Kernel Course: Lecture 17

Interface with user space

Serhii Popovych, Illia Smyrnov

May 26, 2020

GlobalLogic

Agenda

1. Interface with user space
2. Assignments

Interface with user space

Operating system, kernel and user space

- So far we talk mostly about kernel related stuff. But to get stuff that we eventually call “Operating System - OS” we do need `user space tools`.
- user space ecosystem is huge, even bigger than kernel. Actually operating system popularity fully depends on how many user space applications available on it, not maturity of it’s kernel.
- Related to Linux you may already hear, that full name of the Linux OS is GNU/Linux:
    - First part of it’s name is a GNU - that core set of user space applications available on it
    - Second is just Linux Kernel
- From perspective of the user space operating system provides Hardware Abstraction Layer (HAL) and resource sharing/management.

UNIX, Linux and IO model

- To interact with hardware, user space need to follow HAL model implemented by the OS kernel.
- These models vary from one operating system (class) to another. Typical examples of these models are:
  - Most communication with hardware can be represented as IO.
  - Messaging model. Accessing device special communication API.
- It looks obvious that consolidating hardware access with other IO operations, not related to direct hardware communication would have benefits over distinct interfaces.
- In UNIX world most of communications represented through IO.
- Since Linux is a UNIX-like OS it follows IO communication model too.

- In Input/Output model everything is build on top of IO operations.
- Most of the resources in general represented as entries in file system.
- Some of the resources cannot be represented in such way, however their access model is still fit's to general IO model. One of the examples of such resources is BSD Socket Interface used to exchange data over network: application may use standard IO model with hidden packet exchange details.
- To distinguish resources on which operations should be performed (instantiate) we need some handle for each resource.
- This handle represented via signed int with values greater or equal than zero and less than or equal to `INT_MAX` (POSIX).

UNIX, Linux and IO model (cont.)

- To implement IO model for user space Linux Kernel need to implement at least following interfaces:
    - open() - get resource handle (file descriptor)
    - read() - read data from handle to given buffer
    - write() - write data from buffer to handle
    - close() - release resource handle (close file descriptor)
- However sometimes we need to perform some additional actions:
    - llseek() - seek pointer to the data to the given position
    - fsync() - synchronize OS cached buffers to a disk
    - mmap() - map contents of the resource to a memory
    - lock()/unlcok() - acquire/release POSIX advisory/mandatory locking
- There are other operations beside these ones that can’t be treated as IO.

UNIX, Linux and IO model (cont.)

- Not all operations can fit into IO model natively.
- For example when some configuration of the hardware/driver should be made and we can't represent this as an IO.
- That's configuration can be accomplished via special interface(s) implemented by the driver:
  - `ioctl()` - accept resource handle, option number and data in format recognizable by device
  - memory mapped region in user space - quite uncommon, might be hard to implement
- Generic netlink (genetlink) protocol via BSD sockets.
- syscalls - an generic kernel<->user space communication interface (cannot be implemented from modules, one need to modify and rebuild kernel to add new syscall).
- Note that `ioctl(2)` is actually generic system call interface.

UNIX, Linux and IO model (cont.)

- Sometimes kernel code decides to implement control interface using standard IO model. Linux Kernel have wide variety possibilities to help developers implement that:
  - configfs - special filesystem to configure various parameters of driver
  - sysfs - additional interface, that could be used to expose kernel driver information/control to the user space
  - procfs - old, good and proven interface in Linux, has similar objectives as sysfs
- Note that debugfs, we describe briefly earlier in lectures related to the debugging should not be considered as kernel<->user space communication interface. It is intended only for debugging.
- Neither sysfs nor procfs should be used unless implementing generic stuff.

System calls: general

- As you may notice from previous slides all primitives in IO model actually functions implementing kernel<->user space communication.
- This type of communication named system calls.
- While their implementation is architecture and operating system dependent there are few common aspects for all implementations:
    - It uses number to distinguish one call from another
    - There is calling conversion caller and callee must follow to track CPU registers usage
    - Using system call switches execution context from user space to kernel space. That introduces additional overhead to the implementation.
    - No parameters coming from user space should be trusted. Since syscalls executing in kernel context any mistake might compromise system stability and security.

