Instructions For Databar Assignment Tasks B4 And A4: Memory Management

1 Introduction

In this task, you will expand the kernel you investigated in databar assignment task B3. You do not have to have finished task A3 to work on this task. During the work on the task, you will implement two functions in the kernel: kalloc and kfree. kalloc allows you to dynamically allocate memory blocks; kfree is used to free up previously allocated memory blocks. Task A4, if you choose to work on it, involves implementing memory protection.

It is assumed that you are familiar with processes and threads.

2 Goals of the task

The task addresses the following learning objectives:

- Given a skeleton uniprocessor operating system, you can implement a memory management subsystem.

In addition, task A4 addresses:

- Given reference literature, you can write down in text and in your own words definitions or explanations of the following concepts: Swapping; Virtual memory; Paging;.

3 Report and rules

You will have to hand in written reports during the course. The reports should describe all the tasks you have completed. Begin writing on the reports as early as possible. You should also maintain a progress diary!

DTU has a zero tolerance policy on cheating and plagiarism. This also extends to the report and indeed all your work. For example, to copy figure, text passages or source code from someone else without permission or without clearly and properly citing your source is considered plagiarism. See the study hand book for further details.

4 Setting up the environment

The procedure for setting up task 4 is similar to that of task B2. Download the skeleton code from CampusNet. Then find the databar_assignments folder and create a new folder, named task_4 for example, in that folder. Enter the created folder and unpack the skeleton. The discussion below assumes you named the newly created folder: task_4.

Copy over your syscall.c and scheduler.c files from task B3 or task A3 if you have completed it!

Your environment is now set up! Use the same procedure for setting up Eclipse and building the sources as described in previous documents.

5 Source code tree

In the task_4 folder, you will find several folders and files. The overall structure is the same as for the kernel in task B3 and A3. However, some files have been added to hold the memory management sub-system. Here is a brief introduction:

- The Makefile file is a file that the make tool use to orchestrate the compilation of the kernel. The make tool will also, if the compilation is successful, invoke the Bochs emulator.
- Doxyfile is configuration file for a automated documentation system called "Doxygen"

  The doc folder holds documentation extracted from the source code via the Doxygen tool, see http://www.doxygen.org/. You can read the documentation by pointing a browser to doc/html/index.html.
- The objects folder is automatically created and will hold object files during and after compilation.
- The src folder holds all source code files. It has several sub-folders:
  - include which holds files that are used by the user mode processes:
    - scwrapper.h which holds assembly language wrappers for all system calls. The wrappers provide a C function interface to system calls. The wrappers are used by the test case.
    - sysdefines.h which holds type declarations and constant definitions shared by the kernel and the user mode programs.
  - kernel which holds all source code file for the kernel:
    - boot32.s which holds assembly code that puts the CPU into 64-bit mode. This file is not easy to understand.
    - boot64.s which initializes some important CPU registers, initializes and starts the operating system. Like boot32.s, this file is not easy to understand.
    - enter.s which holds the low-level assembly code that handles system calls as well as other pieces of code. Like boot32.s and boot64.s, this file is not easy to understand.
    - kernel.c which is the main kernel source code and holds high-level initialization and system call code. You will also find some of the code that implements the timer queue in kernel.c.
    - mm.h which holds declarations related to the memory sub-system.
    - mm.c which holds the implementation of the memory sub-system.

        - syscall.c which holds the implementations of the system calls.
        - kernel.h which holds various C and C pre-processor definitions and declarations.
        - scheduler.c which holds the implementation of the scheduler.
        - threadqueue.h which holds declarations for an abstract data type called thread queue.
        - threadqueue.c which holds the implementation of an abstract data type called thread queue.
        - link32.ld which is a script used by the ld link editor to generate the 32bit portion of the kernel.
        - link64.ld which is a script used by the ld link editor to generate the 64bit portion of the kernel.
        - relocate.s contains assembler code for calling the entry code of the 64bit portion of the kernel.
    - program_startup_code which holds two files:
        - program_link.ld which is a script used by the ld link editor to generate a ELF executable file.
        - startup.s which holds assembly code that sets up an execution environment that can support the programs' C code and in particular the main function. This is an assembly language file and is not easy to understand.
    - program_0, program_1, program_2 holds the test case programs. Each folder holds a C source code file.
- The Makefile file is a file that the make tool use to orchestrate the compilation of the kernel. The make tool will also, if the compilation is successful, invoke the Bochs emulator.

Take some time to navigate the source tree and read through it. Don't be alarmed, operating system code can be hard to understand. It is not expected of you to understand the assembly code files in detail. However, they are documented so that you can follow their overall structure.

It is well worth to use the documentation generated by Doxygen to navigate the source code.

6 Working on the tasks

Start by finding and reading the changes done to the skeleton. Use for example the diff unix tool to quickly see changes. There are also multiple graphical diff tools. The changes are relatively small and relates to the memory management system.

There are two difficulty levels of this task: the basic level, B4, and the advanced, A4. The advanced level is an extension to the basic level. The advanced level can be selected if you want

a higher grade, see the grading instructions in the “Course_introduction.pdf” document on CampusNet.

Task B4 and A4 are more difficult than previous tasks and will require more effort.

It is very important that you start adding various forms of sanity checks to your code. Implementations will not be deemed correct without an appropriate level of sanity checks.

The kernel has been extended with an array, page_frame_table, that holds information on all page frames in the system. The kernel supports up to 32 megabytes of memory.

