System Calls

Lecture 2

Bibliography

for this section

1. **Alexandru Radovici, Ioana Culic**, *Getting Started with Secure Embedded Systems*
   - Chapter 3 - *The Tock system architecture*
2. **Daniel P. Bovet, Marco Cesati**, *Understanding the LINUX KERNEL*
   - Chapter 10 - *System Calls*

System Calls

- What is a system call?
- What is vDSO?
- How a system call is performed?
- Tock system calls
- Linux system calls

Operating System

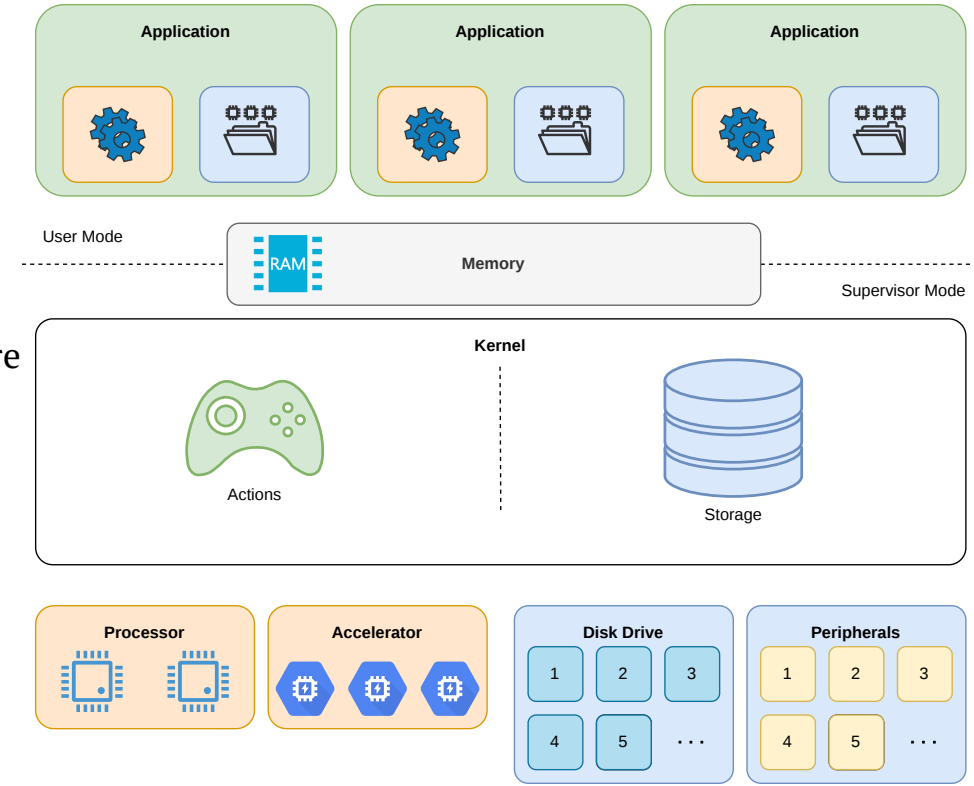
the main role

Allow Portability

- provides a hardware independent API
- applications should run on any hardware

Resources Management and Isolation

- allow applications to access resources
- prevent applications from accessing hardware directly
- isolate applications



