UEFI Development Exploration 41 – Event, Timer and Task Priority

(Please keep it-> Author: Luo Bing https://blog.csdn.net/luobing4365)

As a low-level support system, UEFI does not support interrupts. If you want to support asynchronous operations, you can only do so through events.

In the process of developing Foxdisk, I also encountered events that needed to be processed simultaneously, such as the blinking cursor that prompted the user to input, the automatic display of the system time, etc. I used the clock interrupt (int 1Ch) to implement it, which is a very interesting program.

However, I simply stacked the required functions in int 1Ch, and did not fully implement the mutual exclusion between multiple tasks. It is a "pseudo multitasking" implementation. So, how does UEFI support the simultaneous execution of multiple tasks?

1 Supported service functions

Figure 1 shows the related service functions , a total of 10:

Name	Type	Description
CreateEvent	Boot	Creates a general-purpose event structure
CreateEventEx	Boot	Creates an event structure as part of an event group
CloseEvent	Boot	Closes and frees an event structure
SignalEvent	Boot	Signals an event
WaitForEvent	Boot	Stops execution until an event is signaled
CheckEvent	Boot	Checks whether an event is in the signaled state
SetTimer	Boot	Sets an event to be signaled at a particular time
RaiseTPL	Boot	Raises the task priority level
RestoreTPL	Boot	Restores/lowers the task priority level

Figure 1 Event-related service functions

These functions run in the Boot Services environment and have different task priority requirements (or TPL requirements). In the Boot Services environment, there are three priorities:

TPL_APPLICATION - the lowest priority, the application runs at this level;
TPL_CALLBACK - medium priority, some time-consuming operations, such as disk operations, run at this level;
TPL_NOTIFY - high priority, blocking is not allowed, and should be completed as soon as possible. Usually the underlying IO operations are at this level.

A task running at a high priority can interrupt a task running at a lower priority.

TPL_HIGH_LEVEL is the highest priority level. Interrupts at this level are prohibited. UEFI kernel global variables must be modified at this level.

For the level at which each function and protocol runs, refer to UEFI Spec 2.8 page 141-144.

Events have two mutually exclusive states, "waiting" and "signaled". When an event is created, the firmware sets it to the waiting state. When the event is triggered, the firmware converts it to the triggered state. If the event type is EVT_NOTIFY_SIGNAL, its related notification function will also be placed in the FIFO queue.

There is a processing queue for TPL_CALLBACK and TPL_NOTIFY events. If the TPL of the notification in the queue is equal to or less than the TPL of the current task, it can only wait until the TPL of the current task is reduced, usually by changing the TPL through EFI_BOOT_SERVICES.RestoreTPL.

Generally speaking, events can also be divided into two types: synchronous and asynchronous. A typical example of asynchronous is that in a network device driver, EVT_TIMER events are used to wait for new network packets. Calling EFI_BOOT_SERVICES.ExitBootServices() is a synchronous example. When the function is completed, the EVT_SIGNAL_EXIT_BOOT_SERVICES event will be triggered.

2 Function Description

2.1 CreateEvent()

```
typedef EFI_STATUS (EFIAPI *EFI_CREATE_EVENT) (
IN UINT32 Type,   //Event type
IN EFI_TPL NotifyTpl,   //Priority of NotifyFunction function
IN EFI_EVENT_NOTIFY NotifyFunction, OPTIONAL   //NotifyFunction function
IN VOID *NotifyContext, OPTIONAL  //Parameters passed to NotifyFunction function
OUT EFI_EVENT *Event   //Generated event
);
```

The event types are as follows (from MdePkg\Uefi\UefiSpec.h):

```
// These types can be ORed together as needed - for example,
// EVT_TIMER might be Ored with EVT_NOTIFY_WAIT or
// EVT_NOTIFY_SIGNAL.
//
#define EVT_TIMER                          0x80000000
#define EVT_RUNTIME                        0x40000000
#define EVT_NOTIFY_WAIT                    0x00000100
#define EVT_NOTIFY_SIGNAL                  0x00000200

#define EVT_SIGNAL_EXIT_BOOT_SERVICES      0x00000201
#define EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE 0x60000202

//
// The event's NotifyContext pointer points to a runtime memory
// address.
// The event is deprecated in UEFI2.0 and later specifications.
//
#define EVT_RUNTIME_CONTEXT                0x20000000
```


