重庆大学

CHONGQING UNIVERSITY

操作系统课程作业

time() 系统调用的原理与应用

教师 郭尚伟

姓名 陈骁睿

学号 20225847

班级 强基物理 01

学院 物理学院

2024 年 12 月 31 日

摘　要

本文详细介绍了 Linux 内核中的 `time` 系统调用，包括其功能简介、传递参数、返回值和应用示例。通过阅读内核源码的方式，对比了现代内核中的 `time` 系统调用和 0.11 版本内核中的 `sys_time` 系统调用，分析了它们在实现方式、参数处理、错误处理、时间获取方式和向后兼容性方面的差异。

本文还深入分析了 `ktime_get_real_seconds()` 函数和 `timekeeper` 结构体的实现，并讨论了 `put_user()` 和 `force_successful_syscall_return()` 宏的功能。它们是`time` 系统调用实现中的重要组成部分。

通过这些分析，能更好地理解 Linux 内核时间管理机制的演变和实现细节。

目录

1 基本概念

1.1 硬件时间和系统时间

硬件时间是由计算机硬件提供的时间，通常由计算机的时钟芯片（如实时时钟芯片 RTC）提供。时钟芯片的频率是固定的，通常每秒产生一个时钟中断。通过这些时钟中断，计算机可以计算经过的时间。硬件时间通常在计算机启动时由操作系统读取，并用于初始化系统时间。系统时间是由操作系统维护和提供的时间。操作系统通过处理时钟中断来更新系统时间。每当时钟中断发生时，操作系统的时钟中断处理程序会增加系统时间的计数，从而保持系统时间的准确性。系统时间通常用于时间戳、调度和其他需要时间信息的操作。

1.2 UTC 时间和本地时间

UTC 时间是协调世界时（Coordinated Universal Time）的缩写，是世界标准时间。UTC 时间不受时区和夏令时的影响，是全球统一的时间标准。UTC 时间通常用于计算机系统中，以避免时区和夏令时的问题。本地时间是指当地的时间，受时区和夏令时的影响。本地时间通常是 UTC 时间加上时区和夏令时的偏移量。本地时间通常用于人类的日常生活中。

2 time 系统调用

2.1 功能概述和应用示例

功能简介：

`time` 系统调用用于获取当前系统时间。它返回一个表示当前时间的整数，通常是从 1970 年 1 月 1 日 0 时 0 分 0 秒（UNIX 纪元）开始的秒数。

传递参数：

参数 `tloc` 是一个指向 `__kernel_old_time_t` 类型的用户空间指针。如果不为 `NULL`，则系统调用会将当前时间写入该指针指向的内存位置。

返回值：

成功时，返回当前时间的秒数。如果出错，返回 `-1` 并设置 `errno` 以指示错误类型。

应用示例：

以下是一个使用 `time` 系统调用的示例代码：

```c
#include <stdio.h>
#include <time.h>

int main() {
```

```
    time_t current_time;
    current_time = time(NULL); // 获取当前时间

    if (current_time == ((time_t)-1)) {
        perror("time");
        return 1;
    }

    printf("Current time: %ld seconds since the Epoch\n", (long
       ↪ )current_time);
    return 0;
}
```

2.2 time() 源码解析

来源于 v6.12.6 内核源码 [1]，time 系统调用的实现如下：

Listing 1: time() in time.c

```

/*
 * sys_time() can be implemented in user-level using
 * sys_gettimeofday().  Is this for backwards compatibility?
    ↪ If so,
 * why not move it into the appropriate arch directory (for
    ↪ those
 * architectures that need it).
 */
SYSCALL_DEFINE1(time, __kernel_old_time_t __user *, tloc)
{
    __kernel_old_time_t i = (__kernel_old_time_t)
       ↪ ktime_get_real_seconds();

    if (tloc) {
        if (put_user(i,tloc))
            return -EFAULT;
    }
    force_successful_syscall_return();
    return i;
```

```
}
```

