Middle East Technical University
Department of Computer Engineering

CENG 336

Spring 2022

Take Home Exam 4

Regulations

Due: Tuesday, June 20th 2023, 23:55.

Submission: via ODTUClass (NO LATE SUBMISSION)

The purpose of this assignment is to familiarize you with basic features of PICos18, a small but capable real-time operating system. You will develop several different, concurrently executed tasks with support for preemption.

Any clarifications and revisions to the assignment will be posted to ODTUClass.

Hand Out Instructions

- `simulator.zip` : cooking simulator source files.
- `pic_hash_compute.c` : the hash function that you will need for your implementation

One Serving of RS232 Implementation, Coming Up!

In this assignment, your will help a chef cook food for their customers by sending/receiving messages from a cooking simulator through serial connection. Customers pay money if they get their orders in time. Your mission is to collect a certain amount of money in 90 seconds.

The game consists of 2 difficulty levels, easy and hard. In easy mode, customers and ingredients simply arrive at the restaurant at random times as long as there is an empty spot. In hard mode, the simulation session is punctuated with arrivals of special customers that are called food judges. Food judges will not leave the restaurant unless you offer them a slow-cooked dish, which you can only serve by doing a time-consuming task (in our case, this will be solving hashstrings). The food judges will also devilishly block arrival of new customers and ingredients as they wait for their order.

Simulator

The cooking simulator is written in Python and requires Python 3.6 or higher, with `pygame` and `pyserial` libraries to be installed on your system. The simulator is provided to you in the simulator.zip file.

To install `pygame` and `pyserial` you can use the following command:

```
sudo apt-get install python3-pip
pip install pygame pyserial
```

After installing the tools or on Inek machines you can run the simulator by typing the following command:

```
python3 simulator.py
```

Before running the simulator, make sure that the virtual port the cooking simulator connects to is the one paired with the virtual port PICSimlab is connected. Here are the possible pairs (for more information, refer to this link):

PICSimlab is connected to...	Port for simulator should be set as...
`/dev/tnt0`	`/dev/tnt1`
`/dev/tnt2`	`/dev/tnt3`
`/dev/tnt4`	`/dev/tnt5`

You can set the port for the simulator in the JSON file `cookingSimSettings.json`. If it says 'permission denied', then allow user access to device file with `chmod` command.

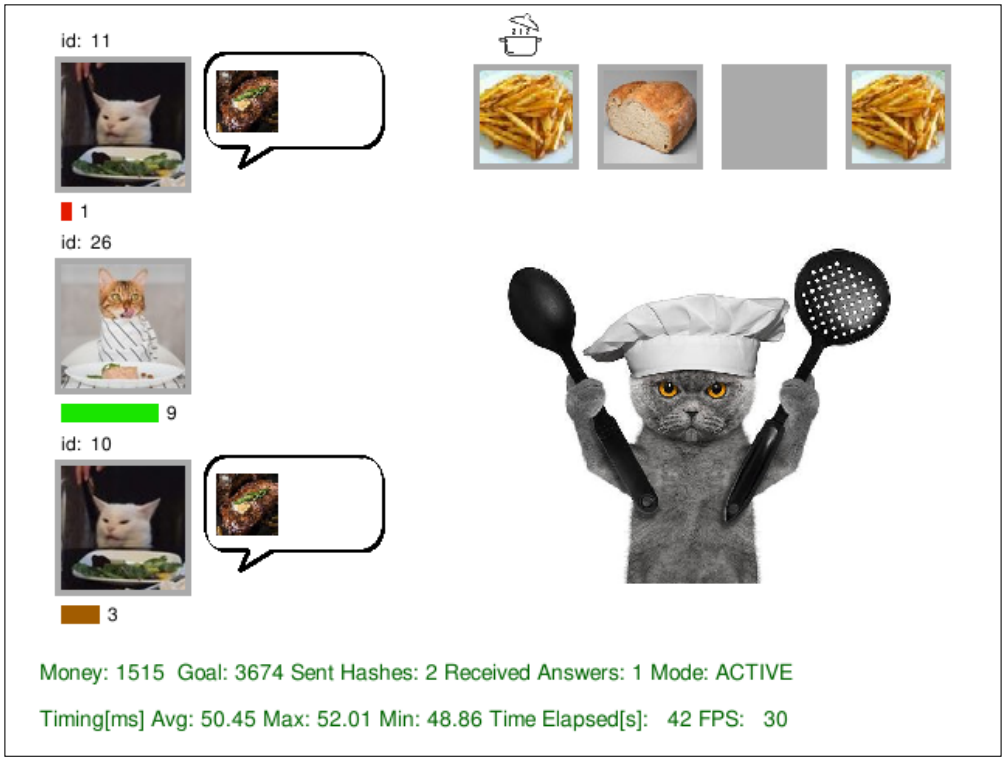

Figure 1: Screenshots of easy and hard mode (at the top and bottom, respectively). The easy mode screenshot features one customer with a dish order containing 2 ingredients. The other customers only ask for baked potatoes, which are in the process of being cooked (indicated by the cooking-pan symbols). In the hard mode, there is food judge in the middle customer slot, waiting for their potato, currently being slow-cooked (indicated by the cooking-pot symbol).