Figure 2 Various types of events

EVT_TIMER: *Ordinary Timer event. After the event is generated, the SetTimer service needs to be called to set the Timer properties.*

EVT_RUNTIME: *If the event is triggered after calling EFI_BOOT_SERVICES.ExitBootServices(), both the event data structure and the notification function must be allocated from runtime memory.*

EVT_NOTIFY_WAIT: *normal event, this event has a notification function. When this event is waited by EFI_BOOT_SERVICES.WaitForEvent() or checked by EFI_BOOT_SERVICES.CheckEvent(), this function will be put into the queue to be executed* ;

EVT_NOTIFY_SIGNAL: *When this event is triggered, its NotifyFunction will be placed in the queue to be executed* ;

EVT_SIGNAL_EXIT_BOOT_SERVICES: *The system issues this event when ExitBootSerices() is called* ;

EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE: *The system issues this event when SetVirtualAddressMap() is executed* ;

The function prototype of the above notification function NotifyFunction is:

*typedef VOID (EFIAPI *EFI_EVENT_NOTIFY)(*
IN EFI_EVENT Event; //Event called by notification function
*IN VOID *Context //Content pointing to notification function (personally, it is similar to the additional string in Windows control ListView)*
);

The CreateEventEx function has one more event group entry parameter than the CreateEvent function. If this parameter is not specified (NULL), this function is the same as CreateEvent.

An event group is a collection of events that share the same EFI_GUID. When any event in the group is triggered, all other events in the group will be triggered, and all notification functions in the same group will be added to the queue to be executed.

For types EVT_SIGNAL_EXIT_BOOT_SERVICES and EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE, the event group is fixed and has been predetermined in UEFI.

Multiple EVT_TIMERs can form an event group, but there is no way to determine which Timer is triggered.

For events in an event group, you can use CloseEvent to remove it.

2.3 CloseEvent() and SignalEvent()

*typedef EFI_STATUS (EFIAPI *EFI_CLOSE_EVENT) (*
IN EFI_EVENT Event //Event that needs to be closed
);

*typedef EFI_STATUS (EFIAPI *EFI_SIGNAL_EVENT) (*
IN EFI_EVENT Event // triggered event
);

The following is an example of triggering an event group. Create an event to add to the event group, and remove the event after triggering:

EFI_EVENT Event;
EFI_GUID gMyEventGroupGuid = EFI_MY_EVENT_GROUP_GUID;
gBS->CreateEventEx (0, 0, NULL, NULL,
&gMyEventGroupGuid,
&Event
);
gBS->SignalEvent (Event);
gBS->CloseEvent (Event);

2.4 WaitForEvent() and CheckEvent()

*typedef EFI_STATUS (EFIAPI *EFI_WAIT_FOR_EVENT) (*
IN UINTN NumberOfEvents, //Number of events in the event array
*IN EFI_EVENT *Event, //Event array*

```
OUT UINTN *Index //Returns the index of the event in the triggered state in the array (counting starts from zero)
);
```

```
typedef EFI_STATUS (EFIAPI *EFI_CHECK_EVENT) (
IN EFI_EVENT Event // event to be checked to see if it is in the triggered state
);
```

WaitForEvent must be called at priority TPL_APPLICATION. If called at any other priority, the function will return EFI_UNSUPPORTED.

WaitForEvent repeatedly checks the events in the event array from front to back until an event is found to be triggered or an error occurs. When an event is detected to be in the triggered state, *Index returns the index of the event in the event array and resets the event to the waiting state before returning.

If an event type is EVT_NOTIFY_SIGNAL, WaitForEvent will return EFI_INVALID_PARAMETER, and *Index indicates the index of this event in the array.

The WaitForEvent function is a blocking function. If you don't want to wait, you can use the CheckForEvent function to check the status of the event.

