CSE 237A Winter 2017: Individual Project

Part 2 of 2: Sensor interaction and energy-efficient scheduler

Detailed Instructions

Many embedded systems have been designed to make diverse user-oriented devices. For example, to design LEDs reacting music, the system has to monitor information from sensors and interact with actuators. Watch these videos:

- https://www.youtube.com/watch?v=LS_oEj7hTtw
- https://www.youtube.com/watch?v=mYM9I4U68MU

Since it is a very time-sensitive task, the sensing management should be executed efficiently. In our project we will be designing an audio monitoring and interacting system for which we will use the following sensors and actuators connected to RPi 3:

1. Button: Button to start/stop the system
2. Auto-flash LED: Indicating the system is running
3. Two audio sensors: Sensing sound amplitude
4. DIP RGB LED: The LED is on when an audio sensor detects high amplitude
5. SMD RGB LED: It is on based on the sensor reading of the other audio sensor

Your goal is to connect these sensors to RPi 3, develop software that correctly interacts with them and a LIST 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 LIST scheduler implementation.

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

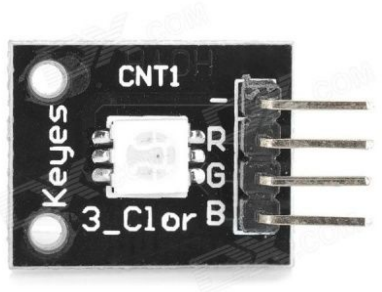
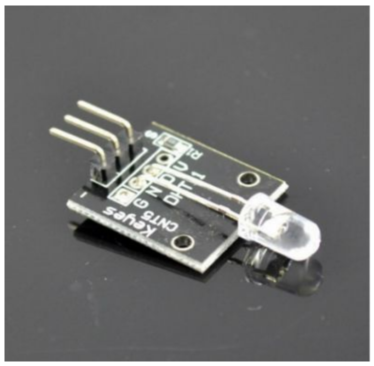
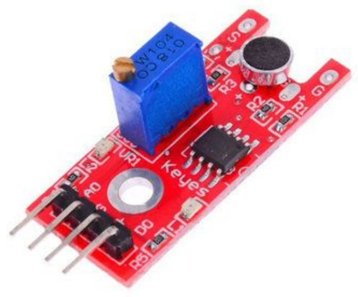
- **Google your error first**
- Are you on the right machine?
- Are you in the right directory?
- Is your PATH variable set correctly? (In ~/.bashrc?)
- Did you need sudo?
- If you're trying to run a binary - is it executable?
- If you're trying to access an external device - is your external drive mounted 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 6 sensors from the sensor kit. The kit may include other sensors which have similar functionalities, but you must use the same sensors specified below:

KY-004 Key Switch Module	KY-009 3-color SMD LED Module (SMD RGB LED)	KY-016 3-color RGB LED Module (DIP RGB LED)
		
KY-034 Flashing LED Module	KY-038 Microphone sound sensor module (Big)	KY-038 Microphone sound sensor module (Small)
		

Section 1. Sensor interaction program implementation

We will be implementing the sensing and actuation system for creating the interactive LED. You will be able to turn on and off the system, and your code will monitor sensors and actuate the LEDs as described below:

1. Once the program starts, the system is in the *running* state:
    a. The auto-flash LED turns on.
    b. The DIP RGB LED is
        i. Blue color {Red, Green, Blue} = {0x00, 0x00, 0xff}) if the small audio sensor gives 0.
        ii. Red color {Red, Green, Blue} = {0xff, 0x00, 0x00}) if the small audio sensor gives 1.
    c. The SMD RGB LED is depending on both the two audio sensors (small and big):

Small Audio Sensor	**Big Audio Sensor**	**Red, Green, Blue**
0	0	(Red) 0xff, 0x00, 0x00
1	0	(Purple) 0xee, 0x00, 0xc8
0	1	(Yellow) 0x80, 0xff, 0x00
1	1	(Cyan) 0x00, 0xff, 0xff