System Call

the OS API

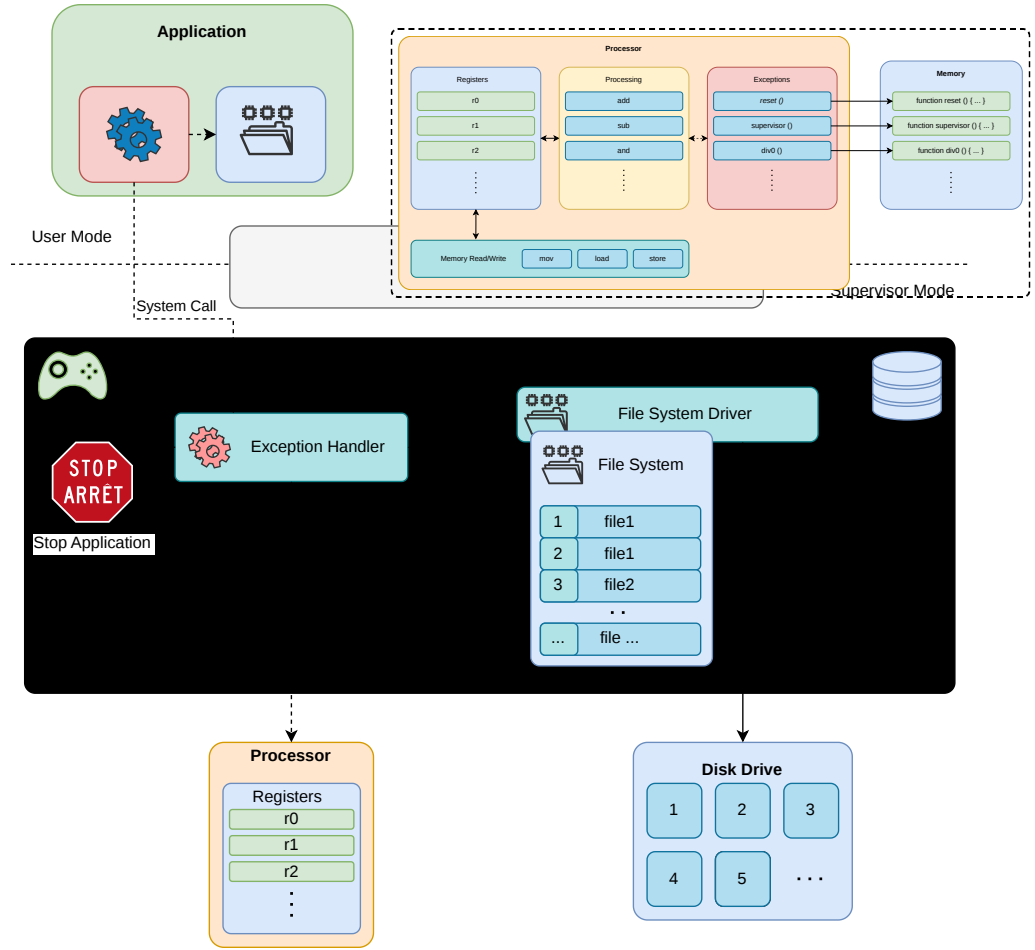
accessing hardware can be **performed** only **by the kernel**

The application:

1. puts values in the registers / stack
2. triggers an exception
   - `svc` instruction for ARM
   - `sysenter` instruction for x86

The kernel:

1. looks at the registers and determines what the required action is
2. performs the action
3. puts the return values in registers / stack



