The University of Waikato

Department of Computer Science

COMPX203 Computer Systems
Exercise 5 — Multitasking Kernel

Submission Due Date: 5 June 2020

Objectives

The objective of this assignment is to develop your understanding of multitasking through the development of a multitasking kernel for the WRAMP processor.

Due to the complexity of this assignment, it is essential to work in small steps, and keep regular backups of code so that you can refer back to it if (when) things go wrong. If you come across anything that doesn't quite make sense, don't just move on and hope it works out OK — it probably won't!

You are strongly encouraged to start this project early.

It is highly recommended that you read this entire specification before beginning to write any code.

Assessment

This exercise contributes **20%** to your **internal grade**, to be assessed in two parts:

- The correctness of your source code, to be submitted via Moodle **(15%)**
- Your completion of an online quiz about the exercise, also on Moodle **(5%)**

Marks for the implementation are based on the code **working** correctly — in other words, you will not get marks for code that doesn't work. For this reason, it is important to ensure that your code for each question is working correctly before moving on.

Please submit **working** codes **only**, up to the question you successfully coded. Any submitted code that does **not** work as expected will incur one grade penalty per submitted code. For example, you have solved all questions up to Q7 correctly, and you submitted non-working codes for Q8 and Q9 anyways, you will incur two grades penalty.

If you are unsure whether your implementation meets the specifications, ask over email. Note that *mark203* cannot mark this assignment — it's simply too complicated!

This is an individual exercise; while you may discuss the assignment with others in general terms, you must write your own code.

Introduction

The primary goal for this assignment is to implement a multitasking kernel for the WRAMP processor. In its final state, this kernel will run three tasks concurrently:

- **Task 1** reads the value of the switches and writes them to the SSDs in either decimal or hexadecimal
- **Task 2** prints the current kernel uptime to serial port 2 in one of three different formats
- **Task 3** contains two text-based games.

You will write tasks 1 and 2 in your choice of C or assembly, although C is strongly recommended. The object files for task 3 are provided in **Moodle**, as well as the files containing the `main` methods for tasks 1 and 2. You should download all of these files, including the Makefile, into a directory in your home folder and start work there.

All of these tasks **must** comply with the WRAMP ABI. An example project which demonstrates the use of I/O device registers in C can be found in **switches.c**. You can use this as a starting point for tasks 1 and 2.

Makefiles

Because the compiling/assembling/linking process will get rather complicated in this assignment, it is recommended to automate it. This will make your life considerably easier. *Make* is a tool for doing exactly that, by managing which files have changed since the last time you compiled your program. It will then recompile only those which need it.

We want to compile several different S-Record files using the same set of source files. A complicated Makefile can be found in **Moodle**. This file is designed for you to update it as you progress through the assignment, so that when you just type `make` it will assemble and link only the questions you have answered. You can also type, for example, `make kernel_q4.srec` to create just the single file you want.

Its format is fairly simple. At the top there are two declarations which tell you to uncomment particular lines depending on which questions you have completed or are working on, and which games you want to include for your third task. The rest of the file shouldn't be changed, but it could be a useful reference for using *Make* in other papers!

Questions

1. **Parallel I/O Task** (1.5 marks)

   Write a program called **parallel_task.c** (or **parallel_task.s** if you'd rather write assembly code) with the functionality below. Rather than using `main` as your entry-point, you should use `parallel_main`. When you compile this code to test it, you should link the **parallel_entry.c** file that was provided. This file has the `main` function for this standalone program, and simply calls `parallel_main`. The provided Makefile handles linking this entry-point automatically.

   Note that using C is highly recommended for both this task and for the task you will write in Question 2! The C compiler will take care of the stack, function definitions, loops, and a number of other things that you are liable to mistakes with when writing in assembly, so it is much faster and less error-prone to get these tasks done. Don't worry, you'll get plenty of assembly written in the later questions!

   If you do choose to use assembly, **do not** write these two tasks using interrupts. It makes it difficult to ensure your code will continue working once they are running as part of the kernel later.

   This program should:

   - Continually (i.e. in an infinite loop) reads the value of the switches, and writes it to the SSDs as a 4-digit number.
   - Allows the user to change the format of that number:
     - Pressing button 0 (the rightmost button) should cause the program to show numbers in base 16
     - Pressing button 1 (the middle button) should cause the program to show numbers in base 10
     - Pressing button 2 should cause the program to exit (i.e. return gracefully from `parallel_main`)

   Once a base has been selected and the button pressed, the program should continue to show numbers in that base until the other is selected. When the program first starts, numbers should initially be shown in base 16.

