CSE 237A Winter 2017: Individual Project

Part 2 of 2: Sensor interaction and real-time scheduler

Detailed Instructions

Many embedded systems have been designed to cope with diverse emergency situations. For example, to safely operate vehicles, the system has to monitor information from sensors and interact with actuators. Since it is a very time-sensitive task, the sensing management should be executed in real-time. In our project we will be designing a car monitoring and emergency detection system for which we will use the following sensors and actuators connected to RPi 3:

1. Button: Ignition button to start/stop the engine.
2. Two-color LED: While the engine is on, the ignition LED is also on.
3. Auto-flash LED: Warning indicator - when the engine temperature is higher than a threshold, it will start flashing.
4. RGB LED: Varies actuation when different warning/alert scenarios are seen
5. Active Buzzer: Audible alarm when temperature threshold is exceeded

Your goal is to connect these sensors to RPi 3, develop software that correctly interacts with them and an EDF scheduler that manages the sensing program in an energy efficient manner. This instruction explains the project part 2 in two sections: (i) Sensor interaction program implementation (ii) Energy-efficient EDF scheduler implementation.

Follow-up for common error messages (in particular "file not found" / "command not found")

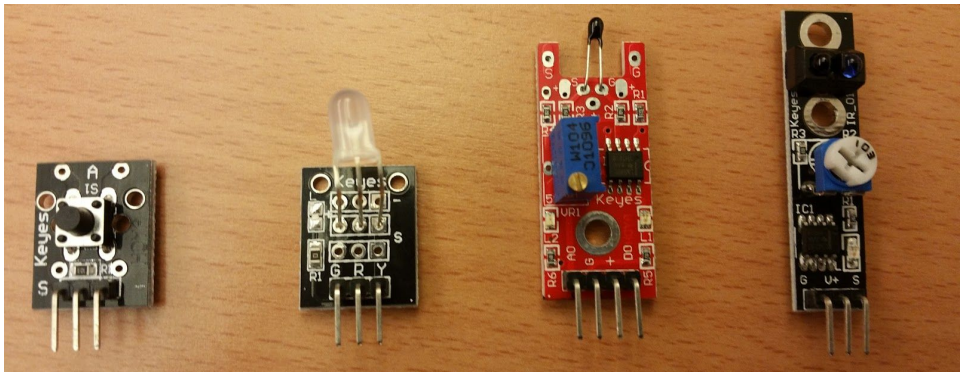
- **Google your error first**
- Are you on the right machine?
- Are you in the right directory?
- Is your PATH variable set correctly? (In ~/.bashrc?)
- Did you need sudo?
- If you're trying to run a binary - is it executable?
- If you're trying to access an external device - is your external drive mounted correctly?
- Do you have the right packages/dependencies? (Especially if it's a *.h file that's missing)

Asking for help on Piazza:

- The best posts are public, include your last commands, the full error message, and a description of your hardware/OS environment. "It does not work" is not a sufficient post for us to help you. You must state clearly what you tried, and what indicates a failure (e.g. exact error message, LED lights, blank screen, etc).
- If someone makes a suggestion that works for you, please report back!
- If you figure it out on your own, please report back!
- "I solved it" or "I made a mistake" is usually not a helpful end to the discussion. Don't be embarrassed by your simple mistakes - we all make them. Please describe the solution and help the next person out!

