|  |
| --- |
| **Project 2: Process Management**  **CSE 330 - Operating Systems** |
| **Instructor: Ming Zhao**  **TAs: Eugene Kuznetsov, Sungho Hong, Yitao (Tom) Chen**  **Part 2** |

1. [Implementing IoT Scheduler](#_xx25qr2aoewt)
2. [Set up Surveillance System](#_ivsobmrth34)
3. [Testing & Evaluating the Scheduler](#_gyr5w6riwivk)
4. [Submission Requirements](#_15ja3j6ulds)

In the second part of this project, you will implement a simple process scheduler named the *IoT* *scheduler*. The objective of this scheduler is to minimize the *response time* for processing events and data collected by the sensors on the device, which is a common task for the IoT devices . We will set up a surveillance system on the Pi, and use it as a case study to develop and evaluate our IoT scheduler. The surveillance system will use a motion sensor to detect motions run the camera to capture pictures when a motion is detected, and use a face detection program to detect possible intrudes in the images.

The major components of Project 2, Part 2 include

1. Implementing the new IoT scheduler that is designed to minimize the response time for processing detected external event and sensed data
2. Setting up the surveillance system using the provided motion sensor, camera, and face detection program
3. Testing the scheduling of the surveillance system programs and evaluating the response time from when a motion is detected to when a face is detected.

Both Part 1 and Part2 of Project 2 are due at the same time. Code needs to be submitted on Canvas by **March 27th 11:59:59pm** and the demo needs to be presented by **March 27th.**

Project #2-2 Linux source code is available in the following link:

|  |
| --- |
| git clone <https://github.com/ychen404/linux.git> -b proj-2-2 |

We have prepared a Pi image pre-installed with the essentials, such as the kernel with the basic template of the scheduler and the surveillance application, that you can use to jump start the project

https://drive.google.com/open?id=1Yg3GS10ntkEiCdGnLMGjIRY0AmR4pIj2

You can flash the ISO file onto your microSD card using Etcher as explained previously in Project #1.

### **Implementing IoT Scheduler**

**Add a new scheduler to kernel**

*You can omit this step if you use the provided Pi image. Jump to the next step:* ***Complete the IoT Scheduler*** *on Page 6.*

Here is the list of modified files for adding a new scheduler.

|  |
| --- |
| // in linux/kernel/sched/sched.h, it is required to declare a real-time priority array, and the essential structures of IoT scheduler.  // declare a priority array and the bitmap that are used for IoT scheduler  // the bitmap is two dimensional, and the two dimensions represent the number of CPU and the range of the real-time priority  struct iot\_prio\_array {  DECLARE\_BITMAP(bitmap, MAX\_IOT\_PRIO+1); // declare the bitmap that maps number of cpu ids and priority array  struct list\_head queue[MAX\_IOT\_PRIO]; // create a linked list for every priority array  }  // declare the structure of iot\_rq that contains the meta-data of the run-queue  struct iot\_rq {  struct iot\_prio\_array active; // used for checking the currently used priority array  unsigned int iot\_nr\_running; // check the total number of running processes in run queue  int iot\_queued; // a flag that is used to check whether the process is queued in the run queue.  };  // the function used in the source code of process scheduler to identify the type of policy of the process  static inline int rt\_policy(int policy)  {  // if the return value is one of these scheduler policies, then return true  return policy == SCHED\_FIFO || policy == SCHED\_RR || policy == SCHED\_IOT;  }  // a structure of the run-queue of a single CPU  struct rq {  ...  struct iot\_rq iot\_rq; // a pointer for accessing the struct iot\_rq which serves as a meta-data for the run-queue  }  // a declaration of the bundle of interfaces of IoT scheduler that is called from inside and outside of the scope of process scheduler  extern const struct sched\_class iot\_sched\_class;  // a declaration of a function that is used to initialize the bundle of interfaces of IoT scheduler.  extern void init\_sched\_iot\_class(void);  // a declaration of a function that is used to initialize the struct iot\_rq  extern void init\_iot\_rq(struct iot\_rq \*iot\_rq);  // define the maximum priority number of IoT scheduler  #define MAX\_IOT\_PRIO 100 |

|  |
| --- |
| // In linux/kernel/sched/Makefile. It is required to add the path of the new IoT scheduler.  // added the compilation path for iot.c (the location where the abstract interfaces of IoT scheduler are declared)  obj-y += idle\_task.o fair.o rt.o deadline.o iot.o |

