DM510 Operating systems
Assignment 1
Linux system call

Frederik List
frlis21@student.sdu.dk

March 12, 2023

Introduction

There is no better way to learn how something works than by taking it apart and messing with it yourself. In this report I outline how I designed, implemented, and tested two system calls in the Linux kernel allowing applications to communicate via a message box.

I created two system calls, namely a `msgbox_put` and a `msgbox_get`, which respectively place and retrieve a single "message" (a byte array) from a global message box. The next section details how I came to the design decisions I did.

Design

There are some constraints we must adhere to when implementing a system call in Linux:

- **Safety** — A bad system call must not cause the kernel to panic. Hereunder lie further points to consider. The system call must:
  - be **robust**; the system call must for example handle malformed input well.
  - handle **concurrency**; a simultaneous invocation of system calls must not break things.

  When something does go wrong, the system call must also specifiy what went wrong with error code conventions.
- **Security** — A malicious sytem call must not allow a compromised process to do bad things.
  - I am not sure what sort of bad things one could possibly do with a malicious system call, but I am sure there is something.
- **Correctness** — A system call must, of course, do what it is supposed to without adverse effects such as memory leaks.
- **Probably a lot of other things** — Adding a system call is not the first thing one should resort to when adding a feature to the kernel.

We can guarantee safety by adding sanity checks throught the code and typing our code well. We will handle security and correctness by attempting to keep our code simple and thus easy to test and audit. We discuss concurrency in the implementation section.

We must also consider the policy and semantics of our system calls:

- What happens when we try to get from an empty message box?
- What happens when a message doesn't fit in the given buffer?
  - Truncate and put rest of truncated message in message box?
  - Fail and put failed message back in message box?
- What should happen when a `NULL` pointer is passed as an argument?
- We use `kmalloc` for the sake of simplicity which should not be used to allocate more memory than the page size of the operating system. Should messages have a size limit?
- Should we put back messages that could not be copied to user memory?

Considering we are only implementing a system call to get wet our feet in the Linux kernel, we will stick to simplicity.

- Error when trying to get from an empty message box.
- Truncate when a message doesn't fit in the given buffer without doing anything extra.
- Ignore `NULL`s.
- No message size limits, though I am sure this has unknown implications.
- Failed messages are lost.

In the next section I discuss how I implemented this design.

Implementation

Looking at online guides for adding a system call to the Linux kernel, it seems to have been a very laborious task for the older kernels. One had to update several files and manually increment a system call counter, among other potentially error-prone things. The guide we were supposed to follow suggests that we do something similar, however, the files it suggests we update seem to be *generated* by the kernel build process (a new feature?). This led me to investigate how other system calls are added in the Linux kernel, ignoring any `generated` directories.

I found that (at least for the `x86` architecture and the `um` architecture which is based on `x86`, it seems to be different for other architectures) one has to add entries to the `entry/syscalls/syscall_64.tbl` file in order to generate the syscall numbers in the `linux/unistd.h` file. Then it seems to be the convention to add the function prototypes to the `linux/syscalls.h` file, so that is what I did as well. Then I simply wrote the implementations in a new source file and added it to the build process. The full source code for the system calls can be viewed at Listing A.1.

Listing 1: `.../entry/syscalls/syscall_64.tbl`

```
373 449     common  futex_waitv             sys_futex_waitv
374 450     common  set_mempolicy_home_node sys_set_mempolicy_home_node
375 451     common  dm510_msgbox_put        sys_dm510_msgbox_put
376 452     common  dm510_msgbox_get        sys_dm510_msgbox_get
```

Listing 2: `.../include/linux/syscalls.h`

```
1055 asmlinkage long sys_memfd_secret(unsigned int flags);
1056 asmlinkage long sys_set_mempolicy_home_node(unsigned long start, unsigned long len,
1057                                             unsigned long home_node,
1058                                             unsigned long flags);
1059 asmlinkage long dm510_msgbox_put(char __user *buf, size_t len);
1060 asmlinkage long dm510_msgbox_get(char __user *buf, size_t len);
```

After that, the necessary files to invoke the two system calls are generated when you rebuild Linux.

There is nothing particularly fancy going on in the source code. A message is given the type `msg_t`. Messages are stored in a stack-like structure, so they keep track of their predecessor in their `previous` attribute. The current top of the stack is stored in a `static struct msg_t *top` initialized to `NULL`. Messages can be `push`ed or `pop`ped from the stack. Inside the `push()` and `pop()` procedures is a locking mechanism for preventing concurrency bugs, namely by saving the processor state and disabling interrupts with `local_irq_save()` and restoring the processor state and reenabling interrupts with

`local_irq_restore()`. This is not very good because interrupts are disabled only for the processor the invoking program is running on and race conditions can still occur while calling `local_irq_*`, but it will do for now.

