Linux API

Presented By:

Amir Reza Niakan Lahiji

IEEE POSIX

- POSIX (pronounced /ˈpɒzɪks/ *POZ-iks*) is a family of standards developed by the IEEE

- POSIX stands for **P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**].

- Of specific interest to this lecture is the 1003.1 operating system interface standard (about Core Services)

- Although the 1003.1 standard is based on the UNIX operating system, the standard is not restricted to UNIX and UNIX-like systems.

Error Handeling

- When an error occurs in one of the UNIX System functions, a negative value is often returned.
- and the integer errno is usually set to a value that gives additional information
- The file <errno.h> defines the symbol errno and constants for each value that errno can assume.
- There are two rules to be aware of with respect to errno.
  - First, its value is never cleared by a routine if an error does not occur. Therefore, we should examine its value only when the return value from a function indicates that an error occurred.
  - Second, the value of errno is never set to 0 by any of the functions, and none of the constants defined in <errno.h> has a value of 0.

File I/O

- **File Descriptors**
  - To the kernel, all open files are referred to by file descriptors.
  - A file descriptor is a non-negative integer
  - When we open an existing file or create a new file, the kernel returns a file descriptor to the process.
  - By convention,*NIX System shells associate
    - file descriptor 0 with the standard input of a process
    - file descriptor 1 with the standard output
    - and file descriptor 2 with the standard error

Open a file

- **#include <fcntl.h>**
- *int* **open**(*const char *path, int oflag, ...*);
  - Return -1 on error
  - Otherwise return file descriptor
  - oflag
    - O_RDONLY   Open for reading only.
    - O_WRONLY   Open for writing only.
    - O_RDWR   Open for reading and writing.

Manual Page

- Where are man pages for linux system calls?
  - Install manpages-dev package
    - sudo apt-get install manpages-dev

Close a file

- #include <unistd.h>
- int close(int filedes);
- Closing a file also releases any record locks that the process may have on the file.
- When a process terminates, all of its open files are closed automatically by the kernel.

Read Function

- #include <unistd.h>
- ssize_t read(int filedes, void *buf, size_t nbytes);
- If the read is successful, the number of bytes read is returned.
- the end of file is encountered, 0 is returned.
- Returns -1 on error

Write Function

- #include <unistd.h>

- ssize_t write(int filedes, const void *buf, size_t nbytes);

- Returns: number of bytes written if OK, -1 on error

Signal

- Signals are software interrupts.
- Signals provide a way of handling asynchronous events
- every signal has a name
  - These names all begin with the three characters SIG
  - Linux 2.6.28 has 32 different signals
    - /usr/include/asm/signal.h

Signal

- We can tell the kernel to do one of three things when a signal occurs.
  - Ignore the signal
    - This works for most signals, but two signals can never be ignored: SIGKILL and SIGSTOP.
    - SIGKILL and SIGSTOP are nonmaskable
  - Catch the signal.
    - To do this, we tell the kernel to call a function of ours whenever the signal occurs.
  - Let the default action apply
    - Every signal has a default action

Signal

- #include <signal.h>
- void (*signal(int signo, void (*func)(int)))(int);
- Returns: previous disposition of signal if OK, SIG_ERR on error
  - #define SIG_ERR (void (*)())-1
  - #define SIG_DFL (void (*)())0
  - #define SIG_IGN (void (*)())1
- Example
  - If(signal(SIGALRM,sig_alarm) == SIG_ERR)
  - Where sig_alarm declaration is
    - void sig_alarm(int signo);

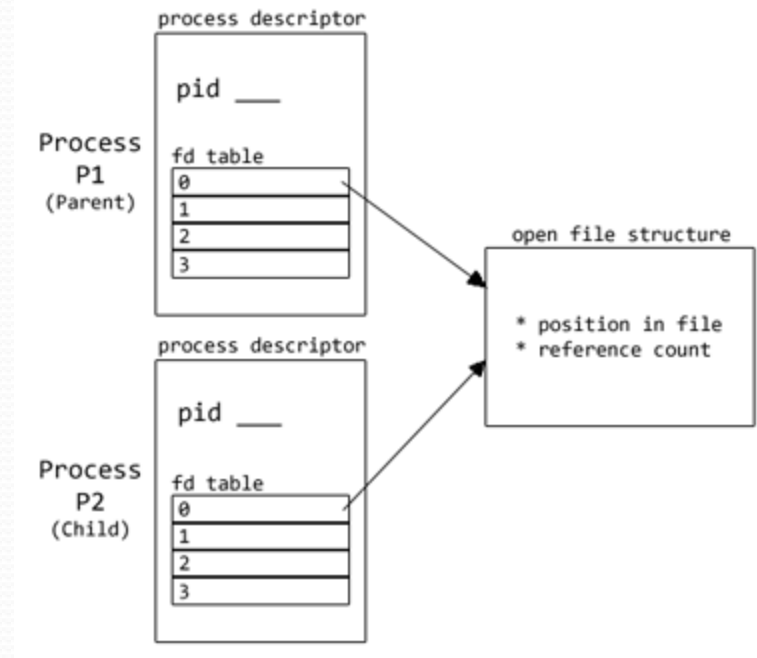# Process

- Process Identifiers
  - Every process has a unique process ID, a non-negative integer.
  - Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse.
  - Process ID 0 is usually the scheduler process and is often known as the swapper
  - Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure.
    - #include <unistd.h>
    - pid_t getpid(void);   Returns: process ID of calling process
    - pid_t getppid(void);   Returns: parent process ID of calling process

Process

- fork
  - An existing process can create a new one by calling the fork function.
  - #include <unistd.h>
  - pid_t fork(void);
  - Returns: 0 in child, process ID of child in parent, -1 on error
  - all file descriptors that are open in the parent are duplicated in the child.

Process

- It is important that the parent and the child share the same file offset

Process

- There are two normal cases for handling the descriptors after a fork.
  - The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors.
  - Both the parent and the child go their own ways. Here, after the fork, the parent closes the descriptors that it doesn't need, and the child does the same thing.

Process

- wait and waitpid
  - When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
  - the termination of a child is an asynchronous event it can happen at any time while the parent is running this signal is the asynchronous notification from the kernel to the parent.
  - a process that calls wait or waitpid can
    - Block, if all of its children are still running
    - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
    - Return immediately with an error, if it doesn't have any child processes

Process

- #include <sys/wait.h>
- pid_t wait(int *statloc);
- pid_t waitpid(pid_t pid, int *statloc, int options);
- Both return: process ID if OK, 0 or -1 on error

Process

- exec functions
  - When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
  - The process ID does not change across an exec
  - There are six different exec functions
  - execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", (char *) 0);

Process

- #include <unistd.h>
- int execl(const char *pathname, const char *arg0, ... /* (char *)0 */);
- int execv(const char *pathname, char *const argv []);
- int execle(const char *pathname, const char *arg0, ... /* (char *)0, char *const envp[] */);
- int execve(const char *pathname, char *const argv[], char *const envp []);
- int execlp(const char *filename, const char *arg0, ... /* (char *)0 */);
- int execvp(const char *filename, char *const argv []);
- All six return: 1 on error, no return on success

Process

- Example

```
#include <unistd.h>
main()
{
    execl("/usr/bin/gedit", "/usr/bin/gedit", (char *) 0);
}
```