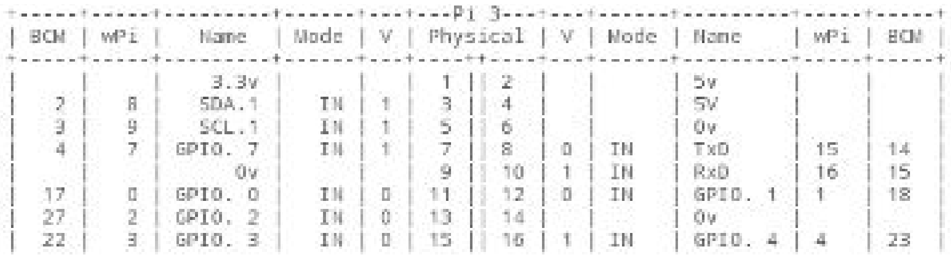
2. If the button is pushed in the *running* state, the system enters in the *pause* state:
    a. The auto-flash LED turns off
    b. Both the DIP and SMD LEDs turn off, i.e., {Red, Green, Blue} = {0x00, 0x00, 0x00}
3. If the button is pushed in the *pause* state, the system reverts back to the *running* 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. In this project, we use WiringPi 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., audio sensors) 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. 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 `section1.h`).
d. Most sensors of the kit already have required resistors. For example, an auto-flash led sensor already has their own resistor to properly work without external resistors.
e. The sensitivity of some sensors (e.g. audio sensors) 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. Increase/decrease the triggering threshold by rotating the potentiometer in a clockwise/counter-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:

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. (This sample code is uploaded with "test.c" in `Section1` folder of the project webpage.

```
#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. So, to control the DIP/SMD RGB LEDs, you need to use Pulse Width Modulated (PWM) digital signals instead. Use the provided functions in the wiringPi library, e.g.:

```
softPwmCreate(PIN_SMD_RED, 0, 0xFF);
softPwmWrite(PIN_SMD_RED, 0xC8); // After calling this, wait for some delay
softPwmStop(PIN_SMD_RED);
```

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 prebuilt library and header files for RPi 3 in the class website.

1. Download the three files of "`wiringPi_prebuilt`" 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 voltage and 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/wi18/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.

`section1.h`	The header file for Section 1. Do not modify it.
`assignment1.c`	The source code you need to implement for sensor interactions. You will modify and SUBMIT them.
`shared_var.h`	
`Makefile.host`	A Makefile for cross compiling (You need to modify a WiringPi 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
```

If you are cross-compiling, make sure that you give the right argument as we did in part 1 (e.g., `` `make CROSS_COMPILE=arm-linux-gnueabihf-` ``)

The original skeleton code will be terminated. You need to modify it to control the 6 sensors on a single thread. ~~using 6 different threads.~~

1. The `main()` function of "`main_section1.c`" executes following steps:
   a. Initialize wiringPi and call `program_init()`function in assignment1.c
   b. Execute `program_body()` function in assignment1.c
   c. Delay 1 milliseconds, and execute again the step (b) in a loop
   d. Call an exit function, `program_exit()`, and terminate the program if the loop is finished
2. **Pin assignments of the sensors are already specified** in **`section1.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 **`program_init()`** function. For any other data initialization, you can implement the function body in `assignment1.c`. (e.g., setting initial INPUT/OUTPUT modes of the WiringPi pins)
4. Upon execution, it calls **`program_body()`** in the loop. You need to implement the functions in `assignment1.c`.
5. The "`SharedVariable`" argument is provided to the **`program_init()`**, **`program_body()`**, and the **`program_exit()`** function. 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.
7. You can also stop the program when it's executing by pressing "Ctrl+c" in the terminal.

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

DOs:

1. Complete the following functions in `assignment1.c`: **`program_init`**`()`, **`program_body()`**, and **`program_exit().`** You can also change the `SharedVariable` structure in `shared_var.h.`
2. Add more variables in the "`SharedVariable`" structure if needed.
2. Only use the following WiringPi functions:
   a. `pinMode, digitalRead, digitalWrite` for the button, audio sensors, and auto-flash LED
   b. `softPwmCreate, softPwmWrite, softPwmStop` for the SMD/DIP LEDs

DON'Ts:

1. Do not submit the main file (`main_section1.c`), the given header file (`section1.h`), and the makefiles. These files 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 RPi3s (we test your code on the TA's RPi 3).
3. Do not modify the given function declarations. For example, if you modify "`void program_init(SharedVariable* sv);`" to "`void program_init(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 (2/6/18)

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 and section1.h. The TA will test the various sensing and actuation scenarios.

