**I/O Part 3 Low Level I/O**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Part 1 provided an overview of files and directories. Part 2 covered standard stream I/O and binary I/O. Part 3 covers file descriptor information and low level I/O.  Why is low level I/O important?   * Allows the user to obtain information about the characteristics of a file (the stat info) * Provides functions to directly access the contents of a directory file. * Useful in interprocess communication. Provides functions for reading and writing pipes. * Some of the conveniences of the standard I/O library can be issues for device programming * Enhances understanding of important internal operating system concepts.   Including subdirectories, copy all files from  **/usr/local/courses/rslavin/cs3423/IO** | |  |  |  | | --- | --- | --- | | **Function** | **Category** | **Purpose** | | **open** | unix low level i/o | opens a file | | **close** | unix low level i/o | closes a file | | **read** | unix low level i/o | reads a specified number of bytes at the current position | | **write** | unix low level i/o | writes a specified number of bytes at the current position | | **lseek** | unix low level i/o | similar to fseek | | **stat**  **fstat** | unix low level i/o | returns the stat structure for a file which includes inodeNr, file type, file mode, number of links, size, etc. | | **opendir** | unix low level i/o | opens the specified directory for reading | | **readdir** | unix low level i/o | reads the next directory entry. | | **closedir** | unix low level i/o | closes a directory | |
| **File Descriptors and File Structures**  The low level I/O functions use **file descriptor numbers** instead of file pointers. **FILE pointers** (used by stdio functions) point to FILE structures which contain information useful in buffering.  **File descriptor numbers** are integer subscripts to a file descriptor table. | The stdio **fopen** returns a FILE pointer. It points to a structure similar to this:  typedef struct \_iobuf  {  int cnt; // num chars left in buffer  char \*ptr; // ptr to next char in buffer  char \*base; // ptr to beginning of buffer  int flag; // fopen mode  int fd; // file descriptor number  } **FILE**;  That was the description in the K&R book, but implementations can change it. |
| **File Descriptor Integration with Linux**  Each process has a file descriptor table. By convention, **stdin** is 0, **stdout** is 1, and **stderr** is 2. Each entry which represents an open file points to an entry in the **System File Table** (SFT). The **fd** attribute in a FILE structure is a subscript into this process specific table.  The **System File Table** (SFT) is a system-wide table with an entry for every open *file.* Notice that process 1201's stdout and stderr point to the same entry which is to the user's monitor. For process 4123, stdout has been redirected to file "out.txt" instead of the monitor.  Within the SFT, we know whether the *file* is being read or written and the current offset. For reading, it tells us the offset to the last byte *physically* read. For writing, it tells us the offset to the last byte *physically* written.  The v-nodes are references to the underlying inode for a file or a particular *device* (e.g., keyboard, monitor). | **File Descriptor Diagram** |
| **Buffering**  When using C stream I/O, the FILE structure contains a pointer in a *buffer* which isn't the same as the offset in the SFT. Notice the results of a first call of **fgets**:   * The **stdio read buffer** is populated with one block of data. * The SFT entry indicates an offset of 4096 which is after that first data block. * The program would have received the first text line (e.g., "text line 1\n") in an input variable. * The FILE structure's **base** pointer points to the front of the stdio read buffer. The FILE structure's **ptr** pointer points to the address immediately after the first text line. * The File structure's **cnt** would be set to the number of characters after the first text line.   What would change with the next call of fgets?   * Change count * ?? * ??   Would the SFT offset change due to that second call of fgets?  no  Physical I/O is much more time expensive than memory movement. Having multiple fgets returning data from a buffer instead of physical reads, greatly improves performance. | **Example 3-1: read buffering** |
