Hardware Clocks

There are three hardware clocks which the Linux kernel makes use of:

- The **Real Time Clock (RTC)** exists in all PCs and operates independently of the CPU and all other chips. It has its own power supply (a battery) and therefore continues to tick even when the computer is shut off. The RTC is read once by the kernel during system initialization to determine the current time and date. After this, the RTC is no longer used by the kernel.
- The **Time Stamp Counter (TSC)** exists as a 64-bit register in Pentium (and later) Intel 80x86 processors. This register is incremented once every cycle. Thus the faster the CPU (in terms of megahertz), the faster this clock ticks.
- The **Programmable Interval Timer (PIT)** is a device that can be programmed by the kernel to issue a *timer interrupt* after a specified length of time. It is this clock that is used to implement process scheduling.

Reading the System Time

UNIX operating systems keep time by keeping track of the number of seconds and microseconds that have passed since Midnight, January 1st, 1970.

The following code (using the system call **`gettimeofday()`**) can be used to get the current time:

```
struct timeval currentTime;
gettimeofday(&currentTime, NULL);
//where timeval  is a structure of the form:
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
}
```

- Create a C program file called **`viewtime.c`.**
- Include the system header file **`sys/time.h`** at the top of the source code.
- Declare a variable called **`currentTime`** of type **`struct timeval`**. You do not need to type the code definition of **`struct timeval`** as given above.
- Make a call to **`gettimeofday()`** as illustrated above.
- Use **`printf()`** functions to print the number of seconds and the number of microseconds that have elapsed since the epoch. Note that to print data of type **`long`**, you must use the format specifier **`"%ld"`**.
- Compile and run **`viewtime.c`**.

Using the `ctime()` function

The time given by **`gettimeofday()`** is not easy for humans to read. There exists a function called **`ctime()`** that converts the number of seconds since the epoch into a more readable string. The **`ctime()`** function takes a pointer to a **`long`**, that specifies the number of seconds that have elapsed since the epoch. It returns a string, with the date and time formatted in a human readable way.

- In your **`viewtime.c`** source code, add the header file **`time.h`**.
- Declare a string variable **`timestring`**.
- Call **`ctime()`**, using the appropriate field of **`currentTime`** as a parameter, and capture the return value in **`timestring`**.
- Use **`printf()`** to print the formatted date and time in **`timestring`**.

OS Source Code for `gettimeofday()`

In this section, you will need to look at the source code for the Linux operating system. You can find the source code files in the directory **`/usr/src/linux`**.

The **`gettimeofday()`** function is a system call. It is implemented in the kernel function **`sys_gettimeofday()`**.

- Open a bash shell window and type `sudo apt install linux-source`
  - This will download a tar file containing the source for Ubuntu.
  - The file is saved in the `usr/src` folder
  - You will need to extract the file to view the contents.
  - Alternatively, you can retrieve sample tar file from the Courses folder on the network and extract to your Desktop.
- After extracting the source code files, navigate to the directory `/linux-source-4.4.0` This is the base directory for the source code files for the Linux operating system.
- Go into the directory `kernel/time`.
- Look for a file called `time.c` and view the contents of this file.
- Search within `time.c` for the function `sys_gettimeofday()`.
- Browse the code of the `sys_gettimeofday()` to get an idea of how it works. You are not expected to understand it thoroughly.

Different computer hardware will require different machine instructions to access the hardware clock. The implementation of **`sys_gettimeofday()`** can be found in the function **`do_gettimeofday()`** in the file **`/linux-source-4.4.0/kernel/time/timekeeping.c`**.

- Go to the directory `/linux-source-4.4.0/kernel/time`
- Locate and view the file `timekeeping.c`

- Search within `timekeeping.c` for the function `do_gettimeofday()`.
- Browse the code of the `do_gettimeofday()` to get an idea of how it works. You are not expected to understand it thoroughly.

The Timer Interrupt Service Routine

Whenever the PIT issues an interrupt, the interrupt is handled by an interrupt service routine (interrupt handler) called **`timer_interrupt()`** (This can be found in **`/linux-source-4.4.0/arch/x86/kernel/time.c`**).

The major activities performed by this handler are:

- Update the time elapsed since system startup.
- Update the time and date.
- Determine how long the current process has been running on the CPU, and preempt it if it has exceeded its quantum.
- Update resource usage statistics
- Check whether the interval of time associated with each software timer has elapsed, and if so, invoke the proper function.

The first activity is urgent and is performed by the interrupt handler directly, with interrupts disabled. The other activities are executed in the so-called “bottom halves” known as **`TIMER_BH`** and **`TQUEUE_BH`**

Linux terminology: A *bottom half* is a low-priority function, usually related to interrupt handling, that is executed when the kernel finds it convenient, i.e. after the kernel has performed all urgent and critical instructions related to the interrupt handler.