The sanity checks mentioned in the design section can be seen doing their job in the actual implementations of the system calls. There are sanity checks for passed arguments, memory allocation, and copying memory from kernel-space to user-space. If a check fails, it returns a corresponding error number.

So much for the implementation of the system calls.

Evaluation

The system calls must be tested for correctness. I wrote 3 user-space programs to assist me in this:

- `msgbox_put` — Puts the first string from the command-line arguments in the message box (see Listing A.2).
- `msgbox_get` — Takes a number n from the command-line arguments and gets (at most n bytes of) a message from the message box (see Listing A.3).
- `test` — Performs various abuses of the new system calls to see how the kernel reacts (see Listing A.4).

There is an accompanying video demonstrating the tests I performed on the system calls using these programs. Note that in the video I am using a wrapper script to start user-mode Linux; this just sets up networking and other goodies for me. I also use a different root filesystem than the one provided, namely a Void Linux root filesystem. This is because I was interested in setting up user-mode Linux myself.

The tests I performed in the video to conclude that the system calls are correct were as follows (with video times in parentheses):

- Attempting to get a message from an empty message box correctly results in an error (00:10).
- Putting and getting random messages (00:14 – 00:45).
- Getting a message with a buffer that is too small truncates the message and removes it from the message box (00:45).
- Invoking any of the system calls with bad arguments correctly results in an error (01:00). "Bad arguments" include
  - `NULL`s,
  - negative numbers,
  - bad addresses.

I believe these tests are sufficient to prove that the system calls are correct.

Message box dynamics

There are multiple scenarios in which multiple processes would want to use the message box at the same time, many of which have problems because of the global nature of our message box:

- **Single producer and single consumer**

  - Process A produces messages.
  - Process B consumes messages, polling for new messages.
  - No other processes are using the message box.

  In this case, the message box functions well. No processes are vying for control over the message box. This is the ideal case.

- **Bidirectional communication**

  - Processes A and B communicate both ways through the message box.

  In this case, the message box does not perform very well. Processes waiting for a reply would occasionally take back their own messages, so they would need to implement random backoff or some other strategy to allow their recipient to read the message. This is extremely inefficient. The processes would be better off having two message boxes, one for each direction of communication that no other processes can access.

- **Pure chaos**

  - Processes A and B want to communicate with each other.
  - Processes C, D, and E want to communicate with each other.
  - etc.

  Here our message box completely breaks down. Processes will spend most of their time accidentally stealing other processes' messages or taking back their own messages because no processes can get the messages intended for them.

Conclusion

Our system call may be useless and crappy, but at it served its purpose well as a gateway to Linux kernel development. That is all!

Appendix A

Source code

Listing A.1: .../kernel/dm510_msgbox.c

```
1  #include <linux/syscalls.h>
2  #include <linux/slab.h>
3  #include <linux/uaccess.h>
4  #include <linux/kernel.h>
5  #include <linux/errno.h>
6  #include <linux/minmax.h>
7
8  struct msg_t {
9          struct msg_t *previous;
10         size_t length;
11         char *message;
12 };
13
14 static struct msg_t *top = NULL;
15
16 static void destroy(struct msg_t *msg)
17 {
18         kfree(msg->message);
19         kfree(msg);
20 }
21
22 static void push(struct msg_t *msg)
23 {
24         unsigned long flags;
25
26         local_irq_save(flags);
27         msg->previous = top;
28         top = msg;
29         local_irq_restore(flags);
30 }
31
32 static struct msg_t *pop(void)
33 {
34         unsigned long flags;
35         struct msg_t *msg;
36
37         local_irq_save(flags);
38         msg = top;
39         top = msg->previous;
40         local_irq_restore(flags);
41
42         return msg;
43 }
44
45 SYSCALL_DEFINE2(dm510_msgbox_put, char __user *, buf, size_t, len)
46 {
47         struct msg_t *msg;
48
```

```
49          if (!buf)
50                  return -EFAULT;
51
52          msg = kmalloc(sizeof(struct msg_t), GFP_KERNEL);
53
54          if (!msg)
55                  return -ENOMEM;
56
57          msg->length = len;
58          msg->message = kmalloc(len, GFP_KERNEL);
59
60          if (!msg->message) {
61                  destroy(msg);
62
63                  return -ENOMEM;
64          }
65
66          if (copy_from_user(msg->message, buf, len)) {
67                  destroy(msg);
68
69                  return -EFAULT;
70          }
71
72          push(msg);
73
74          return 0;
75  }
76
77  SYSCALL_DEFINE2(dm510_msgbox_get, char __user *, buf, size_t, len)
78  {
79          struct msg_t *msg;
80          size_t mlen;
81          long ret;
82
83          // Check that a message exists before whether a buffer is valid
84          // in case we want to clear the message box?
85          if (!top)
86                  return -ENODATA;
87          if (!buf)
88                  return -EFAULT;
89
90          msg = pop();
91          mlen = msg->length;
92          ret = mlen;
93
94          // Copy snprintf behavior and truncate message
95          if (copy_to_user(buf, msg->message, min(mlen, len)))
96                  ret = -EFAULT;
97
98          destroy(msg);
99
100         return ret;
101 }
```

