

# CSCI131

## Introduction to Computer Systems

### Task Group 3: Linux OS interface and OS responsibilities

*Complete the exercises, and try to get them marked, before attempting the assignment!*

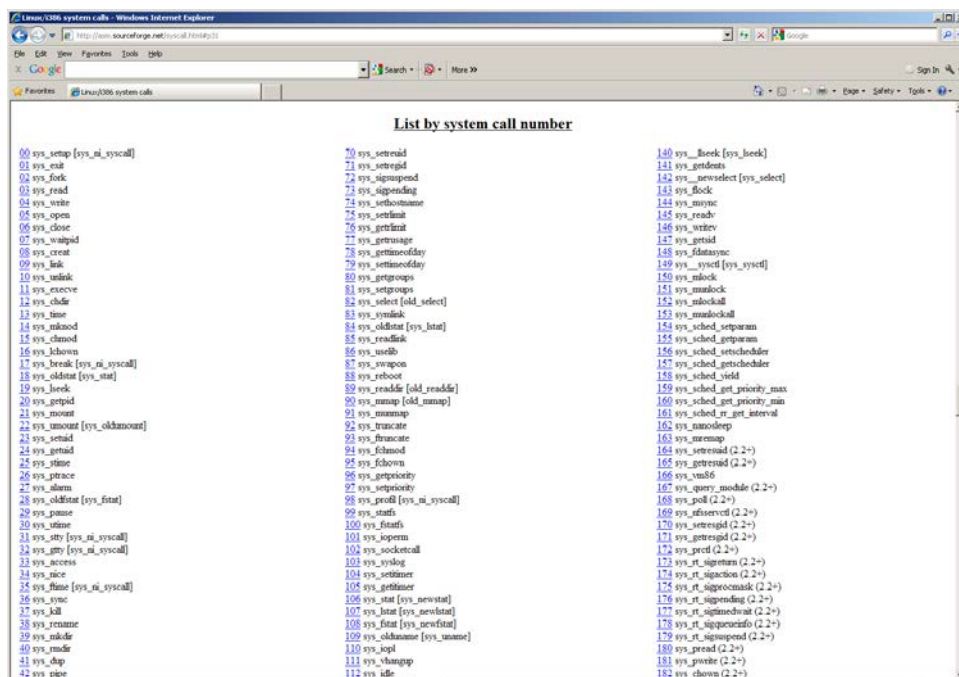
There are several exercise tasks in this task group as detailed below. At the end of this document there is a link to the assignment. You should try to complete all the exercise tasks before attempting the assignment task.

The exercises focus on some of the standard Unix/Linux system calls – calls used to control processes, and calls for handling low-level I/O.

The assignment will be slightly different. For the assignment you will implement a simulation of one of major tasks of an OS – possibly a simple file management system, or a memory management system (something like an implementation of malloc – the standard library function that looks after heap memory as acquired using the “brk” supervisor call), or maybe a process management system (allocation of CPU to ready tasks etc).

### Linux system calls

There are many:



The screenshot shows a web browser window with the title "Linux/UNIX system calls - Windows Internet Explorer". The address bar shows a URL from sourceforge.net. The main content area displays a list of system calls, titled "List by system call number". The list is organized into three columns, showing system call numbers, names, and their corresponding header files in square brackets.

List by system call number		
00 sys_setup [sys_setup]	70 sys_setreuid	140 sys___block [sys_block]
01 sys_exit	71 sys_setregid	141 sys_getdents
02 sys_fork	72 sys_sigpending	142 sys___newselect [sys_select]
03 sys_read	73 sys_sigpending	143 sys_block
04 sys_write	74 sys_setsockopt	144 sys_mmap
05 sys_open	75 sys_setlimit	145 sys_readv
06 sys_close	76 sys_getlimit	146 sys_writev
07 sys_waitpid	77 sys_getrusage	147 sys_getid
08 sys_creat	78 sys_gettimeofday	148 sys_fdatasync
09 sys_link	79 sys_settimeofday	149 sys___sysctl [sys_sysctl]
10 sys_unlink	80 sys_getgroups	150 sys_mlock
11 sys_execve	81 sys_setgroups	151 sys_munlock
12 sys_chdir	82 sys_select [old_select]	152 sys_mlockall
13 sys_time	83 sys_syslink	153 sys_munlockall
14 sys_mknod	84 sys_oldlstat [sys_lstat]	154 sys_sched_getparam
15 sys_chmod	85 sys_readlink	155 sys_sched_getparam
16 sys_lchown	86 sys_uselib	156 sys_sched_setscheduler
17 sys_break [sys_brk]	87 sys_sbrk	157 sys_sched_getscheduler
18 sys_oldstat [sys_stat]	88 sys_reboot	158 sys_sched_yield
19 sys_block	89 sys_readlink [old_readlink]	159 sys_sched_get_priority_max
20 sys_getpid	90 sys_mmap [old_mmap]	160 sys_sched_get_priority_min
21 sys_mount	91 sys_munmap	161 sys_sched_rr_interval
22 sys_umount [sys_oldumount]	92 sys_truncate	162 sys_nanosleep
23 sys_setuid	93 sys_ftruncate	163 sys_mremap
24 sys_getuid	94 sys_fchmod	164 sys_setreuid (2.2+)
25 sys_time	95 sys_fchown	165 sys_getreuid (2.2+)
26 sys_ptrace	96 sys_getpriority	166 sys_valid
27 sys_alarm	97 sys_setpriority	167 sys_query_module (2.2+)
28 sys_oldlstat [sys_lstat]	98 sys_profile [sys_profile]	168 sys_poll (2.2+)
29 sys_pause	99 sys_stats	169 sys_afservic (2.2+)
30 sys_vfsync	100 sys_fstatfs	170 sys_setregid (2.2+)
31 sys_vfsync [sys_vfsync]	101 sys_sopen	171 sys_getregid (2.2+)
32 sys_vfsync [sys_vfsync]	102 sys_socketcall	172 sys_prl (2.2+)
33 sys_access	103 sys_syslog	173 sys_rt_sigreturn (2.2+)
34 sys_mlock	104 sys_setitimer	174 sys_rt_sigaction (2.2+)
35 sys_fstat [sys_fstat]	105 sys_getitimer	175 sys_rt_sigprocmask (2.2+)
36 sys_sync	106 sys_stat [sys_newstat]	176 sys_rt_sigpending (2.2+)
37 sys_kill	107 sys_lstat [sys_newlstat]	177 sys_rt_sigtimedwait (2.2+)
38 sys_rename	108 sys_fstat [sys_newfstat]	178 sys_rt_sigqueueinfo (2.2+)
39 sys_mkdir	109 sys_olduname [sys_uname]	179 sys_rt_sigpending (2.2+)
40 sys_mkdir	110 sys_ioctl	180 sys_pread (2.2+)
41 sys_dup	111 sys_vhangup	181 sys_pwrite (2.2+)
42 sys_pipe	112 sys_fide	182 sys_chown (2.2+)

You will use only a few directly in your future programming – calls for low-level I/O, calls for process management, and calls for inter-process communication. The exercises illustrate some of the more frequently use system calls.

### ***Task 1: Scrambling around a file hierarchy: 2 marks***

For this task, you simply copy my example code for scrambling around a file hierarchy. The actual application is to search for ASCII text files that contain a given string; it searches through the files in a given directory and, recursively, searches through subdirectories.

This application utilises system calls like read, lstat, readdir along with others that now are mainly used via C wrapper functions.

The program operates by:

- Reading (C's scanf on stdin) the search string and the path for the initial directory
- Invoking a recursive search function that operates on a directory:
  - For each entry in the current directory, read in a descriptive data structure:
    - Ignore special files
    - If an entry represents a sub-directory, make a recursive call
    - If an entry is a file, read (at byte level) the file contents; check whether all bytes appear to be ASCII – do not further process files with non-ASCII characters; search ASCII files for the string.

Create a NetBeans C application project:

```
/*
 * File:   main.c
 * Author: neil
 *
 * Scramble recursively through a file hierarchy examining files
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <limits.h>
#include <unistd.h>
#include <dirent.h>
#include <errno.h>
#include <ctype.h>

static char searchstr[150];

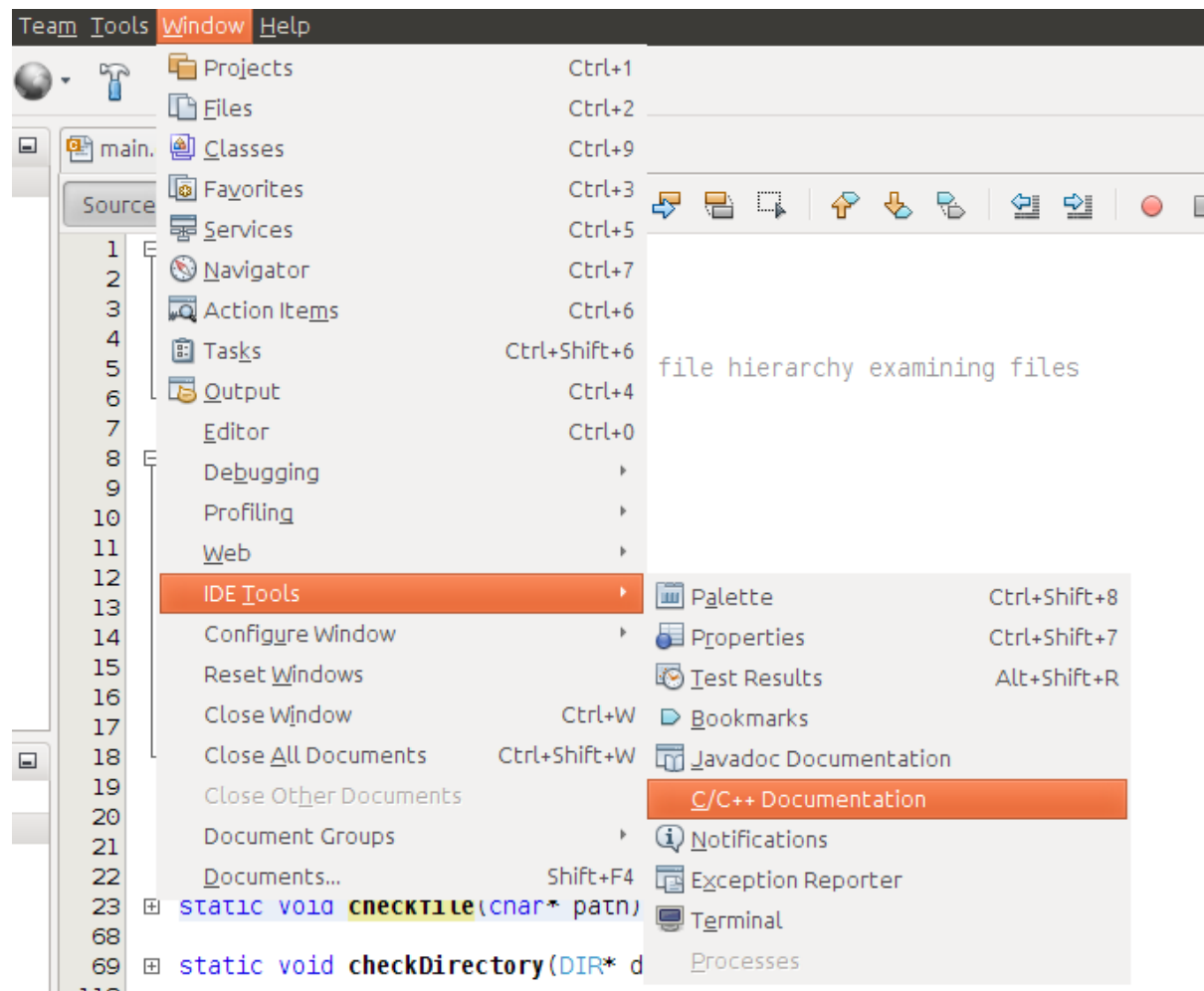
static void checkfile(char* path) { ...45 lines }

static void checkDirectory(DIR* dir, char* name) { ...49 lines }

int main(int argc, char** argv) { ...20 lines }
```

It uses lots of #include files! As Unix/Linux grew, new features were added to the standard library “libc” or placed in new libraries. These new features each had their own header and implementation file.

Of course, you will need to view the definitions of the various system calls and C wrapper functions. These will have man page entries. Fortunately, man pages can be accessed from within the NetBeans IDE. Open the C/C++ documentation window:



This will simplify access to the man page documentation.