vDSO

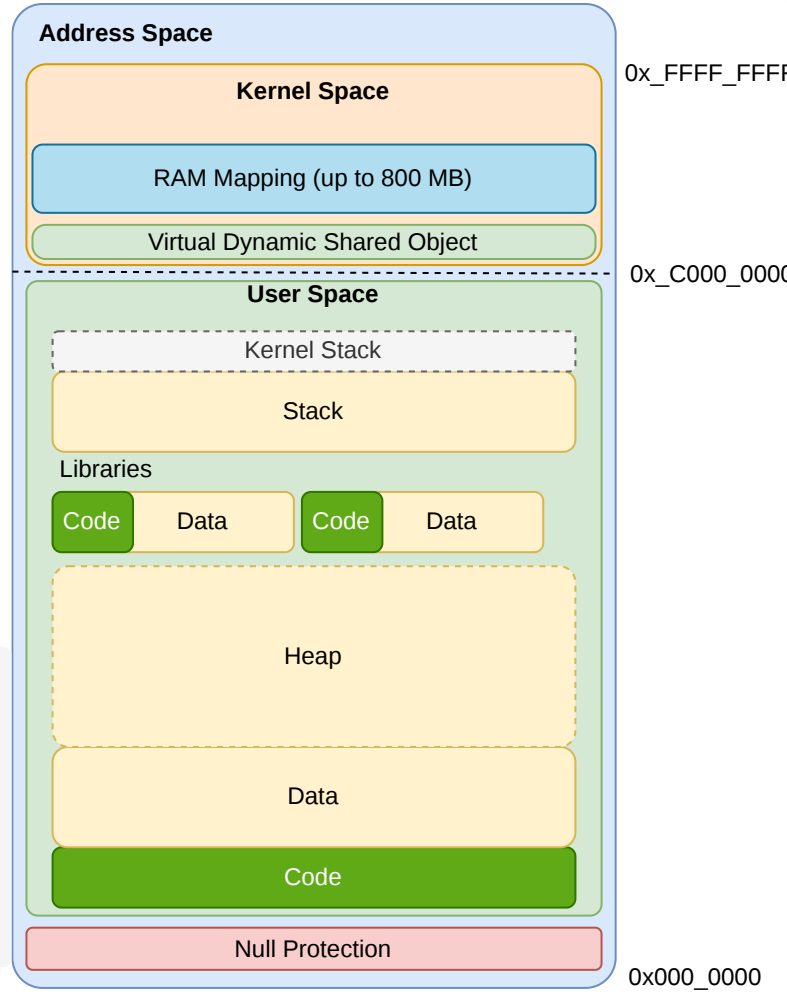
Virtual Dynamic Shared Object

Address Space

with vDSO

- *kernel memory* that can be read from userspace
- system calls that can run in userspace (examples)
  - `getpid`
  - `gettimeofday`
  - `gettime`
- Linux implements it as an ELF object `libvdso.so`
  - lookable by process loaders

```
Symbol table '.dynsym' contains 11 entries:
Num: Value   Size Type  Bind   Name
2: ff700600  727 FUNC    WEAK     clock_gettime@@LINUX_2.6
4: ff7008e0  365 FUNC    GLOBAL   __vdso_gettimeofday@@LINUX_2.6
5: ff700a70   61 FUNC    GLOBAL   __vdso_getcpu@@LINUX_2.6
6: ff7008e0  365 FUNC    WEAK     gettimeofday@@LINUX_2.6
7: ff700a50   22 FUNC    WEAK     time@@LINUX_2.6
...
```



System Call for Tock OS

RISC-style system calls

Memory Layout

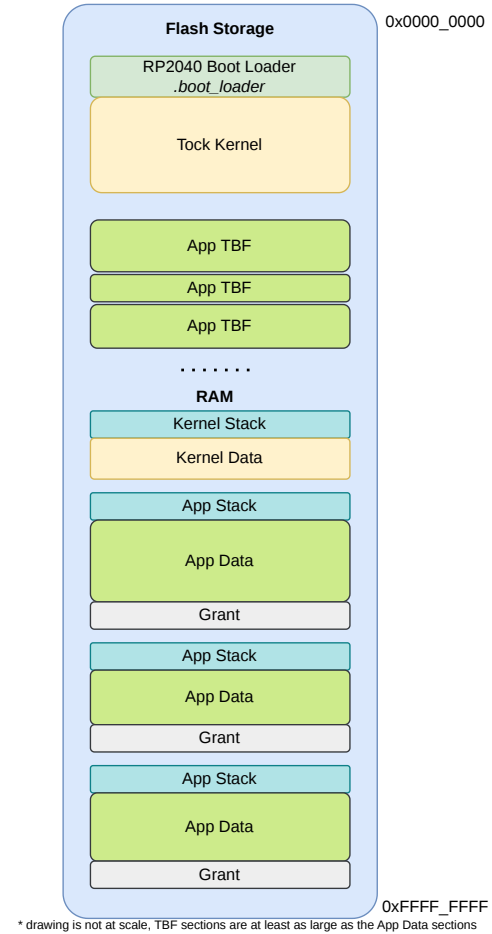
for the RP2040

Kernel

- is written in flash separated from the apps
- loads each app at boot

Applications

- each application TBF is written to the flash separately
- each application has a separate
  - *stack* in RAM
  - *grant* section where the kernel stores data about the app
  - *data* section in RAM



* drawing is not at scale, TBF sections are at least as large as the App Data sections