System calls: general (cont.)

- As said previously system calls are gateways from user to kernel context. Switching from one to another and back has significant overhead.
- It is also known that it is architecture dependent.
- But what is actually user and kernel context? What is actually context in the scope of syscalls?
- Context it is a some, architecture defined state of execution flow that must be preserved/restored upon context switch.
- That state may include, but not limited to:
    - CPU registers, general purpose, stack, segment, FPU, control registers
    - Page table catalog and TLB flush

System calls: general (cont.)

- Okay, saving CPU registers implemented in hardware, switching address space, flushing TLB etc all this is done in hardware. Why there is performance lost?
  - First instructions performing CPU registers save/restore isn't trivial. It is tens of CPU cycles performing various security checks as well as requesting memory access on bus.
  - Second after TLB flush there is no cached entries: MMU need to perform paging (logical -> physical translations) more aggressively before TLB gets populated. This especially true when accessing random pages in new address space.
  - Third CPU cache entries, related to the kernel address space get flushed. This trashes performance when doing syscall after syscall.
- Besides all described above there is another contributor to performance drop when doing syscalls.

System calls: general (cont.)

- As we said previously CPU instructions performing context switch check for permissions. Those checks coming from privilege separation implemented within CPU.
- That’s mean not all processor instructions allowed to be executed in different execution context.
- For example instructions controlling page permissions, processor operational mode, interrupt/exception vectors, access to IO ports can be performed in one context, but are illegal for another one.
- In general these execution contexts are called rings and actual number of them is architecture dependent.

System calls: general (cont.)

- Privilege rings for the x86 available in protected mode [1]:

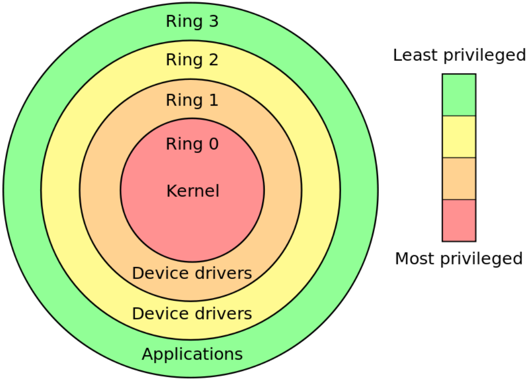

Figure 1: Privilege rings for the x86 available in protected mode

System calls: general (cont.)

- In Linux Kernel it is assumed that architecture provides at least two rings:
  - Supervisor, where all, including privileged, instructions can be executed
  - User, restricted, only “safe”, non-privileged instructions available.
- Other architectures might implement much more and spread instruction privileges across them:
  - x86 implements four (4) rings (0..3)
  - ARM implements three (3) rings (domains of execution)
- Presence of execution contexts completes user space process vs user space process and kernel vs user space process isolation model required to implement secure operating systems.

System calls: general (cont.)

- Now we known what is syscalls, why they needed and their drawbacks.
- It is time to describe how they implemented within Linux Kernel and wrapped in user space.
- By saying “wrapped” we mean following:
    - Kernel and user space might have different idea about system calls being actually executed.
    - For example vfork(2) syscall, described by man-pages project in section 2 actually implemented via different system call clone(2).
    - That means user space (usually libc) wraps call to one function in user space to one (or sometimes to series) of the real kernel system calls.
    - Of course libc (in our case we will look at glibc) might provide some interfaces to invoke kernel system calls directly.

System calls: user space application side

- Before we look on how syscalls implemented on Linux Kernel side we want to look on how to call kernel syscall directly from application bypassing possible library (libc) wrappers.
- In user space most C libraries (we will look at glibc) provide non-standard conformant interface:
  - `_syscall(2)` - deprecated interface, should not be used in new applications
  - `syscall(2)` - BSD 4.4 API conforming system call interface.
- We will look at syscall(2) and how to use it to call any kernel system call directly.
- This libc provided API is nothing else than wrapper that hides architecture specific kernel<->user interface, implemented in assembly, behind C func.

