黄树超

非典型嵌入式行业从业者；C/C++，底层驱动方向

sleep(0)、usleep(0)与sched_yield() 调度

结论：

如果你是为了耗掉一个机器周期 ，那直接asm ("nop")，

如果是为了让权，建议把 所有使用 usleep(0) 换成 sched_yield() ;

最近发现很多hpc 领域的MPI程序中在用usleep(0)，比较差异。 后来问了之前做hpc 的同事 得到的答复是一般用usleep(0) 的主要目的应该是：

CPU交出当前线程的执行权，让CPU去执行其他线程。也就是放弃当前线程的时间片，转而执行其他线程

我感觉很诧异。 Usleep(0) 来做这个事情 是POSIX要求的 还是一个意外的发现呢?

于是有2个问题

1 :usleep(0) 能不能让权，

2 :如果可以，那么和sched_yield 比到底谁更合适

我先man了一下usleep(0) 在linux上，

```
NOTES
       The type useconds_t is an unsigned integer type capable of holding inte
       portable if they never mention this type explicitly.  Use

           #include <unistd.h>
           ...
               unsigned int usecs;
           ...
               usleep(usecs);
```

```
       The  interaction  of  this  function  with  the  SIGALRM  signal,  and  t
       nanosleep(2),  setitimer(2),  timer_create(2),  timer_delete(2),  timer_
       ualarm(3) is unspecified.
```

先来看几个奇怪的现象：

```
strace usleep 0
```

执行shell usleep 0 会明显的看到调用了

```
sched_yield()                           = 0
```

难道
usleep(0) = sched_yield?

而

```
strace usleep 1
```

执行shell usleep x (x!=0) 会去调用naonsleep

这就比较合理了，　　之前猜测　usleep 就应该是调用了 nanosleep，

然后写一个　c 函数调用来看看
会发现　无论是0 还是 !0 都是调用的

```
int
usleep (useconds_t useconds)
{
  struct timespec ts = { .tv_sec = (long
                         .tv_nsec = (long int) (usec

  /* Note the usleep() is a cancellation point.  But
   nanosleep() which itself is a cancellation point w
   to do anything here.  */
  return __nanosleep (&ts, NULL);
}
```

这就比较合理了，　看了glibc源码 也验证了确实是 封装naosleep

那第一个问题在linux 上就变成 naosleep(0,0) 是不是会去让权了，他和sheld_yield 的区别。

在.18 之后 应该naosleep 都是基于 hrtimer的机制实现了 (

==========

do_nanosleep(struct hrtimer_sleeper *t, enum hrtimer_mode mode)

```
1.  {
2.  hrtimer_init_sleeper(t, current);
3.  do {
4.  set_current_state(TASK_INTERRUPTIBLE);
5.  hrtimer_start_expires(&t->timer, mode);
6.  if (!hrtimer_active(&t->timer))
7.  t->task = NULL;
8.  if (likely(t->task))
9.  schedule();
10. hrtimer_cancel(&t->timer);
11. mode = HRTIMER_MODE_ABS;
12. } while (t->task && !signal_pending(current));
13. __set_current_state(TASK_RUNNING);
14. return t->task == NULL;
15. }
```

=======

补充一个　在2.6.9内核 或者可能之前的glibc实现中 usleep(0) 如果是基于 select (0) 这样的实现 在判断入参是0 之后会离开返回 不会调用 schelduer()的

```
1  int do_select(int n, fd_set_bits *fds, long *timeout)
2  {
3    poll_table table, *wait;
4    int retval, i, off;
5    long __timeout = *timeout;
6
7    read_lock(&current->files->file_lock);
8    retval = max_select_fd(n, fds);
9    read_unlock(&current->files->file_lock);
10
11   if (retval < 0)
12     return retval;
13   n = retval;
14
15   poll_initwait(&table);
16   wait = &table;
17   if (!__timeout)
18     wait = NULL;
19   retval = 0;
```

==========

)

根据nanosleep 的 syscall， 发现

很明显的有　schedule(), 于是可以确定 usleep(0) 如果一切顺利确实会让权，那么和sched_yield比呢

于是写了一个　main

```
#include <unistd.h>
#include <sched.h>
int main(){
int j ;
for(j=0; j<100000; j++)
//usleep(0);
sched_yield();
}
```

在sched_yield() 的时候 调用10万次 的耗时如下

```
root@ubuntu:/home/szx/test# time ./time

real    0m3.116s
user    0m0.020s
sys     0m1.528s
```