Figure 1 displays 2 screenshots from the simulator. At the bottom, the cooking simulator displays:

- the current money accumulated by serving customers
- the goal money amount we are supposed to reach
- the number of hashstrings sent by the cooking simulator
- the number of answers (solved hashstrings) received by the cooking simulator
- seconds elapsed after the cooking simulator started
- frames per second drawn by the cooking simulator.

On the other regions, we see 4 slots for ingredients and 3 slots for customers:

- **The ingredients** can be Meat, Potato, or Bread. The arrival times of ingredients are random, and they only arrive if there is an empty slot. The only ways of getting rid of ingredients at hand is either cooking/slow-cooking them for an order or *tossing* them.
- **The customers** arrive at random times, and they only arrive if there is an empty slot. Each customer asks for a dish containing 1 or 2 ingredients (the food judge accepts any single ingredient as long as it is slow-cooked). The customers also have a patience meter

that starts with a random value when they enter the restaurant. This meter depletes over time, and lowers the amount of money they pay if they are served their order. The customers leave if their patience meter becomes zero or they are served their order (the patience meter of the food judge stays constant, they will not leave unless they get their dish). All customers have a unique id displayed on their top right for addressing them in the commands.

Operation Modes of The Cooking Simulator

The cooking simulator has three operation modes:

- IDLE: In this mode, the cooking simulator waits without sending commands or responding to the PIC until the user selects one of the difficulty levels, easy or hard, by pressing 'e' or 'h' buttons on the computer's keyboard. After the user selects a difficulty level, the cooking simulator will send a *GO response* over serial port to the PIC and switch to the ACTIVE mode.
- ACTIVE: In this mode, the cooking simulator expects serial commands from the PIC for controlling the restaurant. The cooking simulator sends an immediate response to every command (the details of these responses/commands are described in the section below). It stays in this mode for 90 seconds. **The microcontroller is expected to send serial messages only during this mode.**
- END: When the cooking simulator enters this mode, it sends an *END response* and terminates further events in the restaurant. The serial message frequency statistics are also finalized.

When user presses the 'ESC' button on the computer's keyboard, the simulation program terminates.

Each command/response starts and ends with the `$` and `:` characters. We will refer to `$` and as the delimiter and terminator. All integers within messages are encoded as genuine unsigned little-endian byte-strings, i.e. when we refer to number 8, we do not mean the char `'8'=0x38`, we mean the byte `0x08`. Similarly, when we mean 2758, we mean the byte-string `0x0abc` with the higher byte as `0x0a`, not the char string `"2758"`.

Responses

- ***GO Response:*** When the user selects the difficulty level in the IDLE mode, the simulator sends a GO response over the serial port to the PIC and switches to the ACTIVE mode. This response consists of the following 4 bytes: `$GO:`
- ***END Response:*** When the 90 second period is over, the simulator transitions from the ACTIVE mode to END mode. This is signaled to the PIC with the END Response. This response consists of the following 5 bytes: `$END:`
- ***Status Response:*** This message is a response to Wait, Cook, Hash-reply, and Toss commands, representing the current state of the simulation. It comprises 2 inner status types:
  - *Ingredient status* is a `char` that is either `M`,`B`,`P`,`C`,`S` or `N` corresponding to Meat, Bread, Potato, Cooking, Slow-cooking, None respectively. `N` represents the absence of an

ingredient within that context. `C` and `S` do not appear in customer orders or newly arrived ingredients.

- *Customer status* is represented with 4 bytes. The first byte contains customer id (an integer between 1-255), The following 2 bytes contain:
  - the status of 2 ingredients within the order (the second byte is `N` if the order contains a single ingredient) for a regular customer.
  - the letters `FJ` for a food judge.