System calls: user space application side

- Here is architecture specific syscall ABI interface between kernel and user space taken from syscall(2) manpage:

```
arch/ABI   instruction           syscall #   retval Notes
-------------------------------------------------------------------------------------
arm/OABI   swi NR                -           a1     NR is syscall #
arm/EABI   swi 0x0               r7          r1
blackfin   excpt 0x0             P0          R0
i386       int $0x80             eax         eax
ia64       break 0x100000        r15         r10/r8
parisc     ble 0x100(%sr2, %r0) r20          r28
s390       svc 0                 r1          r2     NR may be passed directly with

s390x      svc 0                 r1          r2     "svc NR" if NR is less than 256
sparc/32   t 0x10                g1          o0
sparc/64   t 0x6d                g1          o0
x86_64     syscall               rax         rax
```

System calls: user space application side

- Let's talk to the Linux Kernel directly using libc provided syscall(2) wrapper:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <signal.h>

int main(int argc, char *argv[])
{
        pid_t tid;

        tid = syscall(SYS_gettid);
        syscall(SYS_tgkill, getpid(), tid, SIGHUP);

        /* never reached: only to keep compiler happy */
        return 0;
}
```

System calls: user space application side (cont.)

- While syscall(2) considered as low level, hiding architecture specific aspects, API in libc, when talking directly to the kernel from assembly you need to use architecture dependent, not portable ABI.
- Even deeper: there might be additional, not available on all supported by Linux Kernel platforms, fast and convenient interface called vDSO (from Virtual Dynamic Shared Object)
- vDSO, when available, provides high performance interface for some critical, oftenly called syscalls.
- It is actually kernel mapped, shared between other processes, memory area, containing ELF header, symbol table, and misc information (like shared lib).

System calls: user space application side (cont.)

- Why vDSO?
  - It is a way to kernel provide compatible through versioned symbols (through ELF format), zero overhead (to switch between kernel and user context) on call, interface to the application for very frequently called functions.
  - One of the good examples of such functions is gettimeofday(2) that is either called too frequently from application explicitly (e.g. to get timestamps for logging) or implicitly from libc when doing some calls.
  - There is also another, more generic, reason on why vDSO is important: it actually determines fastest syscall mechanism to be used on architecture. Last means that ALL libc (at least in glibc) primitives exploit either vDSO+architecture specific instructions (e.g. syscall/sysenter on x86) or fall back to generic syscall via software interrupt/exception initiation instruction (e.g. `int \$0x80` on x86).

System calls: user space application side (cont.)

- How user space gets address of vDSO?
  - user space application receives at most three arguments to the function `main()`.
  - First two: `argc` and `argv[]` are standardized by ANSI/ISO standard.
  - Last one `envp[]` isn't standardized (maybe in the last POSIX or SUSvX standards?), but widely implemented.
  - But on Linux things quite different: binary loader (e.g. `binfmt_elf`) passes additional, hidden from the `main()`, auxiliary values vector.
- On Linux user space (e.g. `/lib/ld-linux.so.6`) uses auxiliary vector to get address of the vDSO ELF header.
- Why vDSO address isn't mapped at fixed address in application?
  - Considerations are the same as for other Shared Objects (.so): security
  - This is to prevent class of exploits known as return-to-libc.

System calls: user space application side (cont.)

- However auxiliary vector contains bit more useful information that kernel supplies user space. Let's look at that information and how to access it.
- In general, values in auxiliary vector are split into two parts:
  - Common
  - Architecture dependent
- However these common attributes, while present might still contain architecture dependent data.
- There are at least two methods to access auxiliary vector values:
  - Use glibc provided helper getauxval(3)
  - Manually find auxiliary vector start address
- First is simplest, Second is most portable (e.g. when using non GNU libc)

System calls: user space application side (cont.)

- Let’s find start of vDSO using getauxval(3)::

