Analysis of ACE_Task::putq with timeout=0 when queue is full on Linux platform

作者：余祖波 (livelylittlefish@gmail.com)
Blog: http://blog.csdn.net/livelylittlefish, http://www.abo321.org

content

0. Introduction

In this article, we will discuss most actions of calling ACE_Task::putq with timeout=0. It includes ACE, glibc and Linux kernel source analysis, such as ACE_Task::putq, ACE_Message_Queue::enqueue_tail, pthread_cond_wait, sys_futex and schedule.

1. ACE_Task::putq introduction

ACE_Task::putq will put a message into the message queue. It has two parameters, and the second is timeout with default value NULL, as follows. It is declared in file ace/Task_T.h as below.

```
  // For the following five method if @a timeout == 0, the caller will
  // block until action is possible, else will wait until the
  // <{absolute}> time specified in *@a timeout elapses).  These calls
  // will return, however, when queue is closed, deactivated, when a
  // signal occurs, or if the time specified in timeout elapses, (in
  // which case errno = EWOULDBLOCK).

  /// Insert message into the message queue.  Note that @a timeout uses
  /// <{absolute}> time rather than <{relative}> time.
  int putq (ACE_Message_Block *, ACE_Time_Value *timeout = 0);
```

from the comment, we can see that if timeout=0 and the queue is full, the caller will be blocked until action is possible, such as signal. Then it will NEVER return back until you kill the process.

2. call stack

When timeout=NULL, in the test, we will put many messages to fill the queue to the full, then it will be blocked and NEVER return. The call stack of it is as follows. We can see that it is blocked in __kernel_vsyscall at 0x00110402.

- pstack result

```
#0  0x00110402 in __kernel_vsyscall ()
```

```
#1  0x007065d5 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x008aa7ac in ACE_Condition_Thread_Mutex::wait ()
#3  0x008aa7eb in ACE_Condition_Thread_Mutex::wait ()
#4  0x0804b132 in ACE_Message_Queue<ACE_MT_SYNCH>::wait_not_full_cond ()
#5  0x0804d132 in ACE_Message_Queue<ACE_MT_SYNCH>::enqueue_tail ()
#6  0x0804e15a in ACE_Task<ACE_MT_SYNCH>::putq ()
...
```

- gdb bt result

```
(gdb) bt
#0  0x00110402 in __kernel_vsyscall ()
#1  0x007065d5 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
#2  0x008aa7ac in ACE_Condition_Thread_Mutex::wait () from /usr/lib/libACE.so.5.6.2
#3  0x008aa7eb in ACE_Condition_Thread_Mutex::wait () from /usr/lib/libACE.so.5.6.2
#4  0x0804b132 in ACE_Message_Queue<ACE_MT_SYNCH>::wait_not_full_cond (this=0x9d425f8, timeout=0x0)
    at /usr/include/ace/Message_Queue_T.cpp:1588
#5  0x0804d132 in ACE_Message_Queue<ACE_MT_SYNCH>::enqueue_tail (this=0x9d425f8, new_item=0xb7404588, timeout=0x0)
    at /usr/include/ace/Message_Queue_T.cpp:1753
#6  0x0804e15a in ACE_Task<ACE_MT_SYNCH>::putq (this=0x9d42594, mb=0xb7404588, tv=0x0) at
/usr/include/ace/Task_T.inl:36
...
```

3. call stack analysis

3.1 block analysis

Now, we disassemble the codes at 0x00110402 in gdb, like the following,

```
(gdb) disassemble 0x00110402
Dump of assembler code for function __kernel_vsyscall:
0x00110400 <__kernel_vsyscall+0>:       int    $0x80
0x00110402 <__kernel_vsyscall+2>:       ret    //it is blocked at 'ret', which is a return instruction
End of assembler dump.
```

or,

```
(gdb) x/i 0x00110402
0x110402 <__kernel_vsyscall+2>: ret          //it is blocked at 'ret', which is a return instruction
```

This return instruction is at <__kernel_vsyscall+2>, then, <__kernel_vsyscall> is at 0x00110400.

```
(gdb) p __kernel_vsyscall      //we can print this symbol
$1 = {<text variable, no debug info>} 0x110400 <__kernel_vsyscall>

(gdb) x/5i __kernel_vsyscall    //display 5 instructions from this symbol address
0x110400 <__kernel_vsyscall>:   int    $0x80
0x110402 <__kernel_vsyscall+2>: ret
0x110403:       nop
0x110404:       nop
0x110405:       nop

(gdb) x/5i 0x00110400           // display 5 instructions from this symbol address
0x110400 <__kernel_vsyscall>:   int    $0x80
0x110402 <__kernel_vsyscall+2>: ret
0x110403:       nop
0x110404:       nop
0x110405:       nop

(gdb) disassemble 0x110400      //disassemble codes from this address
Dump of assembler code for function __kernel_vsyscall:
0x00110400 <__kernel_vsyscall+0>:       int    $0x80
0x00110402 <__kernel_vsyscall+2>:       ret
End of assembler dump.
```

Besides this way, we can use 'disassemble' directly by default, then, it disassemble the codes surrounding the pc of the selected frame, as below. And this is the most direct way.

```
(gdb) help disassemble
Disassemble a specified section of memory.
Default is the function surrounding the pc of the selected frame.
With a single argument, the function surrounding that address is dumped.
Two arguments are taken as a range of memory to dump.
(gdb) disassemble
Dump of assembler code for function __kernel_vsyscall:
0x00110400 <__kernel_vsyscall+0>:       int    $0x80   //system call by 'operating system trap'
0x00110402 <__kernel_vsyscall+2>:       ret
End of assembler dump.
```

At last, from the disassemble code, we can see that this system call is done by interrupt with 'operating system trap' 'int $0x80'.(通过操作系统陷入进行系统调用，对本例来讲，通过 0x80 软中断实现系统调用). The system call service handler sys_futex is declared as follows.

```
asmlinkage long sys_futex(u32 __user *uaddr, int op, u32 val, struct timespec __user *utime, u32 __user *uaddr2, u32 val3)
```

then, from registers of frame 0, we can determine some parameters of this system call like the following.

```
(gdb) frame 0  //enter frame 0
#0  0x00110402 in __kernel_vsyscall ()
(gdb) info registers
eax            0xfffffe00       -512
ecx            0x80     128                                 //the second parameter op=128
edx            0x3      3
ebx            0x9d42678        164898424
esp            0xbfca1b1c       0xbfca1b1c
ebp            0xbfca1b68       0xbfca1b68
esi            0x0      0                                   //the third parameter utime=NULL
edi            0x3      3
eip            0x110402 0x110402 <__kernel_vsyscall+2>  //this instruction of the address will be executed
eflags         0x200206 [ PF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

How to know these parameters will be discussed below.
Note, each time you run this program, only %ebx, %esp, %ebp are different, other registers' value are all the same.

3.2 system call analysis

Now, we disassemble code at 0x007065d5, that is, pthread_cond_wait@@GLIBC_2.3.2, as follows.