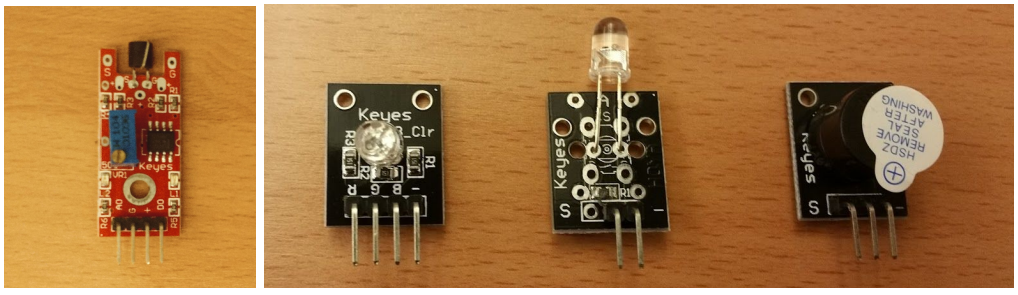
Hardware and software requirements

- Raspberry Pi 3
- Intel ISA, 64 bit machine (x86_64)
- Ubuntu 64bit 14.04.3 LTS desktop edition
    - You may run it as a virtual machine in VirtualBox.
    - If you are familiar with Linux and already have other Linux version (e.g., Ubuntu 12.04 LTS), you might get it to by using additional repos/packages, however, we cannot support that.
- ARM cross-compiler
- Sensor kit: We will use 8 sensors from the sensor kit. The kit may include other sensors which have similar functionalities (e.g., Active buzzer vs. Passive buzzer), but you must use the same sensors specified below:



(Left to right)

- Button Switch (KY-004)
- Two-color LED (KY-011)
- Digital temperature (KY-028)
- Tracking (KY-033)



(Left to right)

- Touch sensor (KY-036)
- RGB LED (KY-016)
- Auto-flash LED (KY-034)
- Active buzzer (KY-012)

Section 1. Sensor interaction program implementation

We will be implementing the sensing and actuation system for running and monitoring a vehicle. You will be able to turn on and off the ignition, and your code will monitor several sensors and actuate warnings/alarms for emergency scenarios as described below:

1. Before pushing the ignition button, the system is in the initial state:
    a. The two-color LED is off.
    b. The RGB LED is blue.
    c. Other actuators (active buzzer and flashing LED) are off.
2. After pushing the ignition button, the system is in a driving state:
    a. The two-color LED is green, and the RGB LED is yellow color {Red, Green, Blue} = {0xff, 0xff, 0x00})
    b. When in a driving state, the system handles three emergency situations as follows:
        i. If the temperature sensor is activated, the auto-flash LED will turn on. When the temperature drops below the threshold, the actuators are turned off.
        ii. If the tracking sensor is activated, the RGB LED is set to the red color. This warning state is only resolved when the vehicle is restarted (button is toggled to off and on).
        iii. If the touch sensor is activated, the RGB LED is set to the purple color ({Red, Green, Blue} = {0xc8, 0x3b, 0xff}) and the buzzer plays a sound. The LED warning state is only resolved when the vehicle is restarted (button is toggled to off and on), but the buzzer turns off if the touch sensing input is removed.
3. If ignition is pushed at any other time, the system reverts back to the initial state, i.e., (1).

1.1 Individual sensor interfacing

The RPi 3 provides multiple General Purpose Input/Output (GPIO) connectors for sensors. You will connect the sensors to the GPIO pins, and implement a user-space program using WiringPi library which supports GPIO communication. It should be installed by default with Raspbian Jessie. If not, get it using these instructions: http://wiringpi.com/download-and-install/

Then, you can check how each GPIO pin number corresponds to the WiringPi pin number as follows:

```
$ gpio readall
```

```
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5V      |     |     |
 |   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 0 | IN   | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | IN   | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 1 | IN   | GPIO. 4 | 4   | 23  |
```

Note that the WiringPi ("wPi" column) numbers are different from the physical GPIO ("Physical" column) pin numbers. Useful information about the pins and the sample code for each sensor are available here: https://tkkrlab.nl/wiki/Arduino_37_sensors

Here is how you can test the sensors:

Step 1: Connect a sensor to proper GPIO pins

a. A sensor has at least 2 pins. For example, an auto-flash LED sensor has two pins, one for ground (0V) and the other for input signal (0/5V). Some sensors (e.g., buzzer) may have 3 pins, one for ground (0V), another one for 5V power, and the other one for an input signal (0/5V). For more detailed guidelines for pins, please refer the included manual in the sensor kit.