| **open**  int **open**(const char \*pszFilename, int iFlags)  int **open**(const char \*pszFilename, int iFlags  , int iMode)   * opens a file and returns an file descriptor number or -1 if it couldn't be opened * If the file couldn't be opened, the variable **errno** is assigned a value and -1 is returned. A message corresponding to errno can be printed by calling **perror**. A string representing the meaning of errno can be returned using **strerror** which requires including **errno.h** and **string.h**. * The filename is the name of the file, possibly qualified with a path * The flags specify what you want to do with the file:   + O\_RDONLY - read only   + O\_WRONLY - write only   + O\_RDWR - both read and write * The following additional constants can be bitwise or'd with one of those values:   + O\_APPEND - if the file exists, append onto the end. If it doesn't exist, it starts at the beginning.   + O\_TRUNC - removes existing data   + O\_CREAT - creates a file if it doesn't exist.   + O\_EXCL - gives an error if the file already exists * iMode is an optional argument which specifies the file permissions. It must be specified if iFlags includes O\_CREAT. Constants according to the POSIX standard:   + S\_IRUSR - read by owner   + S\_IWUSR - write by owner   + S\_IXUSR - execute by owner   + S\_IRGRP - read by group   + S\_IWGRP - write by group   + S\_IXGRP - execute by group   + S\_IROTH - read by others   + S\_IWOTH - write by others   + S\_IXOTH - execute by others   These constants can be bitwise or'd.  Note that the value of the **umask** command can override the values of iMode since it specifies which bits cannot be set. umask 077 specifies that we can't turn on any group or other file permissions when creating a file. | **Example 3-2: open a file for read and another for write (which must not already exist)**  $ vi openclose.c  #include <fcntl.h>  #include <sys/stat.h>  #include <errno.h>  #include <string.h>  void errExit(const char szFmt[], ... );  int main()  {  int iFdRd;  int iFdWr;  // Open a file to read  iFdRd = open("readIt.txt", O\_RDONLY);  if (iFdRd < 0)  errExit("Open file for read: %s", strerror(errno));  // Open the file for create and give an error  // if it exists. Give permissions 00660  iFdWr = open("writeIt.txt", O\_WRONLY|O\_CREAT|O\_EXCL  , S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP);  if (iFdWr < 0)  errExit("Open file for write: %s", strerror(errno));  close(iFdRd);  close(iFdWr);  return 0;  }  $ gcc –o openclose openclose.c errExit.c  $ ./openclose  ERROR: Open file for read: No such file or directory  # create a simple "readIt.txt  $ cat > readIt.txt  how low can you go?  CTRL-D  $ ./openclose  # it created an empty file called writeIt.txt  $ ls –l writeIt.txt  -rw-rw---- 1 clark faculty 0 Sep 6 09:53 writeIt.txt  # Execute openclose again, but this time writeIt.txt exists  $ ./openclose  ERROR: Open file for write: File exists |