```
(gdb) disassemble 0x007065d5
Dump of assembler code for function pthread_cond_wait@@GLIBC_2.3.2:
...
0x007065ab <pthread_cond_wait@@GLIBC_2.3.2+107>:        call   0x709060 <__pthread_enable_asynccancel>
0x007065b0 <pthread_cond_wait@@GLIBC_2.3.2+112>:        mov    %eax,(%esp)      //%eax=0, %ebx=0x862c63c, %ecx=0
0x007065b3 <pthread_cond_wait@@GLIBC_2.3.2+115>:        cmpl   $0xffffffff,0x20(%ebx)
0x007065b7 <pthread_cond_wait@@GLIBC_2.3.2+119>:        sete   %cl
0x007065ba <pthread_cond_wait@@GLIBC_2.3.2+122>:        sub    $0x1,%ecx        //%ecx=-1, that is, 0xffffffff
0x007065bd <pthread_cond_wait@@GLIBC_2.3.2+125>:        and    %gs:0x20,%ecx    //%ecx=0x80, the soft interrupt
0x007065c4 <pthread_cond_wait@@GLIBC_2.3.2+132>:        mov    %edi,%edx        //%edi=1, then %edx=1
0x007065c6 <pthread_cond_wait@@GLIBC_2.3.2+134>:        add    $0x4,%ebx        //%ebx=0x862c640
0x007065c9 <pthread_cond_wait@@GLIBC_2.3.2+137>:        mov    $0xf0,%eax     //%eax=0xf0=240=SYS_futex, the system call number
0x007065ce <pthread_cond_wait@@GLIBC_2.3.2+142>:        call   *%gs:0x10       //jump to 0x00110400 'int $0x80', use stepi to see it
```

```
0x007065d5 <pthread_cond_wait@@GLIBC_2.3.2+149>:        sub    $0x4,%ebx        //the return address of call the system-call-handler
0x007065d8 <pthread_cond_wait@@GLIBC_2.3.2+152>:        mov    (%esp),%eax
0x007065db <pthread_cond_wait@@GLIBC_2.3.2+155>:        call   0x709020 <__pthread_disable_asynccancel>
...
```

Contrast with the source code of glibc, we can find the system call is done at 0x007065ce. Wehre, %eax=0sf0=240 is the system call number.

```
      /* Enable asynchronous cancellation.  Required by the standard.  */
      cbuffer.oldtype = __pthread_enable_asynccancel ();

      /* Wait until woken by signal or broadcast.  */
      lll_futex_wait (&cond->__data.__futex, futex_val, pshared);

      /* Disable asynchronous cancellation.  */
      __pthread_disable_asynccancel (cbuffer.oldtype);
```

```
(gdb) frame 1
#1  0x007065d5 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
(gdb) info registers
eax            0xfffffe00       -512
ecx            0x80     128
edx            0x1      1
ebx            0x93b7678        154891896
esp            0xbfd55c90       0xbfd55c90
ebp            0xbfd55cd8       0xbfd55cd8
esi            0x0      0
edi            0x1      1
eip            0x7065d5 0x7065d5 <pthread_cond_wait@@GLIBC_2.3.2+149>   //this instruction will be executed
eflags         0x200206 [ PF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

if want to get all parameters passed to system call service handler (系统调用服务程序), it is necessary to arrive at the codes before call 'int $0x80'. Then, set a breakpoint at 0x007065ce, and run to this breadpoint to watch all registers.

```
(gdb) b *0x007065ce     //set a break point at the assemble code
Breakpoint r at 0x7065ce
(gdb) r
...
(gdb) info registers
eax            0xf0     240                    //the interrupt number SYS_futex=240, is passed by %eax
ecx            0x80     128                    //the second parameter op=128
edx            0x1      1                      //_val=1
ebx            0x83c9640        138188352      //the first parameter uaddr=0x83c9640
esp            0xb7f74200       0xb7f74200
ebp            0xb7f74248       0xb7f74248
esi            0x0      0                      //the third parameter utime=NULL
edi            0x1      1
eip            0x7065ce 0x7065ce <pthread_cond_wait@@GLIBC_2.3.2+142>
eflags         0x200212 [ AF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) stepi
0x00110400 in __kernel_vsyscall ()             //int $0x80
```

```
(gdb) info registers
eax            0xf0     240                    //the interrupt number SYS_futex=240, is passed by %eax
ecx            0x80     128                    //the second parameter op=128
edx            0x1      1                      //_val=1
ebx            0x83c9640        138188352      //the first parameter uaddr=0x83c9640
esp            0xb7f741fc       0xb7f741fc
ebp            0xb7f74248       0xb7f74248
esi            0x0      0                      //the third parameter utime=NULL
edi            0x1      1
eip            0x110400 0x110400 <__kernel_vsyscall>  //will execute 'int $0x80'
eflags         0x200212 [ AF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

About this system call, it will be detailed discussed later again.

4. analysis of ACE_Task::putq

4.1 the call process of ACE_Task::putq

```
 ACE_Task<ACE_SYNCH_USE>::putq                          <-----------    //timeout=0
->ACE_Message_Queue<ACE_SYNCH_USE>::enqueue_tail           |       |
 ->ACE_Message_Queue<ACE_MT_SYNCH>::wait_not_full_cond   ACE     |
 |->ACE_Condition_Thread_Mutex::wait                        |      |
 | ->ACE_Condition_Thread_Mutex::wait                      |    user space
 |  ->ACE_OS::cond_timedwait                            <---      |
 |   ->pthread_cond_wait                                  |       |    //pthread_cond_wait@@GLIBC_2.3.2,
__pthread_cond_wait
 |    ->lll_futex_wait                                   glibc    |    //macros definition
 |     ->lll_futex_timed_wait                             |       |    //macros definition
 |      ->sys_futex                                     <---------------system call interface
 |       ->do_futex                                       |       |    //from here, timeout=2147483647=LONG_MAX
 |        ->futex_wait                                   kernel kernel space
 |         ->schedule_timeout                             |       |
 |          ->schedule                                  <-----------    //current process will sleep
 -> ACE_Message_Queue<ACE_MT_SYNCH>::enqueue_tail_i       |
```

4.2 each function analysis on the call path

(1) ACE_Task<ACE_SYNCH_USE>::putq

```
//file: ace/Task_T.inl

template <ACE_SYNCH_DECL> ACE_INLINE int
ACE_Task<ACE_SYNCH_USE>::putq (ACE_Message_Block *mb, ACE_Time_Value *tv)  //default value of tv is 0
{
  ACE_TRACE ("ACE_Task<ACE_SYNCH_USE>::putq");
  return this->msg_queue_->enqueue_tail (mb, tv);  //msg_queue_ is the data member of ACE_Task
}
```

(2) ACE_Message_Queue<ACE_SYNCH_USE>::enqueue_tail

