Embedded

System Calls

System Calls

A a system call is an interface to the operating system. Popular examples:

1. malloc() : could invoke memory allocation from OS
2. printf() : OS decides what to do with data you are wanting to "print"
3. Threading : Use the OS to create/launch multiple tasks or threads
4. fopen() : An elaborate function that does several things; more on this below

Why System Calls?

There are surprising number of system calls. The basic and fundamental reason why we need system calls is that you want to protect the OS from any malicious behavior. For example, we do not wish for fopen() to open a file from another user's directory (security reasons).

fopen()

An fopen() function is very elaborate because it is doing several things under the hood and abstracts away all the details:

1. User calls `fopen("C:/file.txt", "r")`;
2. OS code gets invoked through a "Software Interrupt". This essentially switches mode from your program execution to the OS running its internal code. This is basically a switch from user to "kernel mode". The code looks like the following:

```
# Load the address of the filename "example.txt" into r0 (first argument)
    ldr     r0, =.LC0               # r0 = address of "file.txt"

# Load the address of the mode string "r" into r1 (second argument)
    ldr     r1, =.LC1               # r1 = address of "r"
```

```
# Store the intent of what our program is asking the OS to do
# In a typical POSIX OS, value of 2 indicates the intent to fopen()
    ldr     r4, 2
# Call the fopen function
    bl      fopen                   # Branch with link to fopen

# At some point: Invoke an interrupt to the OS code on the spot
    SWI
# Actual fopen() doesn't belong with us, it belongs to the OS
```

When the SWI operation is triggered, your program yields its CPU to the OS code to handle the asynchronous "software interrupt". Inside the interrupt code, the OS would do the following:

1. Pick up the parameters from R0 and R1 to figure out what file and with what mode the file should be opened
2. It calls elaborate file system code, such as FAT32, or exFAT
    1. File system is needed to make sense out of the raw data stored in the SSD drive
3. Read the disk data with File System APIs
4. Enforce permission rules (do you have permission to open this file)
5. If all sanity checks turn out okay, then the file is actually opened to perform Read or Write operations

---

stdio

---

memory

Startup

Basic Concept

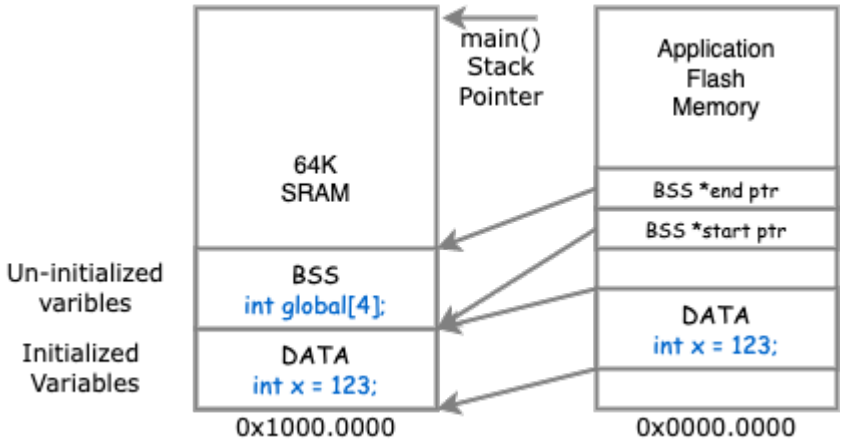
When your CPU starts up, the RAM is not initialized. So some entity needs to initialize the RAM for us. For a system with an operating system, such as linux or windows, your executable starts up, and the OS initializes the RAM before it calls the main() function. On microcontrollers, the startup process is entirely under your control. So a function needs to run before the main() to initialize the RAM.

Let's take a look at an example:

```
int x = 123;       // RAM
const int y = 456; // ROM (flash)
/* "data" section : Any RAM with initial values */
int d1 = 123;
static int d2 = 123;
int d3[10] = {1, 2, 3};
float d4 = 1.23;
/* "BSS" section:
 * Uninitialized RAM section, but
 * according to the C standard, un-initialized global variables need to be zero initialized
 */
int b1;
static int b2;
static float b3;
static char b4[30000];
int main(void) {
  y = 1;             // This will not compile
  * ((int*) &y) = 1; // This will crash
```

```
  // How will this print "x = 123"
  printf("x = %d\n", x);
  return 0;}
```

Illustration



Here are snippets of code the zero initialize the BSS and copy the DATA section from ROM to RAM.

```
static void startup__init_data_sram(void) {
  extern void *_bdata_lma;
  extern void *_bdata_vma;
  extern void *_data_end;
  uint8_t *src_flash = (uint8_t *)&_bdata_lma; // Flash
  uint8_t *dest_ram = (uint8_t *)&_bdata_vma;  // RAM
  while (dest_ram < (uint8_t *)&_data_end) {
    *dest_ram = *src_flash;
    dest_ram++;
    src_flash++;
  }
}
static void startup__init_bss_sram(void) {
  extern void *_bss_start;
  extern void *_bss_end;
  uint8_t *sram_ptr = (uint8_t *)&_bss_start;
  while (sram_ptr < (uint8_t *)&_bss_end) {
    *sram_ptr = 0U;
    sram_ptr++;
```

```
  }
}
```