b. Be careful to connect the power pins of sensors/RPi 3 (0/5V) correctly. You should turn off the RPi 3 to avoid any unexpected problems while working. Some sensors are sensitive to wrong voltage connections (e.g., 5V supply for a ground pin), and may break if mistakes are made while connected to power.

c. As long as the voltage requirements are met, you may use any of the GPIO pin assignments to connect your other sensors pins. In the next section (1.2), you will be required to use the given pin assignments (specified in `assignment1.h`).

d. The sensors of the Sunfounder kit already have required resistors. For example, a auto-flash led sensor already has their own resistor to properly work.

e. The sensitivity of some sensors (e.g. tracking and touch) can be adjusted with a potentiometer on the sensor module. A red LED should be on when the sensor is powered on. A second LED will turn on when the sensing threshold is triggered. Decrease the triggering threshold by rotating the potentiometer in a clockwise direction.

f. The RPi 3 does not support analog inputs/outputs on their GPIO. To connect analog sensors you would need an ADC (analog-to-digital convertor). We do not cover analog sensing in this project.

Step 2: Implement a sensor test program in the following manner:

a. Initialize sensor interfaces before reading/sending digital signals.

b. Access/control a PIN's inputs and outputs. The following sample code controls an LED.

```
#include <wiringPi.h>
#include <stdio.h>

#define  LEDPin    0

int main(void)
{
  if(wiringPiSetup() == -1){
    //if the wiringPi initialization fails, print error message
    printf("setup wiringPi failed !");
    return 1;
  }

  pinMode(LEDPin, OUTPUT);

  while(1){
    digitalWrite(LEDPin, LOW);   //led off
    printf("led off...\n");
    delay(500);
```

```
    digitalWrite(LEDPin, HIGH);  //led on
    printf("...led on\n");
    delay(500);
  }
  return 0;
}
```

The RPi 3 does not support analog inputs, but you can use Pulse Width Modulated (PWM) digital signals instead, to control the multi-color LEDs (http://wiringpi.com/reference/software-pwm-library/), e.g.:

```
softPwmCreate(PIN_RED, 0, 0xFF);
softPwmWrite(PIN_RED, 0xC8);
```

Step 3: Build the sensor test program

Build option 1: Compile on RPi 3 with WiringPi

Use gcc with the WiringPi library on RPi 3:

```
$ gcc test.c -o test -lwiringPi
```

Build option 2: Cross-compile in Ubuntu with WiringPi

To cross-compile programs with the wiringPi library, you also need a cross-compiled version of WiringPi library. We provide the precompiled library and header files for RPi 3 in the class website.

1. Download the three files of "`wiringPi_armhf`" directory from the class website.
2. Place the downloaded files in "`~/RPdev/wiringPi_armhf`" directory in your VM.
3. Using the downloaded files, you can cross-compile the program that uses wiringPi. For example, the above example can be cross-compiled as follows (**one line, with a space after `-lwiringPi`**):

```
$ arm-linux-gnueabihf-gcc test.c -o test -lwiringPi
-I/home/YOUR_ACCOUNT/RPdev/wiringPi_armhf/ -L/home/YOUR_ACCOUNT/RPdev/wiringPi_armhf/
```

Step 4: Execute the test program(s)

Test each sensor prior to proceeding. Pay particular attention to the HIGH/LOW values described in https://tkkrlab.nl/wiki/Arduino_37_sensors for each sensor, as they dictate how to use it. To execute the program binary, you need the root privilege.

```
$ sudo ./test
```

1.2 Sensor interaction program (with skeleton code)

Implement the program based on a skeleton C code that we provide: http://cseweb.ucsd.edu/classes/wi17/cse237A-a/project/part2/section1/ It has 6 files:

File name	***Description***
`main_section1.c`	The `main()` function of the program. Do not modify it.
`assignment1.h`	The source code you need to implement for sensor interactions. You will modify and SUBMIT them.
`assignment1.c`	
`Makefile.host`	A Makefile for cross compiling (You need to modify a path in the Makefile)

`Makefile.rp`	A Makefile for native compile (i.e., on RPi 3)

If you are compiling on the RPi 3, rename "Makefile.rp" to "Makefile"

```
$ cp Makefile.rp Makefile
```

If you are cross-compiling:

```
$ cp Makefile.host Makefile
```

After make, execute as follows:

```
$ sudo ./main_section1
```

Stop the program when it's executing by pressing "Ctrl+c" in the terminal.