```
//file: ace/Message_Queue.h

  /**
   * Enqueue one or more ACE_Message_Block objects at the tail of the queue.
   * If the @a new_item @c next() pointer is non-zero, it is assumed to be the
   * start of a series of ACE_Message_Block objects connected via their
   * @c next() pointers. The series of blocks will be added to the queue in
   * the same order they are passed in as.
```

```
   *
   * @param new_item Pointer to an ACE_Message_Block that will be
   *                 added to the queue. If the block's @c next() pointer
   *                 is non-zero, all blocks chained from the @c next()
   *                 pointer are enqueued as well.
   * @param timeout  The absolute time the caller will wait until
   *                 for the block to be queued.
   *
   * @retval >0 The number of ACE_Message_Blocks on the queue after adding
   *             the specified block(s).
   * @retval -1 On failure.  errno holds the reason. Common errno values are:
   *            - EWOULDBLOCK: the timeout elapsed
   *            - ESHUTDOWN: the queue was deactivated or pulsed
   */
  virtual int enqueue_tail (ACE_Message_Block *new_item,
                            ACE_Time_Value *timeout = 0);

//file: ace/Message_Queue.cpp

// Block indefinitely waiting for an item to arrive,
// does not ignore alerts (e.g., signals).
template <ACE_SYNCH_DECL> int
ACE_Message_Queue<ACE_SYNCH_USE>::enqueue_tail (ACE_Message_Block *new_item,
                                                ACE_Time_Value *timeout)
{
  ACE_TRACE ("ACE_Message_Queue<ACE_SYNCH_USE>::enqueue_tail");
  int queue_count = 0;
  {
    ACE_GUARD_RETURN (ACE_SYNCH_MUTEX_T, ace_mon, this->lock_, -1);

    if (this->state_ == ACE_Message_Queue_Base::DEACTIVATED)
      {
        errno = ESHUTDOWN;
        return -1;
      }

    if (this->wait_not_full_cond (ace_mon, timeout) == -1)  //wait not full condition
      return -1;

    queue_count = this->enqueue_tail_i (new_item);

    if (queue_count == -1)
      return -1;

    this->notify ();
  }
  return queue_count;
}
```

(3) ACE_Message_Queue<ACE_MT_SYNCH>::wait_not_full_cond

```
//file: ace/Message_Queue.cpp

template <ACE_SYNCH_DECL> int
ACE_Message_Queue<ACE_SYNCH_USE>::wait_not_full_cond (ACE_Guard<ACE_SYNCH_MUTEX_T> &,
                                                      ACE_Time_Value *timeout)
{
  int result = 0;

  // Wait while the queue is full.

  while (this->is_full_i ())  //if it is full, do the while-loop
    {
      if (this->not_full_cond_.wait (timeout) == -1)
        {
```

```
          if (errno == ETIME)      //if time expires, wait will return -1 with errno=ETIME=62; if parameter timeout=0, no return, block it
            errno = EWOULDBLOCK;   //if time expires, errno will carry EWOULDBLOCK=11
          result = -1;
          break;                   //if time expires, break the loop and return -1
        }
      if (this->state_ != ACE_Message_Queue_Base::ACTIVATED)
        {
          errno = ESHUTDOWN;
          result = -1;
          break;
        }
    }
  return result;
}
```

Where, is_full_i is a inline function as follows, to judge whether the message queue is full or not.

```
template <ACE_SYNCH_DECL> bool
ACE_Message_Queue<ACE_SYNCH_USE>::is_full_i (void)
{
  ACE_TRACE ("ACE_Message_Queue<ACE_SYNCH_USE>::is_full_i");
  return this->cur_bytes_ >= this->high_water_mark_;  //default value of high_water_mark_ is 16K=16384B
}
```

And where, EWOULDBLOCK is defined in Linux as follows.

- EWOULDBLOCK definition

```
//file: linux/include/asm-generic/errno.h
#define EWOULDBLOCK EAGAIN  /* Operation would block */
```

- EAGAIN definition

```
//file: linux/include/asm-generic/errno-base.h
#define EAGAIN      11      /* Try again */
```

- Other macros

Other macros such as ETIME, ETIMEDOUT are defined in Linux too.

```
//file: linux/include/asm-generic/errno.h
#define ETIME       62  /* Timer expired */
#define ETIMEDOUT   110 /* Connection timed out */
#define ESHUTDOWN   108 /* Cannot send after transport endpoint shutdown */
```

(4) ACE_Condition_Thread_Mutex::wait(const ACE_Time_Value *abstime)

```
//file: ace/Condition_Thread_Mutext.h
  /**
   * Block on condition, or until absolute time-of-day has passed.  If
   * abstime == 0 use "blocking" <wait> semantics.  Else, if @a abstime
   * != 0 and the call times out before the condition is signaled
   * <wait> returns -1 and sets errno to ETIME.
   */
  int wait (const ACE_Time_Value *abstime);

//file: ace/Condition_Thread_Mutex.cpp

int
ACE_Condition_Thread_Mutex::wait (const ACE_Time_Value *abstime)
{
// ACE_TRACE ("ACE_Condition_Thread_Mutex::wait");
  return this->wait (this->mutex_, abstime);
}
```

(5) ACE_Condition_Thread_Mutex::wait(ACE_Thread_Mutex &mutex, const ACE_Time_Value *abstime)

```
//file: ace/Condition_Thread_Mutext.h
  /**
   * Block on condition or until absolute time-of-day has passed.  If
   * abstime == 0 use "blocking" wait() semantics on the <mutex>
   * passed as a parameter (this is useful if you need to store the
   * <Condition> in shared memory).  Else, if @a abstime != 0 and the
   * call times out before the condition is signaled <wait> returns -1
   * and sets errno to ETIME.
   */
  int wait (ACE_Thread_Mutex &mutex, const ACE_Time_Value *abstime = 0);

//file: ace/Condition_Thread_Mutex.cpp

int
ACE_Condition_Thread_Mutex::wait (ACE_Thread_Mutex &mutex,
                                  const ACE_Time_Value *abstime)
{
// ACE_TRACE ("ACE_Condition_Thread_Mutex::wait");
  return ACE_OS::cond_timedwait (&this->cond_,
                                 &mutex.lock_,
                                 const_cast <ACE_Time_Value *> (abstime));
}
```

(6) ACE_OS::cond_timedwait

```
//file: ace/OS_NS_Thread.inl

ACE_INLINE int
ACE_OS::cond_timedwait (ACE_cond_t *cv,
                        ACE_mutex_t *external_mutex,
                        ACE_Time_Value *timeout)
{
  ACE_OS_TRACE ("ACE_OS::cond_timedwait");
# if defined (ACE_HAS_THREADS)
  int result;
  timespec_t ts;

  if (timeout != 0)
    ts = *timeout; // Calls ACE_Time_Value::operator timespec_t().