2. **Serial I/O Task** (1.5 marks)

   Write a program called **serial_task.c** (or **serial_task.s** if you want to use assembly code) with the functionality below. As with the previous task, you should call the function you use as an entry-point `serial_main`, and link it with **serial_entry.c**.

   This task:

   - Has a global variable (one declared outside a function) called `counter` that will store the system uptime
   - Continually reads `counter` and prints it to **Serial Port 2**

   This task, like the parallel task, will support multiple display formats. The format will be selected by a received character from **Serial Port 2**.

   - '1' will set the format to "\rmm:ss", i.e. minutes and seconds
   - '2' will set the format to "\rssss.ss", i.e. seconds printed to two decimal places
   - '3' will set the format to "\rtttttt", i.e. the number of timer interrupts
   - 'q' will quit the program by returning from `serial_main`

You should print to **Serial Port 2** regardless of whether a character has been received. However, you should always check for new characters so that you can update the format if it needs it. Remember to use the **Serial Port Status Register** to do this. You must ignore any characters received that are not one of those four, and continue to print the last format set. Your program should start with a default format as though it had received a '1' — "\rmm:ss".

Assume that `counter` will be incremented 100 times per second. The serial task is not responsible for updating this variable, so at this stage it will never change.

For debugging purposes, it may be useful to initialise `counter` to some non-zero test value to make sure your printing code is working correctly, and verify that you can change between formats by pressing keys 1 to 3 in the **Serial Port 2** terminal, and quit the program using 'q'. You should return to the WRAMPmon prompt when the program quits.

3. **Interrupt Handler and Serial Task** (1 mark)

   Write a program called **kernel_q3.s** that initialises the timer to generate 100 interrupts per second. You'll need to do this part in assembly code, because C doesn't have access to special registers or the instructions needed to use them.

   Immediately after setting up the interrupts, you should `jal` to `serial_main`. The reason we used the `serial_entry.c` file earlier is so that we don't have to rename the serial task's entry-point now!

   The timer interrupt handler should simply increment `counter` by one. You don't need to declare `counter` here, because we already have a global variable with that name in the serial task.

   If everything is working, you should be able to see the serial task showing the current uptime! Do not move on to the next question until you're sure everything is working correctly — debugging is about to get much more difficult.

   Do not worry when you see a General Protection Fault after quitting the serial_task (and the other tasks when you add them later), it will be fixed in Q8.

4. **"Multitasking" with a Single Task** (3 marks)

   Create a copy of **kernel_q3.s** (called **kernel_q4.s**) and update your Makefile accordingly by removing another #. This question is the hardest of the assignment, and should be completed carefully.

   Add to your code a dispatcher and a single process control block (PCB). You do not need to worry about initialising `$ra` in this question (you will do it in question 8). You can assume that 200 words is sufficient for the stack size, unless you've made extensive use of the stack in your serial_task (such as using recursion). Since your PCB is a known, fixed size, you can allocate exactly enough space for it and no more.

   You should initialise `$cctrl` so that interrupts are **disabled**, but the OIE, KU, OKU and IRQ2 bits are **enabled**. This ensures that you don't get stuck in an interrupt loop before you have completed setup.

   Your dispatcher should save the state of the current task, and then load the state of the next task. Since there is only a single task at this stage, the dispatcher will simply restore the same state.

   - You will need to add a current task pointer, and correctly initialise a PCB. You should give your task a time slice of 2: you will only call the dispatcher on every second timer interrupt.
   - The program counter (`$ear`) field of the PCB should be set to `serial_main`.
   - After initialising everything, you should start the first task by jumping into the part of the dispatcher that loads context — **do not jump directly to the task's entry-point as you did in the previous question.**

   The program should *appear* to run exactly the same as it did in the previous question.

5. **Multitasking with Two Tasks** (2 marks)

   Once the kernel is working correctly with a single task, update a copy of code (called **kernel_q5.s**) and update it to run the parallel I/O task at the same time as the serial I/O task. Remember that the entry-point is `parallel_main`.

   To test that the kernel is operating correctly, try a really long time slice, like 100 timer interrupts instead of 2. This should cause each task to run for one second before being switched for the other task. Make sure to change this back before moving on.

6. **Multitasking with Games** (1.5 marks)

   If you've made it this far, congratulations! Reward yourself by playing some text-based WRAMP games.

   If your kernel is working correctly, it should be possible to link with other pre-existing programs. The object files for two games are provided in **Moodle**.

   - **rocks.O** Entry-point is `rocks_main`
   - **breakout.O** Entry-point is `breakout_main`

   In your kernel (now called **kernel_q6.s**), add one of these games as a third task. These games both play via the **Serial Port 1** terminal.