### main

The main function reads in two strings, and then attempts to open the directory with the given pathname:

```

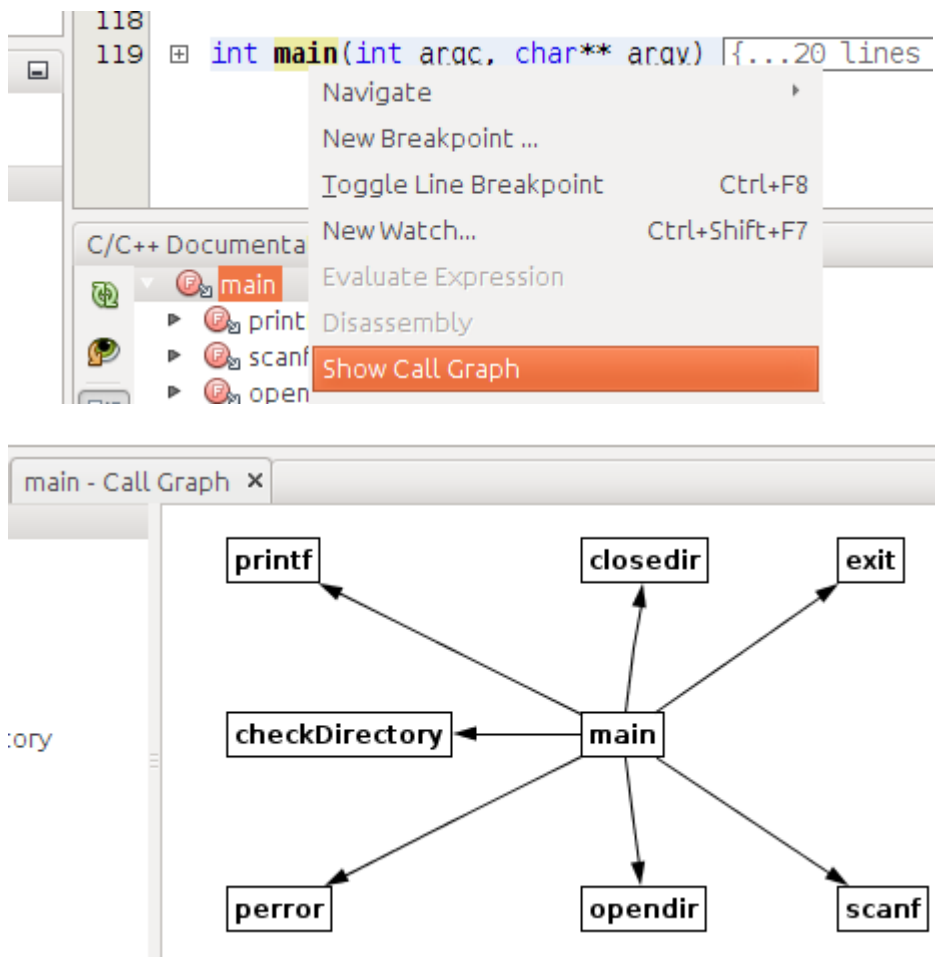
int main(int argc, char** argv) {

    printf("Searching for files with given string that exist in directory or subdirectories\n");
    printf("Enter string whose occurrences are to be found : ");
    scanf("%s",searchstr);
    printf("Enter directory path name : ");
    char rootdirectory[PATH_MAX];
    scanf("%s",rootdirectory);
    DIR *dir = opendir(rootdirectory);
    if (dir == NULL) {
        perror("Failed to open that directory : ");
        exit(1);
    }
    printf("\nStarting search for files with string - %s\n",searchstr);
    printf("#occurrences   File name\n");
    checkDirectory(dir, rootdirectory);

    closedir(dir);
    return 0;
}

```

Another way of looking at the code is via the “call graph”:



(A call graph is often a useful supplement to your documentation on functions that you write.)

The real work starts with the attempt to open the directory with the pathname given as input. This uses the `opendir` function; this is a C wrapper function that uses the `open()` system call on a directory. If successful, the `opendir` function returns a pointer to an opaque data structure (application programmers are not supposed to see how the OS represents directories). Select the

function name to get the man page displayed in the documentation pane of the NetBeans IDE window:

```

127 DIR *dir = opendir(rootdirectory);
128 if (dir == NULL) {
129     perror("Failed to open that directory : ");

```

C/C++ Documentation × main - Call Graph Output

Function **DIR\* opendir(const char\* \_\_name)**

OPENDIR(3) Linux Programmer's Manual OPENDIR(3)

NAME

    opendir, fdopendir - open a directory

SYNOPSIS

```

#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
DIR *fdopendir(int fd);

```

Feature Test Macro Requirements for glibc (see feature\_test\_macros(7)):

```

fdopendir():
    Since glibc 2.10:

```

(Note how man page entries identify the header files that you will need to include.)

You can view the header file with the definition of a function signature, or the declaration of a struct, by selecting the name and right-clicking:

```

119 int main(int argc, char** argv) {
120
121     printf("Searching for files with given string that exist in directory o
122     printf("Enter string whose occurrences are to be found : ");
123     scanf("%s",searchstr);
124     printf("Enter directory path name : ");
125     char rootdirectory[PATH_MAX];
126     scanf("%s",rootdirectory);
127     DIR
128     if
129
130
131

```

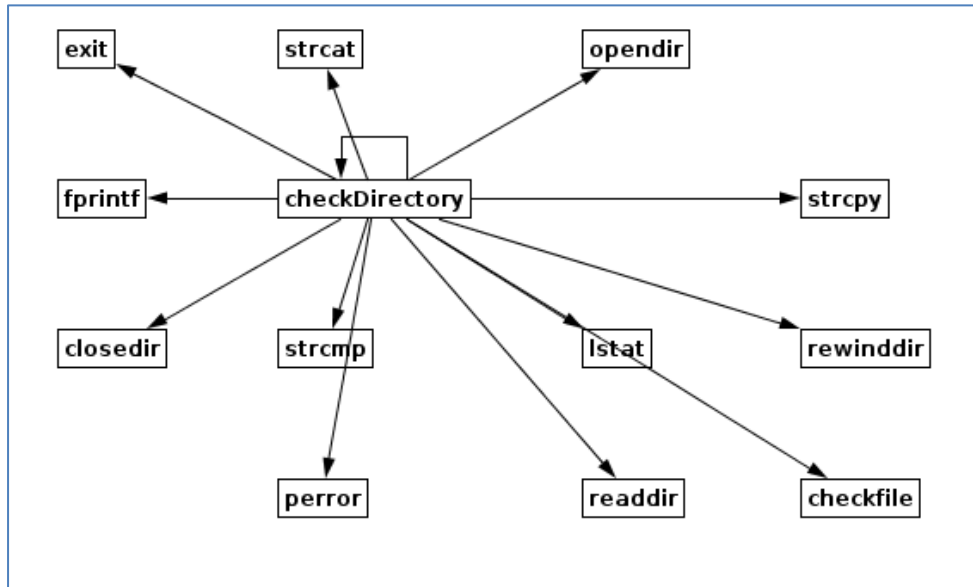
Navigate Go to Declaration/Definition

New Breakpoint ... Go to Override/Overridden Ctrl+Alt+B

Toggle Line Breakpoint Ctrl+F8 Go to Header Ctrl+Shift+A

### *checkDirectory*

The checkDirectory function has the following (recursive) call graph and definition:



```

static void checkDirectory(DIR* dir, char* name) {

    struct dirent *dp;
    rewinddir(dir);
    while ((dp = readdir(dir)) != NULL) {
        char* aname = dp->d_name;
        if (0 == strcmp("../", aname)) continue;
        if (0 == strcmp("./", aname)) continue;

        unsigned char entrytype = dp->d_type;
        if ((entrytype == DT_BLK) || (entrytype == DT_CHR)
            || (entrytype == DT_FIFO) || (entrytype == DT_SOCK)) {
            continue;
        }

        char path[1025];
        strcpy(path, name);
        strcat(path, "/");
        strcat(path, aname);

        struct stat finfo;
        int statresult;
        statresult = lstat(path, &finfo);
        if (statresult != 0) {
            perror("stat failed for file :");
            exit(1);
        }
        if (entrytype == DT_DIR) {
            DIR *subdir;
            subdir = opendir(path);
            if (subdir == NULL) {
                if (errno == EACCES) {
                    fprintf(stdout, "Cannot access %s\n", path);
                    continue;
                } else {
                    perror("opendir failed");
                    exit(1);
                }
            }
            checkDirectory(subdir, path);
            closedir(subdir);
        } else
            if (entrytype == DT_REG) {
                checkfile(path);
            }
    }
}

```

The “while loop” gets the next directory entry.

Directories always have entries representing the directory itself (“.”), and the parent directory (“..”) (well not always a parent, if you are at the top of the directory hierarchy you act as your own parent). Typically, the other entries will represent files or subdirectories. However, there are many strange kinds of object that *can* appear in a directory – FIFOs, sockets, “character special devices”, “block devices”. Remember on Unix/Linux, all I/O is handled through the file system, and so everything – terminals, network connections, ... - gets represented by an entry somewhere in the file system. Generally, such special entries appear only in limited parts of the file system (e.g. under /proc); but they can occur anywhere. You wouldn’t want to try to process any such special file types

– so the code discards these. (What are the types? Well, check via the documentation – just follow the “definition / declaration link”:

```
enum
{
    DT_UNKNOWN = 0,
# define DT_UNKNOWN    DT_UNKNOWN
    DT_FIFO = 1,
# define DT_FIFO       DT_FIFO
    DT_CHR = 2,
# define DT_CHR        DT_CHR
    DT_DIR = 4,
# define DT_DIR        DT_DIR
    DT_BLK = 6,
# define DT_BLK        DT_BLK
    DT_REG = 8,
# define DT_REG        DT_REG
    DT_LNK = 10,
# define DT_LNK        DT_LNK
    DT SOCK = 12,
# define DT SOCK       DT SOCK
    DT_WHT = 14,
# define DT_WHT        DT_WHT
};
```

If the entry corresponds to a sub-directory or a regular file, it will be processed. First, the code builds the full path name for the entry, then it invokes the `lstat()` system call to get more information about the entry. Look up the documentation on `stat()` and `lstat()`. Why did the code use `lstat()`?

#### C/C++ Documentation

Function **int lstat(const char\* \_\_file, stat\* \_\_buf)**

STAT(2)

Linux Programmer's Manual

STAT(2)

#### NAME

stat, fstat, lstat - get file status

#### SYNOPSIS

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

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

(Note that this is a man(2) page; the previously shown documentation for `opendir()` was a man(3) page. The man(2) pages are for system calls that you make directly; the man(3) pages are for C wrapper functions around system calls.)

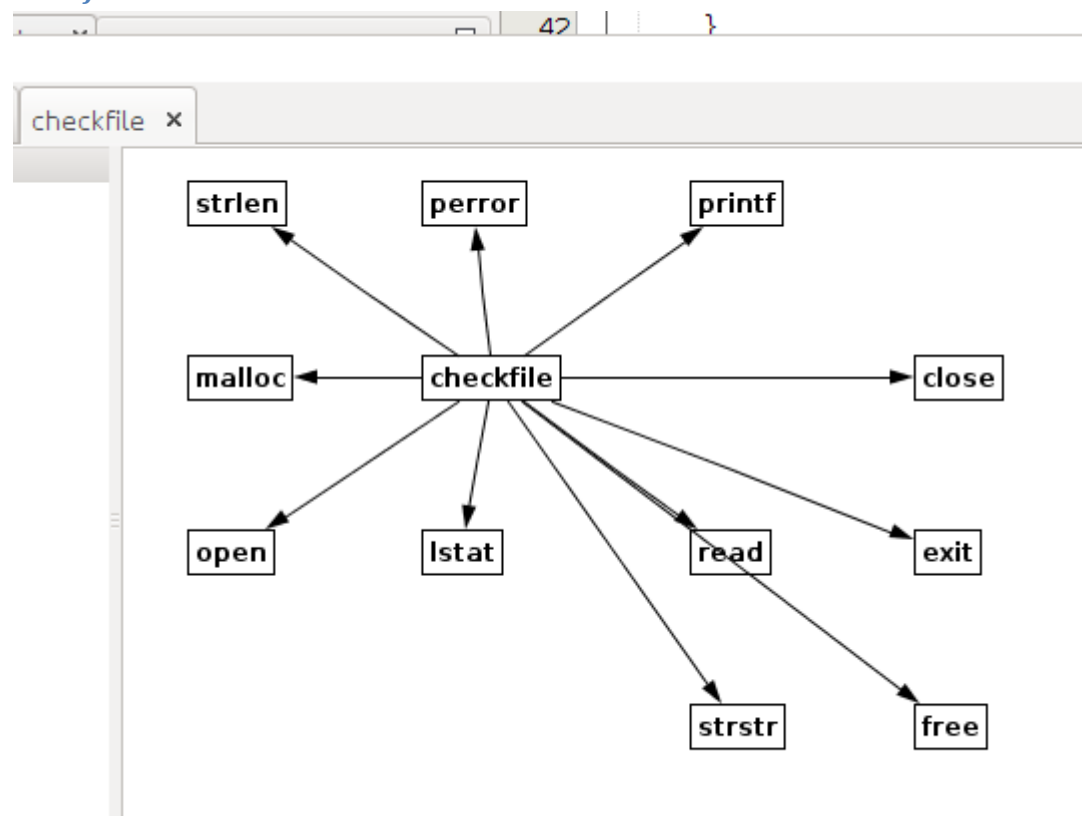
The `lstat` call is really just verifying that the code can access a given directory entry.



