Project 2: Timer & Interrupt

ENEE 447: Operating Systems — Spring 2017
Assigned Date: Wednesday, Feb 22.
Due Date: **Sunday, March 5th. @11:59 p.m.**

Purpose

In this project, we will learn about timer and interrupts. In particular, we will learn what a system interrupt is? How to handler interrupt and how to write an interrupt handler. The second part of the project will focus on implementing two scheduler working modes: 1. real-time scheduling, 2. Non-real time (periodically) scheduling.

Introduction

2.1 Timer

There are two types of timers: physical timer and virtual timer.

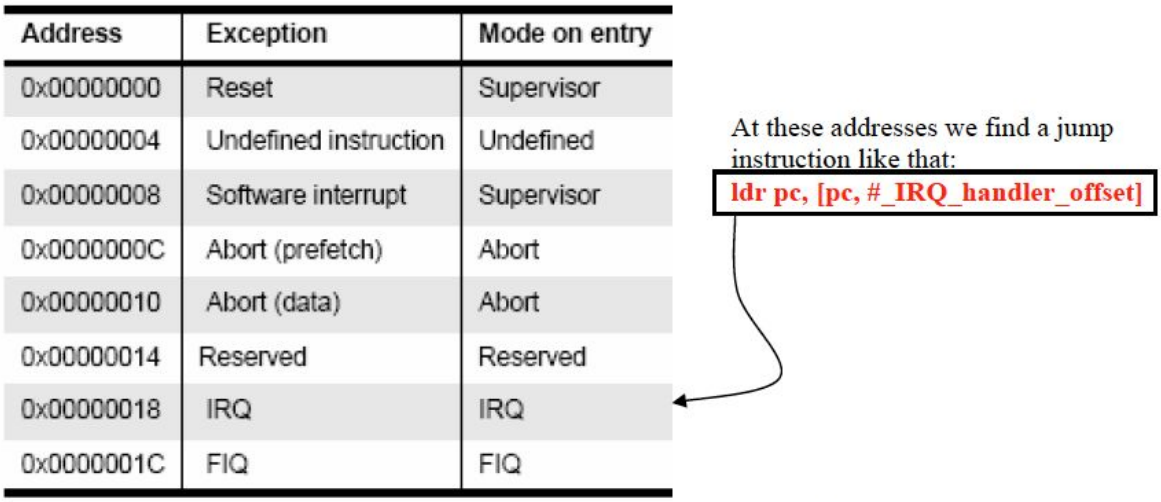
Physical timer is a real timer resides inside your raspberry board. It reads value from a prescaled oscillator. The oscillator runs at a default rate of 19.2Mhz. The default rate of oscillator is too fast to be useful for any timing purpose. In order to make use of this steady oscillator, the default prescaler applied a ratio of 19.2 to clock it down to 1 Mhz. This frequency is more useful, since 1Mhz represent 10^6 times cycles per second. Each tick return us unit of 1 microsecond in time.

Virtual timer, also called the kernel timer, is similar to a count down timer that runs based on the physical timer introduced above. Virtual timer is useful because any timer related function, such as wait(), sleep(), etc all tie to virtual timer.

2.2 Interrupt and Interrupt handler

Interrupt is simply a type of exception. During an exception, the processor causes the program counter (PC) to be set to some predefined values. This predefined value, usually is an address in memory, where the content locates in this address, is capable of handling or resolving this specific type of exception. Exception handler control will return back to previously executing code. Since there could be many exceptions raised during the run time, exception handlers should be concise and efficient.

A value will be loaded into the processor PC when given EXCEPTION occurs. All possible PC values under EXCEPTION, together, combine into a **vector table**. So each address of the vector table is one possible PC value when EXCEPTION occurs. Our operating system need to setup **Vector table content** when the system is up.
To setup the vector table, we simply **put jump instructions in every entry of the table**. These jump instructions goes to different defined handler functions. In this way, on exception, a branch instruction will be execute, and CPU can execute our interrupt handler.



Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

Figure 2 An exact vector table with the branching instruction[5]

Notice that on every entry inside the vector table, there is also a column called “Mode on entry”. Supervisor mode, which is also called privilege mode, is used, for example, when processor is reset on pushing the reset button. On this mode, we are able to write most of the processors’ registers and also able to change processor’s current mode. The only mode that cannot change processor’s mode is User. User mode is only allowed and intended to execute program, where privileged modes are intend for operating system tasks.

There are also two other modes, IRQ mode and FIQ mode, listed on the table above. Our primary focus on this project will be on IRQ mode. **IRQ** stands for **Interrupt Requests** is a special mode and by default begin execution at physical memory address 0x18. Similarly, **FIQ** stands for **Fast Interrupt Request**, which is also a special mode and its default execution is located at physical memory address 0x1C.

