定时器API

传统UNIX定时器

- *int settimer(int which, const struct itimerval *new_value, struct itimerval *old_value);* ⊙ 设置成功是返回0，失败则返回-1
- 系统调用settimer将创建一个定时器，当定时器在未来某个时间到期时，会向进程发送一个信号。信号的具体类型依赖于which参数
- 参数which决定创建何种类型的定时器
  - ITIMER_REAL ⊙ 创建以真实时间倒计时的定时器，到期时将会产生SIGALARM信号
  - ITIMER_VIRTUAL ⊙ 创建用户模式下的CPU时间倒计时的定时器，到期时将产生SIGALARM信号
  - ITIMER_PROF ⊙ 创建一个profiling定时器，以进程时间(用户态+内核态使用CPU时间总和)进行倒计时，到期时将产生SIGPROF信号
- 结构体itimerval ⊙

```
struct timeval {
    time_t      tv_sec;     /* 秒 */
    suseconds_t tv_usec;    /* 微秒 */
}

struct itimerval {
    struct timeval it_value;    /* 定时器到期时间 */
    struct timeval it_interval; /* 当前定时器是否为周期定时器 */
}
```

  - it_interval被设计成timeval类型，用于确定当前定时器是否为周期定时器
    - 若tv_sec或者ev_usec两个字段均为0，则为一次性定时器。当定时器到期后不再重启
    - 只要tv_sec或者ev_usec两个字段有一个不为0，则为周期定时器，间隔时间为it_value
    - 这个设计讲道理挺奇怪的，用一个short类型就能解决的问题非要用timeval类型
- **一个进程只能拥有上述3种定时器的一种，且settimer()调用无法创建出多个不同间隔的定时器，使用起来也不是很便利，所以SUSv4直接干掉了settimer()**
- alarm()调用是比settimer()更为简单的设置定时器接口，只不过其精度要比settimer()稍差
- alarm
  - *unsigned int alarm(usigned int seconds);*
    - alarm调用总是会成功，若进程之前设置过定时器，则alarm()返回定时器剩余时间，若进程未曾设置过定时器，则返回0
    - alarm()设置的定时器到期时仅产生SIGALARM信号
  - 在Linux中，alarm()与settimer()共享同一个定时器，也就是说，两者可以相互修改对方所设置的值
- 局限所在 ⊙ settimer()和alarm()的最大局限就在于同一个进程只能有一个间隔定时器，在多线程或者协程的多道程序设计中，完全排不上用场，连鸡肋都算不上

休眠一段固定时间

- 低分辨率休眠: sleep()
  - *unsigned int sleep(unsigned int seconds);*
    - 通常情况下返回为0，此时表示进程至少休眠了seconds秒
    - 如果因进程接收了到信号而中断了睡眠，sleep()将返回剩余的描述，此时程序可以接收返回值继续睡眠
  - sleep()允许进程暂停执行(睡眠)一段时间，睡眠结束后继续运行 ⊙ 睡眠期间CPU转而执行其它任务(进程or线程)
- 高分辨率休眠: nanosleep()
  - *int nanosleep(const struct timespec *request, struct timespec *remain);* ⊙ 正常睡眠结束将返回0，出现错误或者被信号中断则返回-1
  - nanosleep()调用与sleep()做的事情基本相同，但是具有更高的休眠精度 ⊙ 并且SUSv3规定，nanosleep()不得使用信号实现该函数，所以该函数可以和settimer()或alarm()混用，而不必担心可移植性问题
  - timespec ⊙

```
struct timespec {
    time_t tv_sec;   /* 秒 */
    long   tv_nsec;  /* 纳秒 */
}
```