If the current entry represents a sub-directory, this directory is opened and a recursive function call is made to checkDirectory. If the entry represents a file, the checkfile function is invoked:

```
if (entrytype == DT_DIR) {
    DIR *subdir;
    subdir = opendir(path);
    if (subdir == NULL) {
        if (errno == EACCES) {
            fprintf(stdout, "Cannot access %s\n", path);
            continue;
        } else {
            perror("opendir failed");
            exit(1);
        }
    }
    checkDirectory(subdir, path);
    closedir(subdir);
} else
    if (entrytype == DT_REG) {
        checkfile(path);
    }
```

### checkfile



The `checkfile` function first determines the length of the file (getting this information from a “struct stat” data structure returned by `lstat`). It allocates space, via `malloc`, to hold the contents of the entire file. The file is then read using the `read()` supervisor call. The code tries to make sure that large byte arrays created using `malloc` always get released via `free`.

```

23 static void checkfile(char* path) {
24     struct stat finfo;
25     int statresult;
26
27     statresult = lstat(path, &finfo);
28     if (statresult != 0) {
29         perror("stat failed for directory");
30
31         exit(1);
32     }
33     int filesize = finfo.st_size;
34     void* content = malloc(filesize);
35     int fd = open(path, O_RDONLY);
36     int numread = read(fd, content, filesize);
37     close(fd);
38     if (numread != filesize) {
39         printf("Unable to read %s fully\n", path);
40         free(content);
41         return;
42     }
43     int i;
44     char* p = (char*) content;
45     // Check that it is a simple text file
46     for (i=0; i<numread; i++) {
47         if (!isascii(*p)) {
48             free(content);
49             return;
50         }
51         p++;
52     }
53     int len = strlen(searchstr);
54     int count = 0;
55     p = (char*) content;
56     for(;;) {
57         char* res = strstr(p, searchstr);
58         if (res == NULL) break;
59         count++;
60         p = res + len;
61     }
62     free(content);
63     if (count > 0) {
64         printf("%4d          %s\n", count, path);
65     }
66 }

```

Once the file has been read into memory, the checkfile function scans the bytes checking that all correspond to ASCII characters (simple text). If other characters are found, the file is not further processed.

If it is an ASCII file, the checkfile function then searches for occurrences of the search string and prints a report.

Complete the creation of the project. Build it. Run it against a large directory (I tried a search for files containing the string “cout” starting in my home directory).

```

Searching for files with given string that exist in directory or subdirectories
Enter string whose occurrences are to be found : cout
Enter directory path name : /home/neil

Starting search for files with string - cout
#occurrences  File name
3              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtwinextras/examples/winextras/iconextractor/main.cpp
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdoc/doc/src/diagrams/dependencies.lout
3              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdoc/doc/src/snippets/qdir-filepaths/main.cpp
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdoc/doc/src/snippets/i18n-non-qt-class/myclass.cpp
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtquick1/tools/qmlplugindump/main.cpp
4              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/gnuwin32/man/cat1/flex.1.txt
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/gnuwin32/contrib/bison/2.4.1/bison-2.4.1-src/doc/bison.info
2              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/gnuwin32/contrib/gperf/3.0.1/gperf-3.0.1/check.log
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/gnuwin32/contrib/gperf/3.0.1/gperf-3.0.1-src/ChangeLog
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtscript/tests/benchmarks/script/sunspider/tests/string-tagcloud.
s
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtscript/src/3rdparty/javascriptcore/JavaScriptCore/ChangeLog-20K
-10-14
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdeclarative/tools/qmlimportscanner/main.cpp
2              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdeclarative/tools/qmlbundle/main.cpp
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdeclarative/tools/qmlmin/main.cpp
4              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdeclarative/tools/qmljs/main.cpp
1              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtdeclarative/tools/qmlplugindump/main.cpp
6              /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtbases/tests/manual/filetest/main.cpp
24             /home/neil/Downloads/qt-everywhere-opensource-src-5.2.0/qtbases/tests/manual/qpainter/main.cpp

```

### Gdb practice

Practice use of the debugger by inspecting the operation of the recursive checkDirectory function. Place a break point; then run under gdb control:

```

68
69  static void checkDirectory(DIR* dir, char* name) {
70
71      struct dirent *dp;
72      rewinddir(dir);
73      while ((dp = readdir(dir)) != NULL) {

```

```
67 }
68
69 static void checkDirectory(DIR* dir, char* name) {
70     struct dirent *dp;
71     rewinddir(dir);
72     while ((dp = readdir(dir)) != NULL) {
73         char* aname = dp->d_name;
74         if (0 == strcmp("../", aname)) continue;
75         if (0 == strcmp("./", aname)) continue;
76
77         unsigned char entrytype = dp->d_type;
78         if ((entrytype == DT_BLK) || (entrytype == DT_CHR)
79             || (entrytype == DT_FIFO) || (entrytype == DT SOCK)) {
80             continue;
81         }
82     }
83 }
```

Variables | Call Stack x | Breakpoints

Name
checkDirectory (dir=0x61b0d0, name=0x7ffffffc6d0 \"/home/neil/Downloads/qt-everywhere-oper
checkDirectory (dir=0x613090, name=0x7ffffffc6d0 \"/home/neil/Downloads/qt-everywhere-opensource
checkDirectory (dir=0x60b050, name=0x7ffffffd0b0 \"/home/neil/Downloads\")
checkDirectory (dir=0x603010, name=0x7ffffffd4f0 \"/home/neil\")
main (argc=1, argv=0x7ffffffe5e8)

### “revision”

Make sure that you understand all the system calls and C wrapper functions that are used in this example.

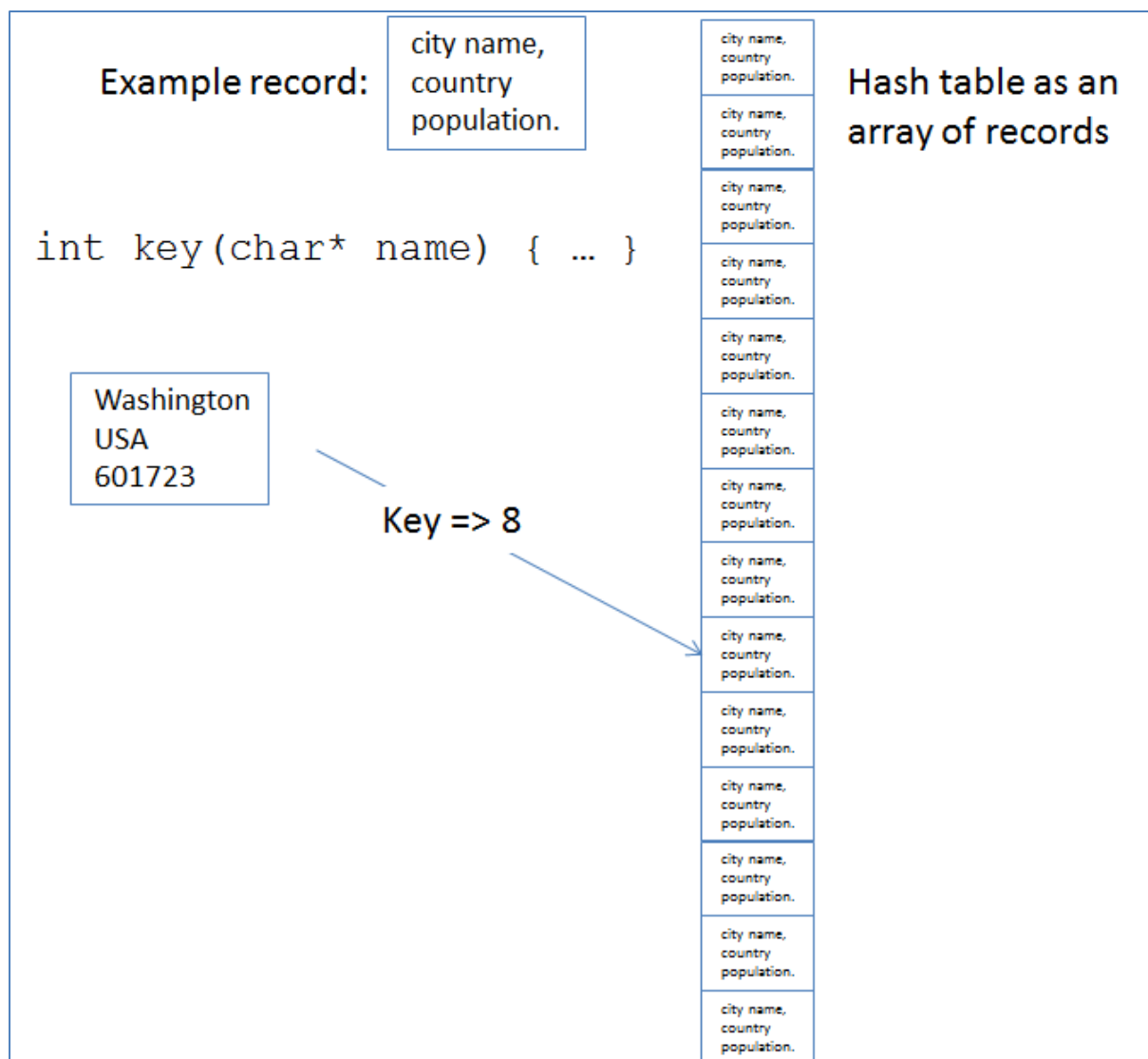
- closedir
- free
- lstat
- malloc
- opendir
- perror
- read (byte read system call)
- readdir (there is an archaic readdir system call, but actually using a C wrapper function)
- rewinddir

## Task 2: Byte level file access: 2 marks

For this task, you simply copy my example code for manipulating a binary file.

This example illustrates a simple “hash table” similar to those that you will probably have learnt about in CSCI103. In CSCI103, you might have had an example where *records were inserted into a hash table, and then subsequently lookup requests were made to retrieve data.*

You will have been shown a hash table that was an array of records. Some of the data in a record could be processed to generate a **key**. This key was to act as the array index. When adding data, the key was where you tried to place a record. The subsequent lookup operations used the key to find the complete record

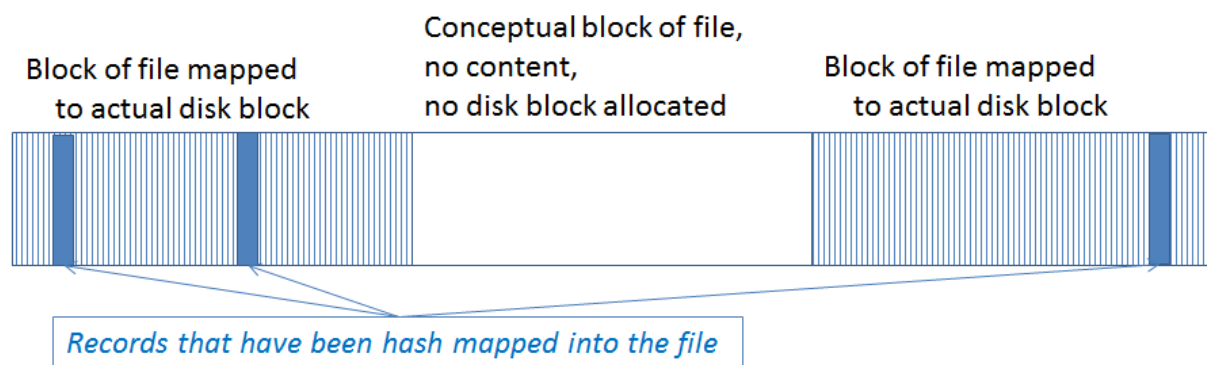


Of course, sometimes you got “hash collisions” where two different records ended up with the same key. In the case of a “hash collision” you had to try to find another place in the table where the record might be stored. You will have been shown some scheme for working cyclically around through the table looking for a suitable spot for the data (and maybe you were also shown other more sophisticated schemes such as “bucket lists”).

The program for this task performs similar operations. There is one difference. *This hash table is kept on disk and could be very large (many megabytes).*

The keys will now determine a byte position in the file where there will be a record-sized block of bytes for the record with that key. These record-sized blocks will be stored in actual disk blocks (disk blocks would typically hold 4kbytes, i.e. a few records, but some Linux systems now use 8kbytes or even 16kbyte blocks and so can hold many records in a single disk block).