Section 2. Energy-efficient LIST scheduler implementation

Implement an energy-efficient LIST scheduler to run sensors and actuators along with given workloads that have to be executed during sensor accesses. We assume that the given workload may represent some preprocessing steps, e.g. sensor ramp-up. The tasks of preprocessing steps have dependencies which need to be managed by a LIST scheduler. 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.

2. Install graphviz

We provide a way to visualize the task graph of the given workloads. This exploits "graphviz" package. Install the package in RPi3:

- `$ sudo apt-get install graphviz`

3. Change default boot option to text console

Since you will implement an energy-efficient 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

We provide the base code that can schedule two threads running on two cores in a user space program with frequency controls. Implement and submit an additional file, "`assignment2.c`".

Step 1. Download the following files from the project folder of the class website, `section2`.
Note that you need to replace `shared_var.h, assignment1.c, and pmu_reader.c` to begin with.

File name	***Description***
`main_section2.c`	The `main()` function of the program. Do not modify it.
`shared_var.h & assignment1.c`	Placeholder for the same code you implemented in Section 1. **Replace with *your* implementation and SUBMIT this.**
`pmu_reader.c`	Placeholder for the same code you implemented in Part 1. **Replace with *your* implementation of part 1** **(It doesn't need to submit. We will test with a TA's version.)**
`assignment2.c`	The source code you need to implement the LIST scheduler. **You will modify and SUBMIT them.**
`section1.h/section2.h`	The header files for each section assignment of part 2. Do not modify it.
`scheduler.h / .o`	The code for scheduler base. Do not modify it.
`workload.h, workload_graph.c`	The code to access the given workloads. Do not modify it.

`cpufreq.h / .c`
`workload_util.h / .c`
`pmu_reader.h`
	The code for CPU frequency control & workload characterization using PMU (same to those provided in part 1)
	Do not modify it.

`Makefile.host`	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_wi18

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 ***both*** 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: workload.o task_graph.pdf
2: workload.o task_graph.pdf
3: workload.o task_graph.pdf
```

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

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-`
```

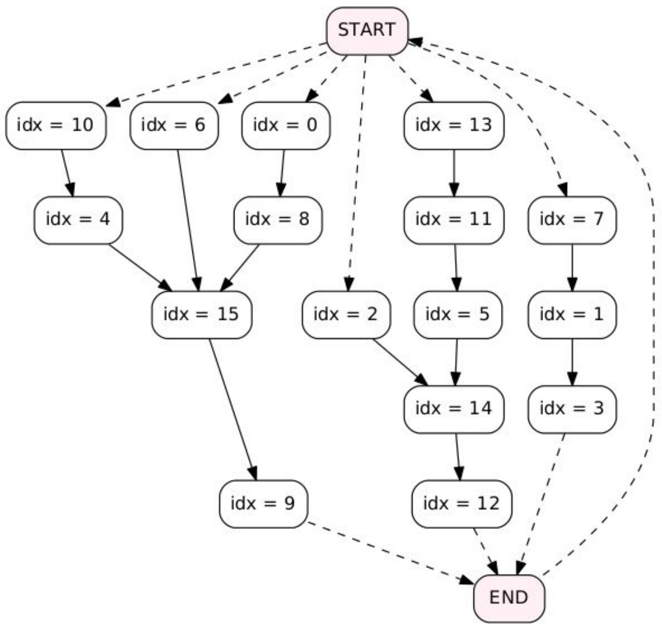
You can see the three compiled binaries: `main_section2_w1`, `main_section2_w2`, and `main_section2_w3`. They were compiled using the three different workloads. If you want to compile an individual binary, you can use:

```
        $ 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 LIST scheduler.

Virtual workload and task graph
There are 16 tasks that you need to run prior to sensing and actuation. They model various tasks that may need to be run in a real system. The provided base code mimics this concept, so the virtual workload, and subsequently, sensor access, may take hundreds of milliseconds to execute. In our system, your `program_body()` function is executed whenever a virtual task completes.
The 16 tasks of a workload set (workload.o) have dependencies with each other. Each of the downloaded three "workload.o" files has different dependencies. You can find the dependencies of the tasks in "task_graph.pdf" file. Here's an example:



In this task graph, the six tasks whose indexes are 0, 2, 6, 7, 10, and 13 can be first executed. However, we will assume that only **two processing cores** can be used in our systems. Thus, you need to choose two of the six tasks. For example, let assume that you choose the two tasks of idx = 0 and 2. Then, once the task of idx = 0 completes, you can select the next task among 6, 7, 8, 10, and 13. Some tasks may have

multiple dependencies. For example, the task 15 will be ready to be executed when all three dependent tasks, 4, 6, and 8, are done. With your implementation, the scheduler has to **run all 16 tasks within 1 second** while satisfying all the dependencies.
In the provided code base, you can access each task using `get_workload(int idx)` function. This function returns a C structure of `WorkloadItem`. They are defined in the provided "workload.h".

