了环境变量路径的问题。  本次作业的收获不仅在于技术层面，更在于系统编程思维的培养。我认识到，操作系统提供的底层机制（如进程控制、IPC）是构建复杂软件的基石，只有深入理解这些机制的原理和限制，才能设计出高效可靠的系统级程序。 | | | |
| **源代码**  Producer.c  // producer.c  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <fcntl.h>  #include <sys/types.h>  #include <sys/wait.h>  #define **FILENAME** "numbers.txt"  // Define the filename to be used  #define **COUNT** 20  // Define the number of random integers to generate  int **main**() {      int fd[2];  // Array to hold pipe file descriptors      if (**pipe**(fd) == -1) {  // Create a pipe  **perror**("Pipe failed");  **exit**(EXIT\_FAILURE);      }      // Create numbers.txt and write 20 random integers to it      int file\_fd = **open**(FILENAME, O\_CREAT | O\_WRONLY | O\_TRUNC, 0644);      if (file\_fd == -1) {  **perror**("Error opening file");  **exit**(EXIT\_FAILURE);      }  **srand**(**getpid**());  // Seed the random number generator with the process ID      int numbers[COUNT];  // Array to hold the random numbers      for (int i = 0; i < COUNT; i++) {          numbers[i] = **rand**() % 100;  // Generate a random number between 0 and 99  **dprintf**(file\_fd, "%d\n", numbers[i]);  // Write the number to the file      }  **close**(file\_fd);  // Close the file after writing      // Open numbers.txt for reading      file\_fd = **open**(FILENAME, O\_RDONLY);      if (file\_fd == -1) {  **perror**("Error opening file for reading");  **exit**(EXIT\_FAILURE);      }      // Output the file descriptor and contents of the file  **printf**("File numbers.txt fd is: %d\n", file\_fd);  **printf**("Contents of file numbers.txt:\n");      // Read the file contents and print them      char buffer[128];      int bytesRead;      while ((bytesRead = **read**(file\_fd, buffer, sizeof(buffer) - 1)) > 0) {          buffer[bytesRead] = '\0';  // Ensure the buffer is null-terminated  **printf**("%s", buffer);      }  **close**(file\_fd);  // Close the file after reading  **printf**("\n");      // Fork to create the consumer process      pid\_t pid = **fork**();      if (pid < 0) {  **perror**("Fork failed");  **exit**(EXIT\_FAILURE);      }      if (pid > 0) {          // Parent process (producer)  **printf**("Parent Process: My pid = %d. I created child pid = %d.\n", **getpid**(), pid);  **close**(fd[0]);  // Close the read end of the pipe          // Reopen the file and send data through the pipe          file\_fd = **open**(FILENAME, O\_RDONLY);          if (file\_fd == -1) {  **perror**("Error reading file");  **exit**(EXIT\_FAILURE);          }          while ((bytesRead = **read**(file\_fd, buffer, sizeof(buffer))) > 0) {  **write**(fd[1], buffer, bytesRead);  // Write data to the pipe          }  **close**(file\_fd);  // Close the file  **close**(fd[1]);  // Close the write end of the pipe  **wait**(NULL);    // Wait for the child process to finish      } else {          // Child process (consumer)  **printf**("Child Process: My pid = %d. My parent pid = %d.\n", **getpid**(), **getppid**());  **close**(fd[1]);  // Close the write end of the pipe  **dup2**(fd[0], STDIN\_FILENO);  // Redirect stdin to read from the pipe  **execlp**("./consumer", "consumer", NULL);  // Execute the consumer program  **perror**("execlp failed");  **exit**(EXIT\_FAILURE);      }      return 0;  }  Consumer.c  // consumer.c  #include <stdio.h>  #include <stdlib.h>  int **main**() {      int num, sum\_odd = 0;  // Variables to hold the number and sum of odd numbers  **printf**("Even numbers: ");  // Print the header for even numbers      // Read numbers from stdin until EOF      while (**scanf**("%d", &num) != EOF) {          if (num % 2 == 0) {  // Check if the number is even  **printf**("%d ", num);  // Print the even number          } else {              sum\_odd += num;  // Add the odd number to the sum          }      }      // Print the sum of odd numbers  **printf**("\nSum of odd numbers: %d\n", sum\_odd);      return 0;  } | | | |