Lab 5 Process Loading and Memory Management

Goals

- Integrate a FAT file system and disk driver (Lab 4 or starter code),
- Develop or integrate a dynamic heap memory manager,
- Integrate capability to load ELF files from disk into dynamically allocated memory,
- Enable static or dynamic linking of ELF executables against driver and library code,
- Develop capability for OS routines to be invoked via supervisor calls (SVC),
- Extend the OS kernel with facilities for process and memory management,
- Add commands to your interpreter to load and launch a user program from disk.

Review

- Chapter 7 in the book,
- Executable and Linkable Format (ELF) specification.

Starter files

- **`RTOS_Lab5_ProcessLoader`** (starter project for Lab 5 OS)
  - **`heap.c`** and **`heap.h`** containing prototypes for a heap memory manager to be implemented in this lab (in **RTOS_Labs_common** folder)
  - **`ff.c`**, **`ff.h`**, **`ffconf.h`** and **`integer.h`** containing a FAT file system implementation (from **`SDCFile_4C123`** starter project)
  - **`eFile.c`** containing an **eFile**-compliant wrapper around the FAT file system API
  - **`loader.c`, `loader.h`** and **`elf.h`** containing an ELF file loader implementation (from https://github.com/gerstl/elfloader)
  - **`loader_config.h`** configuration of the ELF file loader for integration into OS
  - **`Lab5.c`** with the bare-bones OS kernel **main**
- **`RTOS_Lab5_User`** project with user application (feel free to create additional user programs)
  - **`OS.h`** (user-level OS API)
  - **`osasm.s`** (trampolines to call routines in the OS kernel via SVC exceptions/traps)
  - **`Display.c` and `Display.h`** (user-level display driver stub)
  - **`User.c`** (user application)

Background

The goal of this lab is to add facilities to your OS to load a separately compiled user program from disk, dynamically create and launch an associated OS process, enable the user program to make calls to OS routines, and dynamically manage processes and system memory such that memory is allocated for process resources (code and data) when a process is launched and a process is removed from the system and all memory occupied by it is reclaimed when its last thread exits.

As part of this lab, you will write a dynamic memory manager for dynamically allocating and deallocating memory on a system heap. You are free to implement any heap management approach and algorithm as long as it supports the basic operations of allocating, reallocating (growing or shrinking) and freeing arbitrary blocks of bytes. In addition, you should add interpreter commands that will allow printing heap usage statistics (size of the heap and number of bytes currently used or available for allocation). Prototypes of corresponding functions are provided in **`heap.h`** and **`heap.c`** for illustration purposes, i.e. similar to Lab 4, are given to you not as a detailed lab specification, but rather as a way to specify the level of complexity we expect you to achieve in this lab. You are free to change function names and parameters, as long as the basic operations are supported.

For loading of program files from SD cards, you can build on your Lab 4 file system driver. However, you need to be able to transfer user programs compiled on your laptop to the SD card. This is easiest if your SD card uses a standard PC-compatible FAT filesystem. The **`RTOS_Lab5_ProcessLoader`** starter project includes a complete FAT16 and FAT32 implementation (taken from the **`SDCFile_4C123`** starter project) on top of the SD card driver from Lab 4. Unless you already implemented FAT in Lab 4, it is easiest to replace your previous file system with this starter code. The **`RTOS_Lab5_ProcessLoader`** starter project also includes an **`eFile.c`** wrapper around the FAT implementation that provides a plug-in replacement for all routines in **`eFile.h`**. Alternatively, if you want to stick to your non-FAT file system from Lab 4, you need to implement a different way of transferring (binary) files from your laptop to the LaunchPad's SD card, e.g. via the UART connection.

