Interrupt, Timer And Advanced Scheduling Scheme Design

Overview: This passage describes the working procedures of BCM283* interruption, and kernel timer implementation in pi-OS. ARM interrupt is ARM-ARCHITECTURE specific, not chip specific, meaning that the interruption working flow on a certain ARM-arch is independent of chip manufacturer. Only thing that has to do with manufacturer is the interrupt-related register ADDRESS. Timer is implemented through interruption, and we need to bind physical timer with IRQ. So when the timer reaches a certain CLOCK TICK, CPU will goes into corresponding interruption procedure. Virtual timer (or kernel timer) is different from physical timer. Virtual timer is based on physical timer, and the number of virtual timer is not limited. But the number of physical timer is certain on a chip.

1. ARM interrupt basics

Why interrupt?

Interrupt sources:

1) Hardware peripherals, like timer, GPIO
2) Software interrupt (SWI or bad instruction)

processor MODE:

Processor Mode	Description
User (*usr*)	Normal program execution mode
FIQ (*fiq*)	Fast data processing mode
IRQ (*irq*)	For general purpose interrupts
Supervisor (*svc*)	A protected mode for the operating system
Abort (*abt*)	When data or instruction fetch is aborted
Undefined (*und*)	For undefined instructions
System (*sys*)	Operating system privileged mode

Registers under different processor MODE

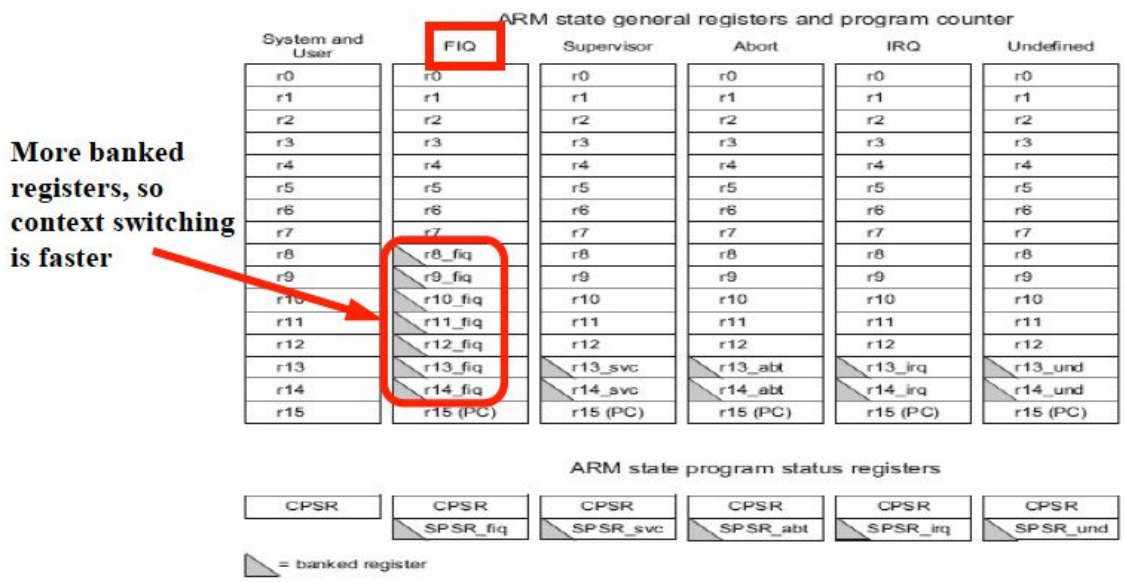


Figure 1: Register Organization in ARM [5]

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).

Interrupt VECTOR TABLE:

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

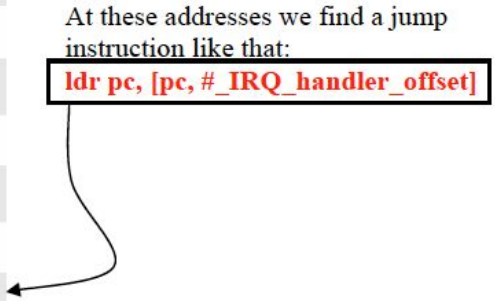


Figure 2 An exact vector table with the branching instruction[5]

We can notice in the vector table that the FIQ exception handler is placed at the end of the vector table, so no need for a branching instruction there; we can place the exception handler directly there so handling begins faster by eliminating the time of branching.

