Bahir Dar University

Operating System and System programming

INDIVIDUAL ASSIGNMENT ON SYSTEM CALLS

Submitted to: Lec. Wendmu.B

Done by: Hailemichael Mulugeta

Id no: 1311614

Date: 17/11/2014

Table Contents page

What Is Futex? 2

Why we need futex? 2

What about the Parameters that futex use? 2

How does it work and What about Operations, implementations and also flags? 3

operations 3

Implementation of some of the operations 4

Implementation of futex wait: 4

Implementation of futex wake: 5

Implementing futex() with our own example. 5

The right way of lock implementation. 6

Flags and their use: 7

Reference 8

What Is Futex?

- Before explaining what futex is, we should look at what mutex is. Mutex is a sync primitive that helps you make sure only one thread can access a critical section (hence the name, mutual exclusion).

So how is a user space mutex implemented? There are different approaches to it, as operating systems and programming libraries can take different directions. But if we have to implement it by our own, we shall use the kernel. After all, only the kernel can put a thread to sleep and awaken it when asked. If we use the kernel, we can figure out a nice interface that takes care of a value for us. That would then allow us to:

- ✓ Check if the lock is taken. If it is, sleep;
- ✓ If it is not taken, take it;
- ✓ And when releasing the lock, wake anyone waiting for it.

However, doing system calls requires expensive context switches. And since our goal is to be as fast as possible in the common case, which is when the lock is not taken. This is where futex design comes in.

- **Futex** system call provides a method for waiting until a certain condition becomes true. It is typically used as a blocking construct in the context of shared-memory synchronization. Instead of asking for the kernel to take care of the mutex value for us, we can keep track of it in user space. Then, depending on the value, we can use the kernel only if we need to sleep or to wake other threads. This is the main design feature of the futex. Most of the time, it doesn't need the kernel, so we have less context switch and can have a lightweight mutex implementation. However, to make this work, the platform needs to support atomic operations in order to ensure the mutex key is atomically modified.

Why we need futex?

- We need futex because it can help us solve one of our big problems and that is: race condition. Since the arrival of multitasking operating systems, different tasks can now work at the same time and compete for the same resources. Using the same data at the same moment can lead to serious bugs and here is where futex come in place. Since its main job is to is to enable a more efficient way for user space code to synchronize multiple threads, with minimal kernel involvement.

What about the Parameters that futex use?

- Futex got a total of 6 parameters each with there on purpose and they are: (udder, op, val, timeout, udder2, val3)

  - ✓ **Uaddr:** points to the futex word. On all platforms, futexes are four-byte integers that must be aligned on a four-byte boundary.
  - ✓ **Op:** this is where the operation to perform on futex is specified.

    The futex_op parameter consists of two parts:

    1. a command that specifies the operation to be performed,
    2. bitwise ORed with zero or more options that modify the behavior of the operation.

  - ✓ **Val:** is a value whose meaning and purpose depends on futex_op.

The remaining arguments (timeout, uaddr2, and val3) are required only for certain of the futex operations, where one of these arguments is not required, it is ignored.

- ✓ For several blocking operations, the **timeout** argument is a pointer to a **timespec** structure that specifies a timeout for the operation. However, notwithstanding the prototype shown above, for some operations, the least significant four bytes of this argument are instead used as an integer whose meaning is determined by the operation. For these operations, the kernel casts the **timeout value** first to **unsigned long**, then to **uint32_t,** and in the remainder of this page, this argument is referred to as **val2** when interpreted in this fashion. Where it is required, the **uaddr2** argument is a pointer to a second futex word that is employed by the operation. The interpretation of the final integer argument, **val3**, depends on the operation.

How does it work and What about Operations, implementations and also flags?

- First, let's define values for our mutex state. Futex can only operate with unsigned int of 32bits:

```
#define UNLOCKED 0
#define LOCKED 1
```

- Now, here's an implementation for the mutex_lock().

```
mutex_lock (unsigned int *mutex)
{
        while (!cmpxchg(mutex, UNLOCKED, LOCKED))
                futex (mutex, FUTEX_WAIT, LOCKED);
}
```

- And then on the unlock side, we have this:

```
mutex_unlock (unsigned int *mutex)
{
        atomic_set (mutex, UNLOCKED);
        futex (mutex, FUTEX_WAKE, 1);
}
```

operations

- Five operations are currently defined:

1) **FUTEX_WAIT**
    - This operation atomically verifies that the futex address uaddr still contains the value val, and sleeps awaiting FUTEX_WAKE on this futex address.
    - If the timeout argument is non-NULL, its contents describe the maximum duration of the wait, which is infinite otherwise. The arguments uaddr2 and val3 are ignored.
    - For futex, this call is executed if decrementing the count gave a negative value (indicating contention), and will sleep until another process releases the futex and executes the FUTEX_WAKE operation.
    - This operation