  The last byte represents the patience of the customer (an integer between 0-9). The absence of a customer is represented with the following 4 bytes: `0x00NN0x00`

The status response is a 21-byte string following these definitions:

```
$R{customer status x 3}{ingredient status x 4}{money}:
```

where customers are listed from top to bottom, ingredients are listed left to right, starting from slot index 0 and ascending. `money` is a 2-byte unsigned integer, representing the current money. Here are some examples:

- In Figure 1, the easy mode example has the status response:

  ```
  $R0x01MB0x050x02PN0x060x03PN0x03NCBC0x0000:
  ```

- The hard mode example has the status response:

  ```
  $R0x0bMN0x010x1aFJ0x09x0aMN0x03SBNP0x05eb:
  ```

- ***Hash Response:*** is sent as a reply to the Slow-cook command. It contains an 8-byte hashstring to be solved by the PIC. The solved version of the hashstring should be sent back with an Hash-reply command for the slow-cooked dish to be served to the food judge. The command consists of 10 bytes: `$H{8-byte hashstring}:`

Commands

- ***Wait Command:*** This command has no effect. It can be used to make the frequency statistics of the commands comply with the specifications of this text. It consists of 3 bytes: `$W:`
- ***Cook Command:*** This command starts cooking 1 or 2 ingredients for a specific customer. If successful, the state of the ingredients specified in the command turns into "being cooked" (their status letter becomes `C`). This state persists for a random amount of time, then the cooking simulator serves the cooked ingredients to the corresponding customer and adds their tip to the total money collected. If the customer leaves when their food is still in the oven, the food is wasted! The cook command is invalid in the following cases:
  - specifying the same ingredient twice in the command
  - the order of the customer to be served and the ingredients specified do not match
  - one of the specified ingredients is already being cooked/slow-cooked

- the customer to be served was the target of a previous valid cook command
- the customer to be served is not in the restaurant or is a food judge / the specified ingredient slot is empty

The permutation of the ingredients within the command does not matter as long as they correspond to the ingredients within the customer's order. The command consists of the following 6 bytes:

```
$C{customer id}{ingredient 1 index or N}{ingredient 2 index or N}:
```

The absence of an ingredient can be marked with putting letter `N` instead of supplying an index (an integer between 0-3, indices ascend from left to right in the visuals of the cooking simulator and status response).

- ***Slow-cook Command:*** This command starts slow-cooking an ingredient for the food judge, turning the state of the slow-cooked ingredient into "being slow-cooked" (the status letter of the ingredient becomes `S`). The cooking simulator replies this command with a Hash response. To take the slow-cooked food from the oven and serve it, the microcontroller has to send a Hash-reply command. The slow-cook command consists of the following 5 bytes:

  ```
  $S{food judge customer id}{ingredient index}:
  ```

  The command is invalid the following cases:

  - The food judge in the restaurant already has slow-cooking food in the oven.
  - The target customer of the command is not a food judge/is not in the restaurant.
  - The target ingredient is already being cooked/slow-cooked or is absent.

- ***Hash-reply Command:*** This command is used to send the solved version of the hash-string obtained with the Hash response. A correct hash-reply command will serve the slow-cooked dish to the food judge, making them pay a huge tip and leave. The command consists of the following 18 bytes: `$H{16-byte hashstring}:`

- ***Toss Command:*** This command can be used to get rid of ingredients that can not be used in any orders, wasting space. The command is invalid if the ingredient specified is being cooked/slow-cooked or the specified slot is empty. The command consists of the following 4 bytes: `$T{ingredient index}:`

LCD Module

The LCD module will also show the information about the status of the restaurant. At the beginning, the LCD Module should show the following screen and your program should wait for a GO response to start the game.

				C	O	O	K	I	N	G					
						S	I	M							

Table 1: Appearance of LCD at the beginning

M	O	N	E	Y	:					0	0	0	0	0
C	:		0				I	N	G	:	N	N	N	N

Table 2: LCD displaying the initial status

After receiving the GO response, you should start the game by using the commands of the simulator explained above. During the game, you will use LCD to show your current money, the number of customers and the current ingredients. Initially, the LCD should look like Table 2.

You are expected to update the LCD regularly. You can make use of the supplied LCD routines. You are not expected to refresh the LCD for every status response or update **immediately** to make synchronization issues more easier for you. (You may use double buffers with periodic updates on the LCD). However, the updates on the LCD should be visible within **1 second.**