Listing A.2: msgbox_put.c

```
1   #include <linux/unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <unistd.h>
5   #include <errno.h>
6   #include <string.h>
7
8   int main(int argc, char *argv[]) {
9           if (argc < 2) {
10                  fprintf(stderr, "no\n");
11                  return 1;
```

```
        }

        if (syscall(__NR_dm510_msgbox_put, argv[1], strlen(argv[1]))) {
                perror(argv[0]);
                return 1;
        }

        return 0;
}
```

Listing A.3: msgbox_get.c

```
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(int argc, char *argv[]) {
        if (argc < 2) {
                fprintf(stderr, "no\n");
                return 1;
        }

        char *endptr;
        int length = strtol(argv[1], &endptr, 10);

        if (length < 0 || argv[1] == endptr) {
                fprintf(stderr, "stop\n");
                return 1;
        }

        char *buffer = malloc(length);
        length = syscall(__NR_dm510_msgbox_get, buffer, length);

        if (length < 0) {
                perror(argv[0]);
                return 1;
        }

        printf("msgbox_get got %d bytes:\n%.*s\n", length, length, buffer);
        free(buffer);

        return 0;
}
```

Listing A.4: test.c

```
#include <errno.h>
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

long _msgbox_put(char *buf, size_t len) {
        long ret = syscall(__NR_dm510_msgbox_put, buf, len);

        if (ret)
                perror("\tmsgbox_put");

        return ret;
}

long _msgbox_get(char *buf, size_t len) {
        int mlen = syscall(__NR_dm510_msgbox_get, buf, len);
```

```
19
20          if (mlen < 0)
21                  perror("\tmsgbox_get");
22          else {
23                  fprintf(stderr, "\tGot %d bytes:\n", mlen);
24                  fprintf(stderr, "\t%.*s\n", mlen, buf);
25          }
26
27          return mlen;
28  }
29
30  static unsigned long test_num = 0;
31
32  #define msgbox_put(buf, len)                                                \
33          {                                                                   \
34                  fprintf(stderr, "%lu. msgbox_put(" #buf ", " #len "):\n", ++test_num); \
35                  _msgbox_put(buf, len);                                      \
36          }
37
38  #define msgbox_get(buf, len)                                                \
39          {                                                                   \
40                  fprintf(stderr, "%lu. msgbox_get(" #buf ", " #len "):\n", ++test_num); \
41                  _msgbox_get(buf, len);                                      \
42          }
43
44  void msg(const char *message) { fprintf(stderr, "%s\n", message); }
45
46  void sep() {
47          fprintf(stderr, "----------------------------------------"
48                          "----------------------------------------\n");
49  }
50
51  #define BUF_LEN 42
52  #define DUMMY_DATA "dummy message"
53  #define DUMMY_DATA_LEN 13
54  #define DUMMY_N 20
55
56  int main(int argc, char *argv[]) {
57          char buffer[BUF_LEN];
58
59          // Clear the msgbox
60          while (syscall(__NR_dm510_msgbox_get, buffer, BUF_LEN) >= 0)
61                  ;
62
63          msg("(cleared msgbox)");
64          sep();
65
66          msg("msgbox_get from empty msgbox?");
67          msgbox_get(buffer, BUF_LEN);
68          sep();
69
70          // Fill the msgbox
71          for (int i = 0; i < DUMMY_N; i++)
72                  syscall(__NR_dm510_msgbox_put, DUMMY_DATA, DUMMY_DATA_LEN);
73
74          fprintf(stderr, "(filled msgbox with %dx%d bytes \"%s\")\n", DUMMY_N,
75                  DUMMY_DATA_LEN, DUMMY_DATA);
76          sep();
77
78          msg("Nulls and stuff?");
79          msgbox_put(NULL, 42);
80          msgbox_get(NULL, 42);
81          sep();
82
83          msg("Invalid addresses?");
84          msgbox_put((char *)-42, 42);
85          msgbox_put((char *)42, 42);
86          msgbox_get((char *)-42, 42);
```

```
87          msgbox_get((char *)42, 42);
88          sep();
89
90          msg("Negative numbers?");
91          msgbox_put(buffer, -42);
92          msgbox_get(buffer, -42);
93          sep();
94
95          msg("And everything should still work after all of that!");
96          msgbox_get(buffer, BUF_LEN);
97          msgbox_put("hello", 5);
98          msgbox_get(buffer, BUF_LEN);
99
100         return 0;
101 }
```