Project Seven

ENEE 447: Operating Systems – Spring 2020
Assigned April 13; Due April 24

Purpose

Up until this point, all the “user” applications that have been run as separate and separately scheduled threads have been code compiled into the kernel binary, just run in user mode. In this project, the user applications are all in separate files. You will have to interact with the SD Card, but all of the driver work, in particular the SD interface and FAT filesystem running atop it, has been done for you.

You will read in application binaries from the SD card to start threads, and you will do this both as the startup thread (the shell) as well as in response to “RUN” commands executed in the shell, which will start up either or both of the “app1” and “app2” binaries.

Reading from the SD Card

The following shows the boot sequence for the code as given to you. In the *kernel.c* module, there is a function called *test_read()* that provides an example on how to interface with the *SDCard.c* module. After initializing the SD card and opening up the *kernel7.img* file (your bootable file on the card), it reads into a local buffer the first 1024 bytes of the *kernel7.img* file, prints out the first 64 words of it, and then goes into a forever loop.

```
[c0|00:02.259] ...
[c0|00:02.261] System is booting, kernel cpuid = 00000000
[c0|00:02.266] Kernel version: [p7, Fri Apr 12 16:13:13 EDT 2019]
[c0|00:02.272] Initializing SD Card ...
[c0|00:02.276] ---------------> sdInitCard [init]
[c0|00:02.280] EMMC: reset card.
[c0|00:02.283] EMMC: setting clock speed to  00061A80
[c0|00:02.288] GO_IDLE_STATE 00000000
[c0|00:02.291] SEND_IF_COND 000001AA
[c0|00:02.295] APP_CMD 00000000
[c0|00:02.298] SD_SENDOPCOND 50FF8000
[c0|00:02.702] APP_CMD 00000000
[c0|00:02.704] SD_SENDOPCOND 50FF8000
[c0|00:02.708] ALL_SEND_CID 00000000
[c0|00:02.711] SEND_REL_ADDR 00000000
[c0|00:02.715] SEND_CSD AAAA0000
[c0|00:02.718] EMMC: setting clock speed to  017D7840
[c0|00:02.722] CARD_SELECT AAAA0000
[c0|00:02.726] APP_CMD AAAA0000
[c0|00:02.729] SEND_SCR 00000000
[c0|00:02.734] SET_BLOCKLEN 00000200
sdTransferBlocks read blk 00000000 len 00000001 addr 0002BD88
c0|00:02.743] READ_SINGLE 00000000
sdTransferBlocks read blk 00002000 len 00000001 addr 0002BD88
c0|00:02.756] READ_SINGLE 00002000
[c0|00:02.770] ---------------> sdInitCard [term]
[c0|00:02.774] SD Card working.
[c0|00:02.777] create_thread:
[c0|00:02.780] NULL thread 00000000
[c0|00:02.783] stack = 00031000
[c0|00:02.786] start = 00000044
[c0|00:02.789] tcb    = 00013B00
[c0|00:02.792] create_thread:
[c0|00:02.795] shell.bin 00000000
```

```
[c0|00:02.798] Eggshell 00000001
[c0|00:02.801] stack = 00032000
[c0|00:02.804] start = 00000000
[c0|00:02.807] tcb     = 00013B64
[c0|00:02.810] ...
[c0|00:02.812] Init complete. Please hit any key to continue.
```

<hit enter>

```
[c0|00:05.890] test_read - SD Card example usage
sdTransferBlocks read blk 00003DCA len 00000001 addr 000070B8
[c0|00:05.909] READ_SINGLE 00003DCA LocateFATEntry:
[kernel7.im]
sdTransferBlocks read blk 00003DCB len 00000001 addr 000070B8
[c0|00:05.923] READ_SINGLE 00003DCB
sdTransferBlocks read blk 0000F6CA len 00000001 addr 000070B8
[c0|00:05.934] READ_SINGLE 0000F6CA
[c0|00:05.941] Reading file into buf at 000136D0
[c0|00:05.945] kernel7.img
sdTransferBlocks read blk 0000F6CB len 00000001 addr 000070B8
[c0|00:05.953] READ_SINGLE 0000F6CB

00000000: EA000006 EA00000E EA00003F EA00000C EA00000B EA00000A EA000029 EA000008
00000020: EE110F10 E3800A01 E3C00A02 EE010F10 EE100FB0 E7E10050 E3500000 0A000002
00000040: EAFFFFFF E320F003 EAFFFFFD E3A00000 E169F000 E162F300 E166F300 E16EF300
00000060: E164F300 E160F300 E16EF200 E3A00000 E12EF300 F1020012 E3A0DA2E F1020011
00000080: E3A0DA2F F1020013 E3A0DA2D F102001F E3A0D90B EB000D7C EB001185 F1020010
000000A0: E59FD010 E59F0008 E1A0F000 EAFFFFE4 00000000 00000000 00000000 00000000
000000C0: 00000000 E51FD010 E8CD7FFF E58DE03C E50FE018 E14FE000 E58DE040 E51FE024
000000E0: E3A0DA2D EB00116B EB000D20 EB001170 E51FD03C E59D0040 E16FF000 E59DE03C

[c0|00:06.018] Compare output to first 256 bytes of kernel7.list
[c0|00:06.023] Done.
```

When you build the code and run it, this is exactly what you should see. If not, there is a problem, and it is most likely with your SD card or the timing between your laptop and your SD card.