#   if defined (ACE_HAS_PTHREADS)

  ACE_OSCALL (ACE_ADAPT_RETVAL (timeout == 0
                                ? pthread_cond_wait (cv, external_mutex)  //timeout=0, call ptheread_cond_wait
                                : pthread_cond_timedwait (cv, external_mutex, (ACE_TIMESPEC_PTR) &ts),  //timeout!=0, call
pthread_cond_timedwait
                                result),
              int, -1, result);
  // We need to adjust this to make the POSIX and Solaris return
  // values consistent.  EAGAIN is from Pthreads DRAFT4 (HP-UX 10.20 and down)
  if (result == -1 && (errno == ETIMEDOUT || errno == EAGAIN))  //ETIMEDOUT=110, EAGAIN=11
    errno = ETIME;                                              //ETIME=62

#   elif defined (ACE_HAS_STHREADS)
  ACE_OSCALL (ACE_ADAPT_RETVAL (timeout == 0
                                ? ::cond_wait (cv, external_mutex)
                                : ::cond_timedwait (cv,
                                                    external_mutex,
                                                    (timestruc_t*)&ts),
                                result),
              int, -1, result);
#   endif /* ACE_HAS_STHREADS */
  if (timeout != 0)
```

```
    timeout->set (ts); // Update the time value before returning.

  return result;
# else
  ACE_UNUSED_ARG (cv);
  ACE_UNUSED_ARG (external_mutex);
  ACE_UNUSED_ARG (timeout);
  ACE_NOTSUP_RETURN (-1);
# endif /* ACE_HAS_THREADS */
}
```

Where, for function pthread_cond_wait, its versionded symbol is pthread_cond_wait@@GLIBC_2.3.2, which is a global symbol. And its local symbol is __pthread_cond_wait, its entry point is 0x00706540, listed as follows.

```
# nm /lib/libpthread.so.0 | grep wait
0070a6e0 t __libc_sigwait
0070a370 t __libc_wait
0070a430 t __libc_waitpid
007090f0 t __lll_lock_wait
007090c0 t __lll_lock_wait_private
007092d0 t __lll_robust_lock_wait
00709330 t __lll_robust_timedlock_wait
00709130 t __lll_timedlock_wait
00709250 t __lll_timedwait_tid
00708930 t __new_sem_trywait
00708820 t __new_sem_wait
00708930 t __old_sem_trywait
007088c0 t __old_sem_wait
00706810 t __pthread_cond_timedwait
00706e20 t __pthread_cond_timedwait_2_0
00706540 t __pthread_cond_wait                  //t/T is in the text (code) section, lower case represents local symbol
00706db0 t __pthread_cond_wait_2_0
0070a6e0 t __sigwait
0070a370 W __wait
007019b0 t __wait_lookup_done
0070a430 t __waitpid
0070a43a t __waitpid_nocancel
0070a640 t do_sigwait
00707190 T pthread_barrier_wait
00706810 T pthread_cond_timedwait@@GLIBC_2.3.2
00706e20 T pthread_cond_timedwait@GLIBC_2.0
00706540 T pthread_cond_wait@@GLIBC_2.3.2      //upper case represents global (external) symbol
00706db0 T pthread_cond_wait@GLIBC_2.0
00708970 T sem_timedwait
00708930 T sem_trywait@@GLIBC_2.1
00708930 T sem_trywait@GLIBC_2.0
00708820 T sem_wait@@GLIBC_2.1
007088c0 T sem_wait@GLIBC_2.0
007088ac t sem_wait_cleanup
00708a57 t sem_wait_cleanup
0070a6e0 W sigwait
0070a370 W wait
0070a430 W waitpid
```

the following sentence is from ‘man nm’.

If lowercase, the symbol is local; if uppercase, the symbol is global (external). (reference: man nm)

By objdump -x, we can see this clearly.

```
# objdump -x /lib/libpthread.so.0|grep wait
00000000 l    df *ABS*  00000000              old_pthread_cond_wait.c
00000000 l    df *ABS*  00000000              old_pthread_cond_timedwait.c
007088ac l     F .text  0000000d              sem_wait_cleanup
00708a57 l     F .text  0000000d              sem_wait_cleanup
```

```
00000000 l    df *ABS*  00000000              wait.c
00000000 l    df *ABS*  00000000              sigwait.c
0070a640 l     F .text  0000009e              do_sigwait
007092d0 l     F .text  00000056              .hidden __lll_robust_lock_wait
007019b0 l     F .text  00000147              .hidden __wait_lookup_done
007090f0 l     F .text  00000035              .hidden __lll_lock_wait
00706540 l     F .text  000001cd              __pthread_cond_wait                    //'l' represents 'local'
007088c0 l     F .text  0000006e              __old_sem_wait
0070a6e0 l     F .text  00000055              __sigwait
00708930 l     F .text  00000032              __new_sem_trywait
0070a430 l     F .text  0000007a              __libc_waitpid
00709250 l     F .text  0000007a              .hidden __lll_timedwait_tid
0070a370 l     F .text  000000b6              __libc_wait
00706810 l     F .text  00000257              __pthread_cond_timedwait
0070a430 l     F .text  0000007a              __waitpid
00708930 l     F .text  00000032              __old_sem_trywait
00709130 l     F .text  000000b7              .hidden __lll_timedlock_wait
0070a43a l     F .text  00000022              __waitpid_nocancel
00706e20 l     F .text  0000007a              __pthread_cond_timedwait_2_0
007090c0 l     F .text  0000002f              .hidden __lll_lock_wait_private
0070a6e0 l     F .text  00000055              __libc_sigwait
00706db0 l     F .text  0000006c              __pthread_cond_wait_2_0
00709330 l     F .text  000000d3              .hidden __lll_robust_timedlock_wait
00708820 l     F .text  0000008c              __new_sem_wait
007088c0 g     F .text  0000006e              sem_wait@GLIBC_2.0
00708930 g     F .text  00000032              sem_trywait@@GLIBC_2.1
00708820 g     F .text  0000008c              sem_wait@@GLIBC_2.1
00706810 g     F .text  00000257              pthread_cond_timedwait@@GLIBC_2.3.2    //'g' represents 'global'
00706e20 g     F .text  0000007a              pthread_cond_timedwait@GLIBC_2.0
00708930 g     F .text  00000032              sem_trywait@GLIBC_2.0
0070a370  w    F .text  000000b6              wait
00708970 g     F .text  000000e7              sem_timedwait
0070a370  w    F .text  000000b6              __wait
00707190 g     F .text  000000f5              pthread_barrier_wait
0070a6e0  w    F .text  00000055              sigwait
00706540 g     F .text  000001cd              pthread_cond_wait@@GLIBC_2.3.2
0070a430  w    F .text  0000007a              waitpid
00706db0 g     F .text  0000006c              pthread_cond_wait@GLIBC_2.0
```

We can verify function __pthread_cond_wait is from 0x00706540 as follows.