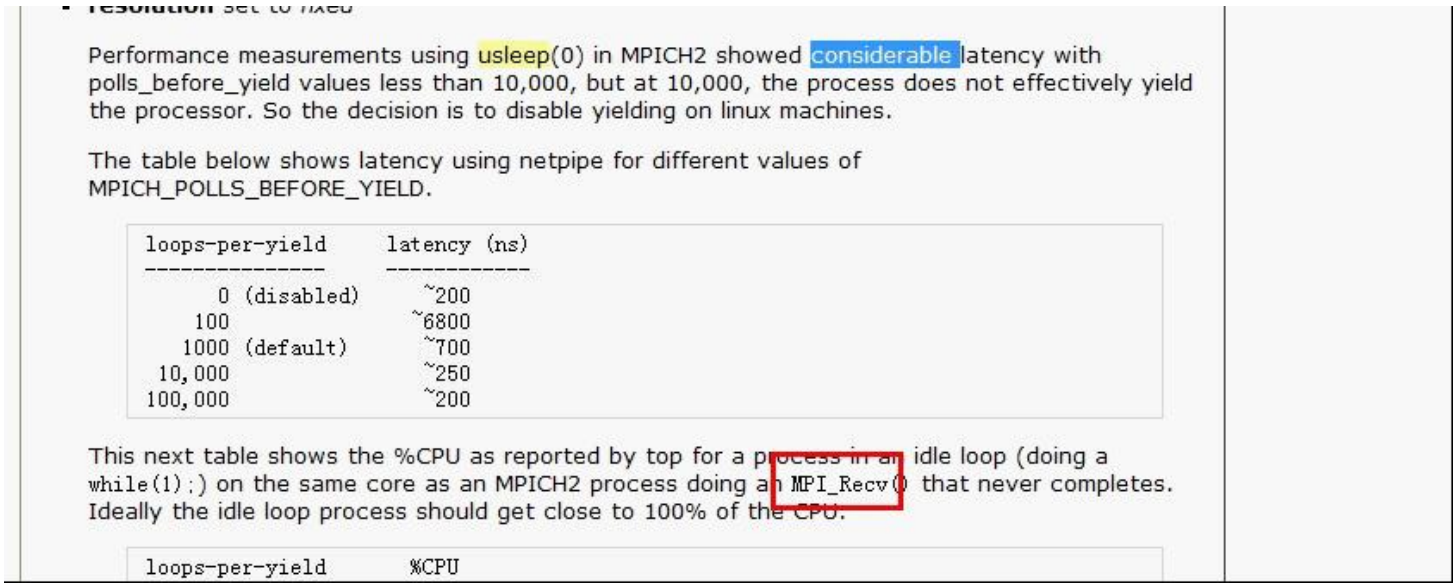
在usleep(0) 的时候 调用10万次 的耗时如下

```
root@ubuntu:/home/szx/test# time ./time

real    8m38.543s
user    0m0.436s
sys     0m11.077s
```

延迟简直不是一个数量级。。　太可怕了，如果用于网络　那要丢多少UDP， TCP要做多少次拥塞避免。

在来看一下MPI中的这个问题



Performance measurements using usleep(0) in MPICH2 showed considerable latency with polls_before_yield values less than 10,000, but at 10,000, the process does not effectively yield the processor. So the decision is to disable yielding on linux machines.

The table below shows latency using netpipe for different values of MPICH_POLLS_BEFORE_YIELD.

loops-per-yield	latency (ns)
0 (disabled)	~200
100	~6800
1000 (default)	~700
10,000	~250
100,000	~200