    - 精度可至纳秒级别，取值范围为0~999999999之间

POSIX时钟

- POSIX时钟提供了纳秒级的时间精度API，包括获取当前真实时间、进程或者线程所使用的CPU时间等
- 获取时钟当前值
  - *int clock_gettime(clockid_t clockid, struct timespec *tp);*
    - 成功时返回0，调用错误则返回-1
    - 调用成功后，某个类型时钟的时间值将写入至tp所指向的timespec结构中
  - 参数clockid
    - CLOCK_REALTIME ⊙ 获取系统当前真实时间，可人为地修改
    - CLOCK_MONOTONIC ⊙ 在Linux中，获取系统启动到当前的运行时间，该值不可修改
    - CLOCK_PROCESS_CPUTIME_ID ⊙ 获取当前进程所使用的CPU时间
    - CLOCK_THREAD_CPUTIME_ID ⊙ 获取当前线程所使用的CPU时间
- 设置时钟的值
  - *int clock_settime(clockid_t clockid, const struct timespec *tp);* ⊙ 成功时返回0，调用错误则返回-1
  - 除了CLOCK_REALTIME时钟时间可由特权级进行修改外，其余类型的时钟时间在Linux中均为只读属性，不可进行修改
- 获取特定进程或线程的时钟ID
  - 如果想要做一个类似于htop的系统资源监控，并获取所有进程或者线程所消耗的CPU时间的话，可以首先获取进程或线程的时钟iID，再利用该ID获取所消耗的CPU时间
  - 获取进程CPU时钟ID ⊙ *int clock_getcpuclockid(pid_t pid, clockid_t *clockid);*
  - 获取线程CPU时钟ID ⊙ *int pthread_getcpuclockid(pthread_t pid, clockid_t *clockid);*
  - *时钟ID均存放于clockid指针所指向的缓冲区中*

```
#define _POSIX_C_SOURCE 199309
#define _XOPEN_SOURCE 600

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

int main() {
    pid_t initPid = 1;
    clockid_t initClockID;
    struct timespec initUseCpuTime;

    if (clock_getcpuclockid(initPid, &initClockID) == -1) {
        perror("get clock ID");
    }

    if (clock_gettime(initClockID, &initUseCpuTime) == -1) {
        perror("get time");
    }

    printf("init process use cpu seconds: %ld\n", initUseCpuTime.tv_sec);
    printf("init process use cpu nano seconds: %ld\n", initUseCpuTime.tv_nsec);

    return 0;
}
```

  - 获取 init 进程所使用的CPU时间

POSIX间隔定时器

- settimer()与alarm()的局限
  - 只能通过发送信号的方式通知定时器到期，并且只能使用标准信号SIGALARM或者SIGPROF，无法自定义实时信号
  - 由于使用不做排队处理的标准信号，当间隔定时器多次到期时，只会调用一次信号处理函数，无法处理定时器溢出的情况
- POSIX重新定义了一套API以及流程来突破这些限制，并明确定时器的生命周期 ⊙ 所划分的阶段为: 创建定时器并定义通知方法，启动定时器，删除不再需要的定时器
- 创建定时器
  - *int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid);* ⊙ 成功时返回0，调用错误则返回-1
  - clockid_t ⊙
    - 可以是CLOCK_REALTIME、CLOCK_MONOTONIC、CLOCK_PROCESS_CPUTIME_ID、CLOCK_THREAD_CPUTIME_ID，或者是通过clock_getcpuclockid()调用返回的进程时钟，pthread_getcpuclockid()返回的线程时钟中的任意值
  - evp
    - 参数evp将决定定时器到期时使用何种方式通知应用进程，结构体sigevent稍微有些复杂

```
union sigval
{
  int sival_int;
  void *sival_ptr;                  /* 通常将timerid的地址赋给该值 */
};


struct sigevent {
    int sigev_notify;               /* 通知方法 */
    int sigev_signo;                /* 采用信号通知时所指定的信号值，可使用实时信号 */
    union sigval sigev_value;       /* 当使用线程作为方式作为通知方式时，该结构将作通给启动的线程 */

    union {
        pid_t _tid;                 /* 通知的进程ID */

        struct {
            void (*_function) (union sigval);   /* 线程通知所调用的函数 */
            pthread_attr_t *_attribute;         /* 线程属性 */
        } _sigev_thread;
    } _sigev_un;
}

#define sigev_notify_function   _sigev_un._sigev_thread._function
#define sigev_notify_attributes _sigev_un._sigev_thread._attribute
```