|  |
| --- |
| // In linux/kernel/sched/iot.c it is required to add the source code so that the abstract interface of IoT scheduler will work.  // refer to the comments in written in the source code |

|  |
| --- |
| // In linux/kernel/sched/core.c, it is required to use the functions of IoT scheduler in the process scheduler  // a function that is used to change the scheduling policy and/or RT priority of a process.  static void \_\_setscheduler(struct rq \*rq, struct task\_struct \*p,  const struct sched\_attr \*attr, bool keep\_boost)  {  ….  // use the abstract interface of the IoT scheduler if the policy of the process is equal to seven  if(attr->sched\_policy == SCHED\_IOT){  p->sched\_class = &iot\_sched\_class;  }  ….  }  // add the IoT scheduler in this syscall which returns the minimum real-time priority that can be used by a given scheduling class.  SYSCALL\_DEFINE1(sched\_get\_priority\_min, int, policy)  {  int ret = -EINVAL;  switch (policy) {  ….  case SCHED\_IOT:  ….  }  return ret;  }  // add the IoT scheduler in this syscall which returns the maximum real-time priority that can be used by a given scheduling class.  SYSCALL\_DEFINE1(sched\_get\_priority\_max, int, policy)  {  int ret = -EINVAL;  switch (policy) {  ….  case SCHED\_IOT:  ….  }  return ret;  }  // call the function that will initialize the bundle of abstract interfaces of IoT scheduler  void \_\_init sched\_init\_smp(void){  ...  init\_sched\_iot\_class();  ...  }  // initialize the structure of iot\_rq and save the address in the structure of run-queue  void \_\_init sched\_init(void)  {  …  // iterate the run queue of possible cpus  for\_each\_possible\_cpu(i) {  struct rq \*rq;  rq = cpu\_rq(i);  raw\_spin\_lock\_init(&rq->lock);  rq->nr\_running = 0;  rq->calc\_load\_active = 0;  rq->calc\_load\_update = jiffies + LOAD\_FREQ;  init\_cfs\_rq(&rq->cfs);  init\_rt\_rq(&rq->rt);  init\_dl\_rq(&rq->dl);  // initialize the structure of run queue used for IoT scheduler  // after allocating the structure of iot\_rq, each cpu’s run queue will save the address of this structure.  init\_iot\_rq(&rq->iot\_rq)  }  ....  } |