根据注释了解到，time 系统调用可以通过 sys_gettimeofday() 在用户空间实现。属于旧版本的系统调用，用于向后兼容。

该代码的主体较为简单，首先通过宏定义 SYSCALL_DEFINE1 定义了 time 系统调用的实现，然后调用 ktime_get_real_seconds() 函数获取当前时间，并转换为 __kernel_old_time_t 类型。如果 tloc!=NULL。则通过 put _user() 函数将其写入用户空间。若写入失败则返回 -EFAULT。最后调用 force_successful_syscall_return() 函数，返回当前时间。

接下来对 time 包含的三个函数调用进行分析。

2.3 ktime_get_real_seconds() 函数

Listing 2: ktime_get_real_seconds() in timekeeping.c

```
/**
* ktime_get_real_seconds - Get the seconds portion of
   ↪ CLOCK_REALTIME
*
* Returns the wall clock seconds since 1970.
*
* For 64bit systems the fast access to tk->xtime_sec is
   ↪ preserved. On
* 32bit systems the access must be protected with the sequence
* counter to provide "atomic" access to the 64bit tk->xtime_sec
* value.
*/
time64_t ktime_get_real_seconds(void)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    time64_t seconds;
    unsigned int seq;

    if (IS_ENABLED(CONFIG_64BIT))
        return tk->xtime_sec;

    do {
        seq = read_seqcount_begin(&tk_core.seq);
        seconds = tk->xtime_sec;
```

```
23
24      } while (read_seqcount_retry(&tk_core.seq, seq));
25
26      return seconds;
27  }
28  EXPORT_SYMBOL_GPL(ktime_get_real_seconds);
```

根据注释了解到，该函数用于获取 UTC 时间的秒数。对于 64 位系统，直接返回 `tk->xtime_sec` 的值。对于 32 位系统，需要使用序列计数器保护 `tk->xtime_sec` 的读写操作。

2.3.1 timekeeper 结构体

struct timekeeper 是 Linux 内核中用于管理时间保持功能的结构体。以下是对该结构体各个成员的详细解释：

Listing 3: timekeeper in timekeeper_internal.h

```
1  struct timekeeper {
2      struct tk_read_base     tkr_mono;
3      struct tk_read_base     tkr_raw;
4      u64                     xtime_sec;
5      unsigned long           ktime_sec;
6      struct timespec64       wall_to_monotonic;
7      ktime_t                 offs_real;
8      ktime_t                 offs_boot;
9      ktime_t                 offs_tai;
10     s32                     tai_offset;
11     unsigned int            clock_was_set_seq;
12     u8                      cs_was_changed_seq;
13     ktime_t                 next_leap_ktime;
14     u64                     raw_sec;
15     struct timespec64       monotonic_to_boot;
16
17     /* The following members are for timekeeping internal use
           ↪ */
18     u64                     cycle_interval;
19     u64                     xtime_interval;
20     s64                     xtime_remainder;
21     u64                     raw_interval;
```

```
    /* The ntp_tick_length() value currently being used.
     * This cached copy ensures we consistently apply the tick
     * length for an entire tick, as ntp_tick_length may change
     * mid-tick, and we don't want to apply that new value to
     * the tick in progress.
     */
    u64                     ntp_tick;
    /* Difference between accumulated time and NTP time in ntp
     * shifted nano seconds. */
    s64                     ntp_error;
    u32                     ntp_error_shift;
    u32                     ntp_err_mult;
    /* Flag used to avoid updating NTP twice with same second
       ↪ */
    u32                     skip_second_overflow;
};
```

