- Varun Dev (2017115608)
- Rogith Y (2017115580)
- Antony Gilon (2017115510)

System Calls – UNIX

stat() - fstat()

You can access the information/status of files with the stat() and fstat() system calls. stat() and fstat()

The format for the i-node struct returned by these system calls is defined in /usr/include/sys/stat.h.

stat.h uses types built with the C language typedef construct and defined in the file /usr/include/sys/types.h, so it too must be included and must be included before the inclusion of the stat.h file.

The prototypes for stat() and fstat() are:

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

#include <sys/stat.h>


int stat(file_name, stat_buf)

char *file_name;

struct stat *stat_buf;


int fstat(file_descriptor, stat_buf)

int file_descriptor;

struct stat *stat_buf;
```

where file_name names the file as an ASCII string and file_descriptor names

the I/O channel and therefore the file. Both calls returns the file's

specifics in stat_buf. stat() and fstat() fail if any of the following conditions hold:

a path name component is not a directory (stat() only).

file_name does not exit (stat() only).

a path name component is off-limits (stat() only).

file_descriptor does not identify an open I/O channel (fstat() only).

stat_buf points to an invalid address.

Following is an extract of the stat.h file from the University's HP-9000. It shows the definition of the stat structure and some manifest constants used to access the st_mode field of the structure.

```
     /* stat.h  */


     struct   stat

     {

        dev_t          st_dev;       /* The device number containing
the i-node */

        ino_t          st_ino;       /* The i-number */

        unsigned short st_mode;      /* The 16 bit mode */

        short          st_nlink;     /* The link count; 0 for pipes
*/

        ushort         st_uid;       /* The owner user-ID */

        ushort         st_gid;       /* The group-ID   */

        dev_t          st_rdev;      /* For a special file, the
device number */

        off_t          st_size;    /* The size of the file; 0 for
special files */

        time_t         st_atime;     /* The access time.  */

        int            st_spare1;

        time_t         st_mtime;     /* The modification time.   */

        int            st_spare2;
        time_t          st_ctime;     /* The status-change time.  */

        int             st_spare3;

        long            st_blksize;

        long            st_blocks;

        uint            st_remote:1;          /* Set if file is remote
*/

        dev_t           st_netdev;            /* ID of device
containing */

           /* network special file */

        ino_t           st_netino;     /* Inode number of network
special file */

        long            st_spare4[9];

     };


     #define  S_IFMT   0170000     /* type of file */

     #define     S_IFDIR  0040000  /* directory */

     #define     S_IFCHR  0020000  /* character special */

     #define     S_IFBLK  0060000  /* block special */

     #define     S_IFREG  0100000  /* regular (ordinary) */

     #define     S_IFIFO  0010000  /* fifo */

     #define     S_IFNWK 0110000   /* network special */

     #define     S_IFLNK  0120000      /* symbolic link */

     #define     S_IFSOCK 0140000      /* socket */

     #define  S_ISUID  0004000     /* set user id on execution */

     #define  S_ISGID  0002000     /* set group id on execution */

     #define  S_ENFMT  0002000   /* enforced file locking (shared
with S_ISGID)*/

     #define  S_ISVTX  0001000     /* save swapped text even after
use */
```

Following is an example program demonstrating the use of the stat() system call to determine the status of a file:

```
/*  status.c  */
/*  demonstrates the use of the stat() system call to determine the
    status of a file.
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#define ERR   (-1)
#define TRUE  1
#define FALSE 0
int main();
int main(argc, argv)
int argc;
char *argv[];
{
   int isdevice = FALSE;
   struct stat stat_buf;
   if (argc != 2)
      {
      printf("Usage:  %s filename\n", argv[0]);
      exit (1);
      }
   if ( stat( argv[1], &stat_buf) == ERR)
      {
      perror("stat");
      exit (1);
      }
   printf("\nFile:  %s  status:\n\n",argv[1]);
```

```
        if ((stat_buf.st_mode & S_IFMT) == S_IFDIR)
           printf("Directory\n");
        else if ((stat_buf.st_mode & S_IFMT) == S_IFBLK)
           {
           printf("Block special file\n");
           isdevice = TRUE;
           }
        else if ((stat_buf.st_mode & S_IFMT) == S_IFCHR)
           {
           printf("Character special file\n");
           isdevice = TRUE;
           }
        else if ((stat_buf.st_mode & S_IFMT) == S_IFREG)
           printf("Ordinary file\n");
        else if ((stat_buf.st_mode & S_IFMT) == S_IFIFO)
           printf("FIFO\n");
                 if (isdevice)
           printf("Device number:%d, %d\n", (stat_buf.st_rdev > 8) &
0377,
                  stat_buf.st_rdev & 0377);
        printf("Resides on device:%d, %d\n", (stat_buf.st_dev > 8) &
0377,stat_buf.st_dev & 0377);
        printf("I-node: %d; Links: %d; Size: %ld\n",stat_buf.st_ino,
               stat_buf.st_nlink, stat_buf.st_size);
        if ((stat_buf.st_mode & S_ISUID) == S_ISUID)
           printf("Set-user-ID\n");
        if ((stat_buf.st_mode & S_ISGID) == S_ISGID)
           printf("Set-group-ID\n");
        if ((stat_buf.st_mode & S_ISVTX) == S_ISVTX)
```