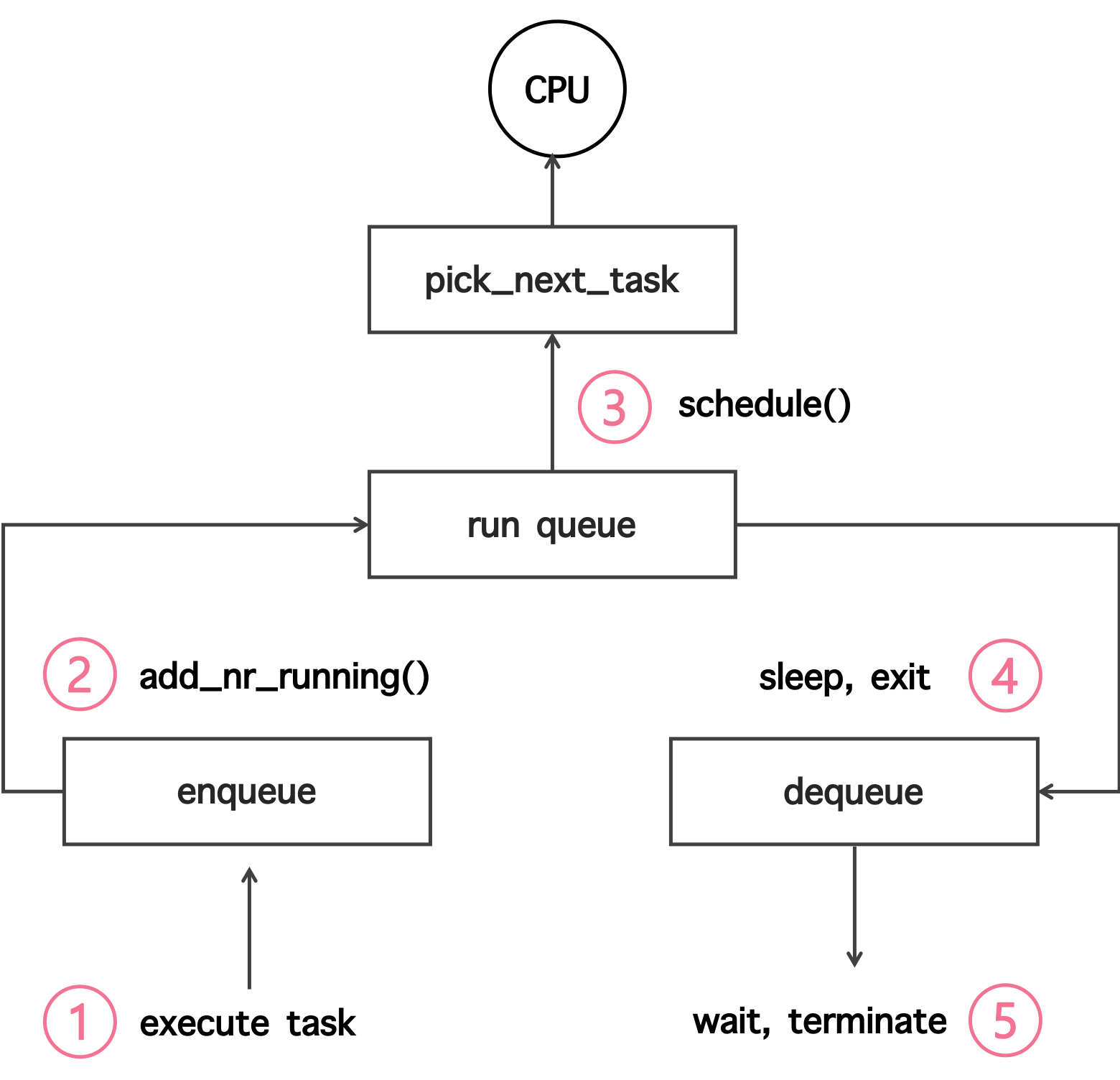
|  |
| --- |
| // In linux/include/uapi/linux/sched.h, it is required to define the number of the IoT scheduler.  …  /\*  \* Scheduling policies  \*/  #define SCHED\_NORMAL 0  #define SCHED\_FIFO 1  #define SCHED\_RR 2  #define SCHED\_BATCH 3  /\* SCHED\_ISO: reserved but not implemented yet \*/  #define SCHED\_IDLE 5  #define SCHED\_DEADLINE 6  // add the policy of the IoT scheduler as number seven  #define SCHED\_IOT 7  …. |

|  |
| --- |
| // In linux/include/linux/sched.h, it is required to declare the essential structs of IoT scheduler and add those pointers to the task\_struct  // create the structure of IoT entity in the structure of the process  struct task\_struct {  ...  const struct sched\_class \*sched\_class;  struct sched\_entity se;  struct sched\_rt\_entity rt;  struct sched\_iot\_entity iot\_t; // allocate the structure of entity of IoT scheduler when the process is allocated  ...  }  // declare the structure of IoT entity which contains the variables to manage the tasks in the process scheduler  struct sched\_iot\_entity {  struct list\_head run\_list; // list\_head used for the linked-list of priority array in IoT scheduler  // setting variables that are usable in algorithms of scheduling policies  unsigned long timeout; // used to check whether the process used all of it's allocated time  unsigned long watchdog\_stamp; // used for debugging purposes  unsigned int time\_slice; // attribute used when the scheduler is using round-robin algorithm.  // setting the attributes that are used to manage the tasks in the process scheduler  unsigned short on\_rq; // a flag that indicates that the task is currently queued in the run queue  unsigned short on\_list; // a flag that indicates that the task is currently in the priority array  struct sched\_iot\_entity \*back; // a pointer connected to the next available task  struct iot\_rq \*rt\_rq; // a pointer directing to the struct iot\_rq  struct iot\_rq \*my\_q; // a pointer directing to the current iot\_rq that the task is queued to  }; |

**Complete the IoT Scheduler**

Your implementation will be focusing on completing the functions of the loT scheduler located at *kernel/sched/iot.c*. The source file, iot.c, includes several functions and you need to complete. Go over the comments in the file carefully and follow the suggestions to complete the necessary functions.