2.5 SetTimer()

```
typedef EFI_STATUS (EFIAPI *EFI_SET_TIMER) (
IN EFI_EVENT Event, //Timer event
IN EFI_TIMER_DELAY Type, //Timer category
IN UINT64 TriggerTime //Timer expiration event, 100ns as a unit
);
```

```
typedef enum { TimerCancel, //Used to cancel the timer trigger event. The timer will not be triggered after setting. TimerPeriodic, //Repetitive timer, the trigger time is TriggerTime * 100ns TimerRelative //One-time timer, the trigger time is TriggerTime * 100ns } EFI_TIMER_DELAY;
```

After this function is called, the previously set event-related time settings will be canceled and the new trigger event settings will be enabled. It can only be used for events of type EVT_TIMER.

2.6 RaiseTPL() and RestoreTPL()

```
typedef EFI_TPL (EFIAPI *EFI_RAISE_TPL) (
IN EFI_TPL NewTpl //New priority, must be higher than or equal to the current task priority
);
```

Related priority values:

```
#define TPL_APPLICATION 4
#define TPL_CALLBACK 8
#define TPL_NOTIFY 16
#define TPL_HIGH_LEVEL 31
```

```
typedef VOID (EFIAPI *EFI_RESTORE_TPL) (
IN EFI_TPL OldTpl //The priority of the task before calling RaiseTPL
)
```

These two functions are paired. After RaiseTPL is called, the priority of the current task will be returned so that RestoreTPL can restore it.

When a task is raised to TPL_HIGH_LEVEL, interrupts are disabled; when it is restored to a level below this priority, interrupts are enabled again.

3 Mechanism Exploration

3.1 Clock Interrupt (Function of Clock 0)

In Foxdisk, I used the default clock 0 for operation. When the system is powered on and initialized, the timer is initialized to send an interrupt request every 55ms. After responding to the interrupt request, the CPU enters the 8H interrupt handler. In the BIOS 8H interrupt, there is an instruction "Int 1ch", so the 1CH interrupt handler is called about 18.2 times per second.

The 8254 programmable clock contains three independent 16-bit clocks. Clock 0 is used for system timing and will be initialized to the above state in the BIOS post stage. The three clocks are assigned three read and write registers, 0x40, 0x41, and 0x42, and also include a common control register 0x43.

My actual code is all in Int 1cH. This method is simple to program and does not require much hardware operation. The way to prevent several tasks from deadlocking is to turn off interrupts. The disadvantage is that the complete event mechanism is not implemented, and it is not a true multi-task scheduling.

Several registers are needed to operate the clock 0 of the 8254. We are mainly concerned with the mode control part, which involves ports 0x40 and 0x43. (In this blog post, the prefix "0x" is generally used to represent hexadecimal, which is the usage of C language; sometimes the suffix "h" is used to represent hexadecimal , which is the usage of assembly language)

```
                8253/8254 PIT - Programmable Interval Timer

Port 40h, 8253 Counter 0 Time of Day Clock (normally mode 3)
Port 41h, 8253 Counter 1 RAM Refresh Counter (normally mode 2)
Port 42h, 8253 Counter 2 Cassette and Speaker Functions
Port 43h, 8253 Mode Control Register, data format:

|7|6|5|4|3|2|1|0|  Mode Control Register
 | | | | | | | `---- 0=16 binary counter, 1=4 decade BCD counter
 | | | | `-------- counter mode bits
 | | `------------ read/write/latch format bits
 `--------------- counter select bits (also 8254 read back command)

Bits
 76 Counter Select Bits
 00  select counter 0
 01  select counter 1
 10  select counter 2
 11  read back command (8254 only, illegal on 8253, see below)
Bits
 54  Read/Write/Latch Format Bits
 00  latch present counter value
 01  read/write of MSB only
 10  read/write of LSB only
 11  read/write LSB, followed by write of MSB

Bits
321  Counter Mode Bits
000  mode 0, interrupt on terminal count;  countdown, interrupt,
     then wait for a new mode or count; loading a new count in the
     middle of a count stops the countdown
001  mode 1, programmable one-shot; countdown with optional
     restart; reloading the counter will not affect the countdown
     until after the following trigger
010  mode 2, rate generator; generate one pulse after 'count' CLK
     cycles; output remains high until after the new countdown has
     begun; reloading the count mid-period does not take affect
     until after the period
011  mode 3, square wave rate generator; generate one pulse after
     'count' CLK cycles; output remains high until 1/2 of the next
     countdown; it does this by decrementing by 2 until zero, at
     which time it lowers the output signal, reloads the counter
     and counts down again until interrupting at 0; reloading the
     count mid-period does not take affect until after the period
100  mode 4, software triggered strobe; countdown with output high
     until counter zero;  at zero output goes low for one CLK
     period;  countdown is triggered by loading counter;  reloading
     counter takes effect on next CLK pulse
101  mode 5, hardware triggered strobe; countdown after triggering
     with output high until counter zero; at zero output goes low
     for one CLK period
```

