System Calls

Outline

- How they work
- File-oriented Linux system calls
    - File descriptors
    - open, read, write, close
- Process-oriented Linux system calls
    - Process IDs
    - fork, execve, wait

System Calls

- Entry points so a process can use kernel services
- Calling them:
  - The actual entry points are wrapped in a library function
  - This sets up the arguments and causes a trap
  - At that point, the kernel gets control and services the request
  - On success, modifies system as appropriate and (possibly) return something
  - On failure, return error, error code

Example Wrapper (PDP-11, from UNIX v6)

```
/ file = open(string, mode)
/
/ file == -1 means error
.globl  _open, cerror
_open:
        mov      r5,-(sp)          / push contents of register r5 onto the stack
        mov      sp,r5             / put stack pointer into register r5
        mov      4(r5),0f          / put first argument into memory location
        mov      6(r5),0f+2        / put second argument into memory location
        sys      0; 9f             / make open system call
        bec      1f                / on success, go to 1 below
        jmp      cerror            / on failure, jump to error routine
1:
        mov      (sp)+,r5          / restore previous value of r5
        rts      pc                / return
.data
9:
        sys      open              / symbolic value of open call (here, it’s 5; see as29.s)
0:..; ..
```

Linux File System

- File system is tree of directories and files on a single partition (device)
    - Files stored on device
    - Kernel identifies files by device number and inode number
    - Directory is really a file with inode, filename pairs identifying files contained in that directory
    - May have 2 entries for same file; cannot cross devices
        - inode numbers the same, but names differ
        - Called *hard link* or *link*
    - One file may simply contain path name of another file; can cross devices
        - Called *symbolic link* or *soft link*
- *Much* more on this later

System Call Errors

- All return –1 on error
- Specific error is given in external variable *errno* (an int)
    - If positive, error occurred
    - Use *perror*(3) to print error message
    - Important: *errno* is ***not*** cleared automatically!

Files

- In programs, represented by file descriptors
  - These are non-negative integers, typically very small
  - Some preassigned
    - 0 for standard input
    - 1 for standard output
    - 2 for standard error
- File pointers point to a structure, one element being the file descriptor
- Kernel maintains file pointer at position of reading/writing in file
  - *This is not the same as the file pointer at user level!!!!*

Accessing File

- First open it
    - This assigns a file descriptor to the file, usually the lowest unused number
    - Returns –1 on error; error code in global variable *errno*
- Then operate on it
    - read puts information into memory
    - write copies information out of memory
- When done, close it
    - This releases the file descriptor so it can be reused

Example: syscall-1.c

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


int main(void)
{
    char *f = "test.py";
    char buf[1024];
    int fd, n;

    if ((fd = open(f, O_RDONLY)) < 0 ||
           (n = read(fd, buf, 1023)) < 0){
        perror(f);
        exit(1);
    }
    (void) close(fd);

    (void) write(1, buf, n);

