System Calls

Library Functions Vs. System Calls

A *library function:*

- Ordinary function that resides in a library external to the calling program.
- A call to a library function is just like any other function call.
- The arguments are placed in processor registers or onto the stack, and execution is transferred to the start of the function's code, which typically resides in a loaded shared library.
- A library function may have a one-to-one correspondence with a system-call (read)

  Or
- The function may format and present data to a system call (printf)

  Or
- The function may not require a system call at all.

Library Functions Vs. System Calls

A *system call:*

- Is implemented in the Linux kernel and performs basic functions that require kernel level privileges
- All activities related to
  - File management
  - User management
  - Process and Memory management

  are handled by the kernel using these system calls.
- Low-level I/O functions such as open and read are examples of system calls on Linux.
- A program running in user mode switches to kernel mode when a system call is made.

System Calls

- When a program makes a system call, the arguments are packaged up and handed to the kernel, the kernel then, takes over execution of the program until the call completes.
- A system call isn't an ordinary function call, and a special procedure is required to transfer control to the kernel.
- The GNU C library (the implementation of the standard C library provided with GNU/Linux systems) wraps Linux system calls with functions so that you can call them easily.
- The set of Linux system calls forms the most basic interface between programs and the Linux kernel. Each call presents a basic operation or capability.
- Note that a library function may invoke one or more other library functions or system calls as part of its implementation.

System Calls

- When an error occurs in a system call:
  - The return value is generally set to -1
  - A global variable errno is set to a positive integer
- This integer
  - Is associated with an error message
  - Is represented by a symbolic constant
- Library functions (like perror) can be used in order to get the information associated with an error.

File Accessibility and Directories

- access
- stat
- chmod
- chown
- chdir
- chroot
- And more…

access

- Determines check user's permissions for a file .
- Can check
  - any combination of read, write, and execute permission,
  - file's existence.
- Syntax:

```
int access(const char *pathname, int mode);
```

- Arguments
  1. Path to the file to check
  2. Bitwise or of R_OK, W_OK, and X_OK, corresponding to read, write, and execute permission

access

- return value
  - 0 if the process has all the specified permissions.
  - -1 if the file exists but the calling process does not have the specified permissions. The global variable, errno is set to EACCES (or EROFS, if write permission was requested for a file on a read-only file system).
  - If the second argument is F_OK, access simply checks for the file's existence.
    - ✓ If the file exists, the return value is 0; if not, the return value is –1 and errno is set to ENOENT
    - ✓ errno may instead be set to EACCES if a directory in the file path is inaccessible

stat

- Provides information regarding a file contained on its i-node, such as
  - The file size,
  - The last modification time,
  - Permissions,
  - The owner.
- **Syntax:**

  int stat (const char * filename, struct stat *buf);
- Parameters:
  - The path to the file and
  - A pointer to a variable of type struct stat. Set the second parameter to a valid stat structure containing the following information:

```
struct stat {
      dev_t     st_dev;     /* device ID, for device files*/
      ino_t     st_ino;     /* inode number*/
      mode_t    st_mode;    /* type & rights */
      nlink_t   st_nlink;   /* number of hard links */
      uid_t     st_uid;     /* user ID of owner */
      gid_t     st_gid;     /* group ID of owner */
      off_t     st_size;    /* total size, in bytes */
      blksize_t st_blksize; /* blocksize for filesystem I/O */
      blkcnt_t  st_blocks;  /* number of 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 change in inode */
    };
```


stat

Return:

- 0: If the call to stat is successful and the fields of the structure are filled with information about that file
- -1 on failure

Few useful fields in struct stat:

- `st_mode` contains the file's access permissions and encodes the type of the file in higher-order bits.
- `st_uid` and `st_gid` contain the IDs of the user and group, respectively, to which the file belongs.
- `st_size` contains the file size, in bytes.
- `st_atime` contains the time when this file was last accessed (read or written).
- `st_mtime` contains the time when this file was last modified

st_mode Macros

- These macros check the value of the st_mode field value to figure out what kind of file stat was invoked on.
- A macro evaluates to true if the file is of that type.
  - S_ISBLK (mode) block device
  - S_ISCHR (mode) character device
  - **S_ISDIR (mode) directory**
  - S_ISFIFO (mode) fifo (named pipe)
  - S_ISLNK (mode) symbolic link
  - **S_ISREG (mode) regular file**
  - S_ISSOCK (mode) socket

Flags in *st_mode* for checking Permissions

S_IRWXU 00700 mask for file owner permissions
S_IRUSR 00400 owner has read permission
S_IWUSR 00200 owner has write permission
S_IXUSR 00100 owner has execute permission

S_IRWXG 00070 mask for group permissions
S_IRGRP 00040 group has read permission
S_IWGRP 00020 group has write permission
S_IXGRP 00010 group has execute permission