2. Interrupt basic working flow

2.4 Entering and exiting an exception handler

Here are the steps that the ARM processor does to handle an exception [5]:

- Preserve the address of the next instruction.
- Copy ***CPSR*** to the appropriate ***SPSR***, which is one of the banked registers for each mode of operation.
- Force the ***CPSR*** mode bits to a value depending on the raised exception.
- Force the ***PC*** to fetch the next instruction from the exception vector table.
- Now the handler is running in the mode associated with the raised exception.
- When handler is done, the ***CPSR*** is restored from the saved ***SPSR.***
- ***PC*** is updated with the value of (***LR*** – offset) and the offset value depends on the type of the exception.

And when deciding to leave the exception handler, the following steps occurs:

- Move the Link Register ***LR*** (minus an offset) to the ***PC***.
- Copy ***SPSR*** back to ***CPSR,*** this will automatically changes the mode back to the previous one.
- Clear the interrupt disable flags (if they were set).

3.3 IRQ and FIQ exceptions

Both exceptions occur when a specific interrupt mask is cleared in the *CPSR*. The ARM processor will continue executing the current instruction in the pipeline before handling the interrupt. The processor hardware go through the following standard procedure:

- The processor changes to a specific mode depending on the received interrupt.
- The previous mode ***CPSR*** is saved in ***SPSR*** of the new mode.
- The ***PC*** is saved in the ***LR*** of the new mode.
- Interrupts are disabled, either IRQ or both IRQ and FIQ.
- The processor branches to a specific entry in the vector table.

Enabling/Disabling FIQ and IRQ exceptions is done on three steps; at first loading the contents of ***CPSR*** then setting/clearing the mask bit required then copy the updated contents back to the ***CPSR***.

3. Interrupt implementation

Upon now, we know the basic characteristics of ARM interrupt. Interrupt is very important to operating system, and many fundamental OS functions are based on interrupt. When interrupt happens, CPU will automatically change into certain mode(IRQ, FIQ). And the CPU will automatically jump to an address in the Exception Vector Table. IRQ happens, it jumps to 0x00000018. Then it's our developers' job to put an instruction in 0x00000018, letting the CPU jumps to our IRQ handler.

We can have many different sources of interrupt, of both hardware type and software type. These sources will generate interrupt with a certain interrupt number. So there is a corresponding relationship between **interrupt source** and **interrupt number** on the chip. When interrupt happens, we can get the interrupt number **by reading some *registers* mapped to memory**(the CPU or GPU would write some hardware info automatically on certain memory address, these address are manufacturer-chip dependent.). Once we get the interrupt number, we can handle correspondingly.

To make the software maintainable, when we want to use certain interrupt, we bind the handler we defined with the interrupt, and this kind of binding info need to be stored in some static structures in the system. When interrupt happens, we just call corresponding handler.

Below is some important functions in pi-OS, describing how interrupt works. You need to see these functions in an IDE like Eclipse. **Be familiar with them, because our timer is based on interrupt system.**

```
66 boolean CInterruptSystem::Initialize (void)
67 {
68     TExceptionTable *pTable = (TExceptionTable *) ARM_EXCEPTION_TABLE_BASE;
69     pTable->IRQ = ARM_OPCODE_BRANCH (ARM_DISTANCE (pTable->IRQ, IRQStub));
70
71     SyncDataAndInstructionCache ();
72
73 #ifndef USE_RPI_STUB_AT
74     PeripheralEntry ();
75
76     write32 (ARM_IC_FIQ_CONTROL, 0);
77
78     write32 (ARM_IC_DISABLE_IRQS_1, (u32) -1);
79     write32 (ARM_IC_DISABLE_IRQS_2, (u32) -1);
80     write32 (ARM_IC_DISABLE_BASIC_IRQS, (u32) -1);
81
82     PeripheralExit ();
83 #endif
84
85     EnableInterrupts ();
86
87     return TRUE;
88 }
```