| **read**  int **read**(int fd, void \*psbBuf, long lCount)   * Beginning with the current file position, **read** reads **lCount** bytes into the buffer. The data read can be binary. * Returns (one of the following):   + number of bytes read   + 0 if there aren't any more bytes to read   + -1 if an error (e.g., file not open for read) * psbBuf is a void pointer so that it can be any structure or a character buffer. * **fd** must be a valid file descriptor number   This C code and the example data can be found in  **/usr/local/courses/rslavin/cs3423/IO**  **copy.c**  **address.txt**  **errExit.c** | **Example 3-3: program to copy a binary file**  $ vi copy.c  #include <stdio.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/stat.h>  #include <errno.h>  #include <string.h>  void errExit(const char szFmt[], ... );  #define BLOCK\_SZ 4096L  int main(int argc, char \*argv[])  {  int iFdRd; // FD for file to be read  int iFdWr; // FD for file to be written  char sbBuff[BLOCK\_SZ]; // buffer for copying data  int iRead; // number of bytes read  int iWrite; // number of bytes written  int iTotal = 0; // total number of bytes written  // check the number of arguments  if (argc < 3)  errExit("Too few arguments for copy");  // Open the file to read  iFdRd = open(argv[1], O\_RDONLY);  if (iFdRd < 0)  errExit("Open file for read: %s", strerror(errno));  // Open the file for create and give an error  // if it exists. Give permissions 00660  iFdWr = open(argv[2], O\_WRONLY|O\_CREAT|O\_EXCL  , S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP);  if (iFdWr < 0)  errExit("Open file for write: %s", strerror(errno));  printf("Files:\nFD Filename\n");  printf("%2d %s\n", iFdRd, argv[1]);  printf("%2d %s\n", iFdWr, argv[2]);  // read until EOF (iRead will be 0)  while ((iRead = read(iFdRd, sbBuff, BLOCK\_SZ)) > 0)  {  iWrite = write(iFdWr, sbBuff, iRead);  if (iWrite < 0)  errExit("writing copy: %s", strerror(errno));  iTotal += iWrite;  printf ("Read %d bytes, wrote %d bytes\n", iRead, iWrite);  }  close(iFdRd);  close(iFdWr);  printf("Total bytes written = %d bytes\n", iTotal);  return 0;  } |
| **write**  int **write**(int fd, void \*psbBuf, long lCount)   * Beginning with the current file position, **write** writes lCount bytes from the buffer to the file. . * Returns (one of the following):   + number of bytes written   + -1 if an error (e.g., file not open for write, ran out of space) * psbBuf is a void pointer so that it can be any structure or a character buffer. * **fd** must be a valid file descriptor number | See the code above for an example using open, read, write, and close.  $ gcc -o copy copy.c errExit.c  $ ./copy address.txt new.txt  Files:  FD Filename  3 address.txt  4 new.txt  Read 4096 bytes, wrote 4096 bytes  Read 484 bytes, wrote 484 bytes  Total bytes written = 4580 bytes  Would it be faster to use stdio binary i/o or the low-level I/O for a copy file utility? Why?  ?? |
| **lseek**  long **lseek**(int fd, long lOffset, int iSeekMode)   * Sets the position in the file based on an offset and iSeekMode. * Returns the new offset in bytes from the beginning of the file or -1 for an error. * Values of iSeekMode:   SEEK\_SET set the position relative to the beginning of the file  SEEK\_CUR seek from the current position  SEEK\_END seek from the end of the file  When using SEEK\_CUR and SEEK\_END, negative values for lOffset are allowed.   * The lOffset is a byte offset (relative to zero). * **fd** must be a valid file descriptor number | **Example 3-4**: lseek - suppose we read the first record and it tells us where the root of a tree (probably B+tree not binary tree) is located. Each node in the tree occupies an entire block. Partial code for btree.c.  iFd = open("btree.dat", O\_RDWR|O\_CREAT  , S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP);  if (iFd < 0)  errExit("Could not open B-Tree: %s", strerror(errno));  iRead = read(iFd, &controlStruc, sizeof(ControlStruc));  switch(iRead)  {  case 0: // file is empty  controlStruc.lRoot = 1;  bTreeNode.iNumEntries = 0;  iWrite = write(iFd, &controlStruc, sizeof(ControlStruc));  if (iWrite < 0)  errExit("write of control: %s", strerror(errno));  lSeekResult = lseek(iFd, sizeof(BTreeNode), SEEK\_SET);  if (lSeekResult < 0)  errExit("lseek to root: %s", strerror(errno));  iWrite = write(iFd, &bTreeNode, sizeof(bTreeNode));  if (iWrite < 0)  errExit("write of root: %s", strerror(errno));  break; // ignored since we exited  case -1:  errExit("B+Tree file read error: %s", strerror(errno));  break;  default: // B+tree already exists  printf("root is at %ld\n", controlStruc.lRoot);  lRBA = controlStruc.lRoot \* sizeof(bTreeNode);  lSeekResult = lseek(iFd, lRBA, SEEK\_SET);  if (lSeekResult < 0)  errExit("lseek to root: %s", strerror(errno));  // get the root node  iRead = read(iFd, &bTreeNode, sizeof(BTreeNode));  if (iRead == 0)  errExit("Could not read root, zero bytes returned");  else if (iRead < 0)  errExit("Root read failed: %s", strerror(errno));  printf("number of keys is %d\n", bTreeNode.iNumEntries);  }  close(iFd); |