The code controls the 8 sensors using 8 different threads.

1. The `main()` function of "`main_section1.c`" executes following steps:
    a. Initialize wiringPi and call other initialization functions
    b. Create 8 threads corresponded to each sensor
    c. Wait until all threads are finished
    d. Delay 1 milliseconds, and execute again the step (b) in a loop
2. **Pin assignments of the sensors are already specified** in `assignment1.h` as macros. You must use these specified numbers, as this is how we test your code. (e.g., `#define PIN_BUTTON 0`)
3. During initialization, it calls the **`init_shared_variable()`** and **`init_sensor()`** functions. For any other data initialization, you can implement the function bodies in `assignment1.c`. (e.g., Initial INPUT/OUTPUT mode of the WiringPi pins)
4. Upon execution, each created thread will call the function, **`body_SENSORNAME()`**. You need to implement **`body_SENSORNAME()`** functions in `assignment1.c`.
5. The "`SharedVariable`" argument is provided to **`init_shared_variable`**`()` and the 8 **`body_SENSORNAME()`** functions. This variable, a C structure, is shared across all function calls and threads. You can extend this data structure as needed.
6. If you want to terminate the program by finishing the main loop, you can set the value of `SharedVariable->bProgramExit` to 1. See `assignment1.c` for an existing example.

The following are constraints to ensure that your implementation can be graded correctly:

DOs:

1. Complete the following functions in `assignment1.c`. Implement each `body_SENSORNAME()` function to control only the corresponding sensor. For example, `body_button()` function has to access and control only the "button sensor" with `PIN_BUTTON`, not the "temperature sensor".

```
void init_shared_variable(SharedVariable* sv);
void init_sensors(SharedVariable* sv);
void body_button(SharedVariable* sv); // Button sensor
void body_twocolor(SharedVariable* sv); // Two color LED sensor
void body_temp(SharedVariable* sv); // Digital Temperature sensor
void body_track(SharedVariable* sv); // Tracking sensor
void body_touch(SharedVariable* sv); // Touch sensor
```

```
void body_rgbcolor(SharedVariable* sv); // RGB LED sensor
void body_aled(SharedVariable* sv); // Auto-flash LED sensor
void body_buzzer(SharedVariable* sv); // Active buzzer sensor
```

2. Add more variables in the "`SharedVariable`" structure if needed.

DON'Ts:

1. Do not submit the main file (`main_section1.c`). This file should not be modified and your implementation must work correctly using the original file.
2. Do not modify the WiringPi PIN numbers. Use the predefined pin numbers for the compatibility of other RPi 3 (we test your code on the TA's RPi 3).
3. Do not modify the given function declarations. For example, if you modify "`void init_sensors(SharedVariable* sv);`" to "`void init_sensors(SharedVariable* sv, int SOME);`", it will not be correctly executed with the provided original main file.
4. Do not implement any code that involves ANSI locking (e.g. `pthread_mutex`) and delay/sleep functions. This could potentially invalidate the scheduler project in Section 2.

Section 1 In-Person Checkpoint Demo (2/7/17 and 2/8/17)

Demo your sensor interaction program to the TA. This is a sanity check for you, as your sensor interaction program will need to be correctly implemented to finish the project part 2. Bring to the demo:

- Your RPi 3 with your working sensor interaction program
- Your sensors, connected to your RPi 3 and ready to run with your program

Build and run your sensor program with an unmodified version of main_section1.c. The TA will test the various sensing and actuation scenarios.

Section 2. Energy-efficient EDF scheduler implementation

Implement an energy-efficient EDF real-time scheduler to run sensors and actuators along with workloads that have to be executed during sensor access (e.g. sensor ramp-up, preprocessing, etc.). In Project Part 1, you already observed that the CPU frequency can be used to lower the energy cost of workload execution.