Jump to IRQ handler

```
29 #define ARM_OPCODE_BRANCH(distance) (0xEA000000 | (distance))
30 #define ARM_DISTANCE(from, to)      ((u32 *) &(to) - (u32 *) &(from) - 2)
```

initial interrupt handler IRQStub:

```
56 * IRQ stub
57 */
58     .globl  IRQStub
59 IRQStub:
60     sub lr, lr, #4          /* lr: return address */
61     stmfd   sp!, {r0-r12, lr}       /* save r0-r12 and return address */
62     bl  InterruptHandler
63     ldmfd   sp!, {r0-r12, pc}^      /* restore registers and return */
```

```
198 void InterruptHandler (void)
199 {
200     PeripheralExit ();  // exit from interrupted peripheral
201
202     CInterruptSystem::InterruptHandler ();
203
204     PeripheralEntry (); // continuing with interrupted peripheral
205 }
```

CInterruptSystem InterruptHandler:

```
159 void CInterruptSystem::InterruptHandler (void)
160 {
161     assert (s_pThis != 0);
162
163 #ifdef ARM_ALLOW_MULTI_CORE
164     if (CMultiCoreSupport::LocalInterruptHandler ())
165     {
166         return;
167     }
168 #endif
169
170     u32 Pending[ARM_IC_IRQ_REGS];
171     Pending[0] = read32 (ARM_IC_IRQ_PENDING_1);
172     Pending[1] = read32 (ARM_IC_IRQ_PENDING_2);
173     Pending[2] = read32 (ARM_IC_IRQ_BASIC_PENDING) & 0xFF;
174
175     for (unsigned nReg = 0; nReg < ARM_IC_IRQ_REGS; nReg++)
176     {
177         u32 nPending = Pending[nReg];
178         if (nPending != 0)
179         {
180             unsigned nIRQ = nReg * ARM_IRQS_PER_REG;
181
182             do
183             {
184                 if (   (nPending & 1)
185                     && s_pThis->CallIRQHandler (nIRQ))
186                 {
187                     return;
188                 }
189
190                 nPending >>= 1;
191                 nIRQ++;
192             }
193             while (nPending != 0);
194         }
195     }
196 }
```

Explain: 2*32 +8 =72 IRQ. need to locate which IRQ is called. Pending[n] is a 32 bit content, reading from 3 irq-mapping registers. Each bit correspond to one irq.

4. physical timer implementation

We use TIMER3 in the BCM chip, and this timer has a certain IRQ number ARM_IRQ_TIMER3. We need to connect corresponding handler in the **CInterruptSystem** class.

```
83 boolean CTimer::Initialize (void)
84 {
85     assert (m_pInterruptSystem != 0);
86     m_pInterruptSystem->ConnectIRQ (ARM_IRQ_TIMER3, InterruptHandler, this);
87
88     PeripheralEntry ();
89
90     write32 (ARM_SYSTIMER_CLO, -(30 * CLOCKHZ));    // timer wraps soon, to check for problems
91
92     write32 (ARM_SYSTIMER_C3, read32 (ARM_SYSTIMER_CLO) + CLOCKHZ / HZ);
93
94     TuneMsDelay ();
95
96     PeripheralExit ();
97
98     return TRUE;
99 }
```

```
92 void CInterruptSystem::ConnectIRQ (unsigned nIRQ, TIRQHandler *pHandler, void *pParam)
93 {
94     assert (nIRQ < IRQ_LINES);
95     assert (m_apIRQHandler[nIRQ] == 0);
96
97     m_apIRQHandler[nIRQ] = pHandler;
98     m_pParam[nIRQ] = pParam;
99
100    EnableIRQ (nIRQ);
101 }
```