The diagram below illustrates the flow of how a task gets scheduled by the kernel:

****

1. When the new task starts running, the new task goes through the enqueue function of a specific scheduler according to the task’s scheduler policy.
2. The new task is added to the run queue (struct rq).
3. When it is time to schedule a process on the CPU, the kernel invokes the schedule() function. The schedule() function invokes the pick\_next\_task to choose the next task that the scheduler will use.
4. The task is dequeued when the task status is set to sleep or exit.
5. The task is terminated when the task is finished, and the task waits when it is set to sleep.

Accordingly, the primary task is to implement the enqueue, dequeue, and pick\_next\_task. Your scheduler can schedule the queued tasks first come first serve (FCFS). Since the IoT scheduler uses the real-time priority, the process scheduler will prioritize handling the tasks with IoT scheduler policy without any sophisticated algorithm.

A better scheduler needs to support preemption. Without preemption, a process run by the IoT scheduler may starve the others if it does not finish or yield the CPU.

Additionally, load-balancing allows the scheduler to utilize multiple available processors by migrating tasks to idle processors and maximizing the system performance.

This assignment does not require you to implement preemption and load balancing, but we encourage you to try them, and you will get bonus points based on how much you complete.

### 

### **Setting up the Surveillance System**

The surveillance system consists of three applications, 1) a program that uses a motion sensor to detect a motion, 2) a program that uses the camera to capture and save the image, and 3) an application that uses a face-detection algorithm to detect a person in the image . The surveillance system works as follows: 1) the motion detection application runs continuously waiting for a motion; 2) once a motion is detected, it invokes the camera application to capture an image and then pass the image to the face detection application; 3) the face detection application will output the number of faces detected in the image. Follow the guide to set up all the components of the system.

**Setup the Camera Module**

You first need to connect the camera module by inserting the cable into your Pi. The connector is located between the ethernet connector and the HDMI connector. Once connected, the silver stripes on the ribbon cable should be facing the HDMI connector.

Access your Raspberry Pi with SSH, and enable camera.

|  |
| --- |
| sudo raspi-config  // In the pop-up window, select **interfaces** ➝ **Camera** ➝ **Enable**  // You will need to reboot your system after the modification |

You can test your camera by running the following command, which will capture an image and store in your current path.

|  |
| --- |
| sudo raspistill -o test.jpg // capture a still image from your camera |

**Setup the Motion Sensor**

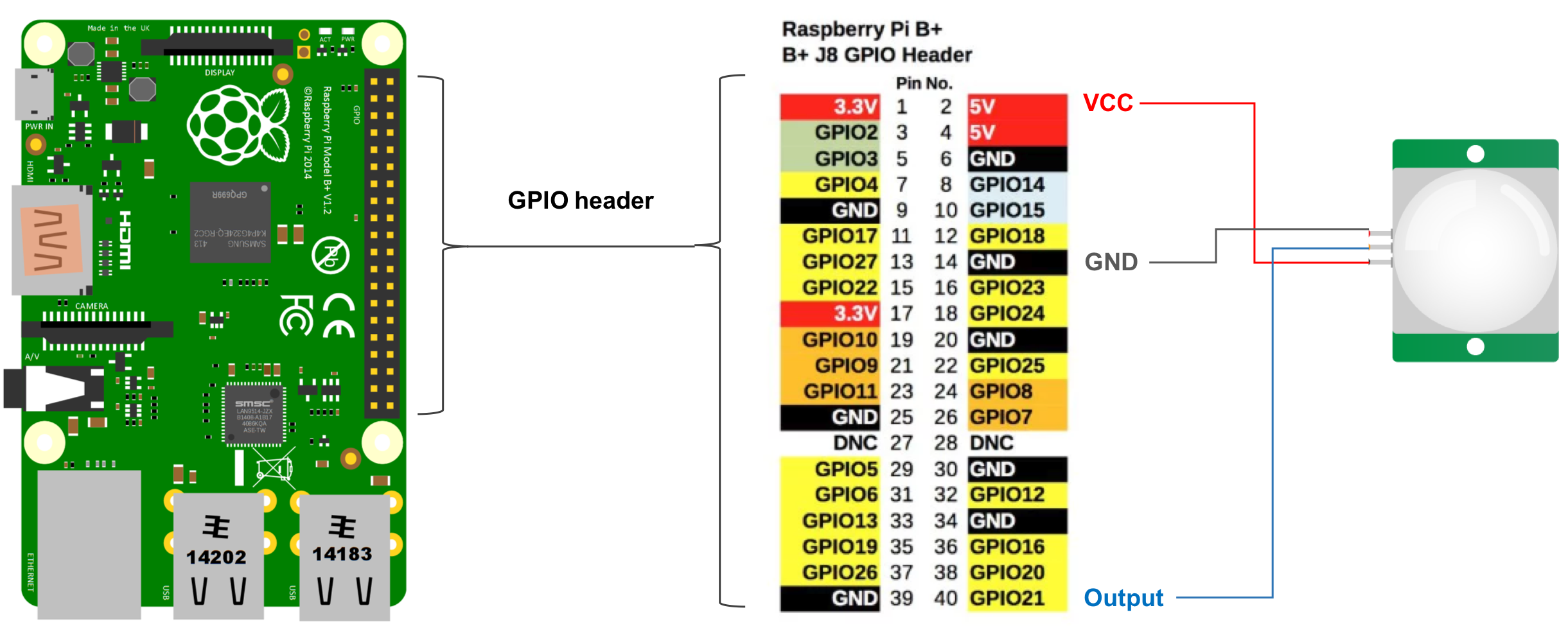
The motion sensor communicates with the Pi through GPIOs.

