Kernel-provided User Helpers

These are segment of kernel provided user code reachable from user space at a fixed address in kernel memory. This is used to provide user space with some operations which require kernel help because of unimplemented native feature and/or instructions in many ARM CPUs. The idea is for this code to be executed directly in user mode for best efficiency but which is too intimate with the kernel counter part to be left to user libraries. In fact this code might even differ from one CPU to another depending on the available instruction set, or whether it is a SMP systems. In other words, the kernel reserves the right to change this code as needed without warning. Only the entry points and their results as documented here are guaranteed to be stable.

This is different from (but doesn't preclude) a full blown VDSO implementation, however a VDSO would prevent some assembly tricks with constants that allows for efficient branching to those code segments. And since those code segments only use a few cycles before returning to user code, the overhead of a VDSO indirect far call would add a measurable overhead to such minimalistic operations.

User space is expected to bypass those helpers and implement those things inline (either in the code emitted directly by the compiler, or part of the implementation of a library call) when optimizing for a recent enough processor that has the necessary native support, but only if resulting binaries are already to be incompatible with earlier ARM processors due to usage of similar native instructions for other things. In other words don't make binaries unable to run on earlier processors just for the sake of not using these kernel helpers if your compiled code is not going to use new instructions for other purpose.

New helpers may be added over time, so an older kernel may be missing some helpers present in a newer kernel. For this reason, programs must check the value of __kuser_helper_version (see below) before assuming that it is safe to call any particular helper. This check should ideally be performed only once at process startup time, and execution aborted early if the required helpers are not provided by the kernel version that process is running on.

kuser_helper_version

Location: 0xffff0ffc

Reference declaration:

```
extern int32_t __kuser_helper_version;
```

Definition:

> This field contains the number of helpers being implemented by the running kernel. User space may read this to determine the availability of a particular helper.

Usage example:

```
#define __kuser_helper_version (*(int32_t *)0xffff0ffc)

void check_kuser_version(void)
{
      if (__kuser_helper_version < 2) {
            fprintf(stderr, "can't do atomic operations, kernel too old\n");
            abort();
      }
}
```

Notes:

> User space may assume that the value of this field never changes during the lifetime of any single process. This means that this field can be read once during the initialisation of a library or startup phase of a program.

kuser_get_tls

Location: 0xffff0fe0

Reference prototype:

```
void * __kuser_get_tls(void);
```

Input:

> lr = return address

Output:

> r0 = TLS value

Clobbered registers:

none

Definition:

Get the TLS value as previously set via the __ARM_NR_set_tls syscall.

Usage example:

```
typedef void * (__kuser_get_tls_t)(void);
#define __kuser_get_tls (*(__kuser_get_tls_t *)0xffff0fe0)

void foo()
{
      void *tls = __kuser_get_tls();
      printf("TLS = %p\n", tls);
}
```

Notes:

- Valid only if __kuser_helper_version >= 1 (from kernel version 2.6.12).

kuser_cmpxchg

Location: 0xffff0fc0

Reference prototype:

```
int __kuser_cmpxchg(int32_t oldval, int32_t newval, volatile int32_t *ptr);
```

Input:

r0 = oldval r1 = newval r2 = ptr lr = return address

Output:

r0 = success code (zero or non-zero) C flag = set if r0 == 0, clear if r0 != 0

Clobbered registers:

r3, ip, flags

Definition:

Atomically store newval in **ptr* only if **ptr* is equal to oldval. Return zero if **ptr* was changed or non-zero if no exchange happened. The C flag is also set if **ptr* was changed to allow for assembly optimization in the calling code.

Usage example:

```
typedef int (__kuser_cmpxchg_t)(int oldval, int newval, volatile int *ptr);
#define __kuser_cmpxchg (*(__kuser_cmpxchg_t *)0xffff0fc0)

int atomic_add(volatile int *ptr, int val)
{
      int old, new;

      do {
              old = *ptr;
              new = old + val;
      } while(__kuser_cmpxchg(old, new, ptr));

      return new;
}
```

Notes:

- This routine already includes memory barriers as needed.
- Valid only if __kuser_helper_version >= 2 (from kernel version 2.6.12).

kuser_memory_barrier

Location: 0xffff0fa0

Reference prototype:

```
void __kuser_memory_barrier(void);
```

Input:

lr = return address

Output:

none

Clobbered registers:

none

Definition:

Apply any needed memory barrier to preserve consistency with data modified manually and __kuser_cmpxchg usage.

Usage example:

```
typedef void (__kuser_dmb_t)(void);
#define __kuser_dmb (*(__kuser_dmb_t *)0xffff0fa0)
```

Notes:

- Valid only if __kuser_helper_version >= 3 (from kernel version 2.6.15).

kuser_cmpxchg64

Location: 0xffff0f60

Reference prototype:

```
int __kuser_cmpxchg64(const int64_t *oldval,
                      const int64_t *newval,
                      volatile int64_t *ptr);
```

Input:

r0 = pointer to oldval r1 = pointer to newval r2 = pointer to target value lr = return address

Output:

r0 = success code (zero or non-zero) C flag = set if r0 == 0, clear if r0 != 0

Clobbered registers:

r3, lr, flags

Definition:

Atomically store the 64-bit value pointed by **newval* in **ptr* only if **ptr* is equal to the 64-bit value pointed by **oldval*. Return zero if **ptr* was changed or non-zero if no exchange happened.

The C flag is also set if **ptr* was changed to allow for assembly optimization in the calling code.

Usage example:

```
typedef int (__kuser_cmpxchg64_t)(const int64_t *oldval,
                                  const int64_t *newval,
                                  volatile int64_t *ptr);
#define __kuser_cmpxchg64 (*(__kuser_cmpxchg64_t *)0xffff0f60)

int64_t atomic_add64(volatile int64_t *ptr, int64_t val)
{
      int64_t old, new;

      do {
              old = *ptr;
              new = old + val;
      } while(__kuser_cmpxchg64(&old, &new, ptr));

      return new;
}
```

Notes:

- This routine already includes memory barriers as needed.

- Due to the length of this sequence, this spans 2 conventional kuser "slots", therefore 0xffff0f80 is not used as a valid entry point.
- Valid only if __kuser_helper_version >= 5 (from kernel version 3.1).