2.1 Prepare your environment

1. Verify your RPi 3 settings

The RPi 3 must use the custom kernel from Project Part 1. The RPi 3 must use only a single core.

2. Change default boot option to text console

Since you will implement a real-time scheduler, your system should execute minimal additional processes. Since the window manager and graphical user interface (GUI) run a number of threads, you should limit the RPi 3 to terminal mode only. If you are using GUI as the default boot option, change the boot option to the text console using raspi-config:

- `$ sudo raspi-config`
- Select "Boot Options"
- Select "Console Text console, requiring login (default)"
- Select "Finish" and reboot your system

If you want to start the GUI again:

```
$ startx
```

Then to return to text console, log out.

When executing the user-space scheduler program that you will implement, always use the text console.

2.2 Download base code and infrastructure for scheduler

The Linux system does not by default support any hard real-time scheduling mechanism. Instead, we provide the base code that can schedule threads in a user space program with frequency controls. Implement and submit the two files "`assignment2.h`" and "`assignment2.c`".

Step 1. Download the following files from the project folder of the class website.

File name	***Description***
`main_section2.c`	The `main()` function of the program. Do not modify it.
`assignment1.h / .c`	Placeholder for the same code you implemented in Section 1. **Replace with *your* implementation and SUBMIT this.**
`assignment2.h / .c`	The source code you need to implement the EDF scheduler. **You will modify and SUBMIT them.**

`governor.h / .c`	The code for CPU frequency control Do not modify it.
`scheduler.h / .o`	The code for scheduling threads in a user-space program Do not modify it.
`workload.h`	Header file for the virtual workload Do not modify it.
`Makefile`	A Makefile for cross compile (You need to modify a path in the Makefile)
`Makefile.rp`	A Makefile for native compile (on RPi 3)

Step 2. Download your own workload at this link: http://seelab.ucsd.edu/cse237a_wi17

You will need to input your PID and download "workload.zip". The workload is automatically generated, so every student gets a different workload set. You will be graded based on the quality of your implementation as tested by your own unique workload ***and*** a common unknown workload that we will use to compare everyone's implementations once you submit.

Copy the zip file into the working directory that contains the files downloaded in the step 1 and unzip. For example, if you downloaded the step 1 files in "part2_scheduler" directory, place the zip file in the same directory. You can unzip the zip file as follows:

```
$ ls
assignment1.c  assignment1.h  ...      workload.zip   ....
$ unzip workload.zip
```

After unzipping, there are three directories, "1", "2", and "3". Each directory has has three different workloads as your test set.

```
$ ls 1 2 3
1: deadlines.c workload.o
2: deadlines.c workload.o
3: deadlines.c workload.o
```

We provided the precompiled "scheduler.o" and "workload.o" files as objects without the source.

Step 3. Compile the provided source code

Compile the program using the provided Makefile. If you want to cross-compile, change the path in the Makefile appropriately.

```
$ make        # or on a host: `make CROSS_COMPILE=arm-linux-gnueabihf-`
```

You can see the three compiled binaries: `main_section2_w1, main_section2_w2`, and `main_section2_w3`. They were compiled using the three different workloads. If you want to compile an individual binary, you can use:

```
$ make w1
$ make w2
$ make w3
```

2.3 Description of scheduler base code

Let's take a look at how the provided code works. Using this, you can implement a soft real-time scheduler. In the same way you did in the sensor interaction program implementation, this program creates 8 threads for sensor interaction.

Worker thread and virtual workload

Since the actual interaction for each sensor of the sensor kit is typically executed in a very short time, we assume that there are 8 virtual workloads which precede each sensing interaction function. For example, if a sensor that needs a non-trivial access time, preprocessing, and setup/hold for ADC conversion. The provided base code mimics this concept, so the virtual workload, and subsequently, sensor access, may take hundreds of milliseconds to execute.

Once each thread which will be scheduled is executed, "`workload_SENSORNAME()`" function and "`body_SENSORNAME()`" function are executed one by one. we will call this thread the worker thread.