This shows the normal initialization sequence, with a new addition: the use of the SD card, which is initialized at the beginning, and then it is read at the end. The initialization sequence is heavily dependent on relative timing of the commands, and **you will need a Class-10 card**, for starters. Please check as soon as possible to see if your system works correctly, because debugging timing issues with drivers and devices can take an enormous amount of time, and it is not something you will want to be doing the weekend that the project is due.

The code that initializes the SD card looks like this:

```
sdInitCard(NULL, NULL, true);
```

And because it may not work initially, we have put it into a *while()* loop that keeps trying until successful. The code that reads the SD card looks like this:

```
fh = sdCreateFile(filename, GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
sdReadFile(fh, (void *)buf, 1024, &bytesRead, 0)
sdCloseHandle(fh);
```

The variable “fh” is a “file handle,” which happens to be an integer index into an array of data structures in the *SDCard.c* module. The *sdCreateFile()* function opens the file up and populates a data structure with information about the file, including where it is located in the file system, and how big it is, etc. The main argument you will use is the first one: the file name, a string with the name of one of the files at the root directory of your SD card.

When the file handle that *sdCreateFile()* returns is passed to the *sdReadFile()* function, the *sdReadFile()* function can use the previously discovered and stored information about the file to go find it and load it. This means that you do not have to know anything about sectors, blocks, or even the FAT filesystem structure.

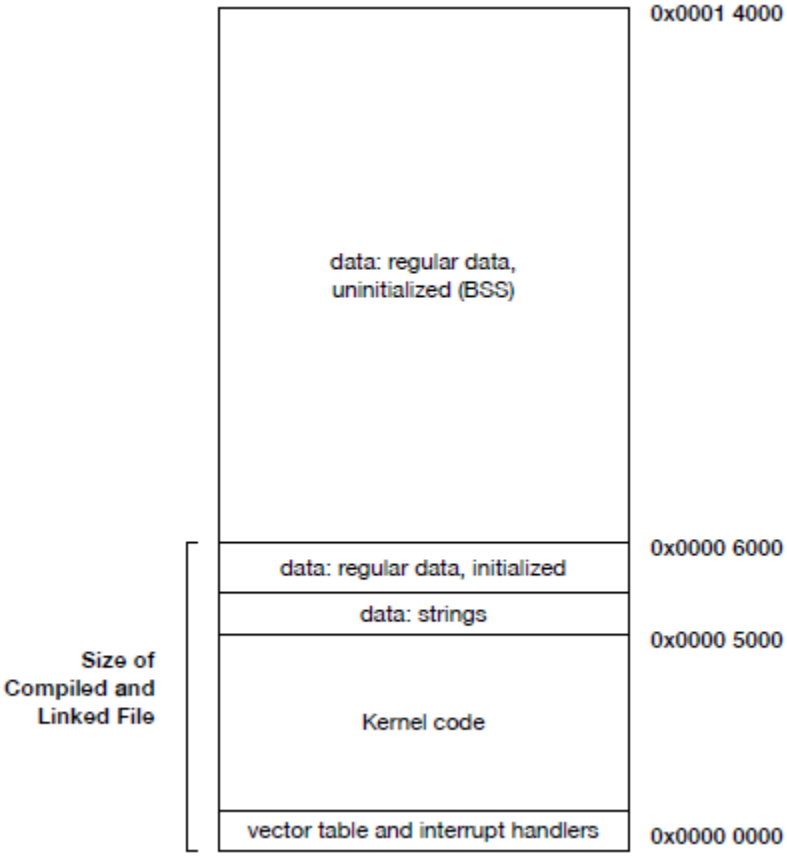
The arguments of the *sdReadFile()* function are as follows:

- file handle - data value returned by the sdCreateFile function
- buffer - address into which the data should be read
- size - the amount of bytes to read from the file into the buffer
- return: bytes read (a pointer to a uint32_t variable) - a return value indicating the amount of data actually read by the function
- unused (leave as 0)

The first three arguments are the ones you will care the most about. Lastly, the *sdCloseHandle()* function should be fairly self-explanatory, and it should be called as soon as you are done using the file.