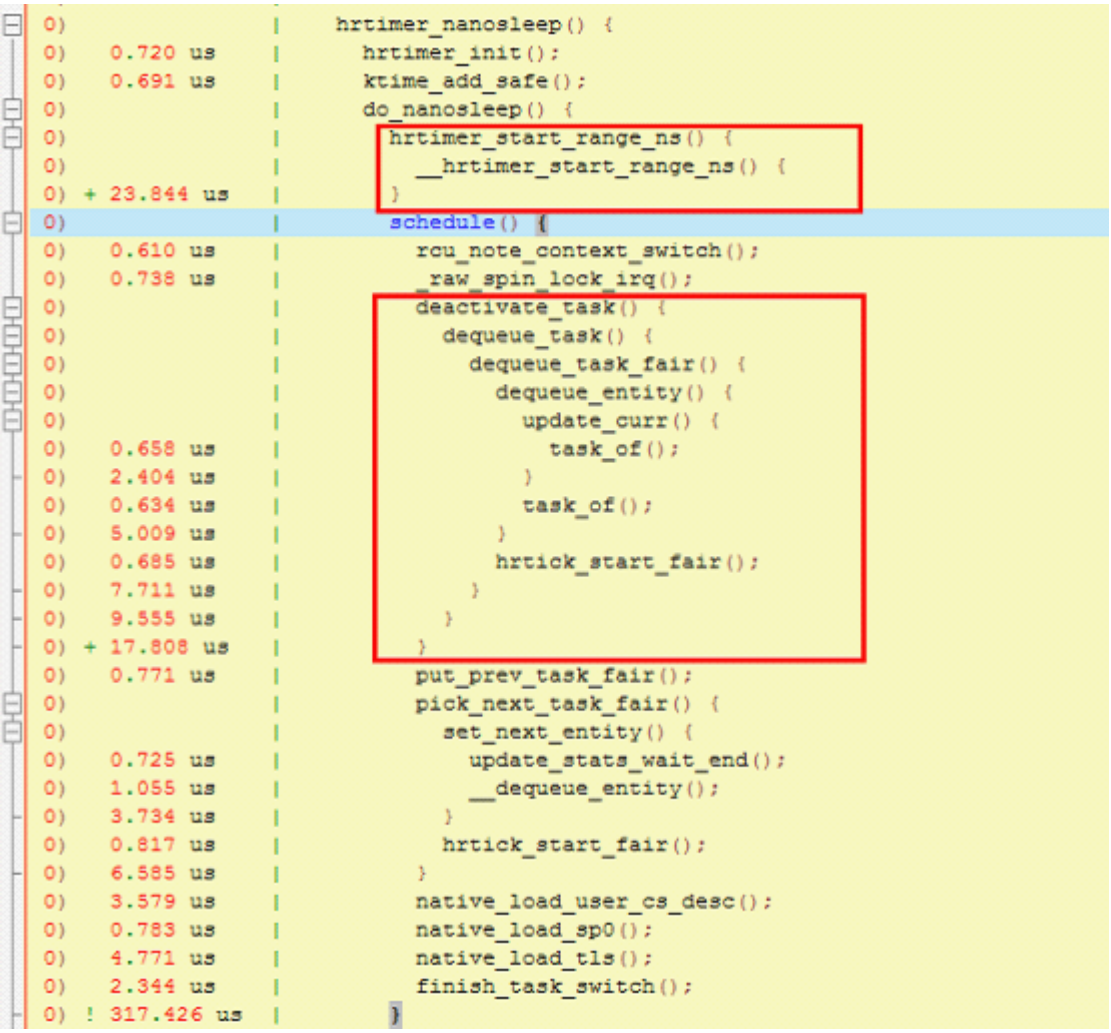
This next table shows the %CPU as reported by top for a process in an idle loop (doing a while(1);) on the same core as an MPICH2 process doing an MPI_Recv() that never completes. Ideally the idle loop process should get close to 100% of the CPU.

loops-per-yield	%CPU

http://trac.mcs.anl.gov/projects/mpich2/ticket/1597\

MPI有个Yield宏，使用了 usleep(0)，但是比较大的延迟
最后一张表的意思是，　应该尽可能的让CPU 100%，这样才算是yield。。

那为什么会造成usleep 如此延迟呢?
先看一下　　trace的信息
Usleep



非常可怕　　因为是非主动让权　调用了　deactivate_task()有简单操作系统知识的都知道简直就恶魔。。。

```
static void __sched __schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;

need_resched:
    preempt_disable();
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    rcu_note_context_switch(cpu);
    prev = rq->curr;

    schedule_debug(prev);

    if (sched_feat(HRTICK))
        hrtick_clear(rq);

    raw_spin_lock_irq(&rq->lock);

    switch_count = &prev->nivcsw;
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        if (unlikely(signal_pending_state(prev->state, prev))) {
            prev->state = TASK_RUNNING;
        } else {
            deactivate_task(rq, prev, DEQUEUE_SLEEP);
            prev->on_rq = 0;

            /*
```

然而　sched_yield()

```
 sys_sched_yield() {
0)    0.613 us    |    _raw_spin_lock();
0)                |    yield_task_fair() {
0)    0.618 us    |      update_curr();
0)    2.099 us    |    }
0)                |    schedule() {
0)    0.593 us    |      rcu_note_context_switch();
0)    0.757 us    |      _raw_spin_lock_irq();
0)                |      put_prev_task_fair() {
0)    0.595 us    |        update_curr();
0)    1.975 us    |      }
0)                |      pick_next_task_fair() {
0)                |        set_next_entity() {
0)    0.616 us    |          update_stats_wait_end();
0)    0.692 us    |          __dequeue_entity();
0)    3.144 us    |        }
0)    0.639 us    |        hrtick_start_fair();
0)    5.660 us    |      }
0) + 12.868 us    |    }
0) + 18.231 us    |  }
```

非常干净　　简直perfect!

我们知道　在hpc 领域 MPI 的终极目地 就是耗尽CPU
像usleep(0) 这么高的延迟 肯定是不能用来做让权的。 而且我也不觉得 usleep(0) 可以用在任何地方， 这是一个没保证，(你知道哪天glibc改了呢) 和极其不高效的方式 。
如果你是为了耗掉一个机器周期　，那直接asm ("nop")， 如果是为了让权建议所有使用usleep(0) (注意是0，不是其他)的地方换成 sched_yield() ;