Instructions For Databar Assignment Task B2: Process Management

1 Introduction

In this assignment, you will expand the kernel you investigated in databar assignment task B1. This assignment is independent of databar assignment task B1. However, while you do not have to finish that assignment to start or work on this assignment, you are encouraged to finish B1 before continuing into this task. During the work on the assignment, you will add two new system calls to the kernel. The two calls allow processes to create new processes and terminate themselves respectively. The capabilities of the processes are limited. Only one thread per process is allowed. Only one process can run at any given time. There is no concurrency between processes. Processes are run until termination before other processes are allowed to run.
It is assumed that you are familiar with how to add system calls to the kernel.

2 Goals of the task

The task addresses the following learning objectives:

- You can explain in your own words how multiple processes can share the same CPU.
- Given a schematic overview of the software stack of a real time system, you can label, in the diagram, key operating system parts. You can also explain in your own words and in text the purpose of those parts.
- Given reference literature, you can write down in text and in your own words definitions or explanations of the following concepts: Process; Address space; System call; Daemon; Thread.
- Given a skeleton operating system, you can implement code for process creation and termination.

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. To copy text passages or source code from someone else without clearly and properly citing your source is considered plagiarism. See the study hand book for further details.

4 Setting up the environment

The following assumes that you have already set up a environment following the instructions for task B1 and, most importantly, have the bochs folder. The instructions below assume the folder structure used in the task B1 instructions. You may have to adjust the paths if you used other folder names.

1. Download the source code skeleton archive file for task 2 from CampusNet. This archive file contains all source code files for the kernel you will study and change when

working with this task. The kernel is a vastly extended version of the kernel used in task B1.

2. Start a terminal
3. Find where your task B1 files are located. cd into the the databar_assignments folder. In many cases the following command will suffice but you will likely need to change it.

```
cd 02333/databar_assignments
```

4. The folder should contain a folder called task_B1. Create a new folder called task_B2

```
mkdir task_B2
```

5. cd into the newly created task_B2 directory and unpack the archive you downloaded

```
tar xzvf <path to the skeleton source code archive>
```

Replace the portion within the brackets with the path to the source code archive you just downloaded, eg. ~/Downloads/source_code_skeleton_archive_file_for_task_B2.tar.gz

6. If you have completed databar assignment task B1 then you can continue extending that assignment's syscall.c. Copy the file by issuing the following command:

```
cp ../task_B1/src/syscall.c src/kernel/
```

Congratulations - you now have a build environment for task B2! Now look through the sources located inside the src folder. You can use Eclipse to edit files as you see fit.
Note: you need some additional work to get Eclipse to build and run the sources, so for the time being, we will be using the Linux terminal to build and run. See instructions for task B1.

5 Source code tree

In the task_B2 folder, you will find several folders and files. The overall structure is different from the one you used in task B1. The kernel has been changed to handle multiple processes. The kernel can load executable programs. The programs are organized as position-independent-executables, PIE, in the ELF64 file format, see http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf . Actually, also the kernel is in ELF format.

The databar_assignments folder contains a folder named task_B2 which holds the skeleton code archive.

- 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”, see the instructions for task B1. NB: doxygen needs to be installed if you want to recreate the documentation.
- The doc folder holds Doxygen generated documentation.
- The objects folder is empty but will hold object files during and after compilation.
- Most of the files from the task_B1 src folder has been moved into the src/kernel subfolder which now contains.

- 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. It also contains the low-level assembly code that handles system calls. 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.
- syscall.c which holds the implementations of the system calls.
- kernel.h which holds various C and C pre-processor definitions and declarations.
- 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.
- The program_startup_code folder 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. Get help from the teaching assistants.
- program_0, program_1, program_2 folders holds the test case programs. Each folder holds a C source code file.

Take some time to navigate the source tree and sift through the source code. Don't be alarmed, operating system code can be hard to understand.
The organization of the system has changed. Draw a sketch of the system and, if you have finished task 1, relate this new sketch to the one you draw in task 1.
How is the system booted? Outline and explain the boot process.

6 Implementing the system calls

To complete the task, you will have to implement two system calls. One, *createprocess*, creates new processes and the other, *terminate*, terminates the currently running thread. Study all of the source code before starting the implementation work. Pay special attention to the process and thread data structures and the initialization code in kernel.c. Information on the executable programs is stored in executable_table. The size, in entries, of the table is stored in the executable_table_size variable. What is the purpose of the current_thread variable?
In this task you will receive a lot of guidance and hints. However, from next task and on, you will have to stand on your own.

6.1 createprocess

The createprocess system call takes a parameter in the rdi register . This parameter is an integer which corresponds to an executable program. The program will be be started as a process by the system call. The system call returns in the rax register a value of ALL_OK if a process was successfully created and run. A value of ERROR is returned if any error occurred which prevented a new process to be created or run.

The following pseudo code outline might help you when you implement the system call:

1. Find a free entry in the process table.
2. Load the program into memory. Hint: Investigate the prepare_process function.
3. Set the parent of the new process to be the calling process.
4. Allocate a new thread. Hint: There is a new function in kernel.c called allocate_thread for this purpose.
5. Set the owner, rflags and rip of the allocated thread. Hint: rflags should be 0.
6. Make the new process valid. Hint: How do you detect that a process is running?
7. Assign the return value of the system call to rax in the callers context.
8. Switch to the new thread and let it execute. Hint: What does the current_thread variable mean?

6.2 terminate

The terminate system call terminates the currently running thread. The system call does not take any parameters and will never return to the caller. It does thus not return any values.

The following pseudo code outline might help you when you implement the system call:

1. Terminate the currently running thread. Hint: How do you know if a thread can be allocated?
2. Terminate the calling process if needed. Hint: How can you check if a process is to be terminated?
3. Cleanup the process if it is to be terminated. Hint: Investigate the cleanup_process function. It is empty right now but will in subsequent tasks it will contain some important code.
4. Let the thread of the parent process of the terminated process run. Hint: You can loop over the thread_table to find the right thread. This is of course inefficient. A linked list structure would eliminate this loop at the expense of more complicated code.

7 Test cases

The skeleton code has a built-in test case. This consists of three programs that together build a chain of processes. The programs print short messages indicating how execution has proceeded. Once the two system calls are properly implemented then the test case prints the following messages:

```
Process 0: Trying to start process 1.
Process 1: Trying to start process 2.
Ta da!
Process 1: Process 2 terminated.
Process 0: Process 1 terminated.
```

8 Reflection

Before continuing with another task, take time and make sure you have answers to the following questions:

1. Draw a figure of the organization of the system. If you have finished task 1, relate this new sketch to the one you draw in task 1.
2. How is the system booted? Outline and explain the boot process.

3. How are processes created from executable files?
4. What information is found in executable files such as ELF files? Why is all the information necessary?
5. A running process make use of multiple memory regions. What regions? Why are all the regions necessary?
6. How does a daemon (process) differ from processes studied in this task?

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 all the questions in section 8. You can then ask for a meeting during a databar session. All group members have to be present at the meeting. Your work will be reviewed for correctness and you will be asked for a brief oral presentation of your solution as well as possibly complementary questions. The purpose of the activity is for you to gain confidence as to what grade you can get on the reports.

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, the contributions of each member of the group must be clear.
You write a single report which handed in at the end of the course. In this report, you document all the tasks you complete during the course.

Good Luck!