```
#define _GNU_SOURCE
#include <sys/auxv.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        void *sysinfo_ehdr = (void *)getauxval(AT_SYSINFO_EHDR);
        if (!sysinfo_ehdr) {
                fputs("no AT_SYSINFO_EHDR: architecture w/o vDSO?\n", stderr);
                return EXIT_FAILURE;
        }
        printf("AT_SYSINFO_EHDR: %p\n", sysinfo_ehdr);
        return EXIT_SUCCESS;
}
```

System calls: user space application side (cont.)

- Finding auxiliary vector by the hand requires knowledge of stack layout after binary loader from the kernel calls application:

```
position            content                       size (bytes) + comment
------------------------------------------------------------------------
stack pointer ->  [ argc = number of args ]     s
                  [ argv[0] (pointer) ]         s   (program name)
                  [ argv[n] (pointer) ]         s   (= NULL)

                  [ envp[0] (pointer) ]         s
                  [ envp[term] (pointer) ]      s   (= NULL)

                  [ auxv[0] (ElfX_auxv_t) ]     s
                  [ auxv[term] (ElfX_auxv_t) ]  s   (= AT_NULL vector)

                  [ padding ]                   0 - 16

                  [ argument ASCIIZ strings ]   >= 0
                  [ environment ASCIIZ str. ]   >= 0
------------------------------------------------------------------------
```

System calls: user space application side (cont.)

- Assuming the above we need to get beyond the end of the envp[]. That could be easily be accomplished by the following code:

```
#include <stdio.h>
#include <elf.h>

int main(int argc, char *argv[], char *envp[])
{
        char **ep = envp;
        /* Use Elf64_auxv_t for 64-bit architectures */
        Elf32_auxv_t *auxv;

        /* Get to the start of ELF auxilary vector */
        while (*ep++)
                ;

        /*...*/
```

System calls: user space application side (cont.)

- and continue code

```
	for (auxv = (Elf32_auxv_t *)ep; auxv->a_type != AT_NULL; auxv++) {
		if (auxv->a_type != AT_SYSINFO_EHDR) {
			void *sysinfo_ehdr = (void *)auxv->a_un.a_val;

			printf("AT_SYSINFO_EHDR: %p\n", sysinfo_ehdr);
			return 0;
		}
	}

	fputs("no AT_SYSINFO_EHDR: architecture w/o vDSO?\n", stderr);
	return 1;
}
```

System calls: user space application side (cont.)

- There are lot more arch-independent information in auxiliary vector:

```
#define AT_NULL         0       /* End of vector */
#define AT_IGNORE       1       /* Entry should be ignored */
#define AT_EXECFD       2       /* File descriptor of program */
#define AT_PHDR         3       /* Program headers for program */
#define AT_PHENT        4       /* Size of program header entry */
#define AT_PHNUM        5       /* Number of program headers */
#define AT_PAGESZ       6       /* System page size */
#define AT_BASE         7       /* Base address of interpreter */
#define AT_FLAGS        8       /* Flags */
#define AT_ENTRY        9       /* Entry point of program */
#define AT_NOTELF       10      /* Program is not ELF */
#define AT_UID          11      /* Real uid */
#define AT_EUID         12      /* Effective uid */
#define AT_GID          13      /* Real gid */
#define AT_EGID         14      /* Effective gid */
#define AT_CLKTCK       17      /* Frequency of times() */
```

System calls: user space application side (cont.)

- There are lot more arch-independent information in auxiliary vector (cont.):

```
#define AT_HWCAP        16        /* Machine-dependent hints about
                                   * processor capabilities.
                                   */
#define AT_SECURE       23        /* Boolean, was exec setuid-like? */
#define AT_RANDOM       25        /* Address of 16 random bytes. */
#define AT_HWCAP2       26        /* More machine-dependent hints
                                   * about processor capabilities.
                                   */
#define AT_EXECFN       31        /* Filename of executable. */

/* Pointer to the global system page used for system calls and other
 * nice things.
 */
#define AT_SYSINFO      32
#define AT_SYSINFO_EHDR 33
```

System calls: user space application side (cont.)

- And there is a trick to get auxiliary vector without writing code:

```
\$ LD_SHOW_AUXV=1 /lib64/ld-linux-x86-64.so.2 /bin/true
        AT_SYSINFO_EHDR: 0x7ffed658e000
        AT_HWCAP:        bfebfbff
        AT_PAGESZ:       4096
        AT_CLKTCK:       100
        AT_PHDR:         0x7fc9fdcc4040
        AT_ENTRY:        0x7fc9fdcc51e0
        AT_UID:          1000
        AT_EUID:         1000
        AT_GID:          1000
        AT_EGID:         1000
        AT_SECURE:       0
        AT_RANDOM:       0x7ffed64f0aa9
        AT_EXECFN:       /lib64/ld-linux-x86-64.so.2
        AT_PLATFORM:     x86_64
```

System calls: user space application side (cont.)

- Additionally when you do something like following:

```
\$ ldd /bin/true
        linux-vdso.so.1 =>  (0x00007fff94573000)
        libc.so.6 => /lib64/libc.so.6 (0x00007efd5c3c4000)
        /lib64/ld-linux-x86-64.so.2 (0x00007efd5c7a6000)
```

- Where to go next with vDSO, auxiliary vector etc:
  - man-page vdso 7 [2]
  - Documentation in Linux Kernel tree contains excellent examples on how to call vDSO. Start with [3]

System calls: kernel space side

- So long we talk about user space applications visible interface. It is time to look at kernel side of implementation.
- Main data structure, holding information about system calls supported on given architecture by given kernel version is `sys_call_table`.
- It is an array of pointers to the specially defined syscall handler function.
- This array is statically defined based on kernel compile time information and stored in write-protected memory page. Main reason for doing that is to protect `sys_call_table` from modifications by rogue kernel modules (e.g. ones implementing rootkit functionality) at runtime.
- This also means that it is impossible to implement syscall in kernel module.

System calls: kernel space side (cont.)

- What is the steps to add a new system call to kernel and possibly merge it?
- First you need to read guidelines in ”Adding a New System Call” [4]
- Next, if you still want to add it as syscall follow implementation requirements and recommendations in that document.
- Finally, if you just want to try to add your own syscall you need following:
    - Define handler function, using special API.
    - Add new syscall number `__NR_hello`, adjust `__NR_syscalls` accordingly
    - Provide information about your syscall in generic syscall table: `kernel/sys_ni.c`
    - Add your syscall to architecture specific array `sys_call_table` for `each` architecture supported by Linux currently.

System calls: kernel space side (cont.)

- Define handler function, using special API.
- First of all: you need to find a place where to put your syscall.
  - Best place is new `.c` file with your syscall implementation in one of the standard locations like `fs/, kernel/, mm/` etc. For example if your call related to filesystem operations put it into `fs/` where `open(), close()` and other filesystem related calls implemented; if you implement something specific related to the bus management (e.g. PCI) put it into `drivers/`.
- Next, decide with number of parameters your function will get. Keep in mind all syscalls implemented using fastcall calling conversion and `never` pass parameters through stack. Number of CPU registers is limited.
- Look at `<linux/syscalls.h>`.
  - You need to declare your syscall handler prototype here.
  - You need to use `SYSCALL_DEFINEn()` macro, where n is the number of syscall parameters.

System calls: kernel space side (cont.)

- Lets add Hello system call for demonstration. It should take two parameters from user space:
  - buffer to return a message;
  - size of the buffer.
- The prototype of new syscall should be added to `include/linux/syscalls.h` as below. Note that name MUST always begin with `sys_` prefix.

```
asmlinkage long sys_hello(char __user *hello_buf, unsigned int len);
```

System calls: kernel space side (cont.)

- The next step - is to add a definition for Hello syscall. Add the following implementation to the new `kernel/sys-hello.c` file.

```c
#include <linux/string.h>
#include <linux/syscalls.h>

#define HELLO_STRING   "Hello from kernel!"

SYSCALL_DEFINE2(hello, char __user *, hello_buf, unsigned int, len)
{
        pr_debug("%s:%d hello_buf 0x%px, len %d\n", __func__, __LINE__,
                hello_buf, len);

        if (strlen(HELLO_STRING) + 1 > len)
                return -EINVAL;

        if (copy_to_user(hello_buf, HELLO_STRING, strlen(HELLO_STRING) + 1))
                return -EFAULT;

        return 0;
}
```

System calls: kernel space side (cont.)

- Add new Kconfig symbol `CONFIG_HELLO_SYSCALL` with dependency on `EXPERT` to `init/Kconfig` file.