Figure 3 Excerpt from the description of 8254

As can be seen from the figure, writing 0x36 to port 0x43 (mode control register) sets clock 0 to mode 3. Just set the corresponding 16-bit count value to port 0x40 (system tick of clock 0, ie read and write count register). When setting, send the low-significant byte of the count value first, and then send the high-significant byte.

Mode 3 is a square wave mode, which is used to generate a periodic square wave output. When the count value is 0, the output period is the largest, and the output appears every 54.9ms. The input frequency of the three clocks is initially 1.1931817MHz. 2^16.

The function for operating clock 0 is in \PcAtChipsetPkg\8254TimerDxe\Timer.c. If you read the code carefully, you will find a lot of interesting things.

Function SetPitCount:

```
  Sets the counter value for Timer #0 in a legacy 8254 timer.

  @param Count    The 16-bit counter value to program into Timer #0 of the legacy 8254 timer.
**/
VOID
SetPitCount (
  IN UINT16  Count
  )
{
  IoWrite8 (TIMER_CONTROL_PORT, 0x36);
  IoWrite8 (TIMER0_COUNT_PORT, (UINT8)(Count & 0xff));
  IoWrite8 (TIMER0_COUNT_PORT, (UINT8)((Count >> 8) & 0xff));
}
```

Figure 4 Setting the count value of clock 0

Function SetPitCount sets the count value for clock 0, and the count value is passed in by Count. This function is called by TimerDriverSetTimerPeriod, and no other function calls it.

The TimerDriverSetTimerPeriod function prototype is:

EFI_STATUS EFIAPI TimerDriverSetTimerPeriod (
  *IN EFI_TIMER_ARCH_PROTOCOL *This,*
  IN UINT64 TimerPeriod
)

The formula between its entry parameter TimerPeriod and the parameter TimerCount passed in when calling SetPitCount is as follows:

TimerCount = ((TimerPeriod * 119318) + 500000)/1000000.

In other words, TimerPeriod multiplied by 100ns is the time for clock cycle 0 output. In the initialization function TimerDriverInitialize, the setting function is called like this:

Status = TimerDriverSetTimerPeriod (&mTimer, DEFAULT_TIMER_TICK_DURATION);

Clock 0 is set here to interrupt once every 10ms (DEFAULT_TIMER_TICK_DURATION=100000) to schedule tasks.

The real working scheduling function is CoreTimerTick, which is located in \MdeModulePkg\Core\Dxe\Event\Timer.c (in UDK2018). Clock 0 calls it once every 10ms to implement the event scheduling mechanism.

As for how this function is registered and how it works, there is a very detailed description in "UEFI Principles and Programming", so I won't repeat it here.

3.2 Myth of SetTimer

The third parameter of SetTimer is in 100ns granularity. If we use 8254's clock 0, whose maximum frequency is 1.19318MHz, the minimum time interval is 840ns, how can this work?

There are two timers used in UEFI corresponding to Intel platform: 8254 and Hpet. How to choose between them in use?