Your OS is required to load and run the test program provided in the **RTOS_Lab5_User** Keil project. The project compiles into a **User.axf** ELF executable that you need to copy onto the SD card. You will add a command to your interpreter to load and execute this or other arbitrarily named program files from SD card. The ELF executable generated by Keil will contain separate segments for code and data (**--split** Linker option is set in the project options of **RTOS_Lab5_User** in Keil). The **RTOS_Lab5_ProcessLoader** starter project includes a reference implementation of an ELF file loader (**loader.c**, **loader.h** and **elf.h** from Github) that can parse and load such files in ELF format as generated by Keil. The ELF loader provides a

```
int exec_elf(const char* filename, const ELFEnv_t *env);
```

routine that you can call from your interpreter to load such executables into memory. The ELF loader code provided in the starter project is configured to internally use the FAT file system interface for all file I/O and the heap functions provided in **heap.h** for dynamic memory allocation (see **loader_config.h**). When called, the **exec_elf** routine will parse the ELF file, dynamically allocate memory on the heap for code and data segments (via calls to **Heap_Malloc**), load both segments from the file into their respective memory regions, perform all necessary relocations (see below) and finally call a

```
int OS_AddProcess(void(*entry)(void), void *text, void *data,
                  unsigned long stackSize, unsigned long priority);
```

routine in your OS to create and launch a corresponding new process. The routine will be passed the starting address of the entry point of code execution, pointers to the program's code and data segments on the heap, a stack size (default 128, can be ignored) and a process priority (default 1). You will need to provide an implementation of **OS_AddProcess** in your OS. Your implementation should allocate a Process Control Block (PCB) for the new process, add a new main thread associated with this process to start executing at the provided entry point, and return with either 0 or 1 in case of error or success.

During its execution, the user program will be able to make calls to OS routines, including **OS_AddThread** to create additional threads using the same code and data. Your OS should keep track of the threads associated with each process and remove the process, i.e. reclaim all of its resources and free its text and data memory regions when its last thread exits (calls **OS_Kill**). Your OS should be designed to handle repeated launching and exiting of processes as well as multiple active processes being loaded and executing at any given time.

The Keil project is setup to compile the test program using position independent code and data (called read-only and read-write position independence in Keil's C/C++, assembler and linker settings). This will allow the program to execute independently of where it is loaded into memory. The compiler will exclusively use PC-relative addressing for all code references (such as branch target addresses or addresses of constants stored in the code region). For accesses to global data variables, ARM defines the R9 register to be the static base (SB) register that points to the address of the program's data region in memory. All variable accesses generated by the compiler will be relative to this register. In the generated code, addresses for load/store instructions are computed at runtime as offsets of the variable location relative to the beginning of the data region added to the value in R9. As such, for the code to work, the OS will need to set R9 to the correct value before starting program execution.

A position-independent approach is not possible for accesses to code or data in the OS kernel. Neither addresses nor offsets of OS symbols are normally known when the user program is compiled. There are several solutions to resolve such external references. The simplest solution is to just provide the code for the OS routines such that they can be compiled into and included in each user program. The test program uses this approach for the **OS_TimeDifference** function, whose source code is provided in **OS.h**. This will, however, not work for routines that need to access OS data structures. It also incurs overhead and should be avoided for larger routines since a separate copy of each such OS routine will be included in each user program.

The test program instead uses supervisor calls (SVCs) to invoke all other OS kernel routines. Special SVC instructions allow triggering of an exception from user software in the ARM. They include an 8-bit immediate value to differentiate exceptions, where each **OS_xxx** call is mapped to a different SVC ID. An **SVC_Handler** in the OS kernel can then catch such exceptions and map them to corresponding OS kernel calls based on the provided ID. Software-triggered SVC exceptions are handled by the processor just like any other interrupt exception. The processor will automatically save R0-R3, R12, LR, PC and PSR registers on the stack before invoking the handler. To realize OS calls via SVC exceptions, you will have to implement an **SVC_Handler** performing the following steps:

1) Get the address of the SVC instruction triggering the exception from the PC return value on the stack.
2) Decode and extract the triggered exception ID from the SVC instruction.

3) Copy the values of R0-R3 saved on the stack into the corresponding registers. This allows up to 4 function parameters to be passed from the user program into OS routines.
4) Call the OS routine matching the triggered ID.
5) Copy the return value in R0 back onto the stack such that it will be made available to the caller.
6) Return from the SVC exception handler.

This approach via SVC traps allows user code to transparently call functions in the OS without needing to know their addresses. However, trapped functions will be executed in an exception context. No blocking or context switches are possible in the middle of such calls.