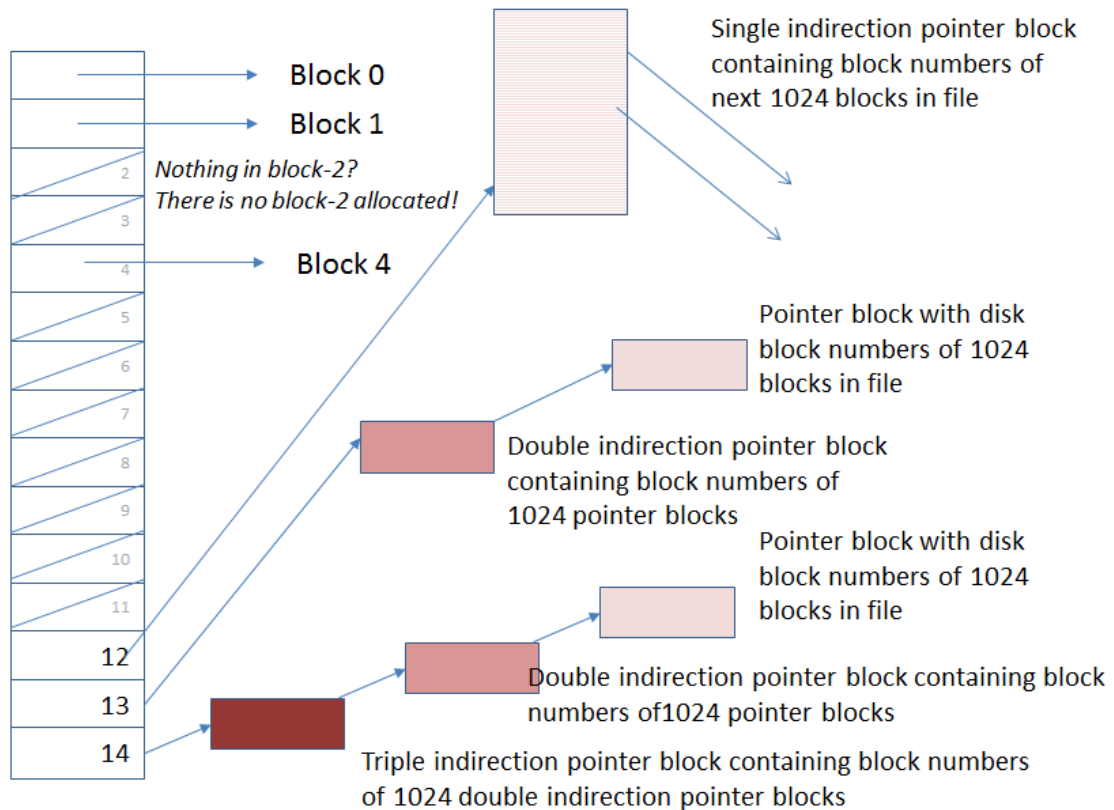
Your hash table, as a memory array, had blank entries where there were no records. Similarly, on disk the file will have blank sections. If a specific disk block contains some actual records, the rest of its space (corresponding to blank sections) will be filled with zero-valued bytes.



What about disk blocks that do not yet contain any records?

This depends on the OS. Unix/Linux allows for “sparse files” (explained in [http://en.wikipedia.org/wiki/Sparse\\_file](http://en.wikipedia.org/wiki/Sparse_file)). A file can conceptually have empty blocks – but these don’t get allocated any space on disk. It all works through the “i-node” pointer scheme that will be discussed briefly in lectures.

# i-node – pointers to blocks



## Example data

The data records contain information about city populations. There are almost 700 (no, you don't have to type them in, the data file is supplied). The data file has city-name, country-name, a 0/1 flag value (1=>capital city), a population, and a continent. The cities are either in Africa or North America. The file should be in <http://www.uow.edu.au/~nabg/131/Resources/cities.txt>.

```
cities.txt x
Ouagadougou Burkina_Faso 1 1475223 Africa
Ouahigouya Burkina_Faso 0 73153 Africa
Solenzo Burkina_Faso 0 16850 Africa
Tenkodogo Burkina_Faso 0 44491 Africa
Bafoussam Cameroon 0 239287 Africa
Bamenda Cameroon 0 269530 Africa
Douala Cameroon 0 1907479 Africa
Garoua Cameroon 0 235996 Africa
Kumba Cameroon 0 144268 Africa
Maroua Cameroon 0 201371 Africa
Ngaoundere Cameroon 0 152698 Africa
Nkongsamba Cameroon 0 104050 Africa
Yaounde Cameroon 1 1817524 Africa
6th_of_October_City Egypt 0 802306 Africa
Alexandria Egypt 0 4358439 Africa
Assyut Egypt 0 991929 Africa
```

The program will use simple structs for these data:

```
44 struct record {
45     char city[40];
46     char country[16];
47     char continent[20];
48     int cap;
49     int pop;
50 };
51
52 typedef struct record Record;
53 static int recordsize = sizeof (Record);
```

### *Program operation*

In the first phase of the program, the data are read, and records are written to a giant hash table represented by a binary file on disk. In the second phase of the program, a user can enter a city name and country, and the program will look for a corresponding record, printing out details if a record is found.

```
Enter city name and countryNames with spaces should be entered with underscore characters
    e.g. 'Baton Rouge' would be entered as 'Baton_Rouge'
Search for:
Paris
France
Sorry, have no data for that city
Search for:
Washington
USA
Washington, USA, in North_America
    Has a population of 601723
    It is the nation's capital
Search for:
Baton_Rouge
USA
Baton_Rouge, USA, in North_America
    Has a population of 229493
Search for:
```

### *System calls and C wrappers used*

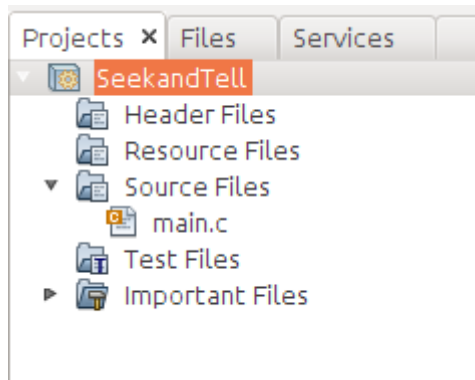
This application uses:

- read and write – supervisor calls used to read or write blocks of bytes
- open – supervisor call, open a file descriptor for byte level data transfers
- fopen – C wrapper working with stdio, open a FILE\* stream for input from the data file
- unlink – supervisor call, it's approximately a "delete" request
- lseek – supervisor call, positions input/output point for a file descriptor (byte level I/O stream)
- fsync – supervisor call, guarantees writes to disk completed (and not left in OS's memory buffers)

The application also illustrates some practical bit-manipulations. These are in the function that generates a hash key from something like a city name.



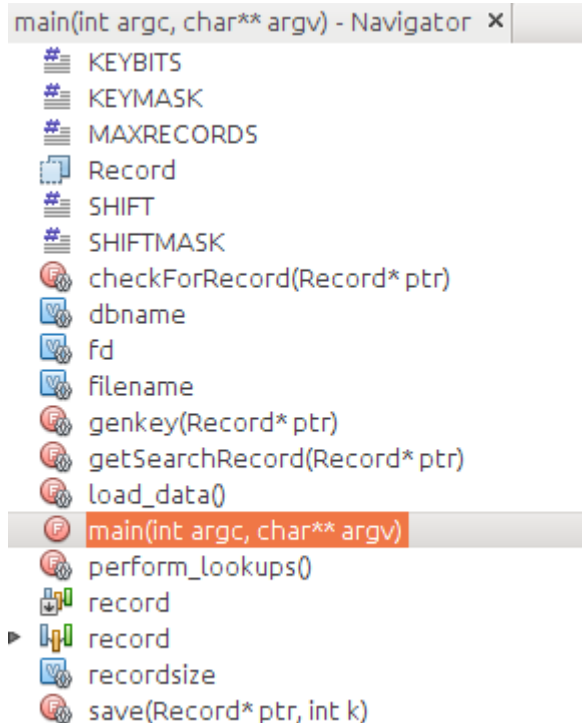
## A new NetBeans C application project



This has some

- #define statements that effectively parameterize the key generation function;
- A struct definition and associate typedef
- Assorted constant 'variables' such as file names and a measure of a record size
- A few functions
- Main()

As program source files get longer (this one is about 300 lines), the “Navigator” pane in NetBeans starts to become useful. You click on an entry in the Navigator and the editor pane will scroll to the definition of that element.



```

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

// Excessive 131072 table
#define MAXRECORDS 0x20000
#define KEYBITS 17
#define SHIFT 3
#define SHIFTMASK 0xe0000
#define KEYMASK 0x1ffff

//...20 lines

struct record {...7 lines };

typedef struct record Record;
static int recordsize = sizeof (Record);

static char* filename = "cities.txt";
static char* dbname = "/tmp/nabgdata";

static int fd; // file descriptor for random access file

static int genkey(Record* ptr) {...26 lines }

static void save(Record* ptr, int k) {...66 lines }

static void load_data() {...31 lines }

static int getSearchRecord(Record *ptr) {...27 lines }

static void checkForRecord(Record *ptr) {...45 lines }

static void perform_lookups() {...14 lines }

int main(int argc, char** argv) {...10 lines }

```

### Main

The main program invokes the load\_data function which builds the binary file. It then lets the user enter search requests that are run against the file. Finally it deletes the file. You should create the file in /tmp. All processes have the right to create files in /tmp. Such files don't count against any quota. The OS is entitled to delete these files whenever it wishes.

```

275 int main(int argc, char** argv) {
276     printf("Record size is %d, ~700 records is %d\n", recordsize, 700 * recordsize);
277     printf("Key length is %d bits, number records %d, potential file size %d\n",
278           KEYBITS, MAXRECORDS, MAXRECORDS * recordsize);
279     fd = open(dbname, O_RDWR | O_CREAT | O_TRUNC);
280     load_data();
281     perform_lookups();
282     unlink(dbname);
283     return (EXIT_SUCCESS);
284 }

```

### #define

```

16 // Excessive 131072 table
17 #define MAXRECORDS 0x20000
18 #define KEYBITS 17
19 #define SHIFT 3
20 #define SHIFTMASK 0xe0000
21 #define KEYMASK 0x1ffff
22
23 // Sensible 32768 table
24 // #define MAXRECORDS 0x8000
25 // #define KEYBITS 15
26 // #define SHIFT 3
27 // #define SHIFTMASK 0x38000
28 // #define KEYMASK 0x1ffff
29
30 // Small 2048 table
31 // #define MAXRECORDS 0x800
32 // #define KEYBITS 11
33 // #define SHIFT 3
34 // #define SHIFTMASK 0x1c00
35 // #define KEYMASK 0x7ff
36
37 // Too small 256 table
38 // #define MAXRECORDS 0x100
39 // #define KEYBITS 7
40 // #define SHIFT 3
41 // #define SHIFTMASK 0x380
42 // #define KEYMASK 0x7f

```

You can try running the application with different table sizes for the hash table and see how this impacts on operation. The different groups of #define statements provide consistent sets of values for key length, “masks” for bit manipulation etc.