- The virtual workload is defined as "`workload_SENSORNAME()`" in workload.h and workload.o. The actual sensor interaction (i.e., "`body_SENSORNAME()`" function) will base on your section 1 implementation.
- For the actual implementation of the thread function, check out the "`thread_def`" macro in "`main_section2.c`".

Scheduler thread

After initialization, the main thread of the base code plays a role for the scheduler thread. It calls three functions repeatedly in a loop. (See `main()` function in "`main_section2.c`" file.)

1. `prepare_tasks()` (closed-source in "`scheduler.o`")
    a. This function is responsible for checking the period and creating the 8 worker threads.
        i. The created worker thread is not executed until it is selected by `select_task()` function. (see below)
    b. The deadline of each thread is same as the period.
        i. If all worker threads are executed completely, it waits until a worker thread's execution is triggered again according to its period.
    c. For your convenience, this function also checks and reports missing deadlines (if any).
        i. It only reports the first 100 errors.
2. `select_task()`
    a. You NEED TO IMPLEMENT this function in `assignment2.c`.
    b. It needs to implement a scheduling algorithm, and return a `TaskSelection` structure (defined in "`scheduler.h`").
    c. The return variable, `TaskSelection` structure, has two fields:
        i. `int task`: Scheduled task (0~7: Index of the selected worker thread)
        ii. `int freq`: Applied CPU frequency (0: Low frequency, 1: High frequency)
3. `execute_task()` (closed-source in "`scheduler.o`")

a. Based on the given `TaskSelection` structure, it executes the selected worker thread at the requested frequency.

The selected worker thread is executed as follows:

- The thread selected by `execute_task()` is guaranteed to be executed for 10 milliseconds after scheduled. Once a worker thread is executed for 10 ms, `prepare_task()` is executed again and another (or same) thread can be selected in `select_task()`.
- If the selected thread finishes execution within the allotted 10 milliseconds, it terminates immediately and goes back to the loop of the scheduler thread.

You also need to consider the following facts when you're designing your scheduler:

- Even if a worker thread misses its deadline, it is not terminated immediately but still schedulable. In this situation, the scheduler considers that the deadline is missed, and doesn't create a new worker thread. You may need to schedule such threads with higher priority.
- The closed-source part of `prepare_task()` and `execute_task()` is implemented so that they run in a very short time (less than 1 ms).
- The worker threads may be not created exactly at their periods. They may have a small delay (at most 10 ms due to the allotted time slot). More specifically, the worker thread is only created when the scheduler thread is first encountered in `prepare_task()`.

Program execution with adjustable run time

When executing the program, you can adjust the running time of the experiment by the program parameter. If the time is not specified, the scheduler will be executed for 10 seconds.

```
$ sudo ./main_section2_w1 <time in seconds>
```

Note again that you should test other workloads, using "`./main_section2_w2`" and "`./main_section2_w3`".

2.4 Energy-efficient EDF scheduler implementation

Implement an energy-efficient EDF scheduler in "`assignment2.c`" file. You should implement two functions:

1. **`void learn_workloads(struct shared_variable* sv);`**

This function is called when initializing the program. It will be the location where you will initialize everything you need for your scheduler. More importantly, you can characterize the workload of each worker thread here, such as the execution time of each worker thread.

Parameters

- The shared variable `sv` is the same one of Section 1. You may expand it for your purpose.

What you should know:

- You may use the "`get_current_time_us();`" function defined in `scheduler.h`. It gives the microseconds since the epoch.

- If required, you can reuse the PMU implementation from Project Part 1 here to profile the 8 worker threads with their workloads.
- You may change the CPU frequency using `set_by_min_freq()` and `set_by_max_freq()`;
    - They are defined in "`governor.h/.c`".
    - To minimize the I/O time to change the frequency, `governor.c/.h` uses a custom system call that is part of your custom kernel.
- The deadline (=scheduling period) of each worker thread is defined by `workloadDeadlines` in `deadlines.c` as an 8-element array.
    - The index of array corresponded to each worker thread is defined in "`workload.h`". (i.e., the order of button, two color LED, temperature, tracking, touch, RGB LED, auto-flashing LED, and buzzer)
- If you want to estimate more accurate execution time in actual scheduling, you may need to consider the execution time of the scheduler thread as well.