```
(gdb) disassemble 0x007065d5
Dump of assembler code for function pthread_cond_wait@@GLIBC_2.3.2:
0x00706540 <pthread_cond_wait@@GLIBC_2.3.2+0>:  push   %edi    //the start address of
pthread_cond_wait@@GLIBC_2.3.2 is 0x00706540
0x00706541 <pthread_cond_wait@@GLIBC_2.3.2+1>:  push   %esi
0x00706542 <pthread_cond_wait@@GLIBC_2.3.2+2>:  push   %ebx
0x00706543 <pthread_cond_wait@@GLIBC_2.3.2+3>:  xor    %esi,%esi
0x00706545 <pthread_cond_wait@@GLIBC_2.3.2+5>:  mov    0x10(%esp),%ebx
0x00706549 <pthread_cond_wait@@GLIBC_2.3.2+9>:  mov    $0x1,%edx
0x0070654e <pthread_cond_wait@@GLIBC_2.3.2+14>: xor    %eax,%eax
0x00706550 <pthread_cond_wait@@GLIBC_2.3.2+16>: lock cmpxchg %edx,(%ebx)
...
0x007065ab <pthread_cond_wait@@GLIBC_2.3.2+107>:        call   0x709060 <__pthread_enable_asynccancel>
0x007065b0 <pthread_cond_wait@@GLIBC_2.3.2+112>:        mov    %eax,(%esp)
0x007065b3 <pthread_cond_wait@@GLIBC_2.3.2+115>:        cmpl   $0xffffffff,0x20(%ebx)
0x007065b7 <pthread_cond_wait@@GLIBC_2.3.2+119>:        sete   %cl
0x007065ba <pthread_cond_wait@@GLIBC_2.3.2+122>:        sub    $0x1,%ecx
0x007065bd <pthread_cond_wait@@GLIBC_2.3.2+125>:        and    %gs:0x20,%ecx
0x007065c4 <pthread_cond_wait@@GLIBC_2.3.2+132>:        mov    %edi,%edx
0x007065c6 <pthread_cond_wait@@GLIBC_2.3.2+134>:        add    $0x4,%ebx
0x007065c9 <pthread_cond_wait@@GLIBC_2.3.2+137>:        mov    $0xf0,%eax
0x007065ce <pthread_cond_wait@@GLIBC_2.3.2+142>:        call   *%gs:0x10
0x007065d5 <pthread_cond_wait@@GLIBC_2.3.2+149>:        sub    $0x4,%ebx
0x007065d8 <pthread_cond_wait@@GLIBC_2.3.2+152>:        mov    (%esp),%eax
```

```
0x007065db <pthread_cond_wait@@GLIBC_2.3.2+155>:        call   0x709020 <__pthread_disable_asynccancel>
...
```

(7) __pthread_cond_wait

//file: glibc/nptl/pthread_cond_wait.c

```
int
__pthread_cond_wait (cond, mutex)
     pthread_cond_t *cond;
     pthread_mutex_t *mutex;
{
  struct _pthread_cleanup_buffer buffer;
  struct _condvar_cleanup_buffer cbuffer;
  int err;
  int pshared = (cond->__data.__mutex == (void *) ~0l)
       ? LLL_SHARED : LLL_PRIVATE;

  /* Make sure we are along.  */
  lll_lock (cond->__data.__lock, pshared);

  /* Now we can release the mutex.  */
  err = __pthread_mutex_unlock_usercnt (mutex, 0);
  if (__builtin_expect (err, 0))
    {
      lll_unlock (cond->__data.__lock, pshared);
      return err;
    }

  /* We have one new user of the condvar.  */
  ++cond->__data.__total_seq;
  ++cond->__data.__futex;
  cond->__data.__nwaiters += 1 << COND_NWAITERS_SHIFT;

  /* Remember the mutex we are using here.  If there is already a
     different address store this is a bad user bug.  Do not store
     anything for pshared condvars.  */
  if (cond->__data.__mutex != (void *) ~0l)
    cond->__data.__mutex = mutex;

  /* Prepare structure passed to cancellation handler.  */
  cbuffer.cond = cond;
  cbuffer.mutex = mutex;

  /* Before we block we enable cancellation.  Therefore we have to
     install a cancellation handler.  */
  __pthread_cleanup_push (&buffer, __condvar_cleanup, &cbuffer);

  /* The current values of the wakeup counter.  The "woken" counter
     must exceed this value.  */
  unsigned long long int val;
  unsigned long long int seq;
  val = seq = cond->__data.__wakeup_seq;
  /* Remember the broadcast counter.  */
  cbuffer.bc_seq = cond->__data.__broadcast_seq;

  do
    {
      unsigned int futex_val = cond->__data.__futex;

      /* Prepare to wait.  Release the condvar futex.  */
      lll_unlock (cond->__data.__lock, pshared);

      /* Enable asynchronous cancellation.  Required by the standard.  */
      cbuffer.oldtype = __pthread_enable_asynccancel ();
```

```
      /* Wait until woken by signal or broadcast.  */
      lll_futex_wait (&cond->__data.__futex, futex_val, pshared);    //macro

      /* Disable asynchronous cancellation.  */
      __pthread_disable_asynccancel (cbuffer.oldtype);

      /* We are going to look at shared data again, so get the lock.  */
      lll_lock (cond->__data.__lock, pshared);

      /* If a broadcast happened, we are done.  */
      if (cbuffer.bc_seq != cond->__data.__broadcast_seq)
    goto bc_out;

      /* Check whether we are eligible for wakeup.  */
      val = cond->__data.__wakeup_seq;
    }
  while (val == seq || cond->__data.__woken_seq == val);

  /* Another thread woken up.  */
  ++cond->__data.__woken_seq;

 bc_out:

  cond->__data.__nwaiters -= 1 << COND_NWAITERS_SHIFT;

  /* If pthread_cond_destroy was called on this varaible already,
     notify the pthread_cond_destroy caller all waiters have left
     and it can be successfully destroyed.  */
  if (cond->__data.__total_seq == -1ULL
      && cond->__data.__nwaiters < (1 << COND_NWAITERS_SHIFT))
    lll_futex_wake (&cond->__data.__nwaiters, 1, pshared);

  /* We are done with the condvar.  */
  lll_unlock (cond->__data.__lock, pshared);

  /* The cancellation handling is back to normal, remove the handler.  */
  __pthread_cleanup_pop (&buffer, 0);

  /* Get the mutex before returning.  */
  return __pthread_mutex_cond_lock (mutex);
}

versioned_symbol (libpthread, __pthread_cond_wait, pthread_cond_wait,
          GLIBC_2_3_2);
```

Versioned symbol will be discussed in another article.

(8) lll_futex_wait, lll_futex_timed_wait

- file: glibc/nptl/sysdeps/unix/sysv/linux/i386/lowlevellock.h

```
#define SYS_futex       240
#define FUTEX_WAIT      0
#define FUTEX_WAKE      1
#define FUTEX_CMP_REQUEUE   4
#define FUTEX_WAKE_OP       5
#define FUTEX_LOCK_PI       6
#define FUTEX_UNLOCK_PI     7
#define FUTEX_TRYLOCK_PI    8
#define FUTEX_PRIVATE_FLAG  128