What Address?

The main issue in this project is figuring out where to put things. Here is a basic structure for the kernel executable file. This is what is in *kernel7.img* and what is shown in human-readable form in *kernel7.list*.



To find this information out, you must look through the file *kernel7.list*. This is extremely important, in general, because it is the easiest way to figure out your code size and code layout. Note that if you simply rely upon the listed file size for the kernel binary, you might be misled into thinking that its size is

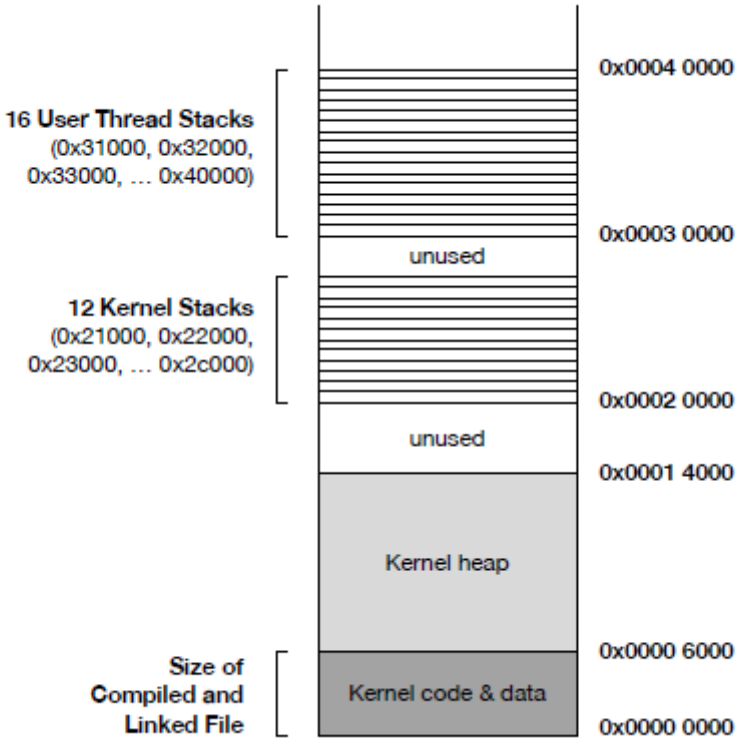
something that it is not. When you look at the compiled size of the kernel file, your laptop will report that the size of the file *kernel7.img* (or *kernel7.bin*) is roughly 25K.

Why is this worth paying attention to? This is why: **0x14000 ≠ 25,000**

The decimal value of 0x14000 is closer to 80K, not 25K. The amount of memory that the kernel uses is more than three times the size of the file as stored on disk. If you tried to put the various stacks and things right after the 25K mark, you would be interfering with the kernel's heap. Moral of the story: don't ever use the binary size as an indication of memory footprint. Anyway, the file ends around 0x06000, and the memory image ends around 0x14000. We have to address the following questions:

- Where should the kernel stacks go?
- Where should the thread stacks go?
- Where should the user application binaries go?

The first two questions have already been answered in previous projects. We have placed the kernel stacks in the 0x0002xxxx range, and we have placed the thread stacks (there are only 16 of them, for now) in the 0x0003xxxx range. This is shown below:



The kernel stacks are assigned statically in the *1_boot.s* module, and the user-thread stacks are assigned statically in the *threads.c* module. Now that the application binaries (shell, app1, and app2 executables) are external, separate files, they are no longer part of the kernel binary. This means that you need to load them explicitly from the SD card, but it also means that you need to decide *where* to put them.

The Raspberry Pi has on the order of 1GB of memory, which means that usable memory addresses *should* exist up to 0x40000000 (or, actually, 1 less than this). This means you have plenty of space to put your applications.

One thing that you will have to do is tell the compiler where they are; this needs to be a static decision. So, for example, if you decide that the shell thread should be loaded at location 0x40000, then you need to edit the shell's *memmap* file to reflect this. This has been done for you, for the shell application. The *memmap* files have the following format:

```
MEMORY
{
        ram : ORIGIN = 0x0000, LENGTH = 0x400000
}

SECTIONS
{
        .text : { *(.text*) } > ram
        .bss : { *(.bss*) } > ram
}
```

This is an extremely simple linker file (do a little research, and you will see … this is wonderfully simple, all thanks to David Welch). The main thing you should work with is the ORIGIN variable in the top part. This tells the linker where the application will start. Because all of the addresses will be different for each application (we are not yet implementing virtual memory), each will have to be loaded into a region that does not overlap with anything, and you will need to modify each application's *memmap* file to reflect the location into which it will be loaded. Yes, this is a pain in the butt, and it is one of the reasons that virtual memory is so awesome. :) Decide where you want app1 and app2 to be loaded, and modify their linker files accordingly.