A third alternative is to statically or dynamically link the user code against actual OS kernel addresses. The test program uses this approach for calls to the **ST7735_Message** display driver function. For static linking, a symbol definition file containing absolute addresses needs to be generated when compiling the OS via a **–symdefs Lab5.def** Linker option. This **Lab5.def** file then needs to be included as object file when compiling the user program (setup is prepared but disabled by default in the provided Keil project). However, this requires recompiling of the user program every time addresses in the kernel change. As an alternative (and default setup of the provided Keil projects), external symbols are declared to be dynamically linked. The compiler inserts dummy addresses for each such symbol during generation of the executable, and marks corresponding as needing to be relocated in the generated ELF file. The ELF loader then patches each such location with correct addresses for the referred symbol when loading the binary. In order for this to work, you will need to provide an array with mappings of symbol names to OS kernel addresses via the **env** parameter to the ELF loader's **exec_elf** call. Passing a symbol table for dynamic linking to the ELF loader is done as follows:

```
static const ELFSymbol_t symtab[] = {
   { "ST7735_Message", ST7735_Message } // address of ST7735_Message
};

void LoadProgram() {
   ELFEnv_t env = { symtab, 1 };   // symbol table with one entry
   if (!exec_elf("User.axf", &env)) { … }
}
```

To relocate a binary after loading it in the memory, the ELF loader will check the relocation table provided in the ELF file. An ELF relocation table lists all locations in the file (and hence memory) that require relocation together with the name of the symbol they should be bound to. For each location listed in the relocation table, the ELF loader will try to find an entry with a matching name in the symbol table provided during the **exec_elf** call and replace the dummy address in memory with the correct address provided in the symbol table entry, all just right before finally executing the program. The advantage of this dynamic and late-binding approach of mapping function calls to correct addresses in the OS (in combination with SVC calls) is that user programs are completely independent of and decoupled from changes in OS kernel internals.

Preparation

1) Design your heap manager. There should be separate **heap.c** and **heap.h** files containing software that implements the heap. Write simple test programs for testing your heap under different corner conditions, including invalid operations (allocating more memory than available, freeing a pointer twice, etc.). Also add interpreter commands to test the heap (allocate/deallocate memory) and print heap usage information.

2) Define the **PCB** data structure and sketch the design of how process management will be performed in your OS. Write pseudo code for **OS_AddProcess** and any modifications you need to make to other OS routines. Write test code that creates new processes with one more threads from within your OS project itself.

3) Draft code for an **SVC_Handler** that handles user program calls to all required OS routines. Write test code for calling of OS routines via SVC traps from within your OS project itself.

4) Use the integrated FAT filesystem code or develop a capability to transfer files from your laptop to your Lab 4 non-FAT filesystem. Test correct filesystem and SD card operation by creating a simple text file on your laptop and printing that file from your OS interpreter.

5) Study the ELF specification. Draw a diagram of the user program layout both in the ELF file and when loaded into the LaunchPad memory. You can examine the generated ELF file using the **fromelf** utility that comes with the Keil installation:

```
C:\Keil_v5\ARM\ARMCC\bin\fromelf.exe –r –y User.axf
```

The **`–r`** parameter will show dynamic linking and relocation information. Illustrate the step-by-step operation of the ELF loader including the relocation process for dynamic linking of the **`ST7735_Message`** symbol.

6) Make sure you understand the operation of the provided ELF loader. In particular, study the code of **`exec_elf()`** routine in **`loader.c`** and single-step through the code in the debugger to observe its operation using dummy implementations of **`Heap_Malloc`** (e.g., just allocating static data regions) and **`OS_AddProcess`** (e.g., doing nothing).

Procedure

1) Implement, test and debug your heap manager (Preparation 1). Use *Testmain1* to verify correct heap operation.

2) Implement and test the process management capabilities of your OS kernel (Preparation 2). Add debugging instruments to profile process creation and completion times. Use *Testmain2* to verify correct process creation and removal.

3) Implement and test the **`SVC_Handler`** (Preparation 3). Add debugging instruments to profile SVC exceptions. Use *Testmain3* to verify correct SVC handling.

4) Add an interpreter command to load and launch a program stored on the SD card via the provided ELF loader. Compile the **`RTOS_Lab5_User`** user program on your laptop, copy the resulting **`User.axf`** ELF file to your SD card, and make sure that you can load it successfully from your interpreter.