#define lll_futex_wait(futex, val, private) \
  lll_futex_timed_wait (futex, val, NULL, private)  //timeout=NULL
```

```
#define lll_futex_timed_wait(futex, val, timeout, private) \
  ({                                                       \
    int __status;                                          \
    register __typeof (val) _val asm ("edx") = (val);      \
    __asm __volatile (LLL_EBX_LOAD                         \
              LLL_ENTER_KERNEL                             \
              LLL_EBX_LOAD                                 \
              : "=a" (__status)                            \
              : "0" (SYS_futex), LLL_EBX_REG (futex), "S" (timeout),\   //timeout will be passed by %esi
                "c" (__lll_private_flag (FUTEX_WAIT, private)),          //__lll_private_flag result will be passed
by %ecx
                "d" (_val), "i" (offsetof (tcbhead_t, sysinfo)) \         //_val will be passed by %edx
              : "memory");                                 \
    __status;                                              \
  })
```

0 indicates it is %eax, the NO. register, and SYS_futex=240 will be passed by %eax;
offsetof result will be passed by %esi;
inlie asm will be discussed in another article.

Where,

```
#ifdef PIC
# define LLL_EBX_LOAD   "xchgl %2, %%ebx\n"
# define LLL_EBX_REG    "D"
#else
# define LLL_EBX_LOAD
# define LLL_EBX_REG    "b"     //b represents %ebx
#endif

#ifdef I386_USE_SYSENTER
# ifdef SHARED
#  define LLL_ENTER_KERNEL  "call *%%gs:%P6\n\t"
# else
#  define LLL_ENTER_KERNEL  "call *_dl_sysinfo\n\t"
# endif
#else
# define LLL_ENTER_KERNEL   "int $0x80\n\t"
#endif
```

- file: glibc/sysdeps/unix/sysv/linux/i386/sysdep.h

```
#if defined USE_DL_SYSINFO \
    && (!defined NOT_IN_libc || defined IS_IN_libpthread)
# define I386_USE_SYSENTER  1
#else
# undef I386_USE_SYSENTER
#endif
```

These macros are defined in the same file as SYS_futex and lll_futex_wait mentioned above.
Where, 系统调用的执行需要从 user space 转换到 kernel space, 不同的平台用不同的指令来完成这种转换，这样的指令也被称为操作系统陷入(operation system trap)指令。
On Linux platform, it use soft interrupt to implement this trap. Concretely, for x86, it uses **soft interrupt 0x80**, that is, instruction '**int $0x80**'.

How on earth is done system calls? Which will be discussed in another article.

```
#if !defined NOT_IN_libc || defined IS_IN_rtld
/* In libc.so or ld.so all futexes are private.  */
# ifdef __ASSUME_PRIVATE_FUTEX
#  define __lll_private_flag(fl, private) \
  ((fl) | FUTEX_PRIVATE_FLAG)          //FUTEX_WAIT | FUTEX_PRIVATE_FLAG = 0 | 128 = 128
# else
```

```
#  define __lll_private_flag(fl, private) \
  ((fl) | THREAD_GETMEM (THREAD_SELF, header.private_futex))
# endif
#else
# ifdef __ASSUME_PRIVATE_FUTEX
#  define __lll_private_flag(fl, private) \
  (((fl) | FUTEX_PRIVATE_FLAG) ^ (private))
# else
#  define __lll_private_flag(fl, private) \
  (__builtin_constant_p (private)                          \
   ? ((private) == 0                                   \
      ? ((fl) | THREAD_GETMEM (THREAD_SELF, header.private_futex))       \
      : (fl))                                          \
   : ({ unsigned int __fl = ((private) ^ FUTEX_PRIVATE_FLAG);            \
    asm ("andl %%gs:%P1, %0" : "+r" (__fl)                     \
         : "i" (offsetof (struct pthread, header.private_futex)));       \
    __fl | (fl); }))
# endif
#endif
```

Then, #ecx=128, which will pass the second parameter ‘op’ to the kernel function.

(9) sys_futex

//file: linux/kernel/futex.c

```
asmlinkage long sys_futex(u32 __user *uaddr, int op, int val,       //system call pass %ecx=128 to op
             struct timespec __user *utime, u32 __user *uaddr2,    //and pass timeout=0 to utime
             int val3)
{
    struct timespec t;
    unsigned long timeout = MAX_SCHEDULE_TIMEOUT;       //MAX_SCHEDULE_TIMEOUT=2147483647
    int val2 = 0;

    if ((op == FUTEX_WAIT) && utime) {              //FUTEX_WAIT=1, when utime=NULL, this if-sentence is not
executed
        if (copy_from_user(&t, utime, sizeof(t)) != 0)
            return -EFAULT;                          // #define EFAULT  14  /* Bad address */
        timeout = timespec_to_jiffies(&t) + 1;
    }
    /*
     * requeue parameter in 'utime' if op == FUTEX_REQUEUE.
     */
    if (op >= FUTEX_REQUEUE)                    //FUTEX_REQUEUE=3, if op=128, then val2 = utime
        val2 = (int) (unsigned long) utime;

    return do_futex((unsigned long)uaddr, op, val, timeout,  //when utime=NULL, then, timeout=2147483647
            (unsigned long)uaddr2, val2, val3);
}
```

Where, the constants are defined in the following file.

- FUTEX_WAIT

```
//file: linux/include/linux/futex.h
#define FUTEX_WAIT (0)
#define FUTEX_WAKE (1)
#define FUTEX_FD (2)
#define FUTEX_REQUEUE (3)
#define FUTEX_CMP_REQUEUE (4)
```

- MAX_SCHEDULE_TIMEOUT

```
//file: linux/include/linux/sched.h
#define MAX_SCHEDULE_TIMEOUT    LONG_MAX
```

- LONG_MAX

```
//file: linux/include/linux/kernel.h
#define LONG_MAX    ((long)(~0UL>>1))  //~ is 'Bitwise negation (按位取反)', UL is unsigned long. LONG_MAX=2147483647
```

(10) do_futex

```
//file: linux/kernel/futex.c
```

```
long do_futex(unsigned long uaddr, int op, int val, unsigned long timeout,
        unsigned long uaddr2, int val2, int val3)       //when utime=NULL, then, timeout=2147483647
{
    int ret;

    switch (op) {
    case FUTEX_WAIT:            //FUTEX_WAIT=0
        ret = futex_wait(uaddr, val, timeout);          //when utime=NULL, then, timeout=2147483647
        break;
    case FUTEX_WAKE:            //FUTEX_WAKE=1
        ret = futex_wake(uaddr, val);
        break;
    case FUTEX_FD:              //FUTEX_FD=2
        /* non-zero val means F_SETOWN(getpid()) & F_SETSIG(val) */
        ret = futex_fd(uaddr, val);
        break;
    case FUTEX_REQUEUE:         //FUTEX_REQUEUE=3
        ret = futex_requeue(uaddr, uaddr2, val, val2, NULL);
        break;
    case FUTEX_CMP_REQUEUE:     //FUTEX_CMP_REQUEUE=4
        ret = futex_requeue(uaddr, uaddr2, val, val2, &val3);
        break;
    default:
        ret = -ENOSYS;          //ENOSYS=38
    }
    return ret;
}
```