```c
typedef struct {
    // An unique index of this workload
    int idx;

    // Three functions of the workload
    WorkloadFunc workload_init;
    WorkloadFunc workload_body;
    WorkloadFunc workload_exit;

    // Next workload
    // If it is NULL_TASK,
    // this workload is one of the last tasks
    // in a scheduling period.
    int successor_idx;
} WorkloadItem;
```

```c
// 1. Get the number of workloads
int get_num_workloads();

// 2. Get a single workload
// for the workload index
const WorkloadItem* get_workload(int idx);
```

- A `WorkloadItem` structure has pointers to three functions, **`workload_init()`**, **`workload_body(),`** and **`workload_exit(),`** in the same way to the workloads characterized in part 1. The function implementations vary over all provided "workload.o" files. Their source code is not provided.
- The task dependencies are defined with the `successor_idx` variable. In the previous example graph, the `successor_idx` variable of task 0 is 8. If it's the last task, (e.g., task 3, 9, and 12 in the previous example,) the `successor_idx` is `NULL_TASK (-1)`.
- In the "assignment2.c" file, we also provide the sample code for i) running the given functions, ii) traversing the task graph, and iii) profiling the tasks using PMU. Feel free to modify/remove/reuse them if needed.

Scheduler base

The `main()` function of Section 2 executes three functions for the provided scheduler base (see main_section2.c.) The scheduler base is implemented in provided "scheduler.h/.o" files. Here is a description how the scheduler base works.

1. `init_scheduler()` *(closed-source)*
    a. This function initializes the provided scheduler base.
    b. This also initializes the given 16 tasks, i.e., calling `workload_init()` on Core 0, in order of the workload index (i.e., from 0 to 15).
2. `schedule()` *(closed-source)*
    a. The main function (in "main_section2.c") executes this function in a loop. For a single `schedule()` function run, you need to schedule the given 16 tasks within 1 second.
    b. This function first creates two threads, called runners, on Core 0 and Core 1.

c. Each thread runner identifies tasks which are currently available in the given task graph, and calls a function that you will implement, **`select_workload()`**. Your function needs to choose a task which will be executed on the target core with a CPU frequency.
d. Based on your decision, it applies the CPU frequency, and executes the body of the selected task, i.e., **`workload_body`**`()`. At the end of the task execution, it also calls the `program_body()` that you implemented in Section 1.
e. Whenever **`workload_body`**`()` of a task is done and there are other tasks to execute, this will repeat from (c).
f. This `schedule()` function will be finished once all 16 tasks are scheduled.

3. `exit_schedule`() *(closed-source)*
    a. This is called when the loop is finished.
    b. This cleans up all 16 tasks by calling **`workload_exit`**`(),` and other relevant data structures.

Program execution with adjustable run time
Load the kernel module "pmuon.ko" that you implemented in part 1, before executing the program.
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 LIST scheduler implementation

Implement an energy-efficient LIST scheduler in "`assignment2.c`" file. Your implementation requires to achieve **the following two goals**.

- **Deadline:** the `schedule()` function running the 16 tasks has to be all executed **within 1 second**.
- **Power efficiency:** Using frequency controls, you need to reduce **average power consumption** as much as possible, while guaranteeing the deadline.

You should implement four functions:

1. **`void learn_workloads(SharedVariable* sv);`**

This function is called when initializing the program. It will be the location where you will plan your strategy to implement the energy-efficient scheduler with the soft deadline of 1 seconds. You can also initialize everything you need for your scheduler. More importantly, you need to characterize the workload of each worker thread here, such as the execution time of each virtual workload.