6.1 Task B4

In task B4 you are to implement two functions in mm.c: kalloc and kfree. You will not implement any system calls.

kalloc takes three arguments: The size in bytes of the requested memory block, the index of the process to which the memory block will be associated and third integer argument holding flags.

Memory allocation is done on a page basis. The total number of pages found in the system is stored in the memory_pages variable. Each page is 4 kilobytes large and you can use the following formula to calculate how many pages you need to reserve: ((memory_block_size_size+4095)>>12). You must find enough contiguous pages!

Information on each page frame in the system is stored in the page_frame_table array. The page frame on index i in the array starts on address i*4096. Each entry in the page frame table holds three members: the identity of the process that owns the frame, the starting index of the reserved memory block and a flag indicating if the block can be de-allocated by the kfree function. The two latter is to be used by the free system call. The code you write must update all members!

The flag argument of the kalloc function has different bits set to indicate the properties of the requested memory block. You can ignore most of these bits in task B4. However, you must handle one. Bit 2 is set if the memory block is to be allocated so that the block cannot be de-allocated with the kfree call. You can test bit 2 using the following expression: (flags & ALLOCATE_FLAG_KERNEL) . The expression is 0 if the bit is 0.

The kalloc function must return the starting address of the allocated memory block or ERROR if not enough contiguous memory could be found.

The kfree function takes a single argument: the address of a memory block previously allocated via kalloc. The kfree function frees up all page frames corresponding to the memory block and makes them available to subsequent kalloc calls. The function must return ALL_OK if the operation was successful or ERROR if not. An attempt to do a kfree on a memory block that i) is not allocated or ii) is not owned by the current process or iii) is not de-allocatable must not be successful!

6.2 Task A4

Task A4 extends on B4 and you are to extend the kernel to add memory protection. Task A4 is a very difficult task. Before starting the task read sections 5-5.1, 5.3, 5.4, 5.6 and 5.7 in “AMD64 Architecture Programmer's Manual Volume 2: System Programming” that you can find on www.amd.com.

Before starting work on the task you should draw a figure of the page table that is used when the kernel boots. You should also draw a figure of the memory map of the system.

You need to do the following to finish task A4:

- Implement the update_memory_protection function. The function changes the memory protection on a memory region in a page table. The memory region is defined by a starting address and a length in bytes. The protection to be used on the region is defined in a bit-field in the flags argument. If bit 0 is set, the region is to be set so that instructions can be executed from it. If bit 1 is set, the region is to be set so that it can be written. If bit 2 is set, the region is to be set so that it can be read. Finally, if bit 3 is set, the region must only be accessible from kernel mode. The NX bit is supported in bochs and must be used. The GET_PTE_ENTRY_POINTER macro is useful when accessing the page table.
- In the initialize_memory_protection function, you need to modify the page table tree used when booting. The address of that tree is in the kernel_page_table_root variable. Update all of the pages in the leaf tables to only be accessible from kernel mode. Also, set the NX bit on relevant pages.
- Honor the third argument passed to the kalloc function. The argument is a set of bits. If bit 0 is set, the pages corresponding to the memory block must be read-only. If bit 1 is set, the pages contain instructions and so the NX bit must not be set. If bit 2 is set, the pages must only be accessible from kernel mode and like in B4 the block must be allocated so that the block cannot be de-allocated with the kfree call. Hint: Do not update the page table if bit 2 is the only bit set in flags.
- Update the page table trees when memory blocks are allocated and freed.

You can use the files mm.h and mm.c to implement all this functionality. The context switch code will set the root of the page table tree to the address in the page_table_root member in the cpu_private_data structure. You need to change the scheduler to change this structure to reflect the thread that is about to execute.

You do not have to implement any demand paging, swapping or virtual memory. You might, however, want to implement some more advanced test cases in the available user level programs.

7 Test cases

The skeleton code has a built-in test case. This consists of one program that stress test the system calls by allocating memory blocks with randomized lengths and flags. Skeletons are available for two additional programs. You can can use these for any test code you may want to add. Do not forget to start processes for the programs with the createprocess system call if you decide to add your own test cases.

8 Reflection

Before continuing with another task, take time to reflect on the following questions and find answers to them:

1. In the system, memory can only be allocated in sizes of multiples (1, 2, 3, ...) of four kilobytes. What are the possible reasons for that?
2. In your implementation, memory is maintained using a bitfield structure. What would the differences be if a linked list structure is used? What are the advantages and disadvantages of using a linked list structure?

In addition, if you did task A4:

1. Draw a figure of the page table that is used when the kernel boots.
2. Draw a figure of the memory map of the system.
3. The system only implements memory protection. What would be needed to implement:
    1. Paging?
    2. Virtual memory?
    3. Swapping?

    Outline the implementation for each case. All three techniques can be combined. Outline how this can be done.

9 Feedback meeting

When you have completed the task and you are able to answer all questions in section 8, you can ask for a feedback meeting. To do that you need to prepare a mini-poster of at most one A3 page, i.e., two A4 pages, with a high level description of your design and implementation and answers to the questions in section 8. All group members have to be present at the meeting. The purpose of the activity is for you to gain confidence as to what grade you can get on the report.

10 Report

In the midterm report, document and describe your implementation and your experiences. You must also address the questions in section 8.

Please note that in the report, each member in the team must write a passage on their contribution to the work.

Good Luck!