   If you prefer, you can link both game object files as well as **gameSelect.O** with your kernel and task object files, and set the entry point of the third task's PCB to `gameSelect_main`. This allows you to choose between games while the kernel is running. To change which files are linked with your kernel and task object files, just uncomment one of the lines at the top of the Makefile. Make sure only one line starting with ``GAMES='' is uncommented. Linking fewer files will make your code load into the WRAMP system faster.

7. **Prioritise Gaming** (1.5 marks)

   You may have noticed that the games run quite slowly. Modify the kernel (**kernel_q7.s**) to allow a different time slice for each task. Give the game task 4 timer interrupts per time slice, and the other tasks 1.

   Hint: Which data structure stores variables related to a task?

8. **Allow Tasks to Exit Cleanly** (1.5 marks)

   Modify your kernel (**kernel_q8.s**) so that tasks are able to exit cleanly by returning from their main functions. There are several approaches for doing this, but all involve initialising `$ra` in each PCB to point to a special subroutine in the kernel. This subroutine should effectively remove the current task from the scheduling queue.

   If you are comfortable manipulating linked lists and pointers, you can do this by updating the link pointers of the relevant PCBs to remove the current task from the list.

   Another approach is to add an "enabled" flag to the PCBs, and modify the scheduler code to skip over any task that isn't enabled. The exit subroutine then only has to set this flag to zero. It can then proceed to wait around doing nothing until the next interrupt happens, or it can manually disable interrupts and jump into the scheduler.

   At this stage, do not worry about what happens when the final task exits.

9. **Idle Task** (1.5 marks)

   Modify your code (**kernel_q9.s**) to run an idle task if all other tasks have exited. On a more advanced CPU, the idle task would put the processor into a low power mode. WRAMP does not have this capability, so you can just run an infinite loop instead. You should do something that provides some indication that the idle task is actually running, such as writing "----" or "IdLE" to the SSDs. Be creative!

   This task should have its own PCB, but should not run if any other tasks are enabled.

Optional Extras

If you want to expand the functionality of your kernel, feel free to do so. However, make sure you keep a copy of all source files related to all questions for submission. Here are some ideas:

- Set up a handler for the user interrupt button that reinitialises any tasks that have exited.
- Allow tasks to give up their time slice early. For example, reading from the switches and writing to the SSDs will not need a full time slice, so it could signal to the kernel that it does not need the remainder of its time. The `syscall` instruction could be useful here — it generates a unique type of interrupt.
- Add a terminal-style task that enables you to run commands, similar to the one included in WRAMPmon. For example, `stop 1` might stop task 1, while `uptime` might print the current kernel uptime.

Model Solutions

For reference, source files for the model solution are (without comments):

- **parallel_task.c** 44 lines
- **serial_task.c** 109 lines
- **kernel_q9.s** 226 lines

These numbers might look intimidating, but much of it is repeated code such as the PCB setup and the dispatcher. However, it only takes a single register out of place to cause the whole program to fail! Do not underestimate the complexity of kernel programming.

Submission

You are required to submit **all files** that are required to compile your complete kernel. Please note that contrary to previous assignments, you **should** submit files that we have provided! Namely, your Makefile and the games' object files should be submitted. We still do **not** want any files that you generated from a tool, e.g. *wcc*, *wasm*, or *wlink*.

Remember that each file you submit should follow the naming conventions below. If you have not changed the Makefile, you should be fine! For this assignment, the required files are:

- **Makefile**
- **wramp.h** (Unless you did not use C at all)
- **breakout.O**
- **rocks.O**
- **gameSelect.O**
- **parallel_task.c** (or parallel_task.s if you did not use C)
- **parallel_entry.c**
- **serial_task.c** (or serial_task.s if you did not use C)
- **serial_entry.c**
- **kernel_q3.s**
- **kernel_q4.s**
- **kernel_q5.s**
- **kernel_q6.s**
- **kernel_q7.s**
- **kernel_q8.s**
- **kernel_q9.s**

Note that the file type for **breakout.O**, **rocks.O** and **gameSelect.O** is **uppercase .O**, not **lowercase .o**.

An easy way to submit the correct files is to simply run `make clobber`, then archive everything left in the folder. This will delete any files generated by tools, including all your S-Records. Run `make all` again before archiving to ensure that all your code actually compiles. Don't forget to `make clobber` one last time after testing that compilation worked!

These files must be compressed into a single **tar.gz** archive called **firstName_lastName.tar.gz** (replace *firstName_lastName with your own name*) before being submitted to Moodle. You can create this archive from the terminal using the commands:

```
make clobber
tar -cvzf firstName_lastName.tar.gz *
```

Or, if you want to be certain you're submitting the right files, you can do the following:

```
tar -cvzf firstName_lastName.tar.gz parallel_*.c serial_*.c kernel_q*.s
          *.O wramp.h Makefile
```