Instructions For Databar Assignment Task B3 And A3: Scheduling

1 Introduction

In this task, you will expand the kernel you investigated in task 2 and it extends on task 2.

During the work on the task, you will implement a round-robin, and if you work on task A3 scheduler into the kernel.

It is assumed that you are familiar with processes, threads and 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.
- Assume a thread can be in three different states: running, blocked, ready. You can draw a diagram of your own showing the transitions between states and explain in text and in your own words what each transition means.
- Given reference literature, you can write down in text and in your own words definitions or explanations of the following concepts: Process; Address space; Thread; Preemtiveness; Scheduling algorithm; System time.
- Given a skeleton operating system, you can implement a scheduler.
- You can, on your own, find information needed to solve problems.
- You can explain in your own words and in text, how scheduling is done in real-time systems to meet deadlines.

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 3 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_3 for example, in that folder. Enter the created folder and unpack the skeleton. The discussion below assumes you named the newly created folder: task_3.

The syscall.c file has changed a little bit. Merge your changes done in tasks B1 and B2 with the new syscall.c.

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_3 folder, you will find several folders and files. The overall structure is the same as for the kernel in task B2. However, some files have been added. The most important change is the addition of a ready queue data structure, a timer queue, code that maintain a system time and code that implements a thread queue abstract data type. 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. This file has been changed to support interrupts. 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. This source code has been changed and code which supports interrupts has been added. The context switch routines have been extended to handle interrupts. A simple idle thread has also been added. 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.
        - 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 is where you implement 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 Implementing the scheduler

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.

While you should read all changes, in particular note:

- Interrupts are enabled and the timer hardware is supported. This means changes to the context switch code and the initialization code. However, more importantly it means that you will have to change your code! You must make sure that all threads' rflags are set to 0x200 when threads or processes are created. You might have set rflags to 0x0 in databar assignment B2 but now you need to set it to 0x200. A value of 0x0 will disable interrupts while a value of 0x200 will enable interrupts. You will have to change your implementation of the createprocess system call.
- The current_thread variable used in databar assignment B2 is gone and is replaced with a ready queue and a data type holding data private to the CPU. The ready queue is of the thread queue abstract data type, see the threadqueue.h and threadqueue.c files. You must change your implementation of the createprocess and terminate system calls to use the ready queue!
- The kernel now manages time. It does so by setting up the timer hardware to generate interrupts approximately 200 times a second. Each timer interrupt, or clock tick as they are often called, causes the kernel to increment the system wall clock time. There are two new system calls: time and pause.
  - time returns the number of clock ticks since the kernel was started.
  - pause force the calling thread to wait for a number of clock ticks. While the thread is waiting, it is in the blocked state.

  The implementation of the two system calls are located in kernel.c.

6.1 B3: Non-preemptive scheduler

First, you are to implement a non-preemptive round-robin scheduler using the ready queue. The scheduler code is to be added to scheduler.c. Hint: What are the steps in the round-robin scheduling algorithm? Hint: You can run the scheduler at each system call!

There is a variable in the system_call_handler function named schedule. The scheduler must reschedule if schedule == 1. All system calls may set schedule to 1. Read all skaleton code to see how it is used in existing system calls.

6.2 B3: Preemptive scheduler

Now, you are to extend the scheduler and make it preemptive. You can add the portions of the scheduler that you want to have run in the timer interrupt handler to pscheduler.c. The CPU_private structure holds a member called ticks_left_of_time_slice that you can use freely when implementing the scheduler. NB: Task B3 consists of both the preemptive and non-preemptive scheduler.

6.3 A3: Priority scheduler

Optionally, and for a higher grade, you may work on Task A3. In this task, you need to implement a preemptive priority scheduler. You will need to change data structures when working on this task. You may use static, per program, priorities. However, you are encouraged to add extra system calls to allow for dynamic priorities. You can freely use a system call id of 1000 or higher for your own calls.

7 Test cases

The skeleton code has a built-in test case. This consists of three programs that all run in never ending loops printing messages. The test case is considered as passed when all three programs are printing their messages. To test the preemptive scheduler you will need to change file src/program_2/main.c according to the instructions in the same file.

Properly testing a scheduler is very hard and it is recommended that you check your code with a few test cases of your own design. For task A3, you will need 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, question 7 only apply if you have completed A3:

1. What is a scheduler? What does it do and how does it work?
2. Assume a thread can be in three different states: running, blocked, ready. Between the states, there are various transitions. Relate the states and transitions to the code and actions in the system.
3. How does the thread queue data type work?
4. How does the timer queue work?
5. How does preemption influence the design of an operation system? How does it influence the scheduler?
6. How would the scheduler change if the system was to support processes with real-time requirements?
7. If you have completed A3, discuss how the type of scheduling algorithm influence kernel data structures. Also, many advanced scheduling algorithms have an constant effective time complexity, O(1). How can that be achieved?

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 meet-

ing. 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. Deliver separate source code for both the preemptive and non-preemptive implementations as well as task A3 if you have completed it.

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

Good Luck!