2. `TaskSelection select_task(struct shared_variable* sv, const int* aliveTasks, long long idleTime);`

This function is called while the scheduler thread is executing. You need to implement the EDF scheduler with a frequency control. Your function eventually has to return a proper worker thread to be executed with a proper frequency using the `TaskSelection` structure. As a sample scheduler, we provide a naive round-robin scheduler in the skeleton code.

Parameters

- The shared variable `sv` is the same one we used in Section 1.
- `aliveTasks` is an 8-integer array which represent which worker threads are currently schedulable. If an element of the array is 1, it means the corresponded thread is still alive, so schedulable. Once a worker thread are finished, the element is set to 0.
    - For example, if BUTTON and BUZZER threads are still alive, then `aliveTasks = {1, 0, 0, 0, 0, 0, 0, 1}`.
- `idleTime` gives a time duration in microsecond. When all tasks are completely scheduled, the scheduler thread (`prepare_task()` function) waits for the next period without any workload. This variable gives the idle time. You will need this variable to compute the energy consumption.

What you should know:

- A rule of thumb is that you have to implement this function in a performance-efficient way. It's a part of the scheduler thread, so any delay of this function consumes the scheduling times repeatedly.
- You may use "`get_scheduler_elapsed_time_us()`" function to get the time passed since each scheduling task began. (i.e., the timer is started from the first `prepare_task()` call.)
- You may know which threads are newly created by using `aliveTasks`.
- You don't need to use the functions that control CPU frequency inside of this function.

    - The frequency is already changed by `execute_task()` function based on the returned `TaskSelection` structure. If you change it again inside this function, it may degrade the scheduler performance.
- Do not use any file IO or interruptible function call here.
    - It may result in unexpected scheduler behavior or system blocking. Please remember again that it's a part of the scheduler which must be executed quickly.
- To avoid IO operations, you must use "`printDBG()`" function instead of "`printf()`" function.
    - The printDBG() function is provided as a part of the base code. (see `scheduler.h`)
    - The usage is same as that of `printf()`. It stores the printed lines into a memory buffer, and the buffer is flushed when the program is finished.
    - The memory buffer size is 1 MB at default. You can adjust the buffer size in `main.c`: `init_deferred_buffer()` function.

2.5 Report on scheduler design and results with actual sensor implementation

- Briefly explain how you implemented the sensor interaction program in Section 1.
- Carefully explain how you designed your energy efficient scheduler.
- Provide a table for the three provided workloads. The table should contain the estimated energy and the result if the scheduler was missing the deadline or not.

Use the table below to calculate the CPU energy values:

CPU Frequency	Power (Total when active)	Power (Idle)
1.2 GHz	600 mW	50 mW
600 MHz	450 mW	50 mW

The active power is the total power consumption while any workload is working. The idle power is the power consumption while no workload is working. The idle time is provided in the third parameter of the `select_task()` function.

Individual Project Submission Instructions (by 23:59:59 PST, 2/16/2017)

Submit the following via TED:

- Submit the four files "`assignment1.c, assignment1.h, assignment2.c,` and `assignment2.h`"
  - Don't submit other source files.
  - Your code must be executed correctly using the predefined wiringPi PIN numbers and the provided other files.
- Report
  - Maximum 3pgs, 12pt Times New Roman font, excluding figures and table.
  - Briefly explain how you implemented the sensor interaction program in Part 1
  - Carefully explain how you designed your energy efficient scheduler
  - Provide a table for the three provided workloads. The table should contain the estimated energy and the result if the scheduler was missing the deadline or not
  - Do not include your source code in the report.
- All files must be zipped up, and the zip file should be titled with <user1_id>_<pid>.proj.part2.zip