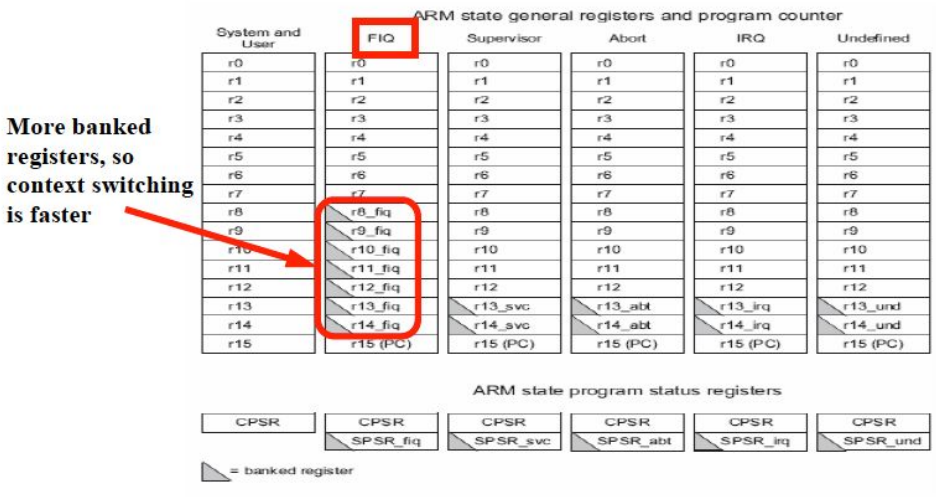


Figure 1: Register Organization in ARM [5]

As we can see the banked registers are marked with the gray colour. We can notice that in the FIQ mode there are more banked registers, this is to speed up the context switching since there will be no need to store many registers when switching to the FIQ mode. We may need only to store the values of registers r0 to r7 if the FIQ handler needs to use those registers, but registers r8_fiq to r14_fiq are specific registers for the FIQ mode and can't be accesses by any other mode (they become invisible in other modes).

FIQ mode is a more advanced mode, which it includes more banked registers than the IRQ mode. On exception, the control flow is transferred away from User mode. Normally, the first thing we would do in the interrupt handler would be to save the registers we are going to use and restore them back to what they were before at the end of interrupt handler. With bank registers in FIQ mode, this save and restore operation is **not needed** since the registers are banked for fast service.

2.3 Scheduler Working Modes

In this project, there will be two types of **scheduling modes**:

- Real time scheduling mode
- Non-Real time scheduling mode

Be careful not to mix up **scheduling modes** with the different **scheduling policies** (FIFO, Priority, etc) we introduced last week. Scheduling policy is about **how** scheduler process tasks within the task queue. Scheduling mode is about **when** scheduler process the takes. In Real time scheduling mode, scheduler processes tasks at real time (almost no delay). When there emerges READY task in the queue, the scheduler would put this READY task into working immediately. On the other hand, Non-Real time scheduling mode will only process existing tasks in the taskQueue periodically. Some

task may sleep a small amount of time, but the scheduling period is very long. So even when the task becomes READY, the task need to wait some time to be rescheduled. For example, the task sleeps 10 ms, but the scheduler only check the queue every 50 ms. In this case, the task may need to wait for 40 ms. But if we use real-time scheduling, the scheduler can know and handle this info immediately.

Why don't scheduler check more frequently? Because it would cost a lot! The scheduler is a regular kernel task in system, checking frequency matters. If there are hundreds of tasks in the queue, checking one by one is huge cost. So real-time scheduling would saves effort for scheduler. But real time scheduling may add too many virtual timer in the system, which can also harm efficiency(Why?).

You will implement both scheduling modes in this project.

Implementation details

3.1 non-blocking countdown timer

```
void addKernelTimer(unsigned nDelay,
                    TKernelTimerHandler *pHandler,
                    void *pParam) {
        Kernel.addKernelTimer(nDelay, pHandler,  pParam);
}
```

addKernelTimer has three input parameters. ***nDelay*** indicates how long the countdown timer should be, the unit is in second. ***pHandler*** is a function pointer to the interrupt handler. ***pParam*** is a pointer point to the task/process that is using the timer.

addKernelTimer is a non-blocking countdown timer, which means once the count down timer is set up, scheduler will be able to schedule the next available task. Upon the timer goes off, the timer will raise an interrupt, then a specific interrupt handler (sleepTimerHandler) will be invoked by the OS.

Important: addKernelTimer() is in unit of **10ms.** For example, if your want to set up a countdown timer of 5 seconds. You need to pass in *nMilliSeconds=500* for addKernelTimer(). [500*10ms = 5000ms=5seconds]

3.2 interrupt handler

```
void sleepTimerHandler (unsigned hTimer, void *pParam, void *pContext)
```

```
{
        Task *pTask = (Task *) pParam;
        pTask->pSysCall->print("sleepTimerHandler is called ");
        // TODO

}
```

In ARM, interrupt handler has three parameters. For sleepTimerHandler() used in this project, you only need to use *pParam*.