There are three pins, *VCC*, *GND*, and *Output*. The VCC and the GND provide the power supply and the output pin will be used to read the results. Detailed connections are as follows:

1. *VCC* is connected to 5V (Pin No. 2)
2. *GND* is connected to GND (Pin No. 14)
3. *Output* is connected to GPIO 21 (Pin No. 40)

*The VCC, GND and Output Pin of the motion sensor device must follow this order like below.*

*Note that the GPIO Pins connected with the device is optional.*



Install the Python libraries for using GPIO pins

|  |
| --- |
| sudo apt-get update  sudo apt-get install build-essential python-dev python-smbus python-pip  sudo pip install RPi.GPIO |

Then, use the following python example to check whether the motion sensor is working.

You should see that the print log will change from “no motion” to “motion detected” once the sensor detects a movement. Note that the response time is quite fast (almost instantaneously), but the motion sensor remains activated for about 2 seconds before changing back to no motion state.

The test program can be found in this link:

https://github.com/SungHoHong2/linux/blob/master/motion\_test.py

**Install the Face Detection Program**

*You can omit this step if you use the provided Pi image.*

Use the following command to install OpenCV and the dependencies which are required by the camera program

|  |
| --- |
| sudo apt-get install build-essential;  sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev \ libswscale-dev;  sudo wget \ http://liquidtelecom.dl.sourceforge.net/project/opencvlibrary/opencv-unix/3.1.0/opencv-3.1.0.zip;  sudo unzip opencv-3.1.0.zip;  cd opencv-3.1.0;  sudo mkdir build;  cd build;  cmake -D CMAKE\_BUILD\_TYPE=Release -D ENABLE\_PRECOMPILED\_HEADERS=OFF -D CMAKE\_INSTALL\_PREFIX=/usr/local ..;  make -j4;  sudo make install; |

Add the following line to the file located at /etc/ld.so.conf.d/opencv.conf

|  |
| --- |
| /usr/local/lib |

Then, in your terminal, type the following command to update the linker and export environment variables.

|  |
| --- |
| sudo ldconfig  PKG\_CONFIG\_PATH=$PKG\_CONFIG\_PATH:/usr/local/lib/pkgconfig  export PKG\_CONFIG\_PATH |

You can verify your OpenCV library configuration with the following command.

|  |
| --- |
| pi@raspberrypi:~ $ pkg-config opencv --cflags --libs |

You should be able to see the below output from your terminal

|  |
| --- |
| -I/usr/local/include/opencv -I/usr/local/include -L/usr/local/lib -lopencv\_shape -lopencv\_stitching -lopencv\_objdetect -lopencv\_superres -lopencv\_videostab -lopencv\_calib3d -lopencv\_features2d -lopencv\_highgui -lopencv\_videoio -lopencv\_imgcodecs -lopencv\_video -lopencv\_photo -lopencv\_ml -lopencv\_imgproc -lopencv\_flann -lopencv\_core |