```
where,
//file: linux/include/asm-generic/errno.h
#define ENOSYS      38  /* Function not implemented */
```

(11) futex_wait

```
//file: linux/kernel/futex.c
```

```
static int futex_wait(unsigned long uaddr, int val, unsigned long time)  //when utime=NULL, then, timeout=2147483647
{
    DECLARE_WAITQUEUE(wait, current);       //create a wait queue
    int ret, curval;
    struct futex_q q;

 retry:
    down_read(&current->mm->mmap_sem);

    ret = get_futex_key(uaddr, &q.key);
    if (unlikely(ret != 0))
        goto out_release_sem;

    queue_me(&q, -1, NULL);                 //add me to this wait queue

    /*
     * Access the page AFTER the futex is queued.
     * Order is important:
     *
     *   Userspace waiter: val = var; if (cond(val)) futex_wait(&var, val);
     *   Userspace waker:  if (cond(var)) { var = new; futex_wake(&var); }
     *
```

```
     * The basic logical guarantee of a futex is that it blocks ONLY
     * if cond(var) is known to be true at the time of blocking, for
     * any cond.  If we queued after testing *uaddr, that would open
     * a race condition where we could block indefinitely with
     * cond(var) false, which would violate the guarantee.
     *
     * A consequence is that futex_wait() can return zero and absorb
     * a wakeup when *uaddr != val on entry to the syscall.  This is
     * rare, but normal.
     *
     * We hold the mmap semaphore, so the mapping cannot have changed
     * since we looked it up in get_futex_key.
     */

    ret = get_futex_value_locked(&curval, (int __user *)uaddr);

    if (unlikely(ret)) {
        /* If we would have faulted, release mmap_sem, fault it in and
         * start all over again.
         */
        up_read(&current->mm->mmap_sem);

        if (!unqueue_me(&q)) /* There's a chance we got woken already */
            return 0;

        ret = get_user(curval, (int __user *)uaddr);

        if (!ret)
            goto retry;
        return ret;
    }
    if (curval != val) {
        ret = -EWOULDBLOCK;
        goto out_unqueue;
    }

    /*
     * Now the futex is queued and we have checked the data, we
     * don't want to hold mmap_sem while we sleep.
     */
    up_read(&current->mm->mmap_sem);

    /*
     * There might have been scheduling since the queue_me(), as we
     * cannot hold a spinlock across the get_user() in case it
     * faults, and we cannot just set TASK_INTERRUPTIBLE state when
     * queueing ourselves into the futex hash.  This code thus has to
     * rely on the futex_wake() code removing us from hash when it
     * wakes us up.
     */

    /* add_wait_queue is the barrier after __set_current_state. */
    __set_current_state(TASK_INTERRUPTIBLE);     //TASK_INTERRUPTIBLE=1
    add_wait_queue(&q.waiters, &wait);           //add me to this wait queue, defined in kernel/wait.c
    /*
     * !list_empty() is safe here without any lock.
     * q.lock_ptr != 0 is not safe, because of ordering against wakeup.
     */
    if (likely(!list_empty(&q.list)))
        time = schedule_timeout(time);      //sleep until timeout, when utime=NULL, then, timeout=2147483647
    __set_current_state(TASK_RUNNING);           //TASK_RUNNING=0

    /*
     * NOTE: we don't remove ourselves from the waitqueue because
     * we are the only user of it.
```

```
     */

    /* If we were woken (and unqueued), we succeeded, whatever. */
    if (!unqueue_me(&q))
        return 0;
    if (time == 0)
        return -ETIMEDOUT;
    /* We expect signal_pending(current), but another thread may
     * have handled it for us already. */
    return -EINTR;

 out_unqueue:
    /* If we were woken (and unqueued), we succeeded, whatever. */
    if (!unqueue_me(&q))
        ret = 0;
 out_release_sem:
    up_read(&current->mm->mmap_sem);
    return ret;
}
```

Where,

```
//file: linux/include/linux/sched.h
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED            4
#define TASK_TRACED             8
#define EXIT_ZOMBIE            16
#define EXIT_DEAD              32

//file: linux/include/asm-i386/current.h
#define __set_current_state(state_value)            \
    do { current->state = (state_value); } while (0)


struct task_struct;

static inline struct task_struct * get_current(void)
{
    return current_thread_info()->task;
}

#define current get_current()
```

(12) schedule_timeout

```
//file: linux/kernel/timer.h
/**
 * schedule_timeout - sleep until timeout
 * @timeout: timeout value in jiffies
 *
 * Make the current task sleep until @timeout jiffies have
 * elapsed. The routine will return immediately unless
 * the current task state has been set (see set_current_state()).
 *
 * You can set the task state as follows -
 *
 * %TASK_UNINTERRUPTIBLE - at least @timeout jiffies are guaranteed to
 * pass before the routine returns. The routine will return 0
 *
 * %TASK_INTERRUPTIBLE - the routine may return early if a signal is
 * delivered to the current task. In this case the remaining time
 * in jiffies will be returned, or 0 if the timer expired in time
 *
```

```
 * The current task state is guaranteed to be TASK_RUNNING when this
 * routine returns.
 *
 * Specifying a @timeout value of %MAX_SCHEDULE_TIMEOUT will schedule
 * the CPU away without a bound on the timeout. In this case the return
 * value will be %MAX_SCHEDULE_TIMEOUT.
 *
 * In all cases the return value is guaranteed to be non-negative.
 */
fastcall signed long __sched schedule_timeout(signed long timeout)  //when utime=NULL, then, timeout=2147483647
{
    struct timer_list timer;
    unsigned long expire;

    switch (timeout)
    {
    case MAX_SCHEDULE_TIMEOUT:    //MAX_SCHEDULE_TIMEOUT=2147483647
        /*
         * These two special cases are useful to be comfortable
         * in the caller. Nothing more. We could take
         * MAX_SCHEDULE_TIMEOUT from one of the negative value
         * but I' d like to return a valid offset (>=0) to allow
         * the caller to do everything it want with the retval.
         */
        schedule();    //call the main scheduler function
        goto out;
    default:
        /*
         * Another bit of PARANOID. Note that the retval will be
         * 0 since no piece of kernel is supposed to do a check
         * for a negative retval of schedule_timeout() (since it
         * should never happens anyway). You just have the printk()
         * that will tell you if something is gone wrong and where.
         */
        if (timeout < 0)
        {
            printk(KERN_ERR "schedule_timeout: wrong timeout "
                   "value %lx from %p\n", timeout,
                   __builtin_return_address(0));
            current->state = TASK_RUNNING;
            goto out;
        }
    }

    expire = timeout + jiffies;

    init_timer(&timer);
    timer.expires = expire;
    timer.data = (unsigned long) current;
    timer.function = process_timeout;

    add_timer(&timer);
    schedule();
    del_singleshot_timer_sync(&timer);

    timeout = expire - jiffies;

 out:
    return timeout < 0 ? 0 : timeout;
}
```

(13) schedule

```
//file: linux/kernel/sched.c
/*
```

```
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
    long *switch_count;
    task_t *prev, *next;
    runqueue_t *rq;
    prio_array_t *array;
    struct list_head *queue;
    unsigned long long now;
    unsigned long run_time;
    int cpu, idx;