Memory Layout

for the RP2040 at runtime

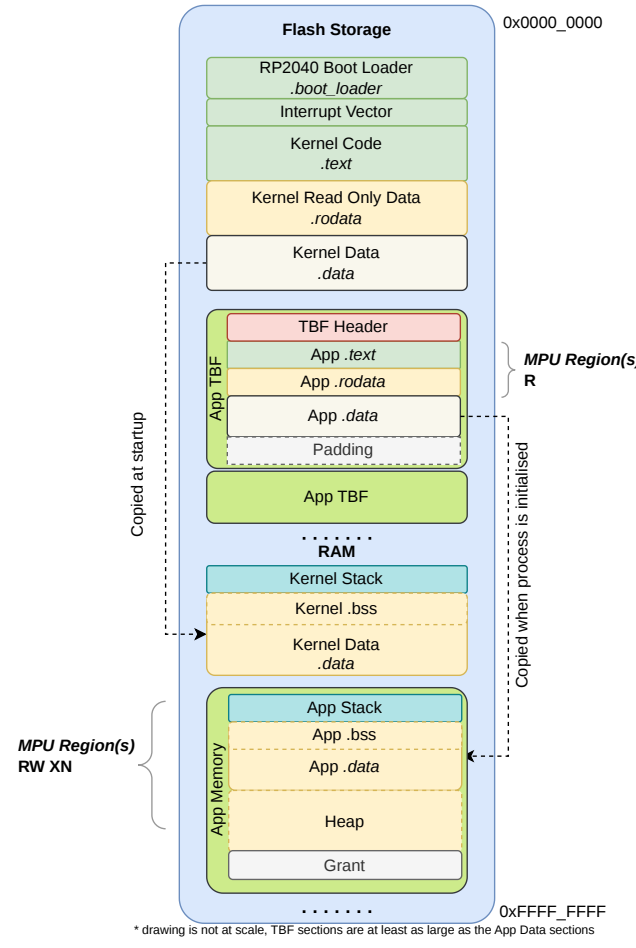
Kernel

- sets up the MPU every time it switches to a process

Applications

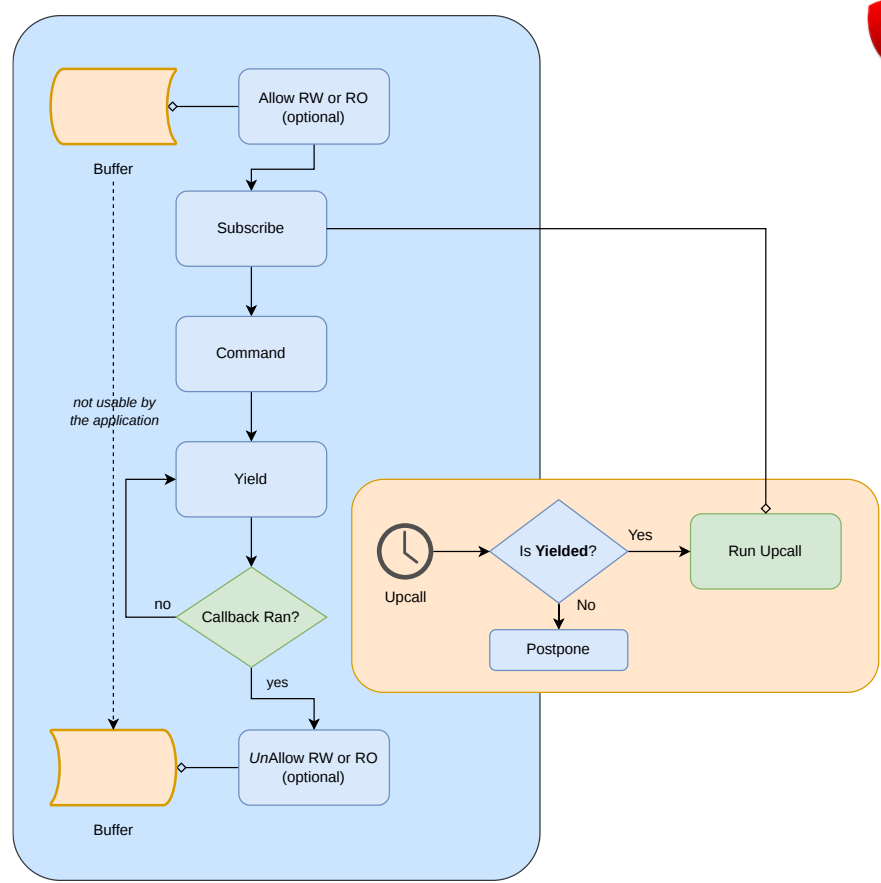
- can read and execute its code
- can read and write its *stack* and *data*
- can read and write the *allocated heap*

Applications are **not allowed** to access the **kernel's memory** or **the peripherals**.



System Calls

0. Yield
1. Subscribe
2. Command
3. ReadWriteAllow
4. ReadOnlyAllow
5. Memop
6. Exit
7. UserspaceReadableAllow



5: Memop

Memop expands the memory segment available to the process, allows the process to retrieve pointers to its allocated memory space, provides a mechanism for the process to tell the kernel where its stack and heap start, and other operations involving process memory.

```
memop(op_type: u32, argument: u32) -> [[ VARIES ]] as u32
```

Arguments

- `op_type` : An integer indicating whether this is a `brk` (0), a `sbrk` (1), or another memop call.
- `argument` : The argument to `brk