Dynamic Thread Creation and Application Loading

Because this does not use virtual memory, the entire application binary must be resident in order to work.

1. Load the binary for the *eggshell* application during the initialization sequence in *kernel.c*
2. Load the binary for either/both of *app1* and *app2* applications when you "RUN" them from the shell Do this by modifying the *create_thread()* function, which should now take the following form:

```
void create_thread(char *name, char *filename, long address);
```

Now, instead of telling the function where in the *kernel* to find the executable, you tell the function where on the *disk* to find the executable image, with the *filename* argument. The *address* argument indicates where in memory to place the application binary. The *create_thread()* function should load the binary into the given address. The RUN command in the shell has the following arguments now:

```
RUN <THREAD> "filename"
```

The "THREAD" argument is up to 3 characters long, like before. The *create_thread()* routine will simply use this as part of the TCB data structure and print it out when doing a PS function. The last argument is the name of the binary file, and it needs to be in quotes for the shell to recognize it as a string. These will be passed to the trap handler, which should then invoke *create_thread()* accordingly.

When working, your output should look something like this:

```
Running the eggshell on core 0.
Available commands:
 RUN    = 004E5552
 PS     = 00005350
 TIME   = 454D4954
 LED    = 0044454C
 LOG    = 00474F4C
 EXIT = 54495845
 DUMP = 504D5544

Please enter a command.
c0> RUN BLK "APP1.BIN"
CMD_RUN [BLK, 000409F5]
[c0|00:18.194] SYSCALL_START_THREAD name = 004B4C42
```

```
[c0|00:18.199] SYSCALL_START_THREAD file = 000409F5
[c0|00:18.204] BLK
sdTransferBlocks read blk 00003DCA len 00000001 addr 00007168
[c0|00:18.211] READ_SINGLE 00003DCA
LocateFATEntry: [APP1.BIN]
sdTransferBlocks read blk 00003DCB len 00000001 addr 00007168
[c0|00:18.225] READ_SINGLE 00003DCB
sdTransferBlocks read blk 000104CA len 00000001 addr 00007168
[c0|00:18.236] READ_SINGLE 000104CA
sdTransferBlocks read blk 000104CB len 00000001 addr 00007168
[c0|00:18.248] READ_SINGLE 000104CB
[c0|00:18.254] create_thread - successful file read into 00060000
[c0|00:18.259] create_thread:
[c0|00:18.262] APP1.BIN 00060000
[c0|00:18.265] BLK 00000002
[c0|00:18.268] stack = 00033000
[c0|00:18.271] start   = 00060000
[c0|00:18.274] tcb     = 00013878

Please enter a command.
c0>
```

Running the “blinker” application, which is in the *app1.bin* file, is done by giving it a name (“BLK”) and pointing the kernel to the *app1.bin* file on disk. At this point, the LED should start blinking in the 1/2/3/4 pattern. The second application prints a pattern to the screen (it counts to 99 by 2-second time steps), which does conflict with the shell, but only for output (it does not read input from the console). It is called the “texter” application, it should look like this:

```
Please enter a command.
c0> RUN TXT "APP2.BIN"
CMD_RUN [TXT, 000409F5]
[c0|00:27.336] SYSCALL_START_THREAD name = 00545854
[c0|00:27.340] SYSCALL_START_THREAD file = 000409F5
[c0|00:27.345] TXT
sdTransferBlocks read blk 00003DCA len 00000001 addr 00007168
[c0|00:27.352] READ_SINGLE 00003DCA
LocateFATEntry: [APP2.BIN]
sdTransferBlocks read blk 00003DCB len 00000001 addr 00007168
[c0|00:27.366] READ_SINGLE 00003DCB
sdTransferBlocks read blk 000105CA len 00000001 addr 00007168
[c0|00:27.378] READ_SINGLE 000105CA
sdTransferBlocks read blk 000105CB len 00000001 addr 00007168
[c0|00:27.389] READ_SINGLE 000105CB
sdTransferBlocks read blk 000105CC len 00000001 addr 00007168
[c0|00:27.401] READ_SINGLE 000105CC
[c0|00:27.408] create_thread - successful file read into 00080000
[c0|00:27.413] create_thread:
[c0|00:27.416] APP2.BIN 00080000
[c0|00:27.419] TXT 00000003
[c0|00:27.422] stack = 00034000
[c0|00:27.425] start = 00080000
[c0|00:27.428] tcb   = 000138DC
[c0|00:27.431] Texter: zero

Please enter a command.
c0> [c0|00:29.433] Texter: one
[c0|00:31.436] Texter: two
[c0|00:33.439] Texter: three
[c0|00:35.442] Texter: four
[c0|00:37.445] Texter: five
```

Build It, Load It, Run It

Once you have it working, show us.