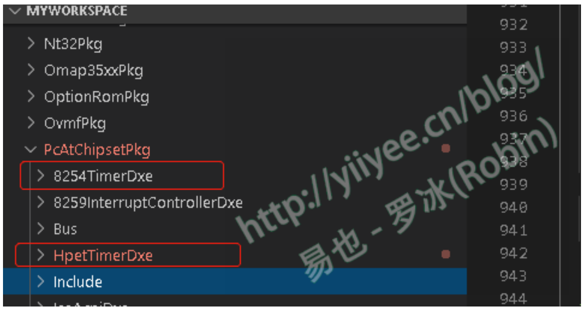

Figure 5 Hpet and 8254

When I debugged with windbg + Qemu0.13, I used 32-bit OVMF.fd. When enumerating TimerDriverInitialize, I located the directory of 8254TimerDxe. I think the Hpet timer should be used normally. It may be related to the Qemu version I am using now. It may not simulate the Hpet timer. (I tried 64-bit OVMF.fd and the same phenomenon occurred)

The interface provided by the Hpet timer is the same as that provided by the 8254. I am not very familiar with Hpet, and I have not had time to study it carefully recently. Others have made a more detailed explanation of Hpet on the Internet, so you can go and have a look.

From the perspective of accuracy, the Hpet timer can certainly meet the setting of 100ns granularity. However, it still does not solve the problem of 8254 setting 100ns granularity. I guess that when using the Hpet timer, 100ns is used as the interrupt interval; when using 8254, the minimum granularity should be 10ms. This is based on the following facts:

1) During debugging, except for calling TimerDriverInitialize during startup, no breakpoints were found here. When tracing SetTimer, no functions such as TimerDriverSetTimerPeriod were called.
2) When tracing SetTimer, this code was found:

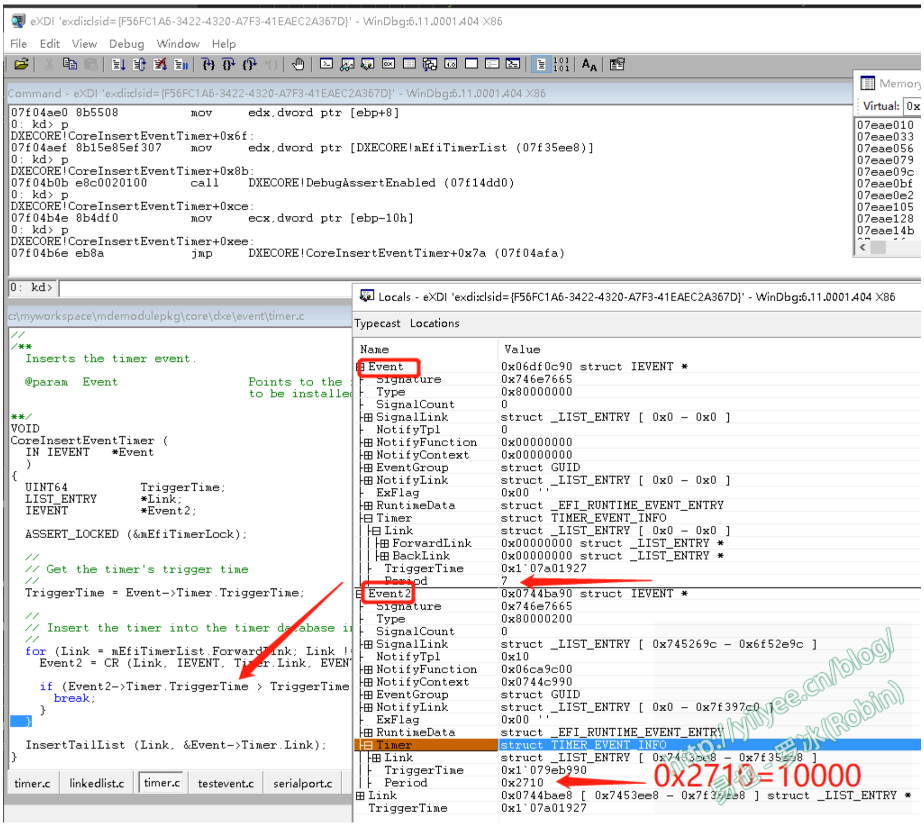

Figure 6 windbg debugging SetTimer

I deliberately set the third parameter of SetTimer to 7, and during debugging, I would compare Event2, and the Period value of Event2 was 10000. It was not added?

However, I haven't found a way to verify this idea, so I'll leave it as a problem that needs to be solved.

4 Build the Program

After tracking events for so long, I should do something with them. It's not interesting to just print characters at regular intervals, so I decided to make something more interesting.