Now you can compile and run the face detection program: https://github.com/ychen404/linux/tree/proj-2-2/facedetect

|  |
| --- |
| cd /home/pi/facedetect  g++ -o facedetect facedetect.cpp `pkg-config opencv --cflags --libs`  ./facedetect /data/lena.jpg |

You should have the following output:

|  |
| --- |
| ./facedetect data/lena.jpg  Detecting face(s) in ../data/lena.jpg  detection time = 219.633 ms  1 face detected! |

### 

### **Testing & Evaluating the Scheduler**

You can use the *chrt* command-line to change the scheduling parameters of a process, and observe how it affects the scheduling decision that a process scheduler makes. We have modified *chrt* to support our IoT scheduler. You can find it on the provided ISO, or download and compile from the source code:

https://github.com/ychen404/linux/blob/proj-2-2/facedetect/chrt.c

|  |
| --- |
| // compile chrt, replace with /usr/bin/chrt  // example of running the application with FIFO scheduler.  chrt -f 98 ./application  // first argument -f : FIFO , -r : Round Robin, -o : CFS, -b : BATCH, -i: IOT  // second argument: the level of priority.  // able to choose the priority level described below  // SCHED\_OTHER min/max priority: 0/0  // SCHED\_FIFO min/max priority: 1/99  // SCHED\_RR min/max priority: 1/99  // SCHED\_BATCH min/max priority: 0/0  // SCHED\_IOT min/max priority: 1/99 |

The Processes scheduled by your IoT scheduler should not be affected by processes scheduled by the other schedulers (the default scheduler is CFS) no matter how busy the system is. To test this, you should create multiple processes that run in a busy while-loop with a CFS scheduler. The surveillance system should have similar response time as when it runs without the busy background processes.

|  |
| --- |
| // i.e /home/sungho/facedetect  // TAs will generate multiple tasks of busy-loop program  ./run\_tasks.sh 100 // run 100 busy processes with CFS scheduler  // run the surveillance system with IoT scheduler  sudo chrt -i 98 python surveillance.py  [motion-sensor] No intruder  [motion-sensor] No intruder  [motion-sensor] Intruder detected  [camera] camera has taken the picture  detection time = 877.001 ms  1 face detected! |

### 

### 

### **Submission Requirements**

**Source code submission on Canvas**

Submit a zip file that contains the source-code of the kernel module and the syscall by **March 27th 11:59:59pm**. Only one submission each team.

The zip file should include the following contents:

* A plain text *README* file that lists the full names of all members of your team;
* A folder named “*project2\_1*” that includes the source code of your new system call and your user-space program.
* A folder named “*project2\_2*” that includes only your IoT scheduler code, *iot.c*.

**Notes:**

* Do not submit any other source code
* Do not submit any binary
* Put everything in a single zip file named by your full names
* Late submissions will not be accepted.

Failure to follow the above instructions will cause penalty to your grade.

**Live demo to the TAs**

In the recitation sessions on **March 25th**, **March 26th**, or **March 27th**, you need to provide a live demo of project to the TAs, using your Pi and laptop (connected through serial port). Make sure to bring all the equipment that you need and be prepared for the demo. Each team will have only five minutes to complete the demo.

TAs will check the response time of the surveillance system running with IoT scheduler in both non-busy and busy environment.TAs will bring a script that will create multiple running processes that will create a CPU busy environment. The response time of the surveillance system should not degrade in such an environment. Here are the expected results from the surveillance system running with the IoT scheduler as compared to running with the default CFS scheduler. The performance below indicates that the response time does not increase as high as the CFS scheduler in a CPU busy environment.

|  |
| --- |
| // IoT scheduler in non-busy environment  Intruder detected  Detecting face(s) in frame.jpg  detection time = 281.652 ms  1 face detected! |

|  |
| --- |
| // CFS scheduler in non-busy environment  Intruder detected  Detecting face(s) in frame.jpg  detection time = 268.819 ms  1 face detected! |

|  |
| --- |
| // IoT scheduler in busy environment  Intruder detected  Detecting face(s) in frame.jpg  detection time = 311.422 ms  1 face detected! |

|  |
| --- |
| // CFS scheduler in busy environment  Intruder detected  Detecting face(s) in frame.jpg  detection time = 5438.24 ms  1 face detected! |