```
/* In fs/Kconfig */
config HELLO_SYSCALL
	bool "Enable hello world system call" if EXPERT
	default y
	---help---
	  Hello world syscall sample.

	  If unsure say Y here.
```

- Add `kernel/Makefile` entry that includes `sys-hello.o` depending on `CONFIG_HELLO_SYSCALL`.

```
obj-\$(CONFIG_HELLO_SYSCALL) += sys-hello.o
```

System calls: kernel space side (cont.)

- Add new system call to the generic list by adding an entry to the list in `include/uapi/asm-generic/unistd.h`. Also update the `__NR_syscalls` count to reflect the additional system call.

```
/* ... */
#define __NR_rseq 294
__SYSCALL(__NR_hello, sys_hello)
/* ... */
#undef __NR_syscalls
#define __NR_syscalls 295
/* ... */
```

- The file `kernel/sys_ni.c` provides a fallback stub implementation of each system call, returning `-ENOSYS`. Add new system call here too.

```
/* ... */
COND_SYSCALL(hello);
/* ... */
```

- Some architectures (e.g. x86, powerpc, ARM (32-bit)) does implement `sys_call_table` using architecture specific implementation, so new system call should be addedto all of them. This might require lot of work, as this isn't standardized per architecture.
- Due to Beagle Bone Black HW used for the course is ARM based and Hello system call is for education purposes only, add it for ARM32 architecture. Add the folloving syscall table entry in `arch/arm/tools/syscall.tbl` file.

```
# ...
400    common  hello                       sys_hello
```

System calls: kernel space side (cont.)

- Finally enable Hello system call - add the following lone to `arch/arm/configs/bbb_defconfig` file. (Or use more correct way to do the same - add this string to `fragments/bbb.cfg` and merge configs).

```
# ...
CONFIG_HELLO_SYSCALL=y
```

- Make config and ensure `CONFIG_HELLO_SYSCALL=y` is in `.config` file.

```
make <...> bbb_defconfig

grep "CONFIG_HELLO_SYSCALL" .config
CONFIG_HELLO_SYSCALL=y
```

- Rebuild kernel and boot the target with new kernel.

System calls: kernel space side (cont.)

- There are different ways to add new syscall to `sys_call_table` for other architectures:
- For x86 - add records to `arch/x86/entry/syscalls/` (see comments on the top of them):
    - `syscall_32.tbl`
    - `syscall_64.tbl`
- For arm64 no additional steps are necessary - it reuses `include/uapi/asm-generic/unistd.h`

System calls: kernel space side (cont.)

- By looking at `<linux/syscalls.h>` we can see that maximum number of arguments syscall can take via fastcall equal to 6. This is hard, and common for all architectures (on x86, which is CISC with reduced number of registers this uses `ebx`, `ecx`, `edx`, `esi`, `edi`, and `eax` to pass syscall number and get status/error after return).
- You may refer to `syscall(2)` to get more information on register usage for syscalls for various ABI.
- But what if we want to pass much more data via syscall?
- For example `uname(2)` syscall takes pointer to struct utsname data structure in address space of the calling process.

Working with data in user space

- It is time to invoke Hello system call using `syscall(2)`.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/syscall.h>
#include <unistd.h>

#define SYS_hello               400
#define HELLO_STR_LEN_MAX       64

int main(void)
{
        int ret;
        char hello_buf[HELLO_STR_LEN_MAX];

        printf("Invoke SYS_hello (%d) demo system call...\n", SYS_hello);

        ret = syscall(SYS_hello, hello_buf, HELLO_STR_LEN_MAX);
        if (ret == -1)
                printf("failed, %s (%d)\n", strerror(errno), errno);
        else
                printf("success, the result is: \"%s\"\n", hello_buf);

        return ret;
}
```

Working with data in user space (cont.)

- And check if new system call works.

```
/ # sys-hello
Invoke SYS_hello (400) demo system call...
success, response is "Hello from kernel!"
```

- As we can see it works as expected - we got ”Hello from kernel!” string in the provided buffer.