5) Test the complete system that repeatedly loads, launches and executes **`User.axf`** via the interpreter.

Checkout (show this to the TA)

Demonstrate loading and launching of the user program from the interpreter. You should be able to demonstrate repeated launching of the program and multiple copies of it being launched to run simultaneously. Stress test the system with multiple concurrent process instances to the point of failure. If at all, your system should fail gracefully (error message instead of crashing). Be prepared to discuss loader and process management operation (Preparation 1 and 2) as well as the logic analyzer profile of program launching and execution.

Deliverables (exact components of the lab report and lab submission)

A) Objectives (1/2 page maximum)
B) Hardware Design (none)
C) Software Design (documentation and code)
   1) Pictures illustrating the loader operation, showing:
      · ELF file layout of compiled user program on disk;
      · ELF loader steps;
      · heap allocation scheme;
      · memory layout of machine after loading the program; and
      · dynamic linking and relocation process
   2) Operating system extensions (C and assembly), including **`SVC_Handler`**
   3) High level software system (the new interpreter commands)
D) Measurement Data
   1) Logic analyzer profile of OS kernel task execution
   2) Logic analyzer profile of user program execution: process creation, SVC traps, toggling of PF2 and PF3 LEDs by the user program's main and child threads, and process completion.
E) Analysis and Discussion (1 page maximum). In particular, answer these questions
   1) Briefly explain the dynamic memory allocation algorithm in your heap manager. Does this implementation have internal or external fragmentation?

2) How many simultaneously active processes can your system support? What factors limit this number, and how could it be increased?

Hints

0) Support for compiling the user test program with position-independent code and data requires a full Keil license. Email the Professor or TAs to receive a Keil license code.

1) You can refer to the class notes and the book for possible heap implementations. The **`Heap_4C123`** starter project provides a reference implementation of the heap discussed in the book. ***However****: students cannot use this heap code or any heap code written by others. You must implement your own heap implementation.*

2) The heap is a resource shared between multiple threads and processes, i.e. you should consider potential critical sections and how to handle them. Also make sure that your heap is large enough to load the given program. Lastly, code and data segments in ELF files are assumed to be aligned to 32-bit word boundaries when loaded into memory. As such, you should design your heap to allocate memory in granularity of 32-bit words and return pointers that are aligned to word boundaries.

3) The FAT reference implementation provided with the starter code is not necessarily fully complete. It is recommended to format the SD card on the Launchpad and only read/write files on the PC to avoid compatibility issues. Note, however, that the provided FAT implementation is also not optimized for speed. In particular the formatting process can take a very long time for SD cards with a high capacity, i.e. a long delay is normal.

4) As mentioned in Lab 4, since the display and SD card share the same SSI port, you need to protect simultaneous port accesses, e.g. by reusing the **`LCDFree`** semaphore to mutually exclude display, file system and now also ELF loader disk accesses.

5) The idle task included in the starter project puts the processor into low-power sleep whenever there is nothing to execute:

```
void IdleTask(void) {
  for(;;) {
    WaitForInterrupt();
  };
}
```

Note that you can extend the idle task to instead measure processor utilization defined as the ration of busy versus idle times. In the same manner as in Lab 3 or the Windows Task Manger, utilization values can then be reported through additional interpreter commands, for example.

6) In Thumb mode, the SVC instruction is 2 bytes long and its immediate value is located within its lower 8 bits. When the **`SVC_Handler`** is invoked, the return address on the stack will point to the next following instruction, i.e. the SVC instruction itself will be located 2 bytes before that address.

7) Due to late arrival and tail chaining in the ARM hardware exception handling, register values can be modified between the execution of the SVC instruction and execution of the SVC handler itself, i.e. only the values saved on the stack are guaranteed to correctly reflect the register content at the point when the SVC instruction was executed.

8) Processes created within the OS will not have separate code and data segments (Procedure 2). For testing purposes, you can just allocate dummy text and data regions on the heap, while having the process' starting address point to code in the ROM itself. See *Testmain2*.

9) The user program (**`RTOS_Lab5_User`**) will be compiled separately into an executable file (**User.axf**) and therefore doesn't need to be touched other than to compile it and understand it. However, you are free to create additional user programs to load and test in your Lab 5 OS.