When ARM_IRQ_TIMER3 interrupts the system, the **initial interrupt handler IRQStub** will goes go **CInterruptSystem InterruptHandler**, then CInterruptSystem **InterruptHandler** check all the IRQ sources, then goes to corresponding handler recorded in its member **m_apIRQHandler[]** . So it will call **CTimer::InterruptHandler.**

```
363 void CTimer::InterruptHandler (void)
364 {
365     PeripheralEntry ();
366
367     assert (read32 (ARM_SYSTIMER_CS) & (1 << 3));
368
369     u32 nCompare = read32 (ARM_SYSTIMER_C3) + CLOCKHZ / HZ;
370     write32 (ARM_SYSTIMER_C3, nCompare);
371     if (nCompare < read32 (ARM_SYSTIMER_CLO))           // time may drift
372     {
373         nCompare = read32 (ARM_SYSTIMER_CLO) + CLOCKHZ / HZ;
374         write32 (ARM_SYSTIMER_C3, nCompare);
375     }
376
377     write32 (ARM_SYSTIMER_CS, 1 << 3);
378
379     PeripheralExit ();
380
381 #ifndef NDEBUG
382     //debug_click ();
383 #endif
384
385     m_TimeSpinLock.Acquire ();
386
387     if (++m_nTicks % HZ == 0)
388     {
389         m_nUptime++;
390         m_nTime++;
391     }
392
393     m_TimeSpinLock.Release ();
394
395     PollKernelTimers ();
396 }
```

CTimer::InterruptHandler then calls PollKernelTimers, which check all the virtual timers.

```
398 void CTimer::InterruptHandler (void *pParam)
399 {
400     CTimer *pThis = (CTimer *) pParam;
401     assert (pThis != 0);
402
403     pThis->InterruptHandler ();
404 }
```