    exit(0);
}
```

And Its Execution

```
bishop@ad3.ucdavis.edu@pc13:~$
```

open System Call Parameters

- first argument is file name
- second argument is one of:
  - O_RDONLY, O_WRONLY, O_RDWR, OAPPEND: red, write, read and write, append
  - O_CREAT: create file if it doesn’t exist ; no error if it does
  - O_EXCL: if O_CREAT called and file exists, give error
- third argument is optional but sets protection mode:
  - S_IRUSR, S_IWUSR, S_IXUSR: turn on owner read, write, execute (respectively)
  - S_IRGRP, S_IWGRP, S_IXGRP: turn on owner read, write, execute (respectively)
  - S_IROTH, S_IWOTH, S_IXOTH: turn on owner read, write, execute (respectively)

read, *write* System Call Parameters

- int read(int *filedescriptor*, void **buffer*, unsigned int *numbytes*)
  - Read *numbytes* from file identified by *filedescriptor*, put then in *buffer*
  - Returns:
    - number of bytes read when anything read (note: may differ from *numbytes*!)
    - 0 on EOF
    - –1 on error; reason put into *errno*
- int write(int *filedescriptor*, const void **buffer*, unsigned int *numbytes*)
  - Write *numbytes* from *buffer* into file identified by *filedescriptor*
  - Returns:
    - number of bytes written when anything written (note: may differ from *numbytes*!)
    - –1 on error; reason put into *errno*
  - Note: write is (usually) to kernel buffer; actual write to device would come later

close System Call Parameters

int close(int *filedescriptor*)

- Dissociate *filedescriptor* from the file
- This closes the file
- If *filedescriptor* is open when the process quits, it is automatically closed
- Returns:
  - 0 on success
  - –1 on failure; reason put into *errno*

Other Useful File System Calls

- int stat(const char **pathname*, struct stbuf **pathinfo*)
  - Puts information about *pathname* in structure *pathinfo*
  - Returns:
    - 0 on success
    - –1 on failure; reason put into *errno*
- int lstat(const char **pathname*, struct stbuf **pathinfo*)
  - Like *stat*, but if pathname is symbolic link, return information about link itself and not target of symbolic link

Other Useful File System Calls

- long int lseek(int filedescriptor, long into offset, int position)
  - Position kernel file pointer to filedescriptor to offset bytes from position
  - position is one of:
    - SEEK_SET: from beginning of file
    - SEEK_END: from end of file
    - SEEK_CUR: from current position of kernel file pointer

Other Useful File System Calls

- int link(const char *oldpath, const char *newpath)
  - Create *newpath* as another name for *oldpath*
  - *oldpath* must exist, or error
  - Returns:
    - 0 on success
    - –1 on failure, reason put into *errno*
- int symlink(const char *oldpath, const char *newpath)
  - Like *link*, but creates a symbolic link rather than a hard link

Other Useful File System Calls

- int unlink(const char *path)
  - Delete link to *path*; if no links remail, and file is not opened, this deletes that file
  - Returns:
    - 0 on success
    - –1 on failure, reason put into *errno*
- int symlink(const char *oldpath, const char *newpath)
  - Like *link*, but creates a symbolic link rather than a hard link

Linux Process-Oriented System Calls

- Processes named by identification number (*pid*)
- Process parent PID available to child
- Process information kept in a table (the *process table*)
  - Older UNIX systems: this was fixed size
  - Current systems: it can be expanded
- Usually limits imposed on number of processes a user may run at the same time
  - Does not apply to *root*
  - Often a configuration option for the *system*; users cannot set it

Linux Process-Oriented System Calls

- int fork()
  - Duplicates the current process, except for:
    - PID; this is unique
    - Parent PID; this is the PID of the process that called fork()
  - In particular, open file descriptors are inherited
    - Basis for interprocess communication

Linux Process-Oriented System Calls

- int execve(const char *path. char *const argv[], char *const envp)
  - Executes file *path* with arguments *argv* and environment *envp*
  - If *envp* omitted, the current environment variables are used
  - Returns:
    - On success, this overlays current process and so does not return
    - –1 on failure; reason put in *errno*
  - File descriptors remain open across *execve*s
    - Exception: a file descriptor can be marked “close-on-exec”

Linux Process-Oriented System Calls

- int wait(int *status)
  - Pauses process until one of its children terminates
  - Status of child returned in *status*
  - Returns:
    - PID of terminating child on success
    - –1 on failure; reason put in *errno*
- int waitpid(int pid, int *status, int options)
  - Like wait() but waits for specific PID
  - If pid set to –1, waits for any child to complete
  - *options* is 0 is none needed, WNOHANG if waitpid should return immediately id no child has exited

Linux Process-Oriented System Calls

- void _exit(int *status*)
  - Terminate the process immediately
  - Any open file descriptors are closed
  - *status* is exit status, sent to parent
    - Only least significant byte of this sent
  - Usually invoked as *exit*(), which is really a library function

Linux Process-Oriented System Calls

- void _exit(int *status*)
  - Terminate the process immediately
    - So it does not return
  - Any open file descriptors are closed
  - *status* is exit status, sent to parent
    - Only least significant byte of this sent
    - Predefined status EXIT_SUCCESS means program worked; by convention this is 0
    - Predefined status EXIT_FAILURE means an error occurred; by convention this is 1
    - Can use any integer

Linux Process-Oriented System Calls

- int getpid(void)
- int getppid(void)
  - These return the process PID or parent process PID
  - Always successful

Other Useful System Calls

- int getuid(), getgid()
  - Returns user ID (UID), primary group ID (GID)
  - Always succeeds
- int setuid(int UID), setgid(int GID)
  - Sets user ID (UID), primary group ID (GID)
  - Returns:
    - 0 on success
    - 1 on failure; reason put into *errno*
- int setreuid(int ruid, int euid), setregid(int rgid, egid)
  - Sets real (ruid) and effective (euid) user ID, primary group real (rgid) and effective (egid) ugroup IDs
  - Returns:
    - 0 on success
    - 1 on failure; reason put into *errno*

Where to Find Information

- Section 2 of the UNIX manual