| **stat**  int **stat**(const char \*pszFilename  , struct stat \*pStat)  int fstat(int fd, struct stat \*pStat)   * We must create a stat structure in our memory and pass its address. * stat and fstat return a inode info stat structure which contains:   st\_dev device containing file  st\_ino inode number  st\_mode file type and mode  st\_nlink number of links  st\_uid user Id of owner  st\_gid group id  st\_rdev device ID if a special type  st\_size total size in bytes  st\_blksize preferred block size for file system I/O  st\_blocks count of 512 byte blocks  st\_atime last access time  st\_mtime last modification time  st\_ctime last status change time   * Functionally returns 0 for success or -1 for error. | **Example 3-5**: provide a command to show the stat info for a file  $ vi statinfo.c  #include <stdio.h>  #include <stdlib.h>  #include <sys/stat.h>  #include <time.h>  #include <errno.h>  #include <string.h>  void errExit(const char szFmt[], ... );  int main(int argc, char \*argv[])  {  struct stat statBuf;  int rc;  // check the number of arguments  if (argc < 2)  errExit("Too few arguments for mystat\n");  rc = stat (argv[1], &statBuf);  if (rc < 0)  errExit("stat: %s", strerror(errno));  printf("stat information for '%s'\n", argv[1]);  printf("dev is %ld\n", statBuf.st\_dev);  printf("inode is %ld\n", statBuf.st\_ino);  printf("mode is %o\n", statBuf.st\_mode&07777);  printf("nlink is %ld\n", statBuf.st\_nlink);  printf("uid is %d\n", statBuf.st\_uid);  printf("gid is %d\n", statBuf.st\_gid);  printf("total size is %ld\n", statBuf.st\_size);  printf("device preferred blksize is %lu\n", statBuf.st\_blksize);  printf("number of 512 blocks is %ld\n", statBuf.st\_blocks);  printf("last accessed at %s", ctime(&statBuf.st\_atime));  printf("last modified at %s", ctime(&statBuf.st\_mtime));  printf("last status change at %s", ctime(&statBuf.st\_ctime));  return 0;  } |
| **We can examine our stat information using stat info.**   * The file permissions (mode) is 660. Prior to running the copy program, I had set umask to 007 which allowed me to use whatever I want for owner and group in my copy program. * The uid and gid are for my user ID and group. * The total size in bytes is the same as the size for address.txt. * The storage device used for most of our CS students is Network File System (NFS). Unfortunately, NFS doesn't provide a reasonable preferred block size. * The file has sixteen 512 byte blocks. That shows that we are actually using two 4096 byte blocks (i.e., 8,192 bytes) although our file only needs 4,580 bytes. | $ gcc –o statinfo statinfo.c errExit.c  $ ./statinfo new.txt  stat information for 'new.txt'  dev is 34  inode is 52693084  mode is 660  nlink is 1  uid is 1000  gid is 1000  total size is 4580 size in bytes  device preferred blksize is 1048576 NFS doesn't provide  number of 512 blocks is 16 512\*16 = 8192 (2 blocks)  last accessed at Wed Sep 6 08:10:58 2017  last modified at Wed Sep 6 08:10:58 2017  last status change at Wed Sep 6 08:10:58 2017 |
