

EMBEDDED OPERATING SYSTEMS

Embedded Linux on Beaglebone Black

System calls: What?

- Programmatic way in which
 - A user space program
 - Requests a service/resource
 - From the OS kernel
- Method for program interaction with kernel
 - The only entry points into the kernel
- Services provided by system calls:
 - Process creation and management
 - Main memory management
 - File access, directory and file system management
 - Device handling (I/O)
 - Protection / security
 - Networking

System calls: Features

- Interface
 - Well-define API for programs to access kernel
- Protection
 - Used to access privileged operations
- Kernel mode
 - User mode program temporarily switched to kernel mode
- Context switching
 - Needs context switch (user«»kernel); impacts performance
- Error handling
 - Return defined error codes to indicate problems
- Synchronization
 - Sync access to shared resources like memory, files, resources

System calls: Advantages

- Access to hardware
 - Streamlined access to hardware through kernel
- Memory management
 - Controlled usage of memory and mem-mapped devices
- Process management
 - Allows programs to create/terminate processes, IPC
- Security
 - Allows secure access to privileged operations
- Standardization
 - Documented interface with specific APIs

Linux syscalls: Process

- Process management
 - `fork()`
 - Create new processes
 - Parent creates child / children
 - `exec()`
 - To run different program from the current one
 - Image changed in place
 - `wait()`
 - Parent waits for child(ren) to terminate
 - For monitoring child(ren)
 - `exit()`
 - Terminate the current process with error code
 - Returns error code to caller

Linux syscalls: File

- File access and operations
 - `open()`
 - Open / create a file on the file system
 - `read()`
 - Read data from an existing file on file system
 - Can be used on pipes, sockets and /dev devices
 - `write()`
 - Send data to a file on file system
 - `close()`
 - Finalizes the file and saves it on file system

Linux syscalls: Network

- Network access and operations
 - `socket()`
 - Create a 'socket' for networked communication
 - `bind()`
 - Bind the socket to a port and address on local network
 - `listen()`
 - Act as a listening server on the socket
 - `accept()`
 - Accept an incoming request on the socket
 - `connect()`
 - Connect to the external endpoint
 - `send()`
 - Send data over the socket
 - `recv()`
 - Receive data over the socket

Linux syscalls: Device

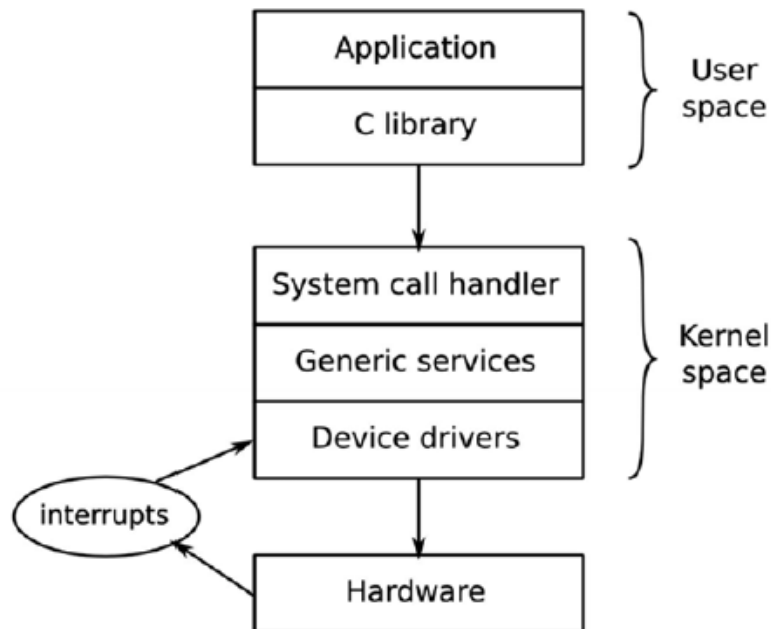
- Device access and operations
 - `ioctl()`
 - Send control commands to underlying device
 - `mmap()`
 - Map a partition of a file into the memory of the process
 - Used for operating on small portions of large files