- **tkr_mono 和 tkr_raw：**
  - `tkr_mono` 和 `tkr_raw` 是 `tk_read_base` 类型的结构体，用于读取单调时钟和原始时钟的基础结构。
- **xtime_sec 和 ktime_sec：**
  - `xtime_sec`：当前时间的秒部分。
  - `ktime_sec`：内核时间的秒部分。
- **wall_to_monotonic：**
  - 从墙上时间（wall time）到单调时间（monotonic time）的偏移。
- **offs_real, offs_boot 和 offs_tai：**
  - `offs_real`：实时时钟的偏移。
  - `offs_boot`：启动时间的偏移。
  - `offs_tai`：国际原子时（TAI）的偏移。
- **tai_offset：**
  - TAI 偏移量。

- **clock_was_set_seq 和 cs_was_changed_seq：**
  - `clock_was_set_seq`：时钟设置序列号。
  - `cs_was_changed_seq`：时钟源改变序列号。
- **next_leap_ktime：**
  - 下一次闰秒的时间。
- **raw_sec：**
  - 原始时间的秒部分。
- **monotonic_to_boot：**
  - 从单调时间到启动时间的偏移。
- **内部使用的成员：**
  - `cycle_interval`：周期间隔。
  - `xtime_interval`：`xtime` 的间隔。
  - `xtime_remainder`：`xtime` 的余数。
  - `raw_interval`：原始时间的间隔。
  - `ntp_tick`：当前使用的 NTP tick 长度。
  - `ntp_error`：累积时间和 NTP 时间之间的差异。
  - `ntp_error_shift`：NTP 错误移位。
  - `ntp_err_mult`：NTP 错误乘数。
  - `skip_second_overflow`：用于避免在同一秒内两次更新 NTP 的标志。

2.3.2 tk_core 变量

其中使用到了一个静态结构体变量 tk_core，相关源码如下：

```
/*
 * The most important data for readout fits into a single 64 byte
 * cache line.
 */
static struct {
    seqcount_raw_spinlock_t     seq;
    struct timekeeper           timekeeper;
```

```
} tk_core ____cacheline_aligned = {
    .seq = SEQCNT_RAW_SPINLOCK_ZERO(tk_core.seq, &
        ↪ timekeeper_lock),
};
```

该代码片段分为 4 个部分。定义并初始化了一个静态变量 tk_core，该变量包含时间保持功能的关键数据，并确保这些数据在内存中对齐到缓存行的边界，以提高访问效率。

注释部分：

这段注释说明了最重要的数据可以适应单个 64 字节的缓存行。这意味着这些数据在访问时可以更高效地利用 CPU 缓存。

结构体定义：

定义了一个匿名结构体，该结构体包含两个成员：

- seq：类型为 seqcount_raw_spinlock_t，用于实现序列计数和自旋锁。
- timekeeper：类型为 struct timekeeper，用于管理时间保持功能。

变量声明：

声明了一个名为 tk_core 的静态变量，该变量的类型为上面定义的匿名结构体。____cacheline_aligned 是一个宏，确保 tk_core 变量在内存中对齐到缓存行的边界，以提高访问效率。

变量初始化：

tk_core 变量的 seq 成员被初始化为 SEQCNT_RAW_SPINLOCK_ZERO(tk_core.seq, ↪ &timekeeper_lock)。

SEQCNT_RAW_SPINLOCK_ZERO 是一个宏，用于初始化 seqcount_raw_spinlock_t 类型的变量。它将 tk_core.seq 初始化为零，并将自旋锁设置为 timekeeper_lock。

2.4 put_user() 函数

Listing 4: put_user() in uaccess.h

```
#define put_user(x, ptr) \
    __put_user_check((__typeof__(*(ptr)))(x), (ptr), sizeof(*(
        ↪ ptr)))
/* other code .........*/
#define __put_user_check(x, ptr, size)                  \
({                                                      \
    long __pu_err = -EFAULT;                            \
    __typeof__(*(ptr)) __user *__pu_addr = (ptr);       \
```

```
    if (__access_ok(__pu_addr, size)) {                 \
        __pu_err = 0;                                   \
        switch (size) {                                 \
          case 1: __put_user_8(x, __pu_addr); break;\
          case 2: __put_user_16(x, __pu_addr); break;\
          case 4: __put_user_32(x, __pu_addr); break;\
          case 8: __put_user_64(x, __pu_addr); break;\
          default: __put_user_unknown(); break;         \
        }                                               \
    }                                                   \
    __pu_err;                                           \
})
```

