tOS Documentation

Authored by: **NGX**

March 10, 2021

Abstract

This documentation describes the boot environment needed to boot the OS, functions that are needed for the OS to work on a different architecture and also some algorithms used by the OS.

Chapter 1

Platform specific and bootloader code Requirements

1.1 Introduction

This sections describes how the OS can be ported to be booted with other bootloader or another CPU architecture

Any function that is required by the OS from platform specific code is prefixed with arch_ e.g. arch_function

1.2 Interfacing with the boot protocol description

To interface with different boot protocols the OS requires a special bootstrap script that would setup everything and provide it in a standardized interface for the OS to work with. The interface is expressed in form of global functions that the OS could call. The link script get's to the OS at compile time - this means that the OS should be compiled and linked with it.

1.3 Boot-loader Functions

Info

The functions that are required by the kernel to be present in boot protocol interpreter file(a c file containing code to translate everything from bootloader to the kernel platform-independent code)

Function to obtain memory map

struct memInfo arch_getMemInfo(size_t count, uint8_t mmap_type)

- **Argument: count** - specifies the entry in the memory map
- **Argument: type** - specifies which type of memory map should be searched(supplied by boot protocol, made by the boot protocol interpreter script to add extra things)
- **Return** - returns a structure containing memory map entry

The function should:

1. Find the entry in the memory map specified by count. If it does not exist(not within the range of indexes) it should return an error(set error flag in meminfo structure and return it)
2. Translate the entry into the tOSs meminfo format by: putting the values from memmap into right fields of the meminfo structure that will be returned(the types should be converted to the tOSs format also)
3. Return translated memmap entry in the form of meminfo structure

- The function should have the area where kernel is located marked as KERNEL memory(there is a special macro for that) in the protocol mem map and not the additional interpreter memory map
- On request for **Boot protocol memory map** it should use count to return the value of the entry located at the index supplied in count in the memory map supplied by the boot protocol used
- On request for **Boot protocol interpreter map** it should do the same as with boot protocol map, but instead use index to find entry in interpreter map
- Specify the things it knows about memory like the frame-buffer, modules... in the interpreter map and not the protocol map

Example use - set some variable to zero and then call getmeminfo with this variable as count, then each call take the supplied data from returned structure and increment the variable(to get to the next entry in the memmap), do this until error flag is set in the returned structure - then finish

Function to do bootloader-specific things

int arch_bootloaderInterface(uint32_t function, void* data)

- **Argument: function** - is a code of the function that the bootloader should call

- **Argument: data** - a pointer that could be used both for additional arguments and the return values(function specific)
- **Return** - returns -1 on error an 0 on success

The functions should:

1. Read the functions code supplied in function variable and check if it has that function (switch: case: case:), if it does run it, else return error

- Should implement an INIT function that would setup the everything it should as the bootloader needs(it could just return success)

Start of the bootloader interprter and kernel start

void _start(void)

The functions should:

1. Start the OS by calling various bootloader interpreter functions that after setup would call the kernel

- Function shouldn't return to it's caller in any case

1.4 Boot process

- First the bootloader(could be any) should load the kernel and call the **_start** function which should be implemented by the bootloader interpreter code and start reading of data supplied by bootloader, interpreting it and doing other bootloader specific things
- Next the bootloader interpreter should call the **kernel_setup** function
- Kernel setup function does platform specific setup and then calls the **kernel_main** which starts the kernel