### Struct and typedef

```
44 struct record {
45     char city[40];
46     char country[16];
47     char continent[20];
48     int cap;
49     int pop;
50 };
51
52 typedef struct record Record;
53 static int recordsize = sizeof (Record);
54
```

### Load\_data

This function loops reading lines from the cities.txt input file. It copies data into a "Record". It next invokes a function that generates the key for that record. The key and record are passed to the save() function that will attempt to write the record to the disk file. The loop terminates at the end of the input file (it really should have closed the file (fclose()) just for tidiness sake).

```
static void load_data() {
    FILE* input = fopen(filename, "r");

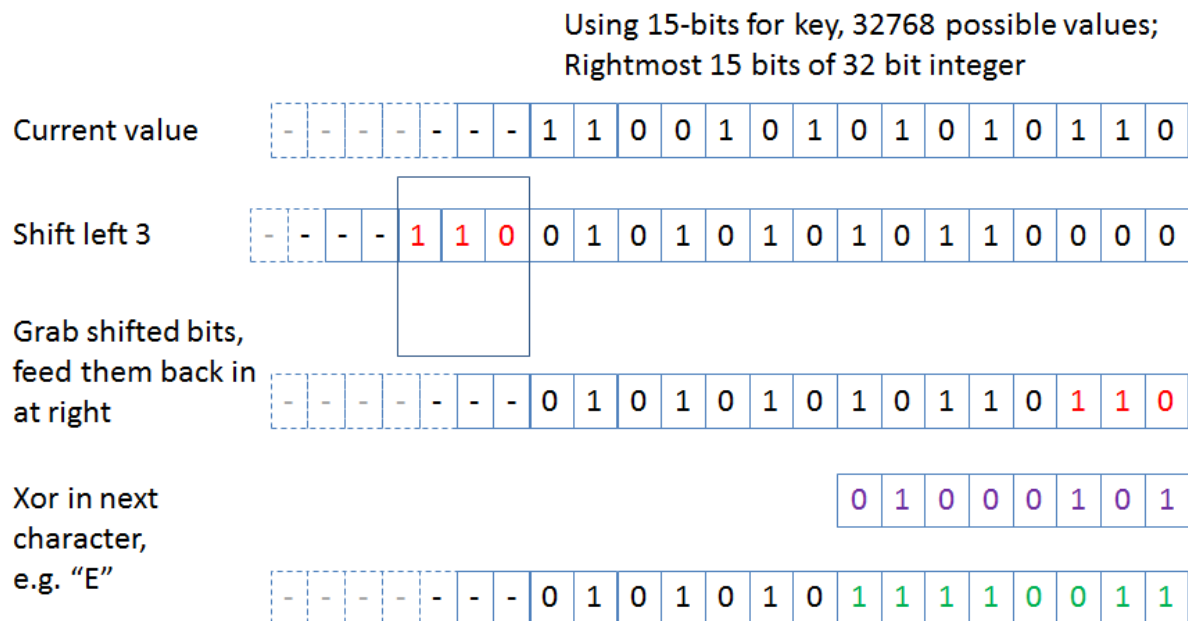
    for (;;) {
        int n;
        char cityname[256];
        char countryname[256];
        int capital;
        int population;
        char continent[256];
        n = fscanf(input, "%s %s %d %d %s", cityname, countryname, &capital,
            &population, continent);
        if (feof(input))
            break;
        if (n != 5) {
            printf("Error at %scityname", cityname);
            exit(1);
        }
        Record r;
        memset(&r, 0, recordsize);
        strncpy(r.city, cityname, 39);
        strncpy(r.country, countryname, 15);
        strncpy(r.continent, continent, 19);
        r.cap = capital;
        r.pop = population;
        int key = genkey(&r);
        save(&r, key);
    }
}
```

(Confirm that you understand the arguments to scanf and the use of functions like memset and strncpy. If these functions are new to you, use the NetBeans C/C++ documentation viewer to read their man page descriptions.)

### genkey

This function just performs a little bit-manipulation to generate a hash key based on both the city name and country name. The approach is standard; you may have been shown something similar in CSCI103.

The key is allocated some number of bits, e.g. 15; it's "masked" against an appropriate bit-pattern to limit it to this size. Characters from a string are xor-ed into the mask. The mask gets shifted leftwards. Bits that move left out of the key are fed back in on the right (if you don't do this, the generated key depends only on the last few characters of the string).



```

static int genkey(Record* ptr) {
    int k = 0;
    // add in letters of city name first
    char* cptr = ptr->city;
    while (*cptr) {
        k = k << SHIFT; // move left
        int save = k & SHIFTMASK; // save bits
        save = save >> KEYBITS;
        k = k & KEYMASK;
        k = k ^ save;
        k = k ^ (*cptr);
        cptr++;
    }
    // and now the country name
    cptr = ptr->country;
    while (*cptr) {
        k = k << SHIFT; // move left
        int save = k & SHIFTMASK; // save bits
        save = save >> KEYBITS;
        k = k & KEYMASK;
        k = k ^ save;
        k = k ^ (*cptr);
        cptr++;
    }
    return k;
}

```

### save

This function attempts to save the new record at an appropriate point in the (disk-based) hash-table.

Of course, it has to handle the possibilities of collisions and of the hash table (disk file) becoming completely full. If the file is full, it can do nothing better than print a warning message and terminate the program. If there was a collision, it must work cyclically round through the table looking for a spot. It's all pretty similar to examples that you should have seen before using an in-memory array of structs.

```

87 static void save(Record* ptr, int k) {
88     int startposn = k;
89     int posn = startposn;
90     int collisions = 0;
91     for (;;) {
92         Find next position
101        Check for record at that point
111        Beyond end of file
129
130        Found spot?
149        Hash collision?
162     }

```

Because this table is on disk, there are a few complications. The key must be converted into a byte offset – not hard, just multiply by record size. We must then check to see if there is already an entry at that point. So, seek to that byte offset into the file (lseek returns a –ve value on error).

```
static void save(Record* ptr, int k) {
    int startposn = k;
    int posn = startposn;
    int collisions = 0;
    for (;;) {
        //<editor-fold defaultstate="collapsed" desc="Find next position">
        long bytuposn = posn*recordsize;
        // Move to that spot
        off_t spot = lseek(fd, bytuposn, SEEK_SET);
        if (spot < 0) {
            perror("Disk seek failed");
            exit(1);
        }
        //</editor-fold>
    }
}
```

Next, try to read in a record (read will return a negative value on error, a zero value, or a count of bytes actually read).

```
91     for (;;) {
92         Find next position
101        //<editor-fold defaultstate="collapsed" desc="Check for record at that point">
102        // Try reading in a record from that spot
103        Record existrec;
104        memset(&existrec, 0, recordsize);
105        int numread = read(fd, &existrec, recordsize);
106        if ((numread < 0) || ((numread > 0) && (numread != recordsize))) {
107            perror("Disk read failed");
108            exit(1);
109        }
110        //</editor-fold>
    }
```

Initially, the file has zero size; so any offset is beyond “the end of file”. Later, we will generate keys that are greater than any existing key and so again end up beyond “the end of file”.

The OS doesn’t really object to your reading beyond the end of file – it simply tells you that zero bytes can be read.

If we do find that we are beyond the end of file, we can write the new record at this position and all is done. (Of course, we have to reposition the file offset to the start of the record before doing the write byte operation.) The file will grow by one real block (and maybe many conceptual but unallocated blocks).

```

91     for (;;) {
92         Find next position
101        Check for record at that point
111        //<editor-fold defaultstate="collapsed" desc="Beyond end of file">
112        if (numread == 0) {
113            // We have found a spot beyond the end of the file
114            spot = lseek(fd, byteposn, SEEK_SET);
115            if (spot < 0) {
116                perror("Disk seek failed");
117                exit(1);
118            }
119            int numwritten = write(fd, ptr, recordsize);
120            if (numwritten != recordsize) {
121                perror("Write to disk failed");
122                exit(1);
123            }
124            fsync(fd);
125            //printf("Put %s, %s at %d\n", ptr->city, ptr->country, posn);
126            return;
127        }
128        //</editor-fold>

```

Another possibility is that we end up at a point in an already allocated disk block. But at a point where there is no actual record. This isn't a problem. The OS has filled the bytes with zeros. We will read in a record that contains all zero bytes. (If we have ended up at in a conceptual block of the file that is empty and so hasn't been allocated an actual disk block, the OS again tells us that it this hypothetical disk block is filled with zeros, and gives a record containing all zero bytes.) In this example, we can make a simple check for an empty record by checking for a zero length city name.

If we find such a record, then once again we have got the point where the record should exist. We can reposition the file descriptor's access point and write the new record.

```

91     for (;;) {
92         Find next position
101        Check for record at that point
111        Beyond end of file
129
130        //<editor-fold defaultstate="collapsed" desc="Found spot?">
131        // Could have read an empty record - a good place
132        // to put the new record
133        if (0 == strlen(existrec.city)) {
134            spot = lseek(fd, byteposn, SEEK_SET);
135            if (spot < 0) {
136                perror("Disk seek failed");
137                exit(1);
138            }
139            int numwritten = write(fd, ptr, recordsize);
140            if (numwritten != recordsize) {
141                perror("Write to disk failed");
142                exit(1);
143            }
144            fsync(fd);
145            //printf("Put %s, %s at %d\n", ptr->city, ptr->country, posn);
146            return;
147        }
148        //</editor-fold>

```

But if we did read in some actual data, then we have had a collision. We must try to find another point by working cyclically around the conceptual hash table.



```

91     for (;;) {
92         Find next position
101        Check for record at that point
111        Beyond end of file
129
130        Found spot?
149        //<editor-fold defaultstate="collapsed" desc="Hash collision?">
150        // A record was found - seem to have a collision
151        printf("%d : Collision at %d\n", ++collisions, posn);
152        printf("\t%s, %s\n", ptr->city, ptr->country);
153        printf("\t%s, %s\n", existrec.city, existrec.country);
154        // Advance to next hash location
155        posn = (posn + 17) % MAXRECORDS;
156        if (posn == startposn) {
157            // This check could fail if maxrecords were a multiple of 17
158            printf("No room in file\n");
159            exit(1);
160        }
161        //</editor-fold>

```

If you run the code with all the printf statements left in you can see how it saves data and deals with collisions.

```

Put Navojoa, Mexico at 30490
Put Netzahualcoyotl, Mexico at 62223
1 : Collision at 128964
    Nuevo_Laredo, Mexico
    Tbessa, Algeria
Put Nuevo_Laredo, Mexico at 128981
Put Oaxaca_de_Juarez, Mexico at 121881
Put Ocotlan, Mexico at 80096

```

### Perform lookups

This function allows the user to enter city and country name combinations attempting to retrieve these. The function `getSearchRecord` attempts to read data from input and uses them to fill a `Record`. The `checkForRecord` function then performs the lookup and search.

```

270 static void perform_lookups() {
271     printf("Enter city name and country");
272     printf("Names with spaces hould be entered with underscore characters\n");
273     printf("\te.g. 'Baton Rouge' would be entered as 'Baton_Rouge'\n");
274     for (;;) {
275         printf("Search for:\n");
276         Record r;
277         memset(&r, 0, recordsize);
278         int inpt = getSearchRecord(&r);
279         if (inpt < 0) return;
280         if (inpt == 0) continue;
281         checkForRecord(&r);
282     }
283 }

```

### getSearchRecord

This simply reads data from stdin.

```

196 static int getSearchRecord(Record *ptr) {
197     // Return -1 if end of input - program to exit
198     // Return 0 if invalid input - program to loop requesting data
199     // Return 1 with filled in record if data ok
200     char city[256];
201     char country[256];
202     int nread = scanf("%s", city);
203     if (nread <= 0) {
204         if (feof(stdin)) return -1;
205         return 0;
206     }
207     nread = scanf("%s", country);
208     if (nread <= 0) return 0;
209     int len = strlen(city);
210     if ((len < 2) || (len > 39)) {
211         printf("Invalid city name\n");
212         return 0;
213     }
214     len = strlen(country);
215     if ((len < 2) || (len > 15)) {
216         printf("Invalid country name\n");
217         return 0;
218     }
219     strncpy(ptr->city, city, 39);
220     strncpy(ptr->country, country, 15);
221     return 1;
222 }

```

### *checkForRecord*

Much of the processing is similar to the save function explained earlier. The record with city name and country name is processed to get a key that acts as the starting point for accessing the disk-based hash table. The function then attempts to read a record at that point.

If a zeroed out record is retrieved, it means that there are no data for the chosen city/country combination; the function informs the user and then returns.

If there was a record, one must check that it is the right one – the right city/country. After all, we could have got to a colliding record. If the record is correct, the other data can be reported. If it's not the right record, move on to the next spot in the table.

```

static void checkForRecord(Record *ptr) {
    int startposn = genkey(ptr);
    int posn = startposn;
    for (;;) {
        long byteposn = posn*recordsize;
        // Move to that spot
        off_t spot = lseek(fd, byteposn, SEEK_SET);
        if (spot < 0) {
            perror("Disk seek failed");
            exit(1);
        }
        // Try reading in a record from that spot
        Record existrec;
        memset(&existrec, 0, recordsize);
        int numread = read(fd, &existrec, recordsize);
        if ((numread < 0) || ((numread > 0) && (numread != recordsize))) {
            perror("Disk read failed");
            exit(1);
        }
        if (numread == 0) {
            // Have read from an unwritten part of file. There is no
            // such city/country combination
            printf("Sorry, have no data for that city\n");
            return;
        }
        // Check that record does match by city name and country name
        int cmp1 = strcmp(existrec.city, ptr->city);
        int cmp2 = strcmp(existrec.country, ptr->country);
        if ((0 == cmp1) && (0 == cmp2)) {
            // Found it
            printf("%s, %s, in %s\n",
                existrec.city, existrec.country, existrec.continent);
            printf("\tHas a population of %d\n", existrec.pop);
            if (existrec.cap) printf("\tIt is the nation's capital\n");
            return;
        }
        // It was simply a hash collision so move on
        posn = (posn + 17) % MAXRECORDS;
        if (posn == startposn) {
            // This check could fail if maxrecords were a multiple of 17
            printf("Sorry, have no data for that city\n");
            return;
        }
    }
}

```

Try to get the whole program to work, and try with different sizes for the hash key.

```
SeekandTell (Build) x SeekandTell (Build, Run) x SeekandTell (Run) x
Record size is 84, ~700 records is 58800
Key length is 17 bits, number records 131072, potential file size 11010048
1 : Collision at 128964
    Nuevo_Laredo, Mexico
    Tbessa, Algeria
Enter city name and countryNames with spaces hould be entered with underscore characters
    e.g. 'Baton Rouge' would be entered as 'Baton_Rouge'
Search for:
Paris
France
Sorry, have no data for that city
Search for:
Washington
USA
Washington, USA, in North_America
    Has a population of 601723
    It is the nation's capital
Search for:
Baton_Rouge
USA
Baton_Rouge, USA, in North_America
    Has a population of 229493
Search for:
```

### *That disk file*

How big would the file be?