S_IRWXO 00007 mask for permissions for others (not in group)
S_IROTH 00004 others have read permission
S_IWOTH 00002 others have write permission
S_IXOTH 00001 others have execute permission

Symbolic Links: lstat

- If you call stat on a symbolic link, stat follows the link and you can obtain the information about the file that the link points to, not about the symbolic link itself.
- This implies that S_ISLNK will never be true for the result of stat. Use the **lstat** function if you don't want to follow symbolic links; this function obtains information about the link itself rather than the link's target.
- If you call lstat on a file that isn't a symbolic link, it is equivalent to stat. Calling stat on a broken link (a link that points to a nonexistent or inaccessible target) results in an error, while calling lstat on such a link does not.
- If you already have a file open for reading or writing, call **fstat** instead of stat. This takes a file descriptor as its first argument instead of a path.

chmod

- Purpose: change mode of file
- The file permission bits of the file named specified by *path* or referenced by the file descriptor *fd* are changed to *mode*.
- The **chmod**() system call verifies that the process owner (user) either owns the file specified by *path* (or *fd*), or is the super-user.
- The **chmod**() system call follows symbolic links to operate on the target of the link rather than the link itself.

chmod

- **Header: sys/types.h , sys/stat.h**
- Syntax

  int **chmod**(*const char *path, mode_t mode*);

- Parameters
  - Path: of the file whose *mode* is to be changed
  - Mode: created from *or'd* permission bit masks defined in *<sys/stat.h>*
- Return Value
  - 0: successful completion; -1: unsuccessful; the global variable *errno* is set to indicate the error.

Bit masks for chmod

```
#define S_IRWXU 0000700 /* RWX mask for owner */
#define S_IRUSR 0000400 /* R for owner */
#define S_IWUSR 0000200 /* W for owner */
#define S_IXUSR 0000100 /* X for owner */

#define S_IRWXG 0000070 /* RWX mask for group */
#define S_IRGRP 0000040 /* R for group */
#define S_IWGRP 0000020 /* W for group */
#define S_IXGRP 0000010 /* X for group */

#define S_IRWXO 0000007 /* RWX mask for other */
#define S_IROTH 0000004 /* R for other */
#define S_IWOTH 0000002 /* W for other */
#define S_IXOTH 0000001 /* X for other */

#define S_ISUID 0004000 /* set user id on execution */
#define S_ISGID 0002000 /* set group id on execution */
#define S_ISVTX 0001000 /* save swapped text even after use */
```


chown

- Purpose: change owner and group of a file
- The chown() system call changes the user and group ownership of a file.
- Only a user having appropriate privileges can change the ownership of a file.
- To get UID corresponding to a user name, use

```
struct password getpwnam (const char *name)
```

chown

- Header: unistd.h
- Syntax:

```
int chown(const char *path, uid_t owner, gid_t
                    group);
```

- Parameters:
  - *path* points to the path name of a file
  - owner ID and group ID of new owner & group
- Return Value:
  - 0: Successful completion.
  - -1: Failure. The owner and group of the file remain unchanged. errno is set to indicate the error.

chdir

- Purpose: change working directory
- Header: **unistd.h**
- Syntax: **`int chdir(const char *`*`path`*`);`**
- Parameters: the new directory path
- Return Value:
  - 0: On success
  - -1: on failure, *errno* is set appropriately.

chroot

- changes the apparent disk root directory for the current running process and its children
- This directory will be used for pathnames beginning with /.
- The root directory is inherited by all children of the current process.
- A program that is re-rooted to another directory cannot access or name files outside that directory

chroot

- Header: unistd.h
- Syntax: **`int chroot(const char *`***`path`***`);`**
- Parameters: path of the new root directory
- Return Value:
  - 0: Successful completion.
  - -1: Failure. The owner and group of the file remain unchanged. errno is set to indicate the error.

Additional

- Try working with these additional File System related System Calls corresponding to simple operations
  - **chdir:** Change to a directory
  - **mkdir**: Creates a directory
  - **rmdir**: Removes an empty directory
  - **rename**: Modifies name in a directory entry

Process Control

- running Linux Commands from C
- fork()
- the exec family
- wait()
- exit()

running Linux Commands from C

- Using the *system* function in the standard C library
  - an easy way to execute a shell command from within a program, much as if the command had been typed into a shell.
  - creates a sub-process running the default shell and hands the command to that shell for execution
  - Dependent upon the system shell /bin/sh
  - preferable to use the fork and exec method for creating processes

system

Header: stdlib.h

Syntax: `int system (const char *command);`

Parameters:

The *command* is executed by issuing **/bin/sh -c** *command*,

Return

- returns the exit status of the shell command
- If the shell itself cannot be run, system returns 127; if another error occurs, system returns –1.