这段代码定义了一个宏 `__put_user_check`，用于将数据从内核空间写入用户空间，并在写入之前进行地址空间检查。如果地址空间检查通过，则调用相应的 `__put_user_x` 函数将数据写入用户空间。如果地址空间检查失败，则返回 `-EFAULT`。

- **宏定义：**
  - `__put_user_check` 是一个宏，用于将数据 `x` 写入到用户空间指针 `ptr` 指向的地址，并根据数据的大小 `size` 选择合适的写入方法。
- **局部变量 `__pu_err`：**
  - 定义一个局部变量 `__pu_err`，初始值为 `-EFAULT`，表示默认情况下写入失败。
- **用户空间指针 `__pu_addr`：**
  - 定义一个用户空间指针 `__pu_addr`，类型为 `__typeof__(*(ptr)) __user *`，并将其初始化为 `ptr`。
- **地址空间检查：**
  - 使用 `__access_ok(__pu_addr, size)` 函数检查 `__pu_addr` 是否是有效的用户空间地址，并且大小为 `size` 的内存区域是否可访问。
  - 如果地址有效，则将 `__pu_err` 设置为 0，表示写入成功。
- **根据大小选择写入方法：**
  - 使用 `switch` 语句根据 `size` 选择合适的写入方法：
    - `case 1`：调用 `__put_user_8(x, __pu_addr)` 写入 1 字节数据。
    - `case 2`：调用 `__put_user_16(x, __pu_addr)` 写入 2 字节数据。

    - case 4：调用 __put_user_32(x, __pu_addr) 写入 4 字节数据。
    - case 8：调用 __put_user_64(x, __pu_addr) 写入 8 字节数据。
    - default：调用 __put_user_unknown() 处理未知大小的数据。

- **返回值：**
  - 宏的最后返回 __pu_err，表示写入操作的结果。如果写入成功，返回 0；如果写入失败，返回 -EFAULT。

2.5 force_successful_syscall_return() 函数

in ptrace.h

```
#define current_pt_regs() \
  ((struct pt_regs *) ((char *)current_thread_info() + 2*
    ↪ PAGE_SIZE) - 1)

#define force_successful_syscall_return() (current_pt_regs()->
  ↪ r0 = 0)
```

force_successful_syscall_return() 宏的主要功能是确保当前系统调用返回成功状态。它通过直接修改当前进程的寄存器状态，将返回值设置为 0，从而强制系统调用返回成功。这在某些情况下非常有用，例如在系统调用的实现中需要确保返回成功时，可以使用这个宏来设置返回值。

current_pt_regs() 宏：

- 该宏用于获取当前进程的寄存器状态（struct pt_regs）。
- current_thread_info() 返回当前线程的信息结构体指针。
- (char *)current_thread_info()+ 2*PAGE_SIZE 计算出寄存器状态在内存中的位置。
- (struct pt_regs *)... - 1 将计算出的地址转换为 struct pt_regs 类型，并指向正确的寄存器状态位置。

force_successful_syscall_return() 宏：

- 该宏将当前进程的寄存器 r0 设置为 0。
- current_pt_regs()->r0 = 0 表示将 r0 寄存器的值设置为 0，其中 r0 寄存器通常用于存储系统调用的返回值。
- 在许多架构中，0 表示系统调用成功。

3 旧版本对比

对比 0.11 版本的 time 系统调用源码 [2]，发现与现代版本有很大不同。

3.1 sys_time 源码

0.11 版本的 sys_time 系统调用源码如下：

Listing 5: sys_time() in sys.c

```
int sys_time(long * tloc)
{
        int i;

        i = CURRENT_TIME;
        if (tloc) {
                verify_area(tloc,4);
                put_fs_long(i,(unsigned long *)tloc);
        }
        return i;
}
```