Detailed Specifications

- **First, and most importantly, your program should make use of Real-Time Operating System principles, creating multiple tasks, setting their priorities and synchronizing them. You will be evaluated on your software design as much as the correctness of your implementation.**
- **You MUST properly comment your code, including a descriptive and informative comment at the beginning of your code explaining your design, choice of tasks and their functionality.**
- It is guaranteed that none of the bytes between the delimiters correspond to ascii value of `$` or `:`.
- You should adjust the LCD module for **no blinking and no cursor options.**
- You should send your commands at every **50ms**, with an acceptable level of accuracy $\pm$ **20 ms shifts** (The time difference between two consecutive commands should not be greater than 70ms, and should not be less than 30ms, while having an average of 50ms). This includes the time when you are solving a hash, you should not block/stop sending commands.
- You are allowed to send Wait command as much as you want. You may need such a case while you are deciding which action you want to take next and while another task is running. Note that if you want to meet the specifications for the command frequency, you must continue to send Wait commands to the simulator.
- Two ingredient dishes have distinct ingredients, i.e. there is no 2 x potato dish in the menu.
- For your implementation to be successful, your program should be able to accumulate the goal money within 90 seconds. The goal money amount changes for easy/hard modes and different seeds in the simulation.
- The tip customers pay increase with their current patience and the amount of ingredients their orders contain. The tips are also inversely correlated with the initial amount of their patience.
- In the hard level, you must use the given hash function for solving hashes within a seperate RTOS task. **In other words you will use the given compute_hash function in pic_hash_compute.c file as an RTOS task.**

- You will receive exactly 3 food judges. Each one will be sent at a random time. You are guaranteed to have exactly one hash to solve at any time (i.e. you will not receive a second food judge when one is in the restaurant).
- Your program should be written for 10 MHz oscillator by using PICos18 operating system.
- USART settings should be 9600 bps, 8N1, no parity.

Resources

- Sample program files provided with homework.
- PIC18F4620 Datasheet
- PICos18 Documents
- PIC Development Tool User and Programming Manual
- Recitation Documents
- ODTUClass Discussions

Hand In Instructions

- You should submit your code as a single file named as the4.zip through ODTUClass. This file should include all of your source and header files.
- By using a text file, you should write ID, name and surname of **all group members and group number**.
- **Only one of the group members should submit the code**. Please pay attention to this, since if more than one member make submission, speed of grading process will be negatively affected.

Grading

Total of the homework is 100 points. For grading we will compile and load your program to the development board. Your program will be considered for grading even if it is incomplete. We advise you to implement your program following the checklist given below.

- A brief and detailed explanation of RTOS logic that you implement
  - Duty of each task,
  - The events activate each task,
  - When does their state change and how? (e.g. from suspended to running state),
  - Their priorities (why did you choose such a prioritization)
  - To sum up, running mechanism of your work (by explaining the preemptions)
- Accuracy of command timings
  - Are you able to send a command at every 50 ms, how much are you good at it?

- Proper usage of the LCD module
  - Is it real-time?

Note that you should structure your program modularly with different tasks, appropriate priorities and proper use of synchronization primitives. It is unacceptable to implement this system as a single task in the form of a cyclic executive with interrupts.

Hints

- **CRUCIAL:** Since PICos18 is clearing PIE1, PIE2, RCON, IPR1 and IPR2 registers inside Kernel/kernel.asm file (instructions between lines 265-269) which are executed after your init function inside main.c, your changes on these registers are becoming ineffective. Therefore you have to configure these registers once inside a related task. If you are experiencing an unexpected stop while receiving characters from serial port (due to overrun error making OERR bit 1), this is probably caused from above issue. In that case you have to configure the bits related with receive interrupt (for example a setting like PIE1bits.RCIE = 1;) inside a task.
- In order to check the serial communication between simulator and PIC you can use `sample.hex` file. **This is not an example solution and its commands are not accurate**. Its purpose is only to check the serial communication.
- Microchip's XC series compilers are not compatible with PICos18. For inek machines, MCC18 compiler is already added, so there is nothing you have to do. For your own Linux or Windows machines, you can follow the instructions in 'PicosSettingsInMplabxManual.pdf'.

Cheating

We have zero tolerance policy for cheating. People involved in cheating will be punished according to the university regulations.

Cheating Policy: Students/Groups may discuss the concepts among themselves or with the instructor or the assistants. However, when it comes to doing the actual work, it must be done by the student/group alone. As soon as you start to write your solution or type it, you should work alone. In other words, if you are copying text directly from someone else - whether copying files or typing from someone else's notes or typing while they dictate - then you are cheating (committing plagiarism, to be more exact). This is true regardless of whether the source is a classmate, a former student, a website, a program listing found in the trash, or whatever. Furthermore, plagiarism even on a small part of the program is cheating. Also, starting out with code that you did not write, and modifying it to look like your own is cheating. Aiding someone else's cheating also constitutes cheating. Leaving your program in plain sight or leaving a computer without logging out, thereby leaving your programs open to copying, may constitute cheating depending upon the circumstances. Consequently, you should always take care to prevent others from copying your programs, as it certainly leaves you open to accusations of cheating. We have automated tools to determine cheating. Both parties involved in cheating will be subject to disciplinary action. [Adapted from http://www.seas.upenn.edu/ cis330/main.html]