A 17 bit key allows for 131072 records, each of 84 bytes and so a file size of 11Mbyte. Of course, there are less than 700 actual records, so the file will contain less than 58000 bytes of real data.

But how big is it?

You can comment out the “unlink” statement that deletes the file and check.

```
neil@neil-OptiPlex-760:/tmp$ ls -lksh nabgdata
2.5M --w--w---T 1 neil neil 11M Nov 27 10:29 nabgdata
```

While officially an 11Mbyte file, it actually occupied 2.5Mbyte on my disk.

Why 2.5Mbyte, there was only 58kbyte of data?

Well, as soon as a single record got written to a block of the file, and that block got mapped to disk, it chewed up a whole block of disk (which could be as much as 16kbyte to hold one 84 byte record).

### *“revision”*

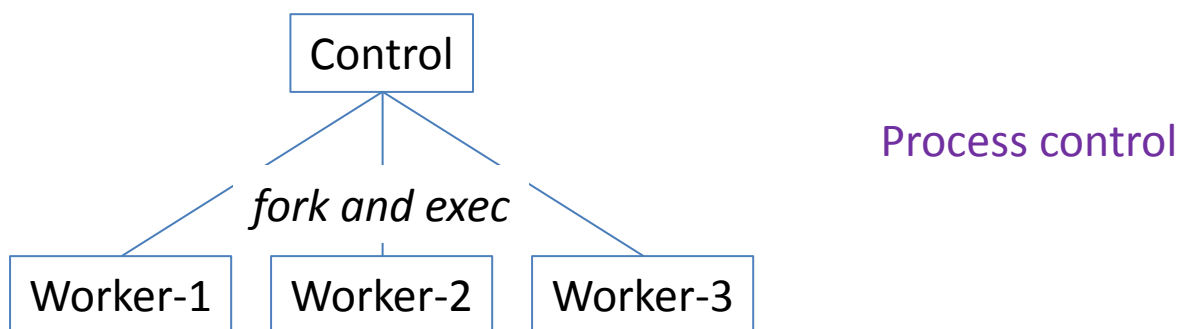
Make sure that you understand all the system calls and C wrapper functions that are used in this example. Also, read up more about how Linux records blocks for files and how it uses those inode structures, and how these allow for sparse files.

### Task 3: Process control and communication: 3marks

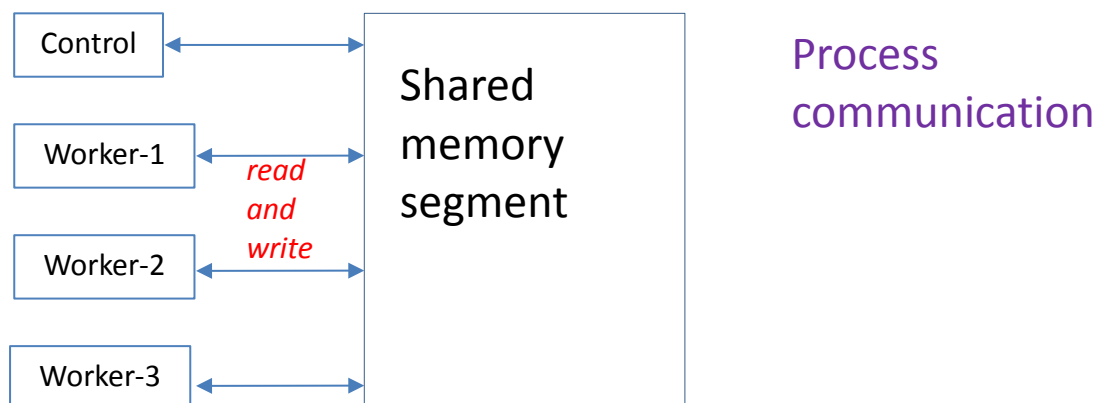
For this task, you simply copy my example code for process control and communication.

Sometime in the future, you may learn about systems such as Hadoop. Hadoop is a very important application for companies such as Yahoo (who paid for its development), Facebook, Amazon and others. It allows large data processing tasks to be split over numerous communicating processes that run on many different computers (at one time, Yahoo was running a Hadoop cluster with 10000 computers). Hadoop works with “mapper” processes that extract information from raw data files such as Apache access logs, and “reducer” processes that combine data from different mappers. Most Hadoop applications are multi-stage. The initial raw data are crudely processed through mappers and reducers to produce data files that then serve as input to a different set of mapper and reducer processes. The (rather complex) Hadoop software system allows you to define the processing in the mapper and reducer processes and then to configure an application engine involving many networked computers running communicating processes. *Such quasi parallel applications are becoming increasingly common.*

This example is more limited. It involves one control (“driver”) process and a number of subordinate processes and these all run on the same computer. The control process starts the other processes using fork-and-exec system calls.



Since they all run on the same computer, these processes communicate through a shared memory segment that is created (and eventually destroyed by the control process).



### The application

It's a demonstration of why you should use the best performing sort function that you can get!

The control process creates a large array of 131072 randomly chosen doubles. It copies these data into a shared memory segment that it creates. The shared memory segment also contains some space for the worker processes to report on the work that they perform.

Once it has created the shared memory segment, the control process forks off four children; each child will then attempt to sort the data. The control program then sits and waits for its children to die. The OS informs the control program of these process deaths. The control program picks up the work reports that each child filed in the shared memory segment prior to its death. These reports are then used to compose a report on comparative performance.

Each of the child processes execs a different program. Each different program works by making a copy of the data array from shared memory. Then it records start time, invokes its sort implementation, gets the end time, and reports the time difference. In this example, the micro-second clock is used to get accurate timings.

The four sort functions:

- Bubble sort
- Insertion sort
- Selection sort
- qsort from libc (the algorithm used for qsort varies with different libc implementations; this version of Linux apparently uses a modified version of quicksort)

### *No NetBeans*

NetBeans builds its executables and then buries them away in deep folder hierarchies (something like Project / dist / Debug / GNU-Linux-x86 / [prog](#)). That's awful inconvenient if you need to specify the pathname for an executable as a string in a program.

So for this task, you will work with a set of files all in a single directory. These files will be created with your favourite text editor. (After the application had been created, the files were imported into NetBeans. The NetBeans editor and Navigator were used when preparing the screen shots shown below.)

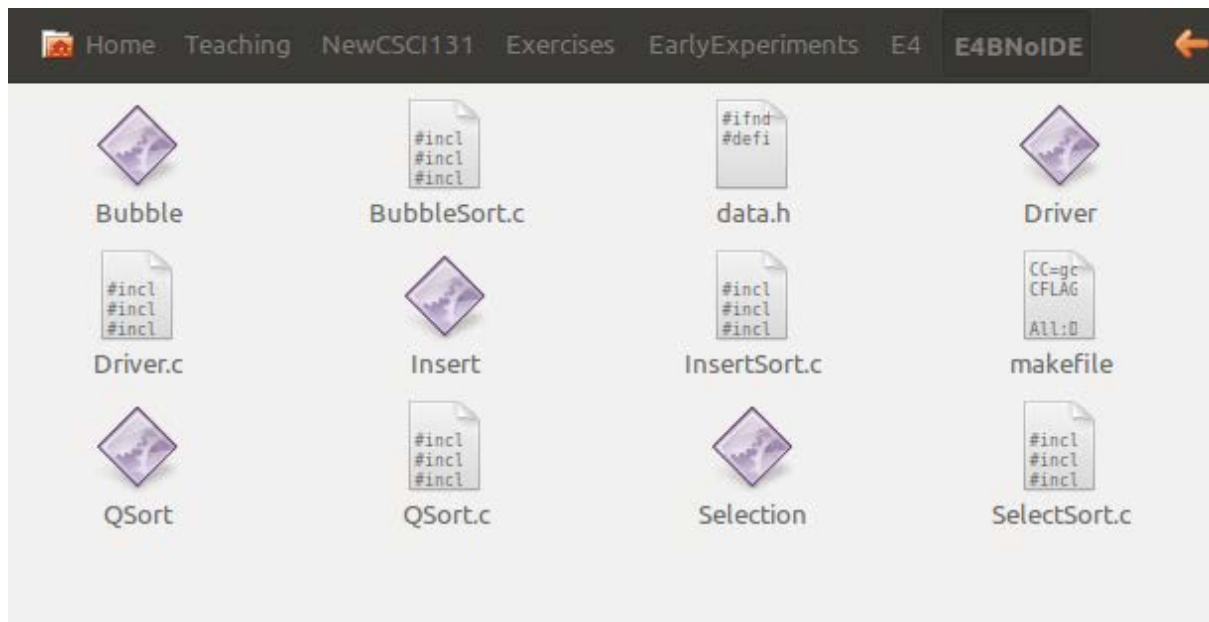
You will need to use `make` and a `makefile` to build the application with its five executables.

### *Your project*

The files that you need to create in an editor are:

- data.h
- Driver.c
- BubbleSort.c
- InsertSort.c
- SelectSort.c
- QSort.c
- makefile

You will use make to build the executable files based on the instructions in the makefile.



When the executables have been built, you run the application by starting the control program (Driver). This launches the other processes; invokes a process status report; waits for its child processes to terminate; and finally prints out a report on the times taken by the different sort algorithms.

You will have learnt something about timings and algorithm complexity in CSCI103 and will learn more in CSCI203. You know that Quicksort is  $O(n \lg n)$  whereas the others are  $O(n^2)$ . So you expect that Quicksort will be faster. Now, you can see it in practice. The Quicksort implementation completes in less than 0.1 second; insertion sort and selection sort take around 30s; and bubble sort – well it takes over a minute.

```
LINUX$ ./Driver
Master process 3089 running for neil
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
neil      3007  0.2  0.1  28304  5384 pts/1    Ss   10:40   0:00 bash
neil      3089  0.0  0.0  13756  1788 pts/1    S+   10:41   0:00 ./Driver
neil      3090  0.0  0.0   6208  2188 pts/1    R+   10:41   0:00 Bubble
neil      3091  0.0  0.0   6208  2192 pts/1    R+   10:41   0:00 Selection
neil      3092  0.0  0.0   6208  2192 pts/1    R+   10:41   0:00 Insert
neil      3093  0.0  0.0      0     0 pts/1    Z+   10:41   0:00 [QSo] <defunct>
neil      3094  0.0  0.0   4400   608 pts/1    S+   10:41   0:00 sh -c ps u
neil      3095  0.0  0.0  22356  1256 pts/1    R+   10:41   0:00 ps u
Qsort
Select
Insert
Bubble
All child processes have finished
Bubble 83314851 musec
Insert 31161370 musec
Select 29524271 musec
Qsort 77896 musec
LINUX$
```

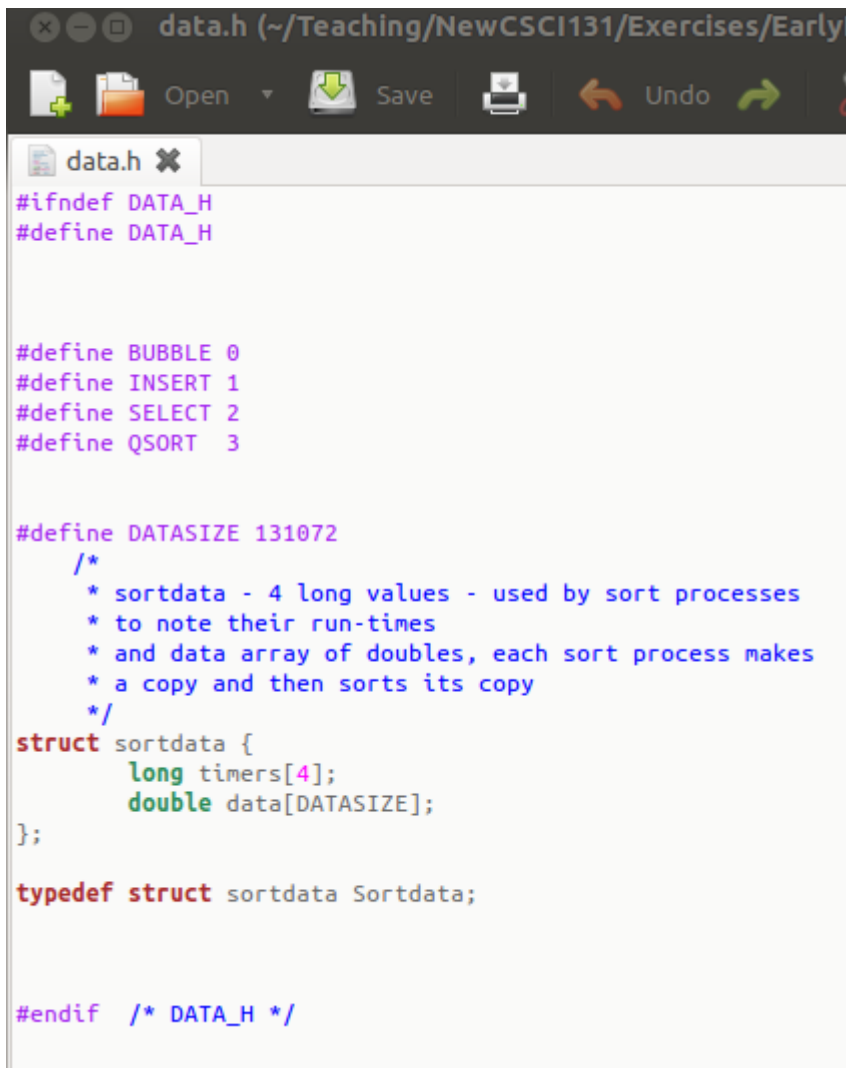
### Linux system calls used

This example illustrates many system calls. There are the process control services – fork, exec, wait. There are “shared memory services” – shmkey, shmget, shmat, shmdt, shmctl, along with ftok. There is also gettimeofday used for the micro-second timer; ‘system’ that is used to launch a subordinate shell process and run a shell command; getpid – identify current process, getuid – get user; and getpwuid – get identity of user.

