UNIVERSITY OF EAST ANGLIA

School of Computing Sciences

CMP-5013A — Architectures and Operating Systems

LABORATORY 9 — Unix File I/O in C

1 Introduction

In this laboratory exercise, we will investigate some Unix system calls used to access the filing system, and their interface in the C programming language. This includes opening, closing, reading and writing, obtaining and modifying attributes and navigating through the hierarchical directory structure. The main programming exercise is in the last section, make sure you allocate sufficient time for its completion.

2 Basic File I/O System Calls

In CMP-5015Y (Programming 2), we have already seen some of the routines for File I/O provided by the C Standard Library (specifically `stdio.h`). As C and Unix developed together, it is not unduly surprising that these functions are closely related to the corresponding Unix system calls. For example, to open the file "`wombat.txt`" using the C standard library in read mode, we would write:

```
FILE *fd = fopen("wombat.txt", "r");
```

On a Unix system, this library routine is typically a just a *wrapper*, and the hard work is done by invoking the POSIX[1] system call `open`, however C also allows us to invoke `open` directly on Unix systems. The declaration of `open` in `fcntl.h` is as follows:

```
int open(const char *path, int flags, mode_t mode);
```

where `path` is a path specifying the file, `flags` is an integer, that serves a similar purpose to the string provided as the second argument to `fopen`, and the optional `mode` argument is used to specify the access permissions of a file created using `open`. The open command returns an positive integer value, which is effectively a file descriptor; if the system call fails, it returns -1 (and `errno` is set to indicate the reason for the failure). The header file provides symbolic constants representing individual bits of the `flags` argument, including (see `man open.2` for a full list):

`O_RDONLY` - open the file in read-only mode.

`O_WRONLY` - open the file in write-only mode.

`O_RDWR` - open the file for reading and writing.

[1]POSIX is a standardisation exercise designed to make programs written for Unix (and other POSIX compliant operating systems) more portable.

O_APPEND - open the file in append mode.

O_CREAT - if the file does not exist, create it as a regular file.

O_TRUNC - If the file already exists and the access mode allows writing (i.e., O_RDWR or O_WRONLY) it will be truncated to length 0.

Only one of O_RDONLY, O_WRONLYO_RDWR can be used, and the remaining flags are optional. The mode argument is optional, and its value is only used if a file is created (e.g. O_CREAT). The following symbolic constants are provided (note this is one of the rare occasions where octal literal constants are useful) :

S_IRWXU = 00700 - user (file owner) has read, write, and execute permission.

S_IRUSR = 00400 - user has read permission.

S_IWUSR = 00200 - user has write permission.

S_IXUSR = 00100 - user has execute permission.

S_IRWXG = 00070 - group has read, write, and execute permission.

S_IRGRP = 00040 - group has read permission.

S_IWGRP = 00020 - group has write permission.

S_IXGRP = 00010 - group has execute permission.

S_IRWXO = 00007 - others have read, write, and execute permission.

S_IROTH = 00004 - others have read permission.

S_IWOTH = 00002 - others have write permission.

S_IXOTH = 00001 - others have execute permission.

It is reasonable to ask why C provides both versions. The main reason is that while Unix systems use small integer values as file descriptors (possibly indices into a table of the attributes of currently open files), other operating systems may require more attributes to describe an open file, so a pointer (perhaps to a struct) provides a flexible approach, where differences between operating systems can be abstracted by the standard library. Direct access to the system call, on the other hand, allow the programmer to exploit features specific to Unix (see man open.2 for details of some of the more advanced features of open).

If we want to create a new file, we can also do this using the creat system call:

```
int creat(const char *pathname, mode_t mode);
```

where the arguments and return value are similar to those for open. When we wish to close an open file, we can use the close system call (declared in unistd.h):

```
int close(int fd);
```

Again a return value of -1 indicates that an error occurred.

