|  |  |  |  |
| --- | --- | --- | --- |
| **《操作系统原理》实验报告** | | | |
| 实验名称 | Copy File | 实验序号 | 1 |
| 实验日期 | 2023/3/24 | 实验人 |  |
| 一、实验题目  在 2.3 节中，我们描述了一个将一个文件的内容复制到目标文件的程序。该程序首先提示用户输入源文件和目标文件的名称。使用 Windows 或 POSIX API 编写此程序。请务必包括所有必要的错误检查，包括确保源文件是否存在。  正确设计和测试程序后，如果你使用支持它的系统，请使用跟踪系统调用的实用程序运行该程序。 Linux 系统提供 strace 实用程序，Solaris 和 Mac OS X 系统使用 dtrace 命令。由于 Windows 系统不提供此类功能，您将不得不使用调试器跟踪此程序的 Windows 版本。 | | | |
| 二、相关原理与知识  Linux系统调用相关基础知识  Linux系统中C文件读写相关基础知识  strace的使用 | | | |
| 三、实验过程  对于文件拷贝，比较常见的思路即读取拷贝指令、读取源文件、写入目标文件。  **在读取拷贝指令时**，由于实验要求模拟Linux cp命令，即：  cp [options] source dest  方便起见，可以通过**命令行**来传递参数，具体代码如下： | | | |
| if (argc != 3) {  fprintf(stderr, "Usage: %s <source\_file> <destination\_file>**\n**", argv[0]);  exit(EXIT\_FAILURE);  }  此处针对传入参数数量有错误检查：对于参数数量不符，即数量不为3的输入，将直接退出程序，并提示正确的输入方法。  **对于文件操作部分**，统一使用open函数打开目标文件。  *// 判断source file是否成功打开*  if ((src\_fd = open(argv[1], O\_RDONLY)) == -1) {  perror("Open Source File Error");  exit(EXIT\_FAILURE);  }    *// 判断destination file是否成功打开*  if ((dst\_fd = open(argv[2], O\_WRONLY | O\_CREAT | O\_TRUNC, 0644)) == -1) {  perror("Open Destination File Error");  exit(EXIT\_FAILURE);  }  使用只读模式打开Source File，使用只写模式打开Destination File，其中open函数的参数含义分别为：  O\_RDONLY：文件打开模式，使用只读模式打开文件  O\_WRONLY：文件打开模式，使用只写模式打开文件  O\_CREAT：选项，若文件不存在则创建，若存在则不起作用  O\_TRUNC：选项，表示打开的文件被截断为0长度，即清空文件  0644：创建新文件时的权限掩码，表示文件所有者有读写权限，其他用户只有读权限。  此处有针对打开文件失败的错误检查，当打开失败时open函数会返回-1.  **对于源文件的读和目标文件的写**，此处分别使用read函数和write函数。  注意到read和write函数的原型分别为： | | | |
| ssize\_t read(int fd, void \*buf, size\_t count);    ssize\_t write(int fd, void \*buf, size\_t count);  需要buf流的长度作为一个参数，虽然可以直接使用一个较大的数来作为第三个参数，但防止出现buf流溢出的错误，决定计算buf流的长度。  查阅资料发现，在 Linux 下我们可以直接使用 fstat() 函数将一个文件的所有相关信息记录到一个stat结构体中，通过其st\_size成员直接获取其长度。故对文件进行读写复制的代码如下：  fstat(src\_fd, &src\_st);    *// buf读取source file长度，并开内存*  if((buf = (char\*)malloc(sizeof(char) \* src\_st.st\_size)) == NULL)  {  perror("Malloc Error");  exit(EXIT\_FAILURE);  }    *// 读source file*  if(read(src\_fd, buf, src\_st.st\_size) == -1)  {  perror("Read Source File Error");  exit(EXIT\_FAILURE);  }    *// 写destination file*  if(write(dst\_fd, buf, src\_st.st\_size) != src\_st.st\_size)  {  perror("Write Destination File Error");  exit(EXIT\_FAILURE);  }  此处首先分配给buf内存，之后将Source File的内容读取到buf中，再将buf写入Destination File，达到文档复制的效果。  此处有三处错误检查：   1. 对buf流进行内存分配的时候，如果分配失败则返回NULL，程序终止 2. 读源文件时若读取失败则返回-1，程序终止 3. 写目标文件时，若写入的字符数不等于buf流字符数，说明写入发生错误，程序终止   以上代码均执行完毕且无错误，将源文件和目标文件关闭并输出复制成功即可。  *// 关闭source file*  if (close(src\_fd) == -1) {  perror("Close Source File Error");  exit(EXIT\_FAILURE);  }    *// 关闭destination file*  if (close(dst\_fd) == -1) {  perror("Close Destination File Error");  exit(EXIT\_FAILURE);  }    printf("File Copied Successfully.**\n**");  此处有两处错误检查，用来判断两文件是否正常关闭。 | | | |
| 四、实验结果  对于以上文档复制的执行，在Unbuntu 20.04版本虚拟机中执行结果如下图： | | | |
| 通过cat命令读取文本内容可知，SourceFile文档被成功复制。  利用strace命令，可以对该程序的运行过程进行跟踪，跟踪结果如下：    对上述跟踪结果进行简要分析：  由上图可以看出在进入到主函数main之前程序首先先通过 mmap 系统调用将ld与libc等文件都映射到该进程的内存空间中，并使用mprotect()函数修改各数据段的读写执行权限。  在上述跟踪过程中，可以看到在**文件操作部分**，打开文件操作我们代码中使用的是open函数，但在跟踪过程中变成了openat函数，查阅资料发现这是因为传入的路径为相对路径而非绝对路径的原因，对于绝对路径openat会退化为open。打开文件操作中可以发现，对本程序，我们可以通过openat=3或4中发现，SourceFile和DestinationFile两个文档的文件描述符分别为3和4，这在之后的read、write、close函数中也都有用到。  之后程序进入**源文件和目标文件的读写部分**，在open函数调用结束之后，程序通过fstat系统调用获取到文件相关信息，期间过程不做复杂描述；直到read函数被调用，从read(3,…)可知，此处对SourceFile中的信息进行了读取，最后的 =79说明read函数一共从SourceFile文档中读取了79 bit的数据。之后调用write函数，从write(4,…)可知此处为对DestinationFile进行写入，=79说明一共写入了79 bit的数据，与读入的数据量相等。  最后**关闭文件流**，即跟踪结果中的close(3)和close(4)，输出“successful copied”。 | | | |