有三个外部调用，分别是CURRENT_TIME、verify_area和put_fs_long。

CURRENT_TIME 是一个宏，用于获取当前时间。在 0.11 版本的内核中，CURRENT_TIME 宏返回一个整数，表示当前时间的秒数。功能与2 ktime_get_real_seconds() 类似。

verify_area 是一个内核函数，用于检查用户空间指针是否指向有效的内存区域。在现代内核中功能已被4 __put_user_check 包含。

put_fs_long 也是一个内核函数，用于将数据从内核空间写入用户空间。功能与4 put_user() 类似。

3.2 CURRENT_TIME 宏

Listing 6: CURRENT_TIME in sched.h

```
#define HZ 100
\*......other codes......*\
#define CURRENT_TIME (startup_time+jiffies/HZ)
```

CURRENT_TIME 是一个宏，用于获取当前时间。在 0.11 版本的内核中，CURRENT_TIME 宏返回一个整数，表示当前时间的秒数。它的计算方式是将系统启动时间（startup_time）与时钟滴答数（jiffies）相结合，除以时钟频率（HZ）得到当前时间的秒数。

其中 startup_time 由内核初始化时设置 [3]。

Listing 7: time_init() in main.c

```
1
2  #define CMOS_READ(addr) ({ \
3  outb_p(0x80|addr,0x70); \
4  inb_p(0x71); \
5  })
6
7  \*......other codes......*\
8
9  static void time_init(void)
10 {
11         struct tm time;
12
13         do {
14                 time.tm_sec = CMOS_READ(0);
15                 time.tm_min = CMOS_READ(2);
16                 time.tm_hour = CMOS_READ(4);
17                 time.tm_mday = CMOS_READ(7);
18                 time.tm_mon = CMOS_READ(8);
19                 time.tm_year = CMOS_READ(9);
20         } while (time.tm_sec != CMOS_READ(0));
21         BCD_TO_BIN(time.tm_sec);
22         BCD_TO_BIN(time.tm_min);
23         BCD_TO_BIN(time.tm_hour);
24         BCD_TO_BIN(time.tm_mday);
25         BCD_TO_BIN(time.tm_mon);
26         BCD_TO_BIN(time.tm_year);
27         time.tm_mon--;
28         startup_time = kernel_mktime(&time);
29 }
```

前面的赋值语句 `CMOS_READ` 是通过读写 CMOS 上的指定端口，依次获取年月日时分秒等信息。CMOS 是主板上的一个可读写的 RAM 芯片，其中存储了硬件时间。

接下来，`BCD_TO_BIN` 用于将 BCD 码转换成二进制数值，因为从 CMOS 获取的这些年月日都是 BCD 码，需要转换成存储在变量中的二进制数值。

最后一步，`kernel_mktime` 根据刚刚获取的时分秒数据，计算从 1970 年 1 月 1 日

0 时起到开机当时经过的秒数。

Listing 8: kernel_mktime() in mktime.c

```
1
2  long kernel_mktime(struct tm * tm)
3  {
4          long res;
5          int year;
6
7          year = tm->tm_year - 70;
8  /* magic offsets (y+1) needed to get leapyears right.*/
9          res = YEAR*year + DAY*((year+1)/4);
10         res += month[tm->tm_mon];
11 /* and (y+2) here. If it wasn't a leap-year, we have to adjust
       ↪ */
12         if (tm->tm_mon>1 && ((year+2)%4))
13                 res -= DAY;
14         res += DAY*(tm->tm_mday-1);
15         res += HOUR*tm->tm_hour;
16         res += MINUTE*tm->tm_min;
17         res += tm->tm_sec;
18         return res;
19 }
```