Linux syscalls: System info

- Get information on system resources
 - getpid()
 - Get process ID (PID) of current process
 - getppid()
 - Get PID of process's parent
 - getuid()
 - Get User ID (UID) of current process
 - getgid()
 - Get Group ID (GID) of current process
 - uname()
 - Get system name, version, release info, etc.
 - sysinfo()
 - Get resource info like free memory, total memory, current processes
 - time()
 - Gets current system time (since 01 Jan 1970)

System calls and C library

- C library and system calls
 - Form the bridge between
 - User space and kernel space



Error handling

- Most library functions and system calls
 - Return codes on success / failure
 - Called error codes
- System calls
 - Return -1 to indicate error
 - Place an integer in a global variable ***errno***
- User space programs should
 - Test return status
 - If error occurred, inspect ***errno*** variable

Using errno for error handling

- Global, positive integer (>0) - each value has a specific meaning
 - `$ sudo apt-get install moreutils`
 - `$ errno -l` (gives all errors)
- Header file: ***errno.h*** (also, *man errno*)
- 2 ways to print errno details
 - `perror()` [*stdio.h, errno.h*]
 - `void perror(const char *s);`
 - Eg. **`perror(str);`** → `str`: No such file or directory
 - `strerror()` [*string.h*]
 - `char *strerror(int errnum);`
 - Eg. **`strerror(2);`** → No such file or directory

File IO: Using C library

- Header
 - `#include <stdio.h>`
- Default streams
 - **`stdout`**, **`stdin`**, **`stderr`**
- Functions
 - File open
 - `FILE *fopen(const char *pathname, const char *mode);`
 - File read
 - `char *fgets(char *s, int size, FILE *stream);`
 - `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
 - File write
 - `int fprintf(FILE *stream, const char *format, ...);`
 - `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
 - File close
 - `int fclose(FILE *stream);`

File IO: Using system calls

- Headers
 - `#include <sys/types.h> // system types`
 - `#include <sys/stat.h> // open`
 - `#include <unistd.h> // read, write`
 - `#include <fcntl.h> // fd manipulation`
- Functions
 - File open
 - `int open(const char *pathname, int flags, mode_t mode);`
 - File read
 - `ssize_t read(int fd, void *buf, size_t count);`
 - File write
 - `ssize_t write(int fd, const void *buf, size_t count);`
 - File close
 - `int close(int fd);`

Syscall: open()

- `int open(const char *pathname, int flags, mode_t mode);`
 - Flags
 - Access modes:
 - `O_RDONLY` / `O_WRONLY` / `O_RDWR`
 - Creation flags:
 - `O_CREAT`, `O_EXCL`, `O_TRUNC`
 - Status flags:
 - `O_APPEND`, `O_SYNC`, `O_NONBLOCK`
 - Modes (*denote permissions*)
 - User: `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRWXU`
 - Group: `S_IRGRP`, `S_IWGRP`, `S_IXGRP`, `S_IRWXG`
 - Others: `S_IROTH`, `S_IWOTH`, `S_IXOTH`, `S_IRWXO`
 - Returns
 - Non-negative number as the fd
 - -1 on failure
- Example:
 - `int fd = read("a.bin", O_RDWR | O_CREAT | O_EXCL | O_TRUNC | O_NONBLOCK | S_IRWXU); // rwx-----`

Syscall: read()

- `ssize_t read(int fd, void *buf, size_t count);`
 - Arguments:
 - fd: file descriptor from *open()*
 - buf: buffer to read into (no null termination!)
 - count: number of bytes to read
 - Returns:
 - >0: number of bytes read (could be < count!)
 - 0: end of file
 - -1: error; check ***errno***

Syscall: write()

- `ssize_t write(int fd, const void *buf, size_t count);`
 - Arguments:
 - fd: file descriptor from *open()*
 - buf: buffer to write from
 - count: number of bytes to write
 - Returns:
 - Number of bytes written (could be < count!)
 - -1: error, check ***errno***

Syscall: close()

- int **close**(int fd);
 - Argument:
 - fd: file descriptor from *open()*
 - Returns:
 - 0: success
 - -1: error, check ***errno***

Syscalls vs C library

- Run the Makefile in bb-codes/syscalls
- Run ./perror-strerror.out
 - Illustrates all defined error codes in Linux
- Run ./io-clib.out
 - Illustrates use of C library functions for file I/O
- Run ./compare.sh
 - Timings for a large file copy
 - Using syscalls as well as C library functions
 - Are compared

THANK YOU!