Using *fork* and *exec*

- Linux provides,
  - one function fork, that makes a child process that is an exact copy of its parent process.
  - another set of functions, the exec family,
    - ✓causes a particular process to cease being an instance of one program and to instead become an instance of another program.
- To spawn a new process,
  - One first uses fork to make a copy of the current process.
  - Then the user uses exec to transform one of these processes into an instance of the program you want to spawn.

fork()

- When a program calls fork, a duplicate process, called the *child process*, is created.
- The parent process continues executing the program from the point that fork was called.
- The child process, too, executes the same program from the same place.
- Differentiation between parent and child can be done on basis of:
  - PID of the current process
  - Return value of fork.

Child / Parent

- The child process is a new process and therefore has a new process ID, distinct from its parent's process ID.
  - One way for a program to distinguish whether it's in the parent process or the child process is to call getpid.
- However, the fork function provides different return values to the parent and child processes
  - one process "goes in" to the fork call, and two processes "come out," with different return values.
  - The return value in the parent process is the process ID of the child.
  - The return value in the child process is zero.
  - This can be used by the program to tell whether it is now running as the parent or the child process.

Example

```
int main()
{
   int pid;
   printf("Before fork...\n");
   pid= fork();
   printf("Process Created\n");
}
```

The value returned by *fork*

The value returned by fork can be used in order to check for the parent or the child process:

-1 in case of Error

```
int main()
{
   int pid;
   printf("Before fork:
    \n");
   pid= fork();
   if (pid==0)
    printf("Child \n");
   else
    printf("Parent \n");
}
```


the exec family

- The exec functions replace the program running in a process with another program.
- When a program calls an exec function,
  - that process immediately ceases executing that program and
  - begins executing a new program from the beginning,
  - assuming that the exec call doesn't encounter an error.
- Within the exec family, there are functions that vary slightly in their capabilities and how they are called.

the exec family

- Functions that contain the letter *p* in their names (execvp and execlp)
  - accept a program name and
  - search for a program by that name in the current execution path;
  - functions that don't contain the *p* must be given the full path of the program to be executed.
- Functions that contain the letter v in their names (execv, execvp, and execve)
  - accept the argument list for the new program as a NULL-terminated array of pointers to strings.

The exec Family

- Functions that contain the letter *l* (execl, execlp, and execle)
  - accept the argument list using the C language's varargs mechanism.
- Functions that contain the letter *e* in their names (execve and execle)
  - accept an additional argument, an array of environment variables.
  - The argument should be a NULL-terminated array of pointers to character strings.
  - Each character string should be of the form "VARIABLE=value".
- Because exec replaces the calling program with another one, it never returns unless an error occurs.

execl

```
int execl(const char *path, const char *arg0,
  ..., const char *argn, (char *)0);

int execle(const char *path, const char *arg0,
  ..., const char *argn, (char *)0, char
  *const envp[]);

int execlp(const char *file, const char *arg0,
  ..., const char *argn, (char *)0);

int execlpe(const char *file, const char
  *arg0, ..., const char *argn, (char *)0,
  char *const envp[]);
```


execl Example Usage

```
int ret;
ret= execl ("/bin/ls", "ls", "-1", (char *)0);

int ret;
char *env={"HOME=/usr/home", "LOGNAME=home",(char *)0};
execle ("/bin/ls", "ls", "-l", (char *)0, env);

int ret;
ret = execlp ("ls", "ls", "-l", (char *)0);
```

execl can similarly be used for running any other application

execv

```
int execv(const char *path, char *const
  argv[]);

int execve(const char *path, char *const
  argv[], char *const envp[]);

int execvp(const char *file, char *const
  argv[]);

int execvpe(const char *file, char *const
  argv[], char *const envp[]);
```


execv Example Usage

```
int ret; char *cmd[] = { "ls", "-l", (char *)0 };
ret = execv ("/bin/ls", cmd);

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] =
  { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
ret = execve ("/bin/ls", cmd, env);

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
ret = execvp ("ls", cmd);
```

execv can similarly be used for running any other application

Process States

- Child dies before parent =>
  - Child becomes (Z)ombie till it is not removed from process table
- Parent dies before parent =>
  - Child becomes (O)rphan temporarily
  - Inherited by PID 1
- Can use (S)leep (n) to make a process wait for n seconds.

The Parent Process

- After creating a child process, the parent process may:
  - Wait for the termination of the child process
    - ✓To gather child process' exit status
  - Continue execution, without waiting for child.
- The parent process uses the ***wait*** system call to wait for the termination of the child process.

wait

Used by a parent process to wait till a child process doesn't terminate.