    /*
     * Test if we are atomic.  Since do_exit() needs to call into
     * schedule() atomically, we ignore that path for now.
     * Otherwise, whine if we are scheduling when we should not be.
     */
    if (likely(!current->exit_state)) {
        if (unlikely(in_atomic())) {
            printk(KERN_ERR "scheduling while atomic: "
                "%s/0x%08x/%d\n",
                current->comm, preempt_count(), current->pid);
            dump_stack();
        }
    }
    profile_hit(SCHED_PROFILING, __builtin_return_address(0));

need_resched:
    preempt_disable();                  //forbit kernel preemption against,禁止内核抢占
    prev = current;                     //assign current to prev
    release_kernel_lock(prev);
need_resched_nonpreemptible:
    rq = this_rq();                     //get local running queue of CPU

    /*
     * The idle thread is not allowed to schedule!
     * Remove this check after it has been exercised a bit.
     */
    if (unlikely(prev == rq->idle) && prev->state != TASK_RUNNING) {
        printk(KERN_ERR "bad: scheduling from the idle thread!\n");
        dump_stack();
    }

    schedstat_inc(rq, sched_cnt);
    now = sched_clock();
    if (likely(now - prev->timestamp < NS_MAX_SLEEP_AVG))
        run_time = now - prev->timestamp;
    else
        run_time = NS_MAX_SLEEP_AVG;

    /*
     * Tasks charged proportionately less run_time at high sleep_avg to
     * delay them losing their interactive status
     */
    run_time /= (CURRENT_BONUS(prev) ? : 1);

    spin_lock_irq(&rq->lock);

    if (unlikely(prev->flags & PF_DEAD))
        prev->state = EXIT_DEAD;

    switch_count = &prev->nivcsw;
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        switch_count = &prev->nvcsw;
```

```
        if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
                unlikely(signal_pending(prev))))
            prev->state = TASK_RUNNING;
        else {
            if (prev->state == TASK_UNINTERRUPTIBLE)
                rq->nr_uninterruptible++;
            deactivate_task(prev, rq);
        }
    }

    cpu = smp_processor_id();
    if (unlikely(!rq->nr_running)) {
go_idle:
        idle_balance(cpu, rq);    /* if there is no runable process in the running queue, then call idle_balance to
                                     move some from another running queue to this local running queue */
        if (!rq->nr_running) {    //if failed to move runnable process, then, switch idle process
            next = rq->idle;
            rq->expired_timestamp = 0;
            wake_sleeping_dependent(cpu, rq);
            /*
             * wake_sleeping_dependent() might have released
             * the runqueue, so break out if we got new
             * tasks meanwhile:
             */
            if (!rq->nr_running)
                goto switch_tasks;
        }
    } else {
        if (dependent_sleeper(cpu, rq)) {
            next = rq->idle;
            goto switch_tasks;
        }
        /*
         * dependent_sleeper() releases and reacquires the runqueue
         * lock, hence go into the idle loop if the rq went
         * empty meanwhile:
         */
        if (unlikely(!rq->nr_running))
            goto go_idle;
    }

    array = rq->active;                     //check there is runnable process or not in active priority array
    if (unlikely(!array->nr_active)) {      //if no runnable process, swap active and expired
        /*
         * Switch the active and expired arrays.
         */
        schedstat_inc(rq, sched_switch);
        rq->active = rq->expired;
        rq->expired = array;
        array = rq->active;
        rq->expired_timestamp = 0;
        rq->best_expired_prio = MAX_PRIO;
    } else
        schedstat_inc(rq, sched_noswitch);

    idx = sched_find_first_bit(array->bitmap);  //find the first runnable process in active array
    queue = array->queue + idx;
    next = list_entry(queue->next, task_t, run_list);

    if (!rt_task(next) && next->activated > 0) {
        unsigned long long delta = now - next->timestamp;

        if (next->activated == 1)
            delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;
```

```
        array = next->array;
        dequeue_task(next, array);
        recalc_task_prio(next, next->timestamp + delta);
        enqueue_task(next, array);
    }
    next->activated = 0;
switch_tasks:
    if (next == rq->idle)
        schedstat_inc(rq, sched_goidle);
    prefetch(next);
    clear_tsk_need_resched(prev);
    rcu_qsctr_inc(task_cpu(prev));

    prev->sleep_avg -= run_time;
    if ((long)prev->sleep_avg <= 0)
        prev->sleep_avg = 0;
    prev->timestamp = prev->last_ran = now;

    sched_info_switch(prev, next);
    if (likely(prev != next)) {
        next->timestamp = now;
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        prepare_arch_switch(rq, next);
        prev = context_switch(rq, prev, next);  //switch the selected process
        barrier();

        finish_task_switch(prev);
    } else
        spin_unlock_irq(&rq->lock);

    prev = current;
    if (unlikely(reacquire_kernel_lock(prev) < 0))
        goto need_resched_nonpreemptible;
    preempt_enable_no_resched();
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
        goto need_resched;
}
```

In this function, because, the current process state is set to TASK_INTERRUPTIBLE=1, and is put into the wait-queue, so the kernel will select other runnable (ready state) process to run, that is, will not schedule this current process again, until a signal comes or an event occurs, it will wake up the process from the wait-queue.

Thus, if timeout is the default value, that is, 0, of ACE_Task::putq, then, the caller of ACE_Task::putq will be blocked by kernel, no return forever,

5. conclusion

In this article, we not only discussed call stack of calling ACE_Task::putq, most actions of calling ACE_Task::putq with timeout=0, but also analyzed system call on the call path, detailed analyzed some functions such as ACE_OS::cond_timedwait and sys_futex. If timeout if NULL, the caller of ACE_Task::putq will be blocked by kernel, NEVER return. This process involves user space and kernel space, and three sets of source codes, ACE, glibc and Linux kernel. Besides, we simply mentioned the difference of system call between kernel versioned 2.6.18 and after it.

Note:
Linux kernel code in this article is version 2.6.11. but the kernel which this test runs on is version 2.6.23.1. What is the difference of system call between kernel versioned 2.6.18 and after it, and how to implement system call will be discussed in another article.
Glibc code in this article is version 2.7.

Reference

ACE-5.6.4 source code

Glibc-2.7 source code
Linux-2.6.11 source code
Linux-2.6.23.1 source code
nm manual
objdump manual
gcc manual