Working with data in user space (cont.)

- This is most general and important part in kernel<->user space communication as it poses security risks, as well as few other issues.
- It does not depend whenever you implementing syscall and need to take data from user space or take data via `read()`/`write()`/`ioctl()`/etc methods.
- Let's look closely how to work with data supplied by user space process in kernel context.
- In general to access any size of user space data from the kernel we need to use special kernel provided primitives that will perform all necessary checks before copying data from user space to kernel space and vise versa.
- Yes, we need to copy data to/from user/kernel context before we using it.

Working with data in user space (cont.)

- We known that when performing system call via any supported method context switch from user to kernel happened and back from kernel to user on syscall return.
- But how kernel can access user space address space if it runs in it's own, kernel address space and on syscall address space should change?
- In fact on context switch address space is not changed. We run in kernel with user space process address space. Therefore can access any page in it.
- This serves two purposes:
  - Simplify access to user space process data
  - Avoid overhead of address space change and TLB flushes on syscalls.

- Wow, but how this might happen? To answer - look at process memory map:

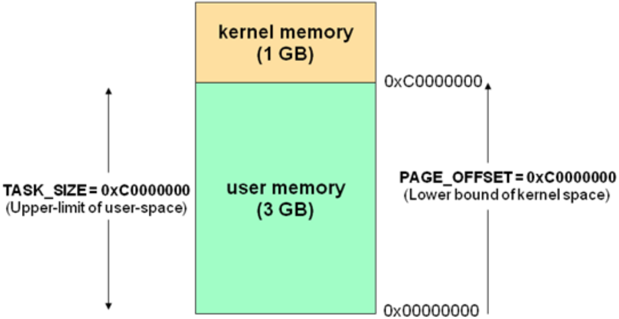

Figure 2: A process memory map

Working with data in user space (cont.)

- Let’s summarize that we say before:
  - In Linux Kernel each process as it’s own address space: that’s implies isolation.
  - Threads, as expected share address space between each other, except for kernel/user stacks.
  - Each process has mapped kernel space at high addresses, but has no permissions to access it (even for read).
  - During context switch due to syscall no address space switch happened: kernel executes syscall handler code in it’s .text pages, able to access all kernel address space + user process address space.
  - Stack is switched from user to kernel when executing syscall as we should not trust stack pointer to be valid when entering to syscall: that’s why kernel always uses fastcall to pass parameters via registers, not memory.
  - User process memory may or may not be valid and present: we need to check this in kernel

Working with data in user space (cont.)

- Okay, we need special checks to be made by kernel before accessing memory. Why we still need to copy from user to kernel memory?
- Consider following scenario:
  - User process has buggy (rogue?) implementation of threading.
  - One thread prepares data in process address space and invokes syscall that takes as parameter data structure that itself contains pointers.
  - On behalf of user process kernel validates supplied pointer to the data structure in memory and all pointers contained in it, ensures at least specified size of data is present in memory (if not, paging is happened and data is loaded into memory).
  - After such validation it starts working with this data on CPU0.
  - On CPU1 second thread corrupts this data while CPU0 working with it. In general results isn't predictable. When "corruption" is crafted specially - this poses huge security risk.

Working with data in user space (cont.)

- Working with user space memory correctly is highly important. But what mechanisms kernel does provide for us?
- It is set of API that should be used to copy data to/from user/kernel space.
- They defined in `<linux/uaccess.h>` and headers they include (e.g. `<asm/uaccess.h>`)
- Implementation is generally architecture dependent, but at least following actions are made when using them:
- Check that pointer to data structure and pointer + data size **`is in user address space`**
- Check there is no conflict between requested access type (`VERIFY_READ`, `VERIFY_WRITE`) and memory access permissions
- As said before user memory might be mapped, but not present (e.g. swapped out to disk) there primitives, unless specified may trigger pagefault.

Working with data in user space (cont.)