`timer_interrupt()` invokes **`do_timer_interrupt()`**, which in turn invokes **`do_timer()`** which can be found in **`/usr/src/linux/kernel/timers.c`**.

The variables in **`do_timer()`** are:

- **`jiffies`**: The number of ticks that have elapsed since the system was started. Each jiffie is 1/100 of a second.
- **`lost_ticks`**: The number of ticks that have occurred since the last update of **`xtime`**
- **`lost_ticks_system`** : The number of ticks that have occurred while the process was running in Kernel Mode since the last update of **`xtime`**.

`do_timer()` increments jiffies and marks **`TIMER_BH`** for execution. When **`timer_bh()`** (the bottom half **`TIMER_BH`**) runs, it updates timers by calling **`update_process_times()`**.

For the system time, the timer bottom half uses the current value of jiffies to compute the current time. It stores the value in **`struct timeval xtime`**, where the value can be read by other kernel functions, such as **`sys_gettimeofday()`**.

- Follow the sequence of calls described above starting with **`timer_interrupt()`**.

Per Process Timers

The kernel accumulates time and manages various timers for each process. These timers are used for example by the scheduling strategy, which depends on each process having a record of the amount of CPU time that it has accrued since it acquired the CPU. Because these time values are associated with each process, they are saved in the process' descriptor.

A Linux Process Control Block (PCB)

In Linux a PCB is called a “process descriptor” and is stored in a structure called **`struct task struct`**, which is located in **`/usr/src/linux/include/linux/sched.h`**.

- Go to the directory **`/usr/src/linux/include/linux`**.
- Open the file **`sched.h`**.
- Search for the definition of **`struct task_struct`**.
- Study the **`task_struct`** structure to look for fields that you would expect to be included in a PCB.

The kernel updates the relevant fields in the descriptor in a function called **`update_process_times()`** which is in **`/usr/src/linux/kernel/timer.c`**. This function calls **`update_one_process()`** which calls other functions to update the values and to decide if a signal should be raised to indicated that a timer has expired.

- Follow the sequence of calls described starting with **`update_process_times()`**

Variables in the PCB used to keep time

In **`task_struct`**, there are six (6) **`unsigned long`** variables, **`it_real_value`**, **`it_prof_value`**, **`it_virt_value`**, **`it_real_incr`**, **`it_prof_incr`**, **`it_virt_incr`** which are known as *interval timers*.

They are actually countdown timers, that is, they can be initialized to some value and then used to reflect the passage of time by counting down toward zero.

When the timer reaches zero, it raises a signal to notify another part of the system that the counter has reached zero. Then it resets the value and begins counting down again. These timers use the kernel time to keep track of the following three intervals of time relevant to every process. (note: each type of interval is represented by a defined constant).

- **`ITIMER_REAL`**: Reflects the passage of real time and is implemented using the **`it_real_value`** and **`it_real_incr`** fields.
- **`ITIMER_VIRTUAL`**: Reflects the passage of virtual time. This time is incremented only when the corresponding process is executing. It is implemented using the

it_virt_value and **it_virt_incr** fields.

- **ITIMER_PROF**: Reflects the passage of time during which the process is active (virtual time) plus the time that the kernel is doing work on behalf of the corresponding process (for example, reading a timer). It is implemented using the **it_prof_value** and **it_prof_incr** fields.

Each timer can be initialized with the **setitimer()** system call, and can be read using the **getitimer()** call. The signatures of these two functions are respectively:

```
int setitimer(int which, struct itimerval *value, struct itimerval *ovalue)
int getitimer(int which, struct itimerval *value)
```

The structure struct **itimerval** is defined by:

```
struct itimerval {
      struct timeval it_interval;
      struct timeval it_value;
};
```

In this structure, **it_interval** stores the value that will be used to reset the timer when it expires. **it_value** contains the current value of the timer. Both of these variables are of type **struct timeval** , whose definition you have already seen in the exercise above with **gettimeofday()** .

The **which** parameter takes one of the following defined constants to indicate which timer is being referenced.

- **ITIMER_REAL**
- **ITIMER_VIRTUAL**
- **ITIMER_PROF**

The next exercise will illustrate how these functions work.

- Create a C file called **mytimer.c** .
- Include the header file **sys/time.h** .
- Declare a variable **v** of type **struct itimerval** .
- Set the **it_interval** field of **v** to 10 seconds and 0 microseconds.
- Set the **it_value** field of **v** to 10 seconds and 0 microseconds.
- Set up a real interval timer by calling **setitimer()**. Simply use **NULL** as the last parameter.
- Write a **for** loop with 1000 iterations.
- Within the **for** loop do the following:
  - Write another **for** loop to waste a bit of time by executing the null instruction 9999999 times.
  - Get the current value of the real timer that you set up by calling **getitimer()**. Use a pointer to **v** as the second parameter.

    - Using **`printf()`**, print the second and microsecond values of the current state of the timer.
- Compile and run your program **`mytimer.c`**