Tasks:

Task 1 Simple task for interrupt & interrupt handler

In the first task, you are asked to complete the implementation for a timer and an interrupt handler. *sleep()* and function is given to you, you **don't** need to modify anything in *sleep()*. You need to fill in both

msSleep(),

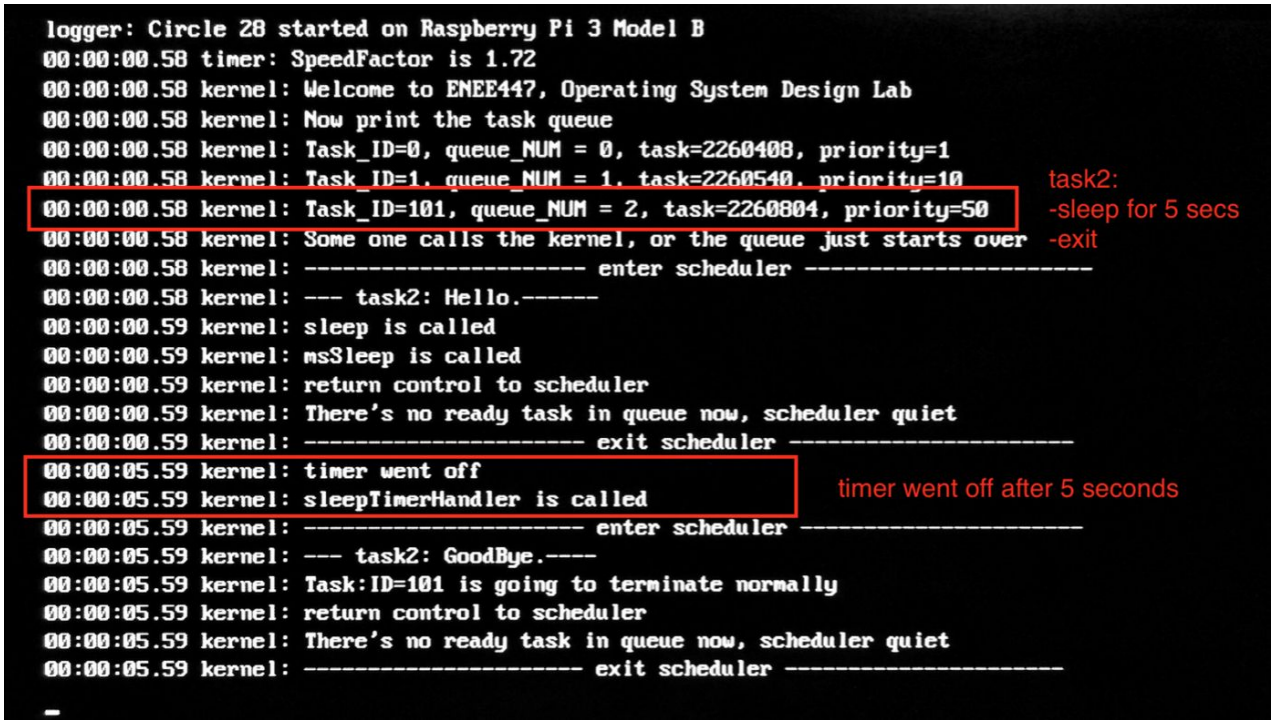
sleepTimerHandler().

You also need to implement

sleepYield(),

which allow user task to yield control back to scheduler during sleep for next ready task.

In this project, use the FIFO scheduling policy for all tasks.



Sample snapshot on task2 provide.

Task 2 Real-time scheduling implementation

In the real time scheme, each task create a virtual timer. This timer handler would modify task state to READY, and notify scheduler that there's some READY task in queue. So the scheduler would come to serve once there is ready task in queue.

Hint:

- don't forget to change your **TaskState** accordingly for sleep and awake.
- **GetNumOfNextReadyTask**() is available for use as well.
- Fill in **scheduleFIFO()**

Snapshot for real-time scheduling with t1&t2 [FIFO].

Task 3 Non real-time scheduling implementation

In the non-real-time scheme, each sleep function only record the WAKE-UP-CLOCK-TICK in each sleeping task. The scheduler has a periodic timer, would periodically check all the sleeping task. If there's some task with READY state, scheduler would re-schedule.

For task 3, once there is no more READY task in the queue, your scheduler will sleep for **4 seconds.** Once scheduler weak up, it will look for READY task again in task queue and process then and sleep again. The provided *schedulerSleepTimerHandler()* needs to be filled in before use.

You need to run the both **task1** and **task2** provided in the project files with FIFO scheduling policy for this task.

Scheduler task could be in either *NEEDTOSCHED* or *SLEEPING* state.

Make sure you **understand** this part of the project before implement it. You grade for this part of project will **purely based** on the screenshot you submitted. Make sure your scheduler is doing the right thing (**timestamps**, **interrupt handling,** etc).

Submission

Plan your time accordingly, **no extension**, any late submission (after **Sunday, March 5th 11:59pm**) will result in penalty. If you submit one day late, 10% will be deducted from your project score. If you submit two days late, 20% will be deducted, etc.

What is inside your zip submission:

- Design/Documentation Report in PDF format. (include **screenshot** for **task3**)
- Make sure your submission is at **task 2 state, which will be used for demo.**
- All Project files, make sure it is compilable, any compile error automatically 10% off.

Demo:

- Demonstrate the **result of task 2** to your TA during lab following the due date.

Honor Code

Never ruin others fun and never show your code to others!