| **Using st\_mode to determine file type**  The following macros (defined in sys/stat.h) will help you determine the type of file based on the st\_mode returned by the stat function:  S\_ISDIR(*st\_mode*)  TRUE if the *st\_mode* shows that it is a directory.  S\_ISLINK(*st\_mode*)  TRUE if the *st\_mode* shows that it is a link.  S\_ISREG(*st\_mode*)  TRUE if the *st\_mode* shows that it is a regular file.  S\_ISFIFO(*st\_mode*)  TRUE if the *st\_mode* shows that it is a pipe. | **Example 3-6**: provide a command to show the type of file  $ vi stattype.c  #include <stdio.h>  #include <stdlib.h>  #include <sys/stat.h>  #include <time.h>  #include <errno.h>  #include <string.h>  void errExit(const char szFmt[], ... );  int main(int argc, char \*argv[])  {  struct stat statBuf;  int rc;  // check the number of arguments  if (argc < 2)  errExit("Too few arguments for mystat\n");  rc = stat (argv[1], &statBuf);  if (rc < 0)  errExit("stat: %s", strerror(errno));  // print the file type  if (S\_ISLNK(statBuf.st\_mode))  printf("link\n");  else if (S\_ISFIFO(statBuf.st\_mode))  printf("pipe\n");  else if (S\_ISREG(statBuf.st\_mode))  printf("file\n");  else if (S\_ISDIR(statBuf.st\_mode))  printf("directory\n");  else  printf("link\n");  return 0;  }  $ gcc –o stattype stattype.c errExit.c  $ ./stattype new.txt  file  $ ./stattype fruit  directory |
| **Directories**  There can be many directories on a file system. We have already discussed that Linux uses a hierarchical directory structure, but what is the actual implementation of a directory? A directory is a single file which contains a list of directory entries:  **file name** name of a file  **inode** index into an array of inodes  Effectively, a directory maps file names to inode values. Once we have a file's inode, we can then reference any of its data. To allow the internal implementation of a directory file to be implementation independent, Linux provides functions for creating, reading, and removing directories. |  |
| **opendir**  DIR \***opendir**(const char \*pszDirname)   * opens a directory for the specified directory name and returns a pointer to a DIR structure. * If the directory couldn't be opened, NULL is returned and the variable **errno** is assigned a value. A message corresponding to errno can be printed by calling **perror**. A string representing the meaning of errno can be returned using **strerror** which requires including **errno.h** and **string.h**. * The directory name is the name of the directory, possibly qualified with a path. * Include **<dirent.h>** to work with directory functions. | **Example 3-7 part 1**: provide a command to show the inode and filename in a directory  $ vi lsdir.c  #include <stdio.h>  #include <string.h>  #include <errno.h>  #include <dirent.h>  void errExit(const char szFmt[], ... );  int main(int argc, char \*argv[])  {  DIR \*pDir;  struct dirent \*pDirent;  if (argc < 2)  errExit("too few arguments, directory name needed");  pDir = opendir(argv[1]);  if (pDir == NULL)  errExit("opendir could not open '%s': %s", argv[1], strerror(errno)); |
| **readdir**  struct dirent \*readdir (DIR \*pDir)   * Given a directory pointer opened by an opendir call, it returns a pointer to the next directory entry. If there are no more entries, NULL is returned. * Each entry contains at least the following information:   + **d\_ino** – inode of the entry   + **d\_name** – filename for the entry * Include **<dirent.h>** to work with directory functions.   Note: there was a **readdir\_r** which was supposed to fix problems in a multi-threaded situation. There were many problems with the interface causing readdir\_r to be deprecated. | **Example 3-7 part 2**: provide a command to contents of a directory  // show information about each directory entry  printf("%-10s %s\n", "INODE", "FILENAME");  while ((pDirent = readdir(pDir)) != NULL)  {  printf("%10ld %s\n", pDirent->d\_ino, pDirent->d\_name);  }  closedir(pDir);  return 0;  }  $ gcc –o lsdir lsdir.c errExit.c  $ ./lsdir fruity  ERROR: opendir could not open 'fruity': No such file or directory  $ ./lsdir fruit  INODE FILENAME  54003713 .  52692795 ..  54003716 orange  54003717 maynard  54003715 Pear  54003718 clark  54003719 slavin |
|  |  |

**©2017 Larry W. Clark,** UTSA CS students may make copies for their personal use