- Ok, but I want to copy to/from user/kernel space in atomic context and it is said before copy primitives might sleep if page faults is enabled and memory isn't present.
- In short:
  - You must ensure memory valid (i.e. `access_ok` is true), present before atomic section
  - Disable page faults via corresponding API in `<linux/uaccess.h>`
  - Copy memory in atomic context using special primitives, enable page faults
  - Enable page faults
- Here is API to disable/enable pagefaults:

```
void pagefault_disable(void);
void pagefault_enable(void);
```

Working with data in user space (cont.)

- Here is general API used for copy to/from user/kernel space:

```
/**
 * get_user: - Get a simple variable from user space.
 * @x:   Variable to store result.
 * @ptr: Source address, in user space.
 *
 * Context: User context only. This function may sleep if pagefaults are
 *          enabled.
 * Returns zero on success, or -EFAULT on error.
 * On error, the variable @x is set to zero.
 */
#define get_user(x, ptr)
```

Working with data in user space (cont.)

- Here is general API used for copy to/from user/kernel space (cont.):

```
/**
 * put_user: - Write a simple value into user space.
 * @x:   Value to copy to user space.
 * @ptr: Destination address, in user space.
 *
 * Context: User context only. This function may sleep if pagefaults are
 *          enabled.
 * @ptr must have pointer-to-simple-variable type, and @x must be assignabl
 * to the result of dereferencing @ptr.
 *
 * Returns zero on success, or -EFAULT on error.
 */
#define put_user(x, ptr)
```

Working with data in user space (cont.)

- Here is general API used for copy to/from user/kernel space (cont.):

```
static inline unsigned long __must_check
copy_from_user(void *to, const void __user *from, unsigned long n);

static inline unsigned long __must_check
copy_to_user(void __user *to, const void *from, unsigned long n);

extern __must_check long
strncpy_from_user(char *dst, const char __user *src, long count);

extern __must_check long strlen_user(const char __user *str);
extern __must_check long strnlen_user(const char __user *str, long n);

extern __must_check unsigned long
clear_user(void __user *mem, unsigned long len);
```

Calling syscalls from kernel

- Since syscall handler are ordinary, visible kernel functions they can be called from the kernel code with few considerations.
- Main consideration is that system call handlers are intended to be called with pointers pointing to user space process memory and as we known primitives that copy data to/from kernel/user space data strictly check that!
- Other consideration is that few (none?) of syscall handler functions defined with `EXPORT_SYMBOL()` and therefore they are not available to direct use by kernel modules: use `kallsyms_lookup_name()` from <linux/kallsyms.h> to find address of syscall handler functions.

References

References

- Protection ring
- vdso(7) vdso - overview of the virtual ELF dynamic shared object
- Documentation/ABI/stable/vdso
- Adding a New System Call
- Robert Love. Linux Kernel Development. Third Edition. 2010. Ch 5: System calls.

Assignments

Assignment

- Add system call `sys_hello2` which should be implemented at least for the target architecture (arm).
- The system call should be called with 2 parameters:
  - pointer to the input data structure instance `struct hello2_in`
  - pointer to the output data structure instance `struct hello2_out`

Assignment (cont.)

- `sys_hello2` should output (print) the fields from an input structure to the kernel ring buffer (dmesg, serial console)

```
pr_info("%s: sys_hello2(name : %s, len : %zu, val : %u\n", __func__, ...);
```

- `sys_hello2` should fill the output structure fields:
  - `nr_calls` - with current number of calls;
  - `rand_data[4]` - with random bytes.

Assignment (cont.)

- Implement error handling
  - In case of success `sys_hello2` should return 0;
  - In case of error `sys_hello2` should return negative error code (e.g. -EFAULT).
- Create user space application to use new system call by `syscall(2)`

Assignment (cont.)

Use the following data structures:

- Input data structure:

```c
struct hello2_in {
        char *name;
        unsigned int len;
        unsigned int val;
};
```

- Output data structure:

```c
struct hello2_out {
        unsigned int nr_calls;              /* num of sys_hello2 calls */
        unsigned int rand_data[4];          /* random bytes */
};
```

Assignment (cont’d)

- Send the following results:
  - `git format-patch ...` (prefferable) or `git diff > sys_hello2.diff` for kernel tree modifications;
  - User space application sources;
  - Short description, etc.

Thank you!