- ✓ **Returns 0** if the process was woken by a FUTEX_WAKE call.
- ✓ In case of timeout, **ETIMEDOUT is returned**.
- ✓ If the futex was not equal to the expected value, the operation **returns EWOULDBLOCK**.
- ✓ Signals (or other spurious wakeups) cause FUTEX_WAIT to **return EINTR.**

2) FUTEX_WAKE

- This operation wakes at most val processes waiting on this futex address (i.e. inside FUTEX_WAIT). The arguments timeout, uaddr2 and val3 are ignored.
- For futex, this is executed if incrementing the count showed that there were waiters, once the futex value has been set to 1 (indicating that it is available).
- This operation **Returns the number of processes woken up.**

3) FUTEX_FD

- To support asynchronous wakeups, this operation associates a file descriptor with a futex. If another process executes a FUTEX_WAKE, the process will receive the signal number that was passed in val. The calling process must close the returned file descriptor after use. The arguments timeout, uaddr2 and val3 are ignored.
- To prevent race conditions, the caller should test if the futex has been upped after FUTEX_FD returns.
- **Returns the new file descriptor associated with the futex.**

4) FUTEX_REQUEUE

- This operation was introduced in order to avoid a "thundering herd" effect (the thundering herd problem occurs when a large number of processes or threads waiting for an event are awoken.) when FUTEX_WAKE is used and all processes woken up need to acquire another futex. This call wakes up val processes, and requeues all other waiters on the futex at address uaddr2. The arguments timeout and val3 are ignored.
- And the **Return value is the number of processes woken up**.

5) FUTEX_CMP_REQUEUE

- There was a race in the intended use of FUTEX_REQUEUE, so FUTEX_CMP_REQUEUE was introduced. This is similar to FUTEX_REQUEUE, but first checks whether the location uaddr still contains the value val3. If not, an error EAGAIN is returned. The argument timeout is ignored
- And also, **it Returns the number of processes woken up**.

Implementation of some of the operations

Implementation of futex wait:

- The futex system call has a timeout parameter which lets user code implement waiting with a time-out. The futex-wait-timeout sample shows this in action. Here is the relevant part of the child process which waits on a futex:

```
printf("child waiting for A\n");
struct timespec timeout = {.tv_sec = 0, .tv_nsec = 500000000};
while (1) {
  unsigned long long t1 = time_ns();
  int futex_rc = futex(shared_data, FUTEX_WAIT, 0xA, &timeout, NULL, 0);
  printf("child woken up rc=%d errno=%s, elapsed=%llu\n", futex_rc,
         futex_rc ? strerror(errno) : "", time_ns() - t1);
  if (futex_rc == 0 && *shared_data == 0xA) {
    break;
  }
}
```

- If the wait takes longer than 500 ms, the process will loop and wait again. The sample lets you configure the length of time the parent process keeps the child waiting and observe the effects.

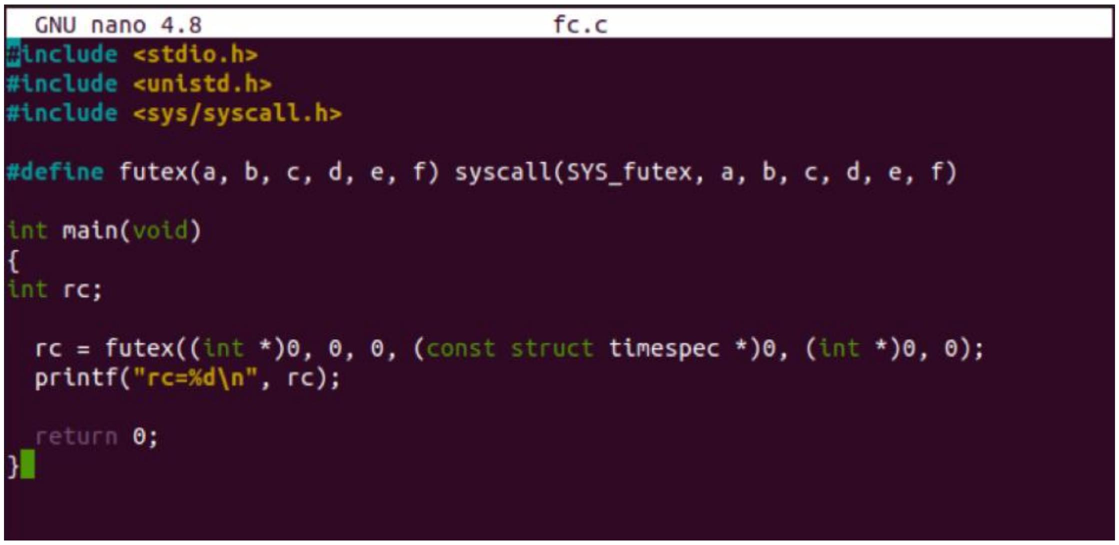
Implementation of futex wake:

- It's a blocking wrapper around FUTEX_WAKE, which will normally return quickly regardless of how many waiters it has woken up.

```
void wake_futex_blocking(int* futex_addr) {
  while (1) {
    int futex_rc = futex(futex_addr, FUTEX_WAKE, 1, NULL,
NULL, 0);
    if (futex_rc == -1) {
      perror("futex wake");
      exit(1);
    } else if (futex_rc > 0) {
      return;
    }
  }
}
```

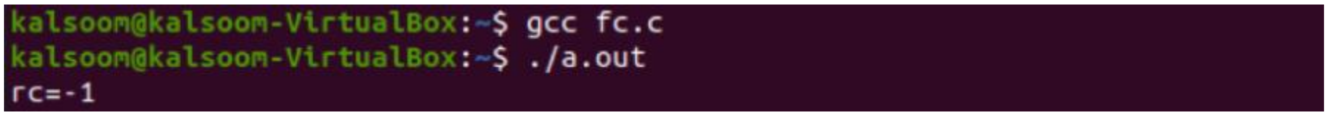
Implementing futex() with our own example.

- First, we booted ubuntu operating system and install compiler and we opened the terminal and wrote **nano fc.c** that created a C file that is named **“fc”** and then we wrote the code that as seen on the picture below. And the result is also shown at the bottom.



```
GNU nano 4.8                                    fc.c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

#define futex(a, b, c, d, e, f) syscall(SYS_futex, a, b, c, d, e, f)

int main(void)
{
int rc;

  rc = futex((int *)0, 0, 0, (const struct timespec *)0, (int *)0, 0);
  printf("rc=%d\n", rc);

  return 0;
}
```

RESULT



```
kalsoom@kalsoom-VirtualBox:~$ gcc fc.c
kalsoom@kalsoom-VirtualBox:~$ ./a.out
rc=-1
```

The right way of lock implementation.

```
class Mutex {
public:
  Mutex() : atom_(0) {}

  void lock() {
    int c = cmpxchg(&atom_, 0, 1);

    if (c != 0) {
      do {

        if (c == 2 || cmpxchg(&atom_, 1, 2) != 0) {
          syscall(SYS_futex, (int*)&atom_, FUTEX_WAIT, 2, 0, 0, 0);
        }
      } while ((c = cmpxchg(&atom_, 0, 2)) != 0);
    }
  }

  void unlock() {
    if (atom_.fetch_sub(1) != 1) {
      atom_.store(0);
      syscall(SYS_futex, (int*)&atom_, FUTEX_WAKE, 1, 0, 0, 0);
    }
  }

private:
  std::atomic<int> atom_;
};
```

Explanation for lock implementation.

- In the upper code at the highlighted numbers the execution will look like the following, In:

#1. If the lock was previously unlocked, there's nothing else for us to do. Otherwise, we'll probably have to wait.

#2. If the mutex is locked, we signal that we're waiting by setting the atom to 2. A shortcut checks is it's 2 already and avoids the atomic operation in this case.

#3. Here we have to actually sleep, because the mutex is actually locked. Note that it's not necessary to loop around this system call; a spurious wakeup will do no harm since we only exit the do...while loop when atom_ is indeed 0.

#4. We're here when either:

(a) the mutex was in fact unlocked (by an intervening thread).

(b) we slept waiting for the atom and were awoken.

So, we try to lock the atom again. We set the state to 2 because we can't be certain there's no other thread at this exact point. So, we prefer to err on the safe side.

#5. 0 means unlocked.

1 means locked, no waiters.

2 means locked, there are waiters in lock().

Flags and their use:

- **futex_clock_realtime_unavailable**: This flag is used to avoid the unnecessary unsupported futex call in the future and to fall back to the previous gettimeofday and relative time implementation.
- **MSG_DONTWAIT**: Enables nonblocking operation; if the operation would block, EAGAIN or EWOULDBLOCK is returned.

Reference

- Futexes Are Tricky, Ulrich Drepper (Red Hat, v1.6, 2011)
- Philip Bohannon and et. al. Recoverable User-Level Mutual Exclusion. In Proc. 7th IEEE Symposium on Parallel and Distributed Systems, October 1995.