The University of Waikato
Department of Computer Science

COMPX203-23A Computer Systems
Multitasking Kernel Assignment

Submission Due Date: 9 June 2023

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 that you can refer back to 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, and to make effective use of the supervised lab sessions and office hours.

Assessment

This exercise contributes **30%** towards your **internal grade.**

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.

If you are unsure whether your implementation meets the specifications, ask during a supervised lab session, lectures, etc. 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 to the SSDs in either decimal or hexadecimal
- **Task 2** prints the current kernel uptime to serial port 2 in three different formats
- **Task 3** contains two text-based games

You will be writing Tasks 1 and 2, while the object files for Task 3 are provided in **/home/compx203/mtk/** on the Tauranga campus computer system and **/courses/compx203/mtk/** on the Hamilton campus computer system.

All of these tasks must comply with the WRAMP ABI. You may write these tasks using WRAMP assembly, but you are strongly encouraged to use C instead. An example project is provided in **/home/compx203/cswitches/** (Tauranga) and **/courses/compx203/cswitches/** (Hamilton) which demonstrates the use of I/O device registers in C. You can use this as a starting point for Tasks 1 and 2.

To compile **cswitches**, run the command `make` in the **cswitches** directory after copying it into your own home directory—this will generate **switches.srec** that you can run on the WRAMP board.

Makefiles (optional)

Because the compiling/assembling/linking process will get rather complicated in this assignment, you may want to automate it. The command line utility '`make`' is a tool for doing exactly that.

The **cswitches** project (above) includes an example Makefile. This file contains a list of instructions for building the program, so instead of using `wcc`/`wasm`/`wlink` directly, we can just run the command `make`, and it will automate the build process for us!

You do not need to use `make` for this assignment, but you are strongly encouraged to do so—it will save a lot of time and limit operational mistakes later on. Simply copy ***Makefile*** from the **cswitches** example, and modify it to suit your needs. It will need to be updated as you work through each question.

Questions

1. **Parallel I/O Task**

   Write a program called **parallel_task.c** (or **parallel_task.s** if you'd rather write assembly code) that:

   - 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 should cause the program to show numbers in base 10.
     - Pressing button 1 should cause the program to show numbers in base 16.
     - Pressing button 2 should cause the program to exit (i.e. return gracefully from `main`)

   Once a base has been selected and the button depressed, 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 10.

2. **Serial I/O Task**

   Write a program called **serial_task.c** (or **serial_task.s** if you'd rather write assembly code) that:

   - Has a global variable called `counter` that will store the system uptime.
   - Continually reads `counter` and prints it to serial port 2.

   The serial task, like the parallel task, will support multiple display formats. The format will be selected by a received character from the vt320/serial port 2 terminal:

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

   You should print to serial port 2 regardless of whether a character has been received. However, always check for new characters so that you can update the format if it needs it. You are free to ignore any characters received that are not one of those four. Your program should start with a default format as though it had received a '1' - "\rssss.ss".

   Assume that `counter` will be incremented 100 times per second. The serial task is not responsible for updating the `counter` 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'.

3. **Interrupt Handler and Serial Task**

   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 direct access to registers or special instructions.

   Immediately after setting up the interrupts, you should `jal` to the serial task's `main` function… which you'll need to rename to something else, e.g. `serial_main`.

   The timer interrupt handler should simply increment `counter` by one. You don't need to declare `counter` in **kernel_q3.s**, because we already have a 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 happy that everything is working correctly—debugging is about to get much more difficult.

4. **"Multitasking" with a Single Task**

   Create a copy of **kernel_q3.s** (called **kernel_q4.s**) and update your Makefile accordingly. 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 can assume that 200 words is sufficient for the stack size, unless you've made extensive use of the stack in your serial code (e.g. you've used recursion).

   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.

   a. You will need to add a current task pointer, and correctly initialise a PCB. You should give your task a time slice of 2, i.e. you will only call the dispatcher on every second timer interrupt.
   b. The program counter (`$ear`) field of the PCB should be set to the main function of the **serial I/O task**. Note: 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. Double-check that your Makefile is indeed compiling/assembling/linking the right files!

5. **Multitasking with Two Tasks**

   Once the kernel is working correctly with a single task, update a copy of the 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.

   To test that the kernel is operating correctly, try a really long time slice, e.g. 100 timer interrupts instead of 2. This should cause each task to run for one second before being switched for the other task.

6. **Multitasking with Games**

   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 respectively in **/home/compx203/mtk/** and **/courses/compx203/mtk/** for Tauranga and Hamilton

   - **breakout.o**: entry point is called `breakout_main`
   - **rocks.o**: entry point is called `rocks_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 (using `wlink`), 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.

7. **Prioritise Gaming**

   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**

   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.

The easier 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.

At this stage, don’t worry about what happens when the final task exits.

9. IDLE Task

Modify your code (**kernel_q9.s**) to run an idle task if all other tasks have exited. On a real CPU, the IDLE task would put the processor into a low power mode. However, WRAMP does not have this capability, so just run an infinite loop instead. You may wish to do something that provides some indication that IDLE is actually running; for example, by writing “- -“ to the seven-segment displays.

The IDLE task should have its own PCB, but should not be scheduled 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 **question 9** for marking/verification purposes.

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 its full time slice, so it could signal to the kernel that it does not need the remainder of its time slice. The *syscall* instruction may be useful here.
- Add a terminal-style task that enables you to run commands. For example, “stop 1” might stop task 1, while “uptime” might print the current kernel uptime.

Model Solution

For reference, source files for the model solution are:

- **parallel_task.c** 67 lines
- **serial_task.c** 139 lines
- **kernel.s**: 306 lines

While these files are not enormous, 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 source files that **you** have written; you do not need to submit any files that we have provided, nor any files that have been generated by a tool, e.g. *wcc*, *wasm* or *wlink*. Each file must follow the naming convention indicated below.

For this assignment, the required files are:

- **parallel_task.c** (or parallel_task.s if you did not use C)
- **serial_task.c** (or serial_task.s if you did not use C)
- **kernel_q3.s**
- **kernel_q4.s**
- **kernel_q5.s**
- **kernel_q6.s**
- **kernel_q7.s**
- **kernel_q8.s**
- **kernel_q9.s**

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

tar -zcf firstName_lastName.tgz parallel_task.c serial_task.c kernel_q*.s