Behaviour:

- If the process calling wait doesn't have any child processes
  - ✓wait returns -1
- If the calling process has a Zombie child
  - ✓PID of the Zombie is returned to the parent
  - ✓Zombie child is removed from the process table
- If the calling process has a child that hasn't terminated yet
  - ✓Parent process is suspended till the child doesn't terminate
  - ✓When the child terminates, a signal is received by the parent that resumes its execution.
  - ✓A zombie child, if present, is also removed at this point.

Exit Status of a Child Process

- Apart from returning PID of a child process wait also tells about the termination status of the child process

  pid_t wait (int * status);

- The value of *status can be used in order to check the exit status of the child process.

Threads

- Threads, like processes are mechanism to allow a program to do more than one thing at a time.
- As with processes,
  - threads appear to run concurrently;
  - the Linux kernel schedules them asynchronously,
  - interrupting each thread from time to time to give others a chance to execute.

Threads Basics

- Conceptually, a thread exists within a process.
- Threads are a finer-grained unit of execution than processes. When a program is invoked,
  - Linux creates a new process
  - and in that process creates a single thread, which runs the program sequentially.
  - This thread can create additional threads;
  - all these threads run the same program in the same process,
  - but each thread may be executing a different part of the program at any given time.

What Processes Share…

- We've seen how a program can fork a child process.
- The child process is initially running its parent's program, with its parent's virtual memory, file descriptors, and so on copied.
- The child process can modify its memory, close file descriptors, and the like without affecting its parent, and vice versa

…Threads Don't

- When a program creates another thread, though, **nothing is copied**. The creating and the created thread **share**
  - the same memory space,
  - file descriptors, and
  - other system resources as the original.
- If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value.
- Similarly, if one thread closes a file descriptor, other threads may not read from or write to that file descriptor.
- Because a process and all its threads can be executing only one program at a time, if any thread inside a process calls one of the exec functions, all the other threads are ended (the new program may, of course, create new threads).

pthreads

- GNU/Linux implements the POSIX standard thread API (known as *pthreads*).
- All thread functions and data types are declared in the header file <`pthread.h`>.
- The `pthread` functions are not included in the standard C library. Instead, they are in `libpthread`,
- Thus, one needs to add `-lpthread` to the command line while linking a program that creates threads.

Thread IDs and Function

- Each thread in a process is identified by a *thread ID*.When referring to thread IDs in C or C++ programs, use the type pthread_t.
- Upon creation, each thread executes a *thread function*.
- This is just an ordinary function and contains the code that the thread should run.
- When the function returns, the thread exits.

Thread Argument

- On GNU/Linux, thread functions take a single parameter, of type `void*`, and have a `void*` return type. The parameter is the *thread argument*:
- GNU/Linux passes the value along to the thread without looking at it.
- Your program can use this parameter to pass data to a new thread.
- Similarly, your program can use the return value to pass data from an exiting thread back to its creator.

pthread_create

The `pthread_create` function creates a new thread. The parameters are:

1. A pointer to a `pthread_t` variable, in which the thread ID of the new thread is stored.
2. A pointer to a *thread attribute* object. This object controls details of how the thread interacts with the rest of the program. If you pass NULL as the thread attribute, a thread will be created with the default thread attributes. (Thread attributes are beyond the scope of current discussion)
3. A pointer to the thread function. This is an ordinary function pointer, of this type: `void* (*) (void*)`
4. A thread argument value of type `void*`. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

- A call to `pthread_create` returns immediately, and the original thread continues executing the instructions following the call.
- Meanwhile, the new thread begins executing the thread function.
- Linux schedules both threads asynchronously, and your program must not rely on the relative order in which instructions are executed in the two threads.
- The program in the example that follows, creates a thread that prints x's continuously to standard error.
- After calling `pthread_create`, the main thread prints o's continuously to standard error.

Thread creation-a simple implementation

```
#include <pthread.h>
#include <stdio.h>
/* Prints x's to stderr. The parameter is unused. Does not
   return. */
void* print_xs (void* unused)
{
    while (1) { fputc ('x', stderr); }
    return NULL;
}
/* The main program. */
int main ()
{
   pthread_t thread_id;
/* Create a new thread. The new thread will run print_xs
   function. */
   pthread_create (&thread_id, NULL, &print_xs, NULL);
/* Print o's continuously to stderr. */
   while (1) { fputc ('o', stderr); }
   return 0;
}
```


Exiting From a Thread

- Under normal circumstances, a thread exits in one of two ways. One way, as illustrated previously, is by returning from the thread function.
- The return value from the thread function is taken to be the return value of the thread.
- Alternately, a thread can exit explicitly by calling `pthread_exit`.
- This function may be called from within the thread function or from some other function called directly or indirectly by the thread function.
- The argument to `pthread_exit` is the thread's return value.

Further reading…

…clone to create threads / processes

And decide what is shared.