2.1 Reading and Writing Data

The POSIX standard provides the system calls `read` and `write` (declared in `unistd.h`) for reading/writing data from/to an open file:

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Again, these are very similar in nature to the `fread` and `fwrite` functions provided by the standard C library, except that the file descriptor is an integer value rather than a `FILE*` pointer. In both cases, the system calls return the number of bytes successfully read/written, and -1 if an error occurred (and `errno` set appropriately). Data is read into, or taken from, a buffer, `buf`, of size `count` bytes. Below a minimal program is given that copies one file into another, where the files are specified by the command line arguments (note error checking has been omitted for brevity, however systems code should always check, where necessary, for errors generated by system calls):

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE (4096)

int main(int argc, char *argv[])
{
   int infd  = open(argv[1], O_RDONLY);
   int outfd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);

   char buffer[BUFFER_SIZE];

   int count;

   while ((count = read(infd, buffer, BUFFER_SIZE)) > 0)
   {
      write(outfd, buffer, count);
   }

   close(outfd);
   close(infd);

   return EXIT_SUCCESS;
}
```

If time permits, as an additional programming exercise, extend this minimal program to provide appropriate error handling and error messages, however it is more important that you first tackle the more substantive programming exercises found below.

2.2 Random Access Files

We are not limited to accessing files in strict sequential order. The `lseek` system call (declared in `unistd.h`) can be used to change the current position in the file at which reads and writes take place:

```
off_t lseek(int fd, off_t offset, int whence);
```

The first argument is the file descriptor and the second is an offset specifying the desired location in the file. The third argument is an integer specifying how the `offset` is to be interpreted and the following symbolic constants are provided:

`SEEK_SET` - the offset is relative to the start of the file.

`SEEK_CUR` - the offset is relative to the current position within the file.

`SEEK_END` - the offset is relatvie to the current end of the file.

Note the `offset` can be both positive and negative, allowing us to travel backwards as well as forwards through the file (`off_t` is a signed integral type guaranteed a to be large enough to accommodate any file size the filing system can support). The return value is an integer giving the offset relative to the start of the file after a successful seek. If the seek fails, a negative value is returned and `errno` set accordingly.

James Joyce's "Ulyssess" is a rather long novel, and it contains a rather long palindromic word. If we know it starts with the 1,421,941st character in the file `ulysses.txt`, we can use `lseek` to find it, without having to load the megabyte of data that precedes it. The program `raf.c` prints that palindrome on the console (note again the error handling has been omitted for brevity):

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define BUFFER_SIZE (256)

int main(int argc, char *argv[])
{
   int infd  = open("ulysses.txt", O_RDONLY);

   char buffer[BUFFER_SIZE];

   lseek(infd, 1421941, SEEK_SET);

   read(infd, buffer, BUFFER_SIZE);
```

```
    // replace the first space with the null terminator character

    *strchr(buffer, ' ') = '\0';

    printf("%s\n", buffer);

    close(infd);

    return EXIT_SUCCESS;
}
```

2.3 Files with Holes

Random access files can be very useful in implementing a database, where the computer has insufficient memory to load the whole database into memory at once. Instead we can save the database as a single file, where the file begins with an index, where each record is described by a “key” used to retrieve the record, and an offset into the file specifying where the record itself can be found. This way only the index needs to be loaded into memory, and specific records located using the index and then read in, as required, using `lseek` and `read` (or `write` if the record is to be updated). One problem with this is that we don’t necessarily know how many records we might want to store in the database at the time the database file is created. One solution is to allocate perhaps the first 1Mb of the file for the index, in the hope that this will be sufficient, at least in the short term. However, this is rather wasteful of disk-space if the actual index is much smaller.

Unix file systems allow us to create files with a “hole” in them, using `lseek`. As we saw in the lectures, each file is described by an *i-node*, which stores the attributes of the file and also the block numbers of the disk blocks comprising the contents of the file. For very large files, i-nodes have single- double- and triple-index blocks. They key point is that these blocks are only assigned to the i-node when we write to the corresponding section of the file, and otherwise they remain at zero (or a value that doesn’t correspond to a valid block on the disk). If we write a short index at the start of the file, and then use `lseek` to move to a position 1Mb from the start before writing the database records, we have created a file with an apparent size of over 1Mb, but only enough disk blocks have been allocated to hold the data we have actually written. This means we haven’t actually wasted much, if any disk-space by allocating 1Mb to hold the index!