The functions are as follows:
1) Generate random numbers as screen coordinates;
2) Set a repetitive Event to be triggered every 200ms;
3) Draw a square at the random coordinates each time it is triggered;

Then you can just daydream, empty your mind, and see when the screen fills up.

This is a useless Buddha nature program.

Starting from UEFI spec 2.4, a random number generation protocol - EFI_RNG_PROTOCOL is provided:

GUID

```
#define EFI_RNG_PROTOCOL_GUID \
{ 0x3152bca5, 0xeade, 0x433d,\
 {0x86, 0x2e, 0xc0, 0x1c, 0xdc, 0x29, 0x1f, 0x44}}
```

Protocol Interface Structure

```
typedef struct _EFI_RNG_PROTOCOL {
  EFI_RNG_GET_INFO    GetInfo
  EFI_RNG_GET_RNG     GetRNG;
} EFI_RNG_PROTOCOL;
```

Parameters

GetInfo	Returns information about the random number generation implementation.
GetRNG	Returns the next set of random numbers.

Figure 7 Protocol *for generating* random numbers

I wrote the relevant functions in the test code and planned to use them to implement random number generation. However, I found that I could not find them in the simulation environment of TianCore and OvmfPkg during the test. I suspect that it may not be implemented in the simulation environment. I don't know whether it is supported in the actual environment. I will try again later.

For now, I'll write a pseudo-random function by hand. I took the random function rand() from StdLib and modified it a little bit, so it can be used directly. You can see the code for the specific implementation.

As for the usage of Event, the previous function description has been quite detailed. In addition, you can also refer to the examples in "UEFI Principles and Programming".

The achieved effects are as follows:

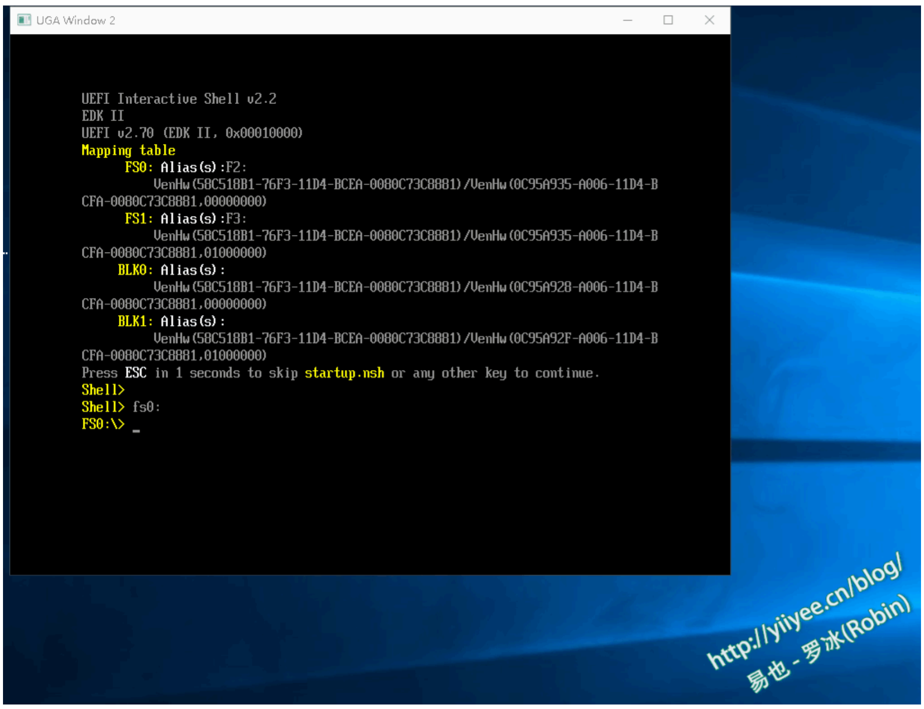

Figure 8 Using Event to achieve random drawing

Gitee address: https://gitee.com/luobing4365/uefi-explorer
The project code is located in: / FF RobinPkg/RobinPkg/Applications/RngEvent
(from this article on, all codes will be compiled in the self-made Package unless otherwise specified)