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 16 tasks with their workloads.
- You may change the CPU frequency using `set_by_min_freq()` and `set_by_max_freq()`.
- This function should be finished within 5 minutes.
- In the provided skeleton code, we provide three sample code which are useful to implement your own `learn_workloads()`.

2. `TaskSelection select_task(SharedVariable* sv, const int core, const int num_workloads, const bool* schedulable_workloads, const bool* finished_workloads);`

This function is called while the scheduler runner threads are executing, i.e., when i) `schedule()` of the scheduler base starts and ii) whenever an assigned task completes on one of the two cores. You need to implement the LIST scheduler with a frequency control. Your function eventually has to return a proper task to be executed with a proper frequency using the `TaskSelection` structure. As a sample scheduler, we provide a naive scheduler that satisfies the task dependency in the skeleton code.

Parameters

- The shared variable `sv` is the same one we used in Section 1. You may expand it for your purpose.
- The integer variable, `core`, is the core index (0 or 1) of the runner thread. Your selected task will be executed on the core.
- The integer variable, `num_workloads`, is the number of tasks. This is alway 16 in this project.
- `schedulable_workloads` is a 16-boolean array that represents which tasks are currently schedulable. If an element of the array is *true*, it means the corresponded task is schedulable, i.e., ready to execute. For the unscheduled tasks, the elements are *false*.
    - For example, if the tasks of idx =3, 9, and 12 are schedulable, then `schedulable_workloads = {false, false, false, true, false, false, false, false, false, true, false, false, true, false, false, false}`.
- `finished_workloads` is a 16-boolean array that represents which tasks are finished.

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.
- For a task, if the corresponding values in `schedulable_workloads` and `finished_workloads` are both *false*, it means that the dependencies of the task are not satisfied yet.
- You don't need to use the functions that control CPU frequency inside of this function.
    - The frequency will be changed inside the scheduler base with the returned `TaskSelection` structure. If you change it again inside this function, it may degrade the performance.
- Do not use any file IO or interruptible function call here, since it's a part of the scheduler which must be executed quickly.

3. **`void start_scheduling(SharedVariable* sv); / void finish_scheduling(SharedVariable* sv);`**

These two functions are called in the main function of “main_section2.c” before scheduling 16 tasks / after scheduling all the tasks. You may use these functions to verify if you met the deadline and to evaluate the power efficiency of your scheduler.

Parameters

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

What you should know:

- The execution time consumed in these functions will be also considered as a part of the scheduling time for the efficiency evaluation. So, make them simple and efficient as well.

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 with average CPU power estimation and if the scheduler made or missed deadlines.
- Do not include your source code in the report.
- Use the table below to calculate the average CPU power consumption:

CPU Frequency	Power (Total)
1.2 GHz	1050 mW
600 MHz	450 mW

Individual Project Submission Instructions (by 23:59:59 PST, 2/15/2017)

Submit the following via TED:

- Submit the three files “`assignment1.c`”, “`assignment2.c`”, and “`shared_var.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 tables
  - Briefly explain how you implemented the sensor interaction program
  - Discuss the design choices for your energy efficient scheduler
  - Provide a table for the three provided workloads with estimated average CPU power consumption and if the scheduler made or missed deadlines
  - Do not include your source code in the report.
- All files must be zipped up, and the zip file should be titled with `<user_id>_<pid>.proj.part2.zip`