```
           printf("Sticky-bit set -- save swapped text after
use\n");

        printf("Permissions: %o\n", stat_buf.st_mode & 0777);



        exit (0);

     }
```

signal()

The UNIX system provides a facility for sending and receiving software interrupts, also called SIGNALS. Signals are sent to a process when a predefined condition happens. The number of signals available is system dependent.

Programs can respond to signals three different ways. These are:

1. Ignore the signal.

2. A signal can be set to its default state, which means that the   process will be ended when it receives that signal.

3. Catch the signal.

You define how you want to respond to a signal with the signal() system call.
The prototype is:

```
     #include <sys/signal.h>

     int (* signal ( signal_name, function ))
     int signal_name;
     int (* function)();
```

where signal_name is the name of the signal from signal.h and function is any of SIG_IGN, meaning that you wish to ignore the signal when it occurs;
SIG_DFL, meaning that you wish the UNIX system to take the default action when your program receives the signal; or a pointer to a function that returns an integer. The function is given control when your program receives the signal, and the signal number is passed as an argument. signal() returns the previous value of function, and signal() fails if any of the following conditions hold:

signal_name is an illegal name or SIGKILL.

function points to an invalid memory address.

Once a signal is caught, the UNIX system resets it to its initial state (the default condition). In general, if you intend for your program to be able to catch a signal repeatedly, you need to re-arm the signal handling mechanism.

You must do this as soon after receipt of the signal as possible, namely just after entering the signal handling routine.

You should use signals in your programs to isolate critical sections from interruption.

The state of all signals is preserved across a fork() system call, but all caught signals are set to SIG_DFL across an exec system call.

kill()

The UNIX system sends a signal to a process when something happens, such as typing the interrupt key on a terminal, or attempting to execute an illegal instruction. Signals are also sent to a process with the kill() system call.

Its prototype is:

```
int kill (process_id, signal_name )
int process_it, signal_name;
```

where process_id is the ID of the process to be signaled and signal_name is the signal to be sent to that process. If process_id has a positive value,that value is assumed to be the process ID of the process to whom signal_name signal is to be sent. If process_id has the value 0, then signal_name signal is sent to all processes in the sending process' process group, that is all processes that have been started from the same terminal. If process_id has the value -1 and the process executing the kill() system call is the superuser, then signal_name is sent to all processes excluding process 0 and process 1 that have the same user ID as the process executing the kill().kill() fails if any of the following conditions hold:

signal_name is not a valid signal.

there is not a process in the system with process ID process_id.

even though the process named by process_id is in the system, you cannot send it a signal because your effective user ID does not match either the real or effective user ID of process_id.

alarm()

Every process has an alarm clock stored in its system-data segment. When the alarm goes off, signal SIGALRM is sent to the calling process. A child inherits its parent's alarm clock value, but the actual clock isn't shared.

The alarm clock remains set across an exec.

The prototype for alarm() is:

```
unsigned int alarm(seconds)
unsigned int seconds;
```

where seconds defines the time after which the UNIX system sends the SIGALRM signal to the calling process. Each successive call to alarm() nullifies the previous call, and alarm() returns the number of seconds until that alarm would have gone off. If seconds has the value 0, the alarm is canceled.

alarm() has no error conditions.

The following is an example program that demonstrates the use of the signal() and alarm() system calls:

```
     /*  timesup.c  */

     #include <stdio.h>
     #include <sys/signal.h>

     #define  EVER  ;;

     void main();
     int times_up();

     void main()
     {
        signal (SIGALRM, times_up);         /* go to the times_up
function  */
                                            /* when the alarm goes
off.     */
        alarm (10);                         /* set the alarm for 10
seconds */

        for (EVER)                          /* endless loop.
*/
           ;                                /* hope the alarm works.
*/
     }

     int times_up(sig)
     int sig;                               /* value of signal
*/
     {
        printf("Caught signal #< %d >n", sig);
        printf("Time's up!  I'm outta here!!\n");
        exit(sig);                          /* return the signal
number     */
     }
```