3.3 time 和 sys_time 的差异分析

实现方式：

- **time：** 在现代内核中，`time` 系统调用通过 `SYSCALL_DEFINE1` 宏定义，并调用 `ktime_get_real_seconds()` 函数获取当前时间的秒数。该函数直接访问内核中的时间数据结构，并返回从 1970 年 1 月 1 日 0 时起到当前时间的秒数。
- **sys_time:** 在 0.11 版本的内核中，`sys_time` 系统调用通过直接访问 `CURRENT_TIME` 宏获取当前时间。该宏计算系统启动时间和时钟滴答数的和，并除以时钟频率（HZ）得到当前时间的秒数。

参数处理：

- **time：** 接受一个指向 `__kernel_old_time_t` 类型的用户空间指针 `tloc`，如果不为 `NULL`，则将当前时间写入该指针指向的内存位置。使用 `put_user` 函数将数据从内核空间写入用户空间。

- **sys_time**：接受一个指向 long 类型的用户空间指针 tloc，如果不为 NULL，则使用 verify_area 函数检查用户空间指针是否有效，并使用 put_fs_long 函数将当前时间写入用户空间。

错误处理：

- **time**：如果写入用户空间失败，返回 -EFAULT 并设置 errno 以指示错误类型。
- **sys_time**：如果写入用户空间失败，返回 -EFAULT。

时间获取方式：

- **time**：使用 ktime_get_real_seconds() 函数获取当前时间，该函数根据系统架构（32 位或 64 位）选择合适的方式访问时间数据。
- **sys_time**：使用 CURRENT_TIME 宏获取当前时间，该宏通过计算系统启动时间和时钟滴答数的和来确定当前时间。

向后兼容性：

- **time**：现代内核中的 time 系统调用可以通过 sys_gettimeofday() 在用户空间实现，主要用于向后兼容。
- **sys_time**：在 0.11 版本的内核中，sys_time 系统调用是直接实现的，没有通过其他系统调用实现。

综上所述，time 和 sys_time 系统调用在实现方式、参数处理、错误处理、时间获取方式和向后兼容性方面存在显著差异。现代内核中的 time 系统调用更加灵活和健壮，能够适应不同的系统架构，并提供更好的错误处理机制。

4 总结

本文通过详细分析和对比，探讨了 Linux 内核中 time 系统调用的实现和演变。

现代内核中的 time 系统调用通过 SYSCALL_DEFINE1 宏定义，并调用了 ktime_get_real_seconds() 函数获取当前时间的秒数。相比之下，0.11 版本内核中的 sys_time 系统调用通过 CURRENT_TIME 宏获取当前时间。

在参数处理方面，现代内核使用 put_user 函数将数据从内核空间写入用户空间，而旧版本内核使用 verify_area 和 put_fs_long 函数。现代内核提供了更好的错误处理机制，如果写入用户空间失败，会返回 -EFAULT 并设置 errno 以指示错误类型。

时间获取方式上，现代内核的 ktime_get_real_seconds() 函数根据系统架构选择合适的方式访问时间数据，而旧版本内核通过计算系统启动时间和时钟滴答数的和来确定当前时间。

通过对比分析，本文展示了 Linux 内核时间管理机制的演变过程，现代内核中的 `time` 系统调用更加灵活和健壮，能够适应不同的系统架构，并提供更好的错误处理机制。这些改进使得时间管理在现代计算机系统中更加高效和可靠。

参考文献

[1] Linus Torvalds and contributors. Linux Kernel v6.12.6. https://cdn.kernel.org/pub/linux/kernel/v6.x/, December 2024. Accessed: 2024-12-30.

[2] Linus Torvalds and contributors. Linux Kernel v0.11. https://www.kernel.org/pub/linux/kernel/v0.1x/, September 1992. Accessed: 2024-12-30.

[3] 闪客. *Linux 源码趣读*. 电子工业出版社, September 2023.