### data.h

This file simply contains the definition of a struct that gets placed in shared memory, along with some #defines that assign identifier numbers for the different sort implementations.

This header file is imported by all the C application programs.

A screenshot of a code editor window titled 'data.h (~/Teaching/NewCSCI131/Exercises/EarlyE...'. The editor shows the following C code:

```
#ifndef DATA_H
#define DATA_H

#define BUBBLE 0
#define INSERT 1
#define SELECT 2
#define QSORT 3

#define DATASIZE 131072
/*
 * sortdata - 4 long values - used by sort processes
 * to note their run-times
 * and data array of doubles, each sort process makes
 * a copy and then sorts its copy
 */
struct sortdata {
    long timers[4];
    double data[DATASIZE];
};

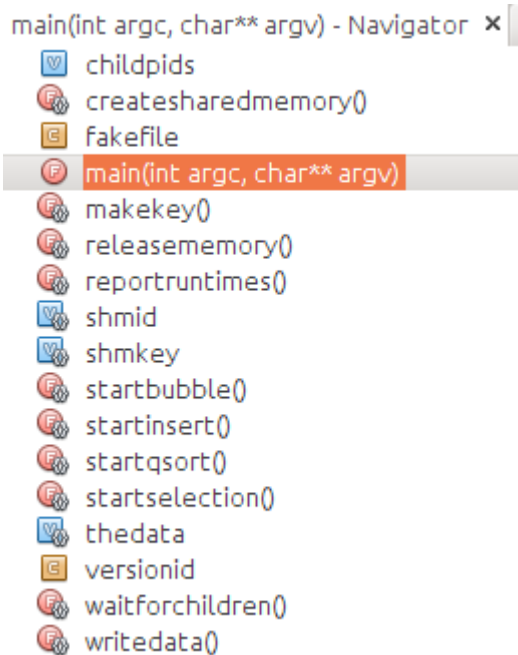
typedef struct sortdata Sortdata;

#endif /* DATA_H */
```

The timers will be filled in by the sort processes before they complete. The data array is filled with random data by the driver program; each sort process makes its own copy.



## Driver.c



The control program, “Driver”, must:

- Create a shared memory segment and “attach” it
- Fill in the random numbers
- Start the child processes
- Run the “ps” status report – identifying process numbers etc
- Wait for all children to finish
- Generate a report
- Release the shared memory

```
int main(int argc, char** argv) {
    pid_t processid = getpid();
    struct passwd *passwd = getpwuid(getuid());

    printf("Master process %d running for %s\n", processid, passwd->pw_name);
    makekey();
    createsharedmemory();
    writedata();
    startbubble();
    startselection(0);
    startinsert();
    startqsort();
    system("ps u");

    waitforchildren();
    reportruntimes();

    releasememory();
    return (EXIT_SUCCESS);
}
```

```

#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include "data.h"

/* Change name of this file - it must be a file that does
const char* fakefile = "/home/neil/tmp/sorters";
const int versionid = 1;

static key_t shmkey;
static int shmid;

static Sortdata *thedata;

int chldpids[4];

static void makekey() { ...7 lines }

static void createsharedmemory() { ...17 lines }

static void releasememory() { ...9 lines }

static void writedata() { ...7 lines }

static void startbubble() { ...15 lines }

static void startinsert() { ...15 lines }

static void startselection() { ...15 lines }

static void startqsort() { ...15 lines }

static void waitforchildren() { ...12 lines }

static void reportruntimes() { ...7 lines }

int main(int argc, char** argv) { ...20 lines }

```

```

makekey();
createsharedmemory();

```

### *Set up shared memory*

Creating a shared memory segment involves a number of rather exotic system calls. First an “identifier key” must be created – this requires reference to an actual file (the file isn’t really used for anything; it simply acts as part of the identifier). (You will have to create a file in your own file space and use the full pathname of your file.) Then the program has to ask the OS to create the shared data structure in memory – it gets back an integer identifier. Next, this has to be turned into an address before the program can read or write data. Shared memory will always be treated as just a big application defined struct; so the address that is returned is cast to a “pointer to struct”.

```

static void makekey() {
    shmkey = ftok(fakefile, versionid);
    if (shmkey == (key_t) - 1) {
        printf("ftok() for shm failed\n");
        exit(1);
    }
}

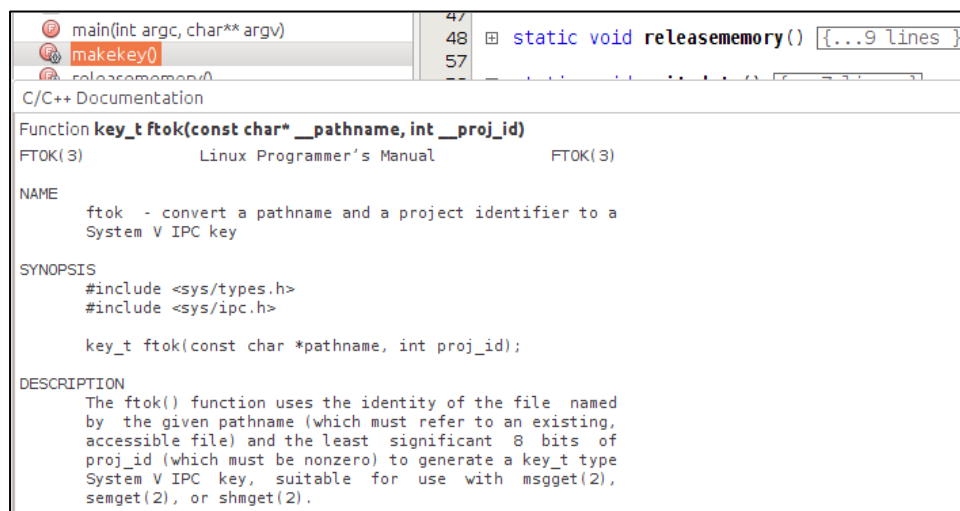
static void createsharedmemory() {
    int howbig = sizeof (Sortdata);
    void *where;
    shmid = shmget(shmkey, howbig, 0666 | IPC_CREAT | IPC_EXCL);
    if (shmid == -1) {
        perror("shmget() failed\n");
        exit(1);
    }

    where = shmat(shmid, NULL, 0);
    if (where == NULL) {
        perror("shmat() failed\n");
        exit(1);
    }

    thedata = (Sortdata*) where;
}

```

You should read the documentation on these system calls and try to develop an understanding of how the Linux OS is supporting inter-process communication:



The screenshot shows a code editor with a project structure on the left and the documentation for the `ftok()` function. The project structure includes `main(int argc, char** argv)`, `makekey()`, and `releasememory()`. The documentation for `ftok()` is displayed below the code editor.

**C/C++ Documentation**

Function **key\_t ftok(const char\* \_\_pathname, int \_\_proj\_id)**  
 FTOK(3)      Linux Programmer's Manual      FTOK(3)

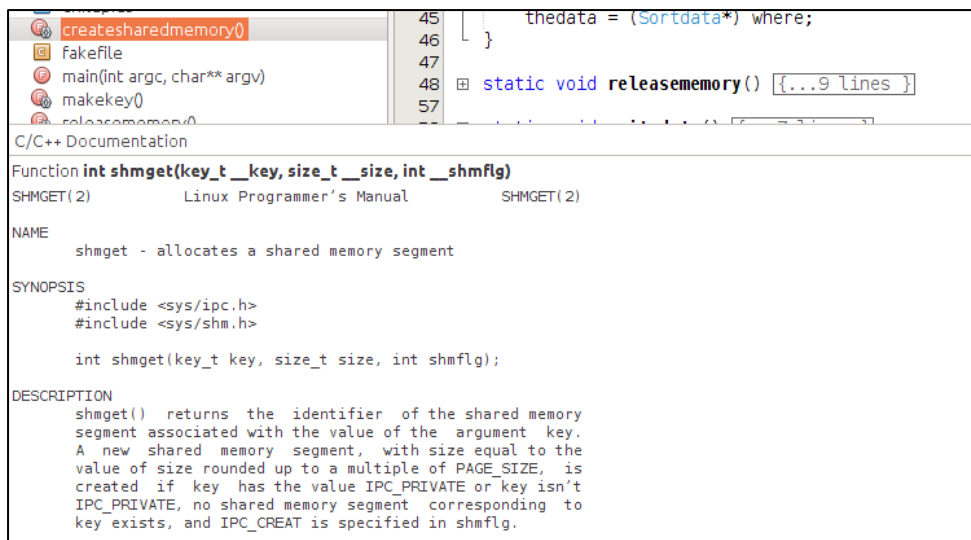
**NAME**  
 ftok - convert a pathname and a project identifier to a System V IPC key

**SYNOPSIS**  

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
```

**DESCRIPTION**  
 The `ftok()` function uses the identity of the file named by the given pathname (which must refer to an existing, accessible file) and the least significant 8 bits of `proj_id` (which must be nonzero) to generate a `key_t` type System V IPC key, suitable for use with `msgget(2)`, `semget(2)`, or `shmget(2)`.



C/C++ Documentation

Function **int shmget(key\_t \_\_key, size\_t \_\_size, int \_\_shmflg)**  
 SHMGET(2) Linux Programmer's Manual SHMGET(2)

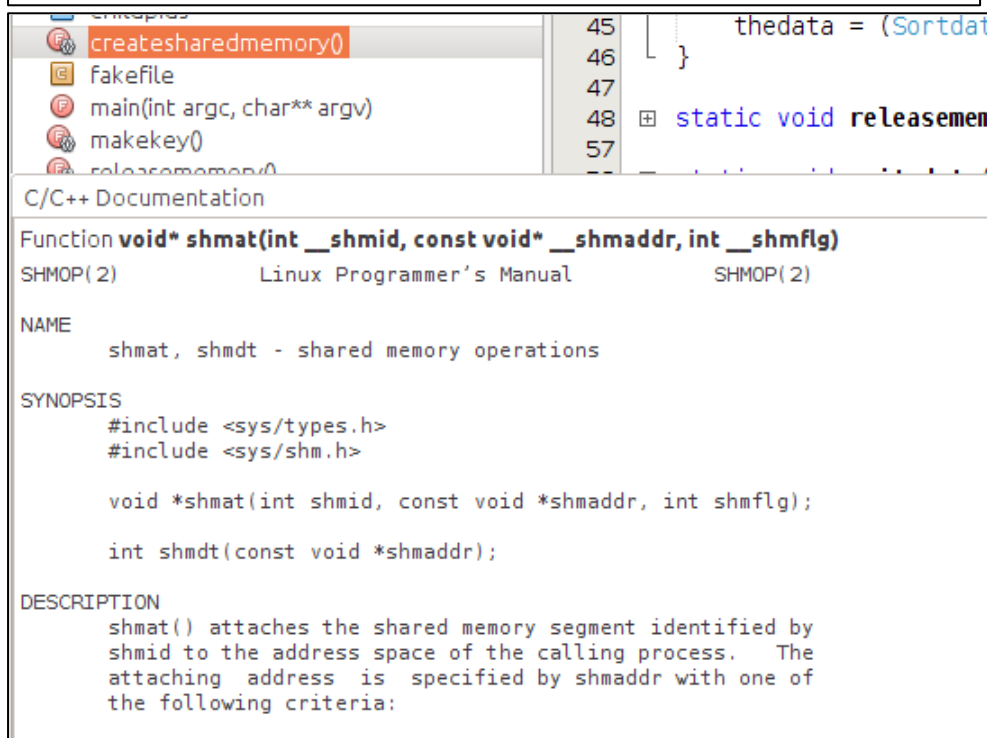
NAME  
 shmget - allocates a shared memory segment

SYNOPSIS  

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

DESCRIPTION  
 shmget() returns the identifier of the shared memory segment associated with the value of the argument key. A new shared memory segment, with size equal to the value of size rounded up to a multiple of PAGE\_SIZE, is created if key has the value IPC\_PRIVATE or key isn't IPC\_PRIVATE, no shared memory segment corresponding to key exists, and IPC\_CREAT is specified in shmflg.



C/C++ Documentation

Function **void\* shmat(int \_\_shmid, const void\* \_\_shmaddr, int \_\_shmflg)**  
 SHMOP(2) Linux Programmer's Manual SHMOP(2)

NAME  
 shmat, shmdt - shared memory operations

SYNOPSIS  

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

DESCRIPTION  
 shmat() attaches the shared memory segment identified by shmid to the address space of the calling process. The attaching address is specified by shmaddr with one of the following criteria:

Create data `writedata();`

This function simply fills in the struct:

```
static void writedata() {
    int i;
    for (i = 0; i < DATASIZE; i++)
        thedata->data[i] = (double) rand();
    for (i = 0; i < 4; i++)
        thedata->timers[i] = 0;
}
```

```

        startbubble();
        startselection(0);
        startinsert();
        startqsort();

```

**Start child processes**

The code for these functions is almost the same, differing only in the path names for the executables that are to be run, and the (#defined) sort identifier used when recording the child's process identifier. (The zero argument to startselection is a typographic error!)

Only startbubble() is shown here:

```

static void startbubble() {
    pid_t childpid = fork();
    /* Fork to create child process */
    if (childpid == -1) {
        /* Fork request failed? Too many processes is most likely reason */
        perror("fork");
        exit(1);
    }

    if (childpid == 0) {
        execl("./Bubble", "Bubble", (char*) 0);
    }

    childpids[BUBBLE] = childpid;
}

```

You must ensure that you understand how the fork and exec system calls work.

**fork** – OS creates a duplicate of the current process. Would you want two copies of Driver? The return value from fork identifies their roles. A return value of 0 means that this is the new child process that is to run some other code. The parent process resumes after the fork call gets the process id of the newly created child.

**execl** – The OS discards the code segment, static data segment, stack segment and heap of the current process. It then loads code from the executable whose pathname is passed as the first argument. (The second argument is effectively a process name; there can be many other arguments that act as command line arguments; a final null string is needed after any command line arguments.)

Here the execl command specifies the code for the Bubble sort application; this application has no command line arguments.

```

        waitforchildren();
        reportruntimes();

```

**Finishing up** releasememory();

The control process must wait for its children to terminate. (One way of crashing a Linux system is to keep creating child processes that terminate while failing to 'wait' for their termination signals.) When all the children have finished, the control process can pick up their sort times from the shared data structure. (The OS uses signals to report on the death of processes. The "wait()" call hides messy signal processing details.)

```

static void waitforchildren() {
    int countfinished = 0;
    do {
        int status;
        pid_t childfinished = wait(&status);
        countfinished++;
        if(childfinished==childpids[BUBBLE]) printf("Bubble\n");
        if(childfinished==childpids[INSERT]) printf("Insert\n");
        if(childfinished==childpids[SELECT]) printf("Select\n");
        if(childfinished==childpids[QSORT]) printf("Qsort\n");
    } while (countfinished < 4);
}

static void reportruntimes() {
    printf("All child processes have finished\n");
    printf("Bubble %ld msec\n", thedata->timers[BUBBLE]);
    printf("Insert %ld msec\n", thedata->timers[INSERT]);
    printf("Select %ld msec\n", thedata->timers[SELECT]);
    printf("Qsort %ld msec\n", thedata->timers[QSORT]);
}

```

The control process must detach the shared memory segment, and then further tell the OS that this segment is no longer required:

```

static void releasememory() {
    void *where = (void*) thedata;
    int res = shmdt(where);
    if (res != 0) {
        perror("Error detaching shared memory\n");
        exit(1);
    }
    shmctl(shmid, IPC_RMID, 0);
}

```

## Sort processes – BubbleSort, InsertSort, SelectSort, QSort

These have a common structure, illustrated here for BubbleSort.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <sys/time.h>
#include "data.h"

/* Change name of this file - it must be a file that does exist in your file space */
/* Must be same as name used in Driver.c */
const char* fakefile = "/home/neil/tmp/sorters";
const int versionid = 1;

static key_t shmkey;
static int shmid;

static Sortdata *thedata;

static double datacopy[DATASIZE];

static void makekey() { ...7 lines }

static void getdata() { ...18 lines }

static void copydata() { ...5 lines }

static void releasesharedmemory() { ...8 lines }

static void bubblesort() { ...14 lines }

static void timedsort() { ...12 lines }

int main(int argc, char** argv) {
    getdata();
    copydata();

    timedsort();

    releasesharedmemory();
    return (EXIT_SUCCESS);
}
```

The sort processes must first map in the shared memory segment. They use the same code to generate the identifier key, and then ask the OS to attach to an existing segment with that key (the Driver program had to create the segment).

```

static void makekey() {
    shmkey = ftok(fakefile, versionid);
    if (shmkey == (key_t) - 1) {
        printf("ftok() for shm failed\n");
        exit(1);
    }
}

static void getdata() {
    makekey();
    int howbig = sizeof (Sortdata);
    void *where;
    shmkey = shmget(shmkey, howbig, 0666);
    if (shmkey == -1) {
        perror("shmget() failed\n");
        exit(1);
    }

    where = shmat(shmkey, NULL, 0);
    if (where == NULL) {
        perror("shmat() failed\n");
        exit(1);
    }

    thedata = (Sortdata*) where;
}

```

The values that are to be sorted must be copied to a local array.

```

static void copydata() {
    int i;
    for (i = 0; i < DATASIZE; i++)
        datacopy[i] = thedata->data[i];
}

```

Calls to the OS's microsecond timer are made before and after the invocation of the sort routine. The time used by the sort routine is determined by the difference between the values. The elapsed time is copied into the appropriate slot in the shared data structure.

```

static void timedsort() {
    struct timeval starttime;
    gettimeofday(&starttime, NULL);
    bubblesort();

    struct timeval endtime;
    gettimeofday(&endtime, NULL);
    long msec_diff = (endtime.tv_sec - starttime.tv_sec)*1000000 +
        (endtime.tv_usec - starttime.tv_usec);

    thedata->timers[BUBBLE] = msec_diff;
}

```

The bubble sort implementation should be familiar:



```

static void bubblesort() {
    int swaps;
    do {
        swaps = 0;
        int i;
        for (i = 1; i < DATASIZE; i++)
            if (datacopy[i - 1] > datacopy[i]) {
                double temp = datacopy[i];
                datacopy[i] = datacopy[i - 1];
                datacopy[i - 1] = temp;
                swaps++;
            }
    } while (swaps > 0);
}

```

The **QSort.c** program invokes qsort, passing the address of the function used to compare elements:

```

static int cmpfunc(const void * a, const void * b) {
    double diff = (*(double*) a - *(double*) b);
    int idiff = (int) diff;
    return idiff;
}

static void timedsort() {
    struct timeval starttime;
    gettimeofday(&starttime, NULL);
    qsort(datacopy, DATASIZE, sizeof (double), cmpfunc);

    struct timeval endtime;
    gettimeofday(&endtime, NULL);
    long musec_diff = (endtime.tv_sec - starttime.tv_sec)*1000000 +
        (endtime.tv_usec - starttime.tv_usec);

    thedata->timers[QSORT] = musec_diff;
}

```

The versions for InsertSort.c and SelectSort.c are left as exercises; they are of course similar to the bubblesort implementation.

Before a sort process terminates, it should detach from the shared memory segment. (Deletion of the shared memory resource is only performed in the driver program).

```

static void releasesharedmemory() {
    void *where = (void*) thedata;
    int res = shmdt(where);
    if (res != 0) {
        perror("Error detaching shared memory\n");
        exit(1);
    }
}

```

## makefile

The application (all 5 executables) is built using **make** and a makefile. The next group of tasks have more illustrations of make as they have more complex needs such as accessing non-standard libraries.

This makefile is fairly simple. But **make** is a very old program; one that is very fussy about the format of its input file. Consequently, care must be taken when composing makefiles.

```
CC=gcc
CFLAGS=-Wall -g
All:Driver Bubble Insert Selection QSort

clean:
rm -f Driver Bubble Insert Selection QSort

Driver: Driver.c data.h
$(CC) $(CFLAGS) Driver.c -o Driver

Bubble: BubbleSort.c data.h
$(CC) $(CFLAGS) BubbleSort.c -o Bubble

Insert: InsertSort.c data.h
$(CC) $(CFLAGS) InsertSort.c -o Insert

Selection: SelectSort.c data.h
$(CC) $(CFLAGS) SelectSort.c -o Selection

QSort: QSort.c data.h
$(CC) $(CFLAGS) QSort.c -o QSort
```

**Must use tab - spaces  
will confuse 'make'**

You can try copying this makefile source code, but make sure that your editor does not turn tabs into spaces and remember to leave blank lines after each rule.

```
CC=gcc
CFLAGS=-Wall -g
```

```
All:Driver Bubble Insert Selection QSort
```

```
clean:
    rm -f Driver Bubble Insert Selection QSort
```

```
Driver: Driver.c data.h
    $(CC) $(CFLAGS) Driver.c -o Driver
```

```
Bubble: BubbleSort.c data.h
    $(CC) $(CFLAGS) BubbleSort.c -o Bubble
```

```
Insert: InsertSort.c data.h
```

**`$(CC) $(CFLAGS) InsertSort.c -o Insert`**

Selection: SelectSort.c data.h

**`$(CC) $(CFLAGS) SelectSort.c -o Selection`**

QSort: QSort.c data.h

**`$(CC) $(CFLAGS) QSort.c -o QSort`**

Some explanations:

- **`$(CC)`** – value of this variable is name of compiler that is to be used; and that name is set in the directive at the top **`CC=gcc`**; these applications are to be compiled with the gcc C compiler.
- **`$(CFLAGS)`** – value of this variable sets compilation options. Here they are `-Wall` and `-g`. The `-Wall` option tells the compiler to print all warnings about possibly dubious code. The `-g` option keeps symbol table data in the executables – these data would be needed if you had to run with gdb.
- **All**: This is a target definition. As it is the first target, it is the default. If you run `make` on this makefile without specifying a target, then `make` picks All. How is it to build “All”? The dependency data after the colon specify what it has to do. `Make` must build the targets: Driver, Bubble, Insert, Selection, and QSort. There are no other rules given for the “All” target; `make` must work out exactly what it has to do.
- `clean`: If you invoke “`make clean`”, `make` will execute the rule “`rm -rf Driver Bubble Insert Selection QSort`”; i.e. it will delete all the executables.
- `Driver`: This target first says that the Driver executable must be rebuilt if either it doesn’t exist, or if the existing copy is older than either Driver.c or data.h. The build rule on the next line says how to build the Driver target. `Make` must compile Driver.c with gcc, it must then link the generated code with `libc` (this part of the rule is implicit), and rename the resulting executable as Driver.
- `Bubble`: `Insert`: `Selection`: `QSort`: These target definitions are similar to that for Driver.

## Build the application

### Run the application

(Timings based on totally random data are those that are typically used to compare sort algorithms. But in practice, it is more common for applications to need to sort partially ordered data – it’s common for data to have been sorted, then for a few records to get added, updated, or deleted so giving a data set that is largely ordered with just a few entries out of place. Partial ordering can have a significant impact on sort performance. Try running the application again with data that are partially ordered, or reverse ordered, and see how this impacts on sort performance.)

### *“revision”*

Make sure that you understand the fork and exec system calls of Linux; you will be making heavy use of these system calls in 200-level subjects (especially CSCI212).

You must also get familiar with the different mechanisms that Linux provides for inter-process communication:

- Signals
- Pipes and FIFOs
- Shared memory (and shared semaphores)
- Ports and sockets

---

## Exercises complete

Show the tutor that you have completed all the parts.

(That was 7 very easy marks wasn't it – just copy my code. Hopefully, you learnt a little about the services provided by the Unix/Linux OS.)

---

## Assignment

The [assignment](#): simulation of some aspect of the work of an OS.