5. Virtual timer (Kernel timer) implementation

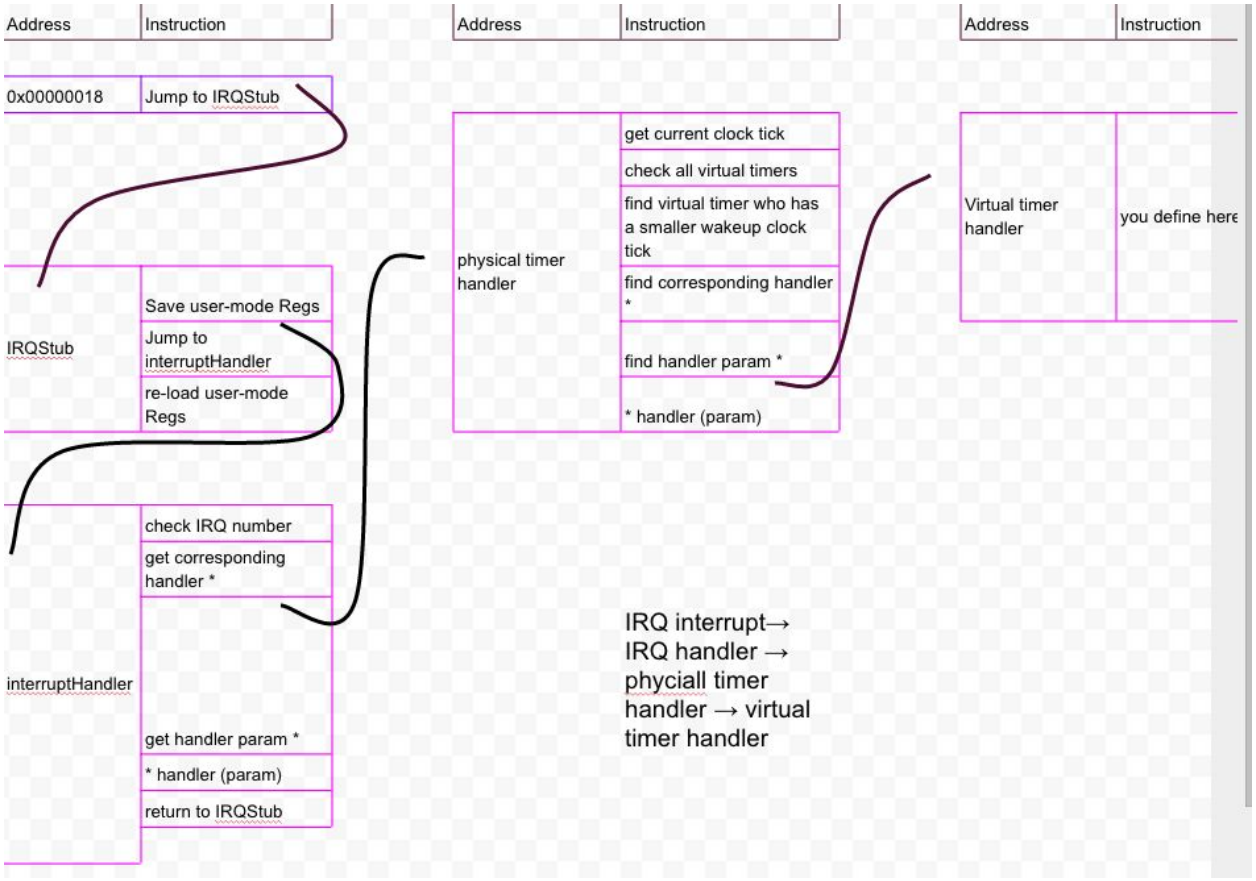
When we want to add a kernel timer, we need to prepare timer handler function, and call CTimer::StartKernelTImer

```
253 unsigned CTimer::StartKernelTimer (unsigned nDelay,
254                     TKernelTimerHandler *pHandler,
255                     void *pParam,
256                     void *pContext)
257 {
258     TKernelTimer *pTimer = new TKernelTimer;
259     assert (pTimer != 0);
260
261     unsigned nElapsesAt = m_nTicks + nDelay;
262
263     assert (pHandler != 0);
264 #ifndef NDEBUG
265     pTimer->m_nMagic     = KERNEL_TIMER_MAGIC;
266 #endif
267     pTimer->m_pHandler   = pHandler;
268     pTimer->m_nElapsesAt = nElapsesAt;
269     pTimer->m_pParam     = pParam;
270     pTimer->m_pContext   = pContext;
271
272     m_KernelTimerSpinLock.Acquire ();
273
274     TPtrListElement *pPrevElement = 0;
275     TPtrListElement *pElement = m_KernelTimerList.GetFirst ();
276     while (pElement != 0)
277     {
278         TKernelTimer *pTimer2 = (TKernelTimer *) m_KernelTimerList.GetPtr (pElement);
279         assert (pTimer2 != 0);
280         assert (pTimer2->m_nMagic == KERNEL_TIMER_MAGIC);
281
282         if ((int) (pTimer2->m_nElapsesAt-nElapsesAt) > 0)
283         {
284             break;
285         }
286
287         pPrevElement = pElement;
288         pElement = m_KernelTimerList.GetNext (pElement);
289     }
290
291     if (pElement != 0)
292     {
293         m_KernelTimerList.InsertBefore (pElement, pTimer);
294     }
295     else
296     {
297         m_KernelTimerList.InsertAfter (pPrevElement, pTimer);
298     }
```

Physical timer handler will call PoolKernelTimers, then it will check each Virtual Timer. If a virtual timer's WAKE-UP clock tick has been reached, then the corresponding virtual timer handler would be called:

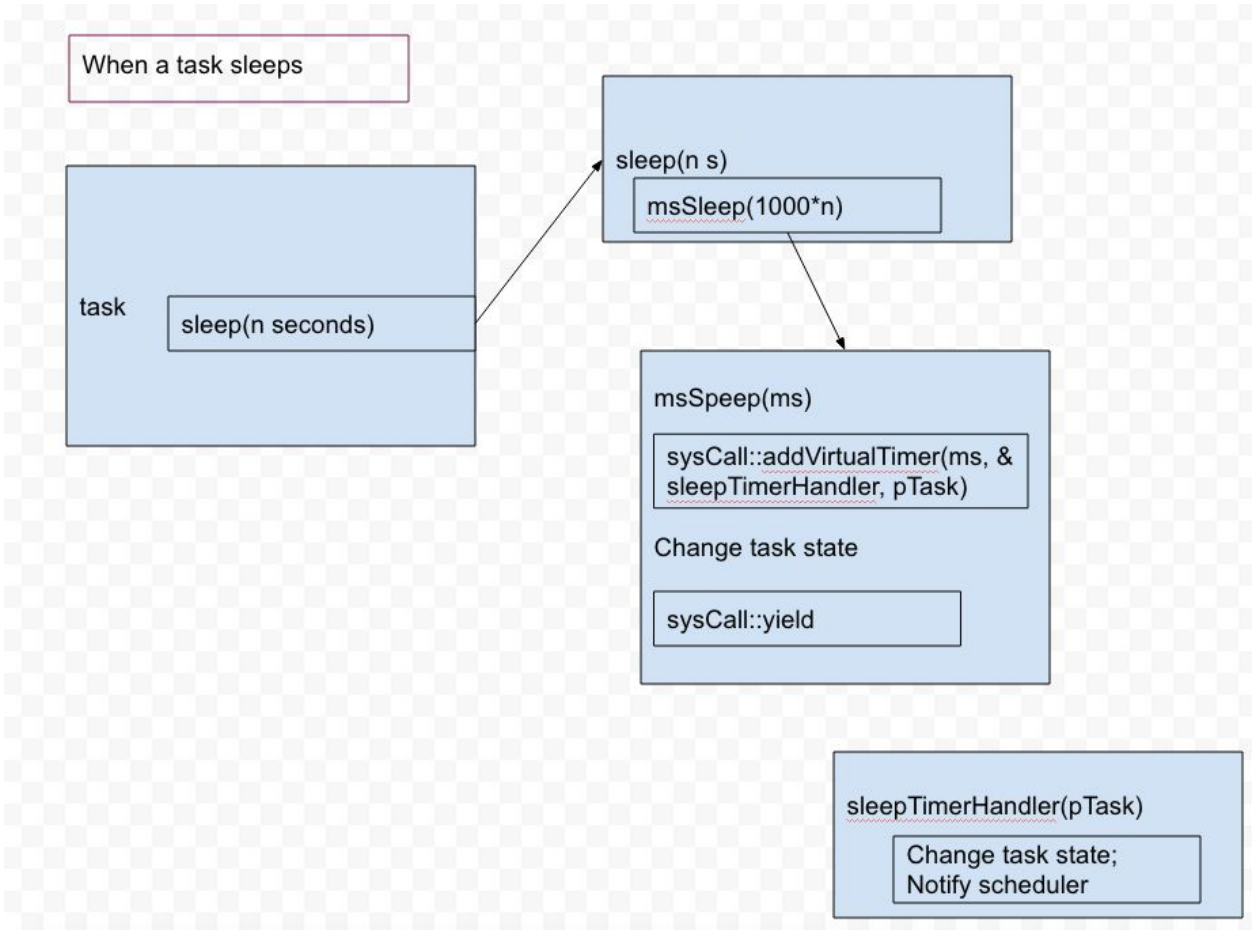
```
328 void CTimer::PollKernelTimers (void)
329 {
330     m_KernelTimerSpinLock.Acquire ();
331
332     TPtrListElement *pElement = m_KernelTimerList.GetFirst ();
333     while (pElement != 0)
334     {
335         TKernelTimer *pTimer = (TKernelTimer *) m_KernelTimerList.GetPtr (pElement);
336         assert (pTimer != 0);
337         assert (pTimer->m_nMagic == KERNEL_TIMER_MAGIC);
338
339         if ((int) (pTimer->m_nElapsesAt-m_nTicks) > 0)
340         {
341             break;
342         }
343
344         TPtrListElement *pNextElement = m_KernelTimerList.GetNext (pElement);
345         m_KernelTimerList.Remove (pElement);
346         pElement = pNextElement;
347
348         m_KernelTimerSpinLock.Release ();
349
350         TKernelTimerHandler *pHandler = pTimer->m_pHandler;
351         assert (pHandler != 0);
352         (*pHandler) ((unsigned) pTimer, pTimer->m_pParam, pTimer->m_pContext);
353
354 #ifndef NDEBUG
355         pTimer->m_nMagic = 0;
356 #endif
357         delete pTimer;
358
359         m_KernelTimerSpinLock.Acquire ();
360     }
361
362     m_KernelTimerSpinLock.Release ();
363 }
```