The program `hole.c`, shown on the following page, demonstrates the construction of a file, `hole.txt`, with two substantial holes in the middle. Again error handling has been omitted for brevity, your code should always include appropriate error handling whenever system calls are made.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define BUFFER_SIZE (256)

int main(int argc, char *argv[])
{
   int outfd = open("hole.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR);

   char buffer[BUFFER_SIZE];

   snprintf(buffer, BUFFER_SIZE, "this is a very large file,\n");

   write(outfd, buffer, strlen(buffer));

   // seek 1Mb in from the start of the file

   lseek(outfd, 1<<20, SEEK_SET);

   snprintf(buffer, BUFFER_SIZE, "with two very large holes,\n");

   write(outfd, buffer, strlen(buffer));

   // seek 2Mb in from the start of the file

   lseek(outfd, 1<<21, SEEK_SET);

   snprintf(buffer, BUFFER_SIZE, "so it takes up little disk space!\n");

   write(outfd, buffer, strlen(buffer));

   close(outfd);

   return EXIT_SUCCESS;
}
```

If we run this program, and list the files in the directory, we obtain:

```
[gcc@cmp-pi32 labsheet9]$ ./hole
[gcc@cmp-pi32 labsheet9]$ ls -ls
total 1540
   4 -rw-r--r-- 1 gcc cmpsupport     446 Nov 22 17:16 copy.c
  12 -rwxr-xr-x 1 gcc cmpsupport    8448 Nov 24 12:01 hole
   4 -rw-r--r-- 1 gcc cmpsupport     831 Nov 24 12:00 hole.c
  12 -rw------- 1 gcc cmpsupport 2097186 Nov 24 12:01 hole.txt
   4 -rw-r--r-- 1 gcc cmpsupport     238 Oct  8 14:50 open_test.c
   4 -rw-r--r-- 1 gcc cmpsupport     439 Nov 24 11:13 raf.c
1500 -rw-r--r-- 1 gcc cmpsupport 1534156 Nov 24 11:06 ulysses.txt
[gcc@cmp-pi32 labsheet9]$
```

Note that although the new file `hole.txt` is much larger than `ulysses.txt` (sixth column), it only takes up 12Kb in blocks actually allocated on the disk, whereas `ulysses.txt` takes up 1500K in allocated blocks. In this case, as `hole.txt` consists of three short sentences, with large gaps inbetween, it seems reasonable to suppose that only three blocks on the disk were allocated, which suggests that the block size used on the disk is 4Kb. If we use `cat` to display the contents of `hole.txt` on the console, we see:

```
[gcc@cmp-pi32 labsheet9]$ cat hole.txt
this is a very large file,
with two very large holes,
so it takes up little disk space!
[gcc@cmp-pi32 labsheet9]$
```

However, we notice a significant pause after the first and second lines. This is because if we try to `read` from a hole in a file, it is as if we were reading from a block that contained zeros, and these take time to `read` and for `cat` to process 1Mb of zeros, hence the pauses. Fortunately, zero is the null-terminator character, `'\0'`, which `cat` sensibly ignores!

3 Navigating Through the Hierarchical Directory Structure

To find out the current working directory in which a process is operating, we can use the `getcwd` system call (declared in `unistd.h`:

```
   char *getcwd(char *buffer, size_t size);
```

The caller must provide a sufficiently large `buffer` (of `size` bytes) to hold the absolute path of the current working directory. The system call returns the address of the buffer if successful and NULL if it fails (for instance because the buffer is too small). We can use this call to implement a simplified version of the unix command `pwd` that simply prints the current working directory on the screen (note, as usual, the error handling has been omitted for brevity):

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE (256)

int main(int argc, char *argv[])
{
   char cwd[BUFFER_SIZE];

   printf("%s\n", getcwd(cwd, BUFFER_SIZE));

   return EXIT_SUCCESS;
}
```