    - sigevent ⊙
      - sigev_notify
        - SIGEV_NONE ⊙ 定时器到期后将不通知进程
        - SIGEV_SIGNAL ⊙ 发送用户定义的sigev_signo信号给进程
        - SIGEV_THREAD ⊙ 定时器到期时将使用_sigev_thread相关字段创建一个新的线程并执行
        - SIGEV_THREAD_ID ⊙ 定时器到期时将发送sigev_signo信号给_sigev_in._tid所标识的线程
      - **总的来说，POSIX定时器的通知方式有4种: 不通知，发送自定义信号给进程，启动一个新的线程执行，发送自定义信号给特定的线程**
  - timerid ⊙ 用于标识该定时器所用，如同pid标识一个进程一样，在后续的timer_settime()、timer_delete()中使用
- 启动/停止定时器
  - *int timer_settime(time_t timerid, int flags, const struct itimespec *value, struct itimerspec *old_time);* ⊙ 成功时返回0，调用错误则返回-1
  - timerid ⊙ 即timer_create()调用所返回的定时器句柄， 用于指代某一个定时器
  - flags ⊙ 决定当前定时器使用相对时间还是绝对时间
    - flags == 0 ⊙ 定时器的到期时间为始于timer_settime()调用时间点的相对值，较为常用
    - flags == TIMER_ABSTIME ⊙ 定时器的到期时间为从时钟0开始的绝对时间

```
struct timespec {
    time_t tv_sec;   /* 秒 */
    long   tv_nsec;  /* 纳秒 */
}

struct itimerspec {
    struct timespec it_interval;
    struct timespec it_value;
}
```

  - itimerspec ⊙
    - 该结构和itimerval非常类似，只不过内部使用的timespec时间精度至纳秒，而timeval的时间精度只能达到微秒
    - 如果想要停止一个定时器的运行，只需要将value.it_value的所有字段指定为0
- 删除定时器 ⊙ *int timer_delete(timer_t timerid);*

文件描述符定时器: timerfd API

- Linux内核提供了一种从文件描述符中读取其所创建定时器到期通知的机制，始于内核版本2.6.25，为Linux所特有，通常与select、poll或者epoll等多路复用函数共同使用
- 创建定时器
  - *int timerfd_create(int clockid, int flags);* ⊙ 调用成功时将返回指代该定时器的文件描述符，失败则返回-1
  - clockid ⊙
    - CLOCK_REALTIME
    - CLOCK_MONOTONIC
  - flags
    - TFD_CLOEXEC ⊙ 作用同open()系统调用的O_CLOEXEC位掩码
    - TFD_NONBLOCK ⊙ 时后续的读操作为非阻塞
- 启动/停止定时器
  - *int timerfd_settime(int fd, int flags, const struct itimerspec *new_value, struct itimerspec *pld_value);* ⊙ 成功时返回0，调用错误则返回-1
  - fd ⊙ timerfd_create()调用所返回的文件描述符
  - flags ⊙ 决定当前定时器使用相对时间还是绝对时间
    - flags == 0 ⊙ 定时器的到期时间为始于timerfd_settime()调用时间点的相对值
    - flags == TFD_TIMER_ABSTIME ⊙ 定时器的到期时间为从时钟0开始的绝对时间
- 删除定时器 ⊙ 由于使用文件描述符来指代定时器，所以直接调用close()即可
- 当定时器到期时，使用read()调用将正常返回，传递给read()调用的缓冲区必须足以容纳一个无符号8字节整数，用于表示定时器到期的次数