The control flow when timer interrupt happens in pi-OS:



6. Application of Timer: sleep() function

When a task want to wait for a period of time before continue running, it needs to call system::sleep. Sleeping will change the taskState to STATE_SLEEPING. Then kernel (or scheduler) regain the control, select next READY task to run. When there's no READY task in queue, scheduler would quit and kernel regains the CPU control.

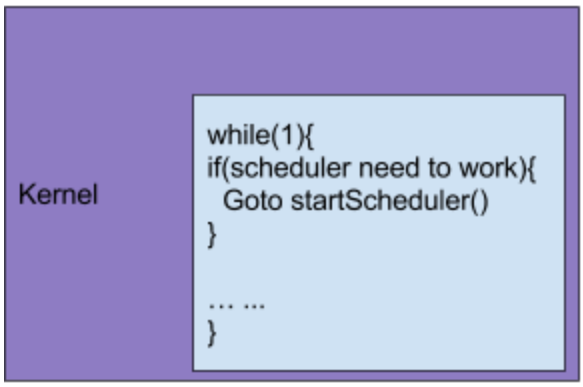


How to notify the scheduler?
We can change the scheduler state to NEEDTOSCHED. The kernel would check scheduler from time to time, and once it goes to scheduler if the state is NEEDTOSCHED. Keep in mind that **schedulerState** must be **volatile type**, because **it's static in memory** and can **be written and read by different functions**.

7. Real-time scheduling

When there's no READY task in the queue, but there're some SLEEPING tasks, the scheduler needs to know when to re-schedule. There can be two scheduling scheme here: real-time and non-real-time.
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 in the first time.

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.




```
typedef enum {
    SCHEDULING,
    NOREADYTASK,
    NEEDTOSCHED
}SCHEDState;

typedef struct {
    Task * taskQueue[MAXTASK];
    Task * pCurrentTask;
    Task * pLastTask;
    TaskSwitchReason lastTaskSwitchReason;
    volatile SCHEDState SchedulerState ;

}Tscheduler;
```

Scheduling ALGORITHM:

Check the task queue, to find some task **with READY state**;
… …
No task with READY state, return

8. Some pitfalls

schedulerState need to be volatile:

```
typedef struct {
    Task * taskQueue[MAXTASK];
    Task * pCurrentTask;
    Task * pLastTask;
    TaskSwitchReason lastTaskSwitchReason;
    volatile SCHEDState SchedulerState ;

}Tscheduler;
```

Task state need to be volatile:

```
33 typedef struct TTask
34 {
35     u32          ID;
36     char     name[32];
37     volatile     TTaskState State;
38     struct TTask * pSelf;
39     unsigned priority;
40     void (*Run)(void * pTask);
41     void (*taks_entry)(void * pTask);
42     unsigned    WakeTicks;
43     TTaskRegisters     Regs;
44
45     u8          *pStack;
46     unsigned        StackSize;
47     sysCall * pSysCall;
48
49
50 }Task;
```

9. Goals

Understanding ARM interrupt procedures.
Know how the system initialize interrupt vector table.
Know when the system register the physical timer handler.
Know difference between physical timer and virtual timer.
(You should be able to find corresponding codes in pi-OS.)

Implement sleep function.
Implement real-time scheduling scheme.
Implement non-real-time scheduling scheme.