Running this program gives an output simmilar to that shown below:

```
[gcc@cmp-pi32 labsheet9]$ ./pwd
/nfs/pihome/gcc/labsheets/labsheet9
[gcc@cmp-pi32 labsheet9]$
```

If we want to change the current working directory, this can be accomplished via the `chdir` system call (again, declared in `unistd.h`):

```
int chdir(const char *path);
```

If the system call fails, perhaps because the directory at the specified `path` does not exist, the return value is a negative integer, and `errno` is set approriately. If we want to create a new directory, the `mkdir` system call is used (declared in `sys/stat.h`):

```
int mkdir(const char *path, mode_t mode);
```

where `mode` is used to specify attributes, such as access permissions. A variety of other basic system commands are defined by the POSIX standard, and it is well worth investigating the API documents to doscover what is available.

3.1 Reading a Directory

A directory on a Unix file system is essentially just a file that contains the names of the files and directories it contains, each with an i-node number, specifying the i-node that contains its attributes and the block numbers of the blocks used to store the contents (in the case of a regular file). Obviously access to these files has to be strictly controlled, and this is achieved using the POSIX system calls `opendir`, `readdir` and `closedir` (the relevant declarations are found in `sys/types.h` and `dirent.h`. The system calls `opendir` and `closedir` are analogous to the C standard library calls for opening and closing files, `fopen` and `fclose`, except a pointer to a `DIR` structure is used as a directory descriptor.

```
DIR *opendir(const char *name);

int closedir(DIR *dirp);

struct dirent *readdir(DIR *dirp);
```

Once a directory has been opened, the entries it contains, each representing a file or sub-directory, can be read sequentially using the `readdir` system call. This returns a structure containing the following information about each file:

```
struct dirent {
   ino_t          d_ino;
   off_t          d_off;
   unsigned short d_reclen;
   unsigned char  d_type;
   char           d_name[256];
};
```

Only two fields are standard and present on all Unix systems: `d_ino`, which records the i-node number of the file and `d_name`, which is a null-terminated string containing it's name. For the sake of portability, it is probably best to ignore the other fields. When there are no more entries left, `readdir` returns a `NULL` pointer. This is one of the system calls where it is important to set `errno` to zero before making the system call, because a `NULL` return is also used to indicate that the call failed, but if that happened, it will have set `errno` to an appropriate value. The program shown below lists the i-node numbers and names of all of the files contained in the directory `/usr/bin` (there are a *lot* of them!).

```
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   DIR *dp = opendir("/usr/bin");
   struct dirent *entry;

   while ((entry = readdir(dp)) != NULL)
   {
      printf("%8d - %s\n", entry->d_ino, entry->d_name);
   }

   closedir(dp);

   return EXIT_SUCCESS;
}
```

4 Reading and Modifying File Attributes

As well as holding the block numbers of the disk blocks used to store the contents of a file, the i-nodes also store the file attributes, and there are system calls that allow us to read and (in some cases) modify those attributes. The key system call is `stat` (the necessary declarations can be found in `sys/types.h`, `sys/stat.h` and `unistd.h`

```
int stat(const char *pathname, struct stat *statbuf);
```

The first argument is the pathname for the file, and the second is a pointer to a structure in which to store the attributes, with the following key fields[2] (see `man stat.2` for further details):

```
struct stat {
   dev_t     st_dev;       /* ID of device containing file  */
   ino_t     st_ino;       /* Inode number                  */
   mode_t    st_mode;      /* File type and mode            */
   nlink_t   st_nlink;     /* Number of hard links          */
   uid_t     st_uid;       /* User ID of owner              */
   gid_t     st_gid;       /* Group ID of owner             */
   dev_t     st_rdev;      /* Device ID (if special file)   */
   off_t     st_size;      /* Total size, in bytes          */
   blksize_t st_blksize;   /* Block size for filesystem I/O */
   blkcnt_t  st_blocks;    /* No. of 512B blocks allocated  */
   time_t    st_atime;     /* Time of last access           */
   time_t    st_mtime;     /* Time of last modification     */
   time_t    st_ctime;     /* Time of last status change    */
}
```

The most important field is probably `st_mode`, which indicates whether the file is a regular file, or a directory, or a block special file etc. Fortunately `sys/stat.h` provides several macros that we can use to test for particular types of file, for instance `S_ISDIR()` returns `true` if the argument represents a directory and `false` otherwise, and `S_ISREG` returns `true` if it represents a regular file. The following program, `numblocks.c` computes the total number of 1Kb blocks allocated to all of the files in `/use/bin` (error handling ommited for brevity):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
```

[2]It is actually a bit more complicated than shown here, specifically modern versions of Unix allow times to be recorded with precision up to the nearest nanosecond (although it isn't always actually available). However, it is backwards-compatible with times recorded to the nearest second, using the fields shown here, but this explains the difference between that shown here and that by `man stat.2` .

```c
#define BUFFER_SIZE (256)

int main(int argc, char *argv[])
{
   DIR           *dp = opendir("/usr/bin");
   char           buffer[BUFFER_SIZE];
   struct dirent *entry;
   blkcnt_t       blocks = 0;

   while ((entry = readdir(dp)) != NULL)
   {
      struct stat sb;

      snprintf(buffer, BUFFER_SIZE, "/usr/bin/%s", entry->d_name);

      stat(buffer, &sb);

      if (S_ISREG(sb.st_mode) != 0)
      {
         blocks += sb.st_blocks;
      }
      else
      {
         printf("ignoring '%s'\n", entry->d_name);
      }
   }

   closedir(dp);

   printf("%d blocks are allocated to regular files in /usr/bin\n",
      blocks);

   return EXIT_SUCCESS;
}
```

Running this program gives the following output:

```
[gcc@cmp-pi32 labsheet9]$ ./numblocks
ignoring '.'
ignoring '..'
ignoring 'core_perl'
ignoring 'vendor_perl'
ignoring 'site_perl'
530552 blocks are allocated to regular files in /usr/bin
[gcc@cmp-pi32 labsheet9]$
```

Note that the directories `.` and `..`, representing the directory itself and the parent directory have entries in the directory file, as well as there being three regular sub-directories in `/usr/bin`.

The field `st_mode` also contains the file access permissions for the user, group and others, stored in the lowest nine bits, just as in the `mode` argument for the `open` system call, described on page 1. The following system calls and library functions are useful for interpreting some of the other fields:

`getpduid` - can be used to obtain a string containing the name of the user with the specified user ID (`st_uid`). This is stored in the password file (`/etc/passwd`) and the other fields can also be retrieved (don't worry, it isn't possible to get access to the unencrypted password this way!)

`getgrgid` - can be used to obtain a string containing the name of the primary group to which the owner of the file belongs (the group ID is stored in the field `st_gid`). This is stored in the local group file (`/etc/group`).

`strftime` - can be used to obtain a string representing a date and time encoded as a structure where the year, month, day, hour, minute and seconds are broken down into different fields.

`localtime` - used to break up a time specified by a `time_t` value (representing the elapsed seconds since the start of 01/01/1970 into a `struct` that can be used with `strftime`.

In addition, the attributes of a file may be updated using system calls such as `chmod` and `utime`, etc.

5 Programming Exercise

The programming task is to implement a program that prints information about the files and directories in a specified directory, using the same format used by the Unix command `ls -ls`. The directory to be listed must be specified as a command line argument; if no command line arguments are provided, list the files in the current working directory.

Most of the system calls required for this task have been introduced in this labsheet, except the system calls relating to times and dates. A key skill in systems programming is learning to use the `man` command to find information about system calls, and this provides an opportunity to hone those skills (hint: `man -k time`). Note that modification times are shown differently for recently modified files and those that have not been modified for some time (hint: the Unix `touch` command can be used to set the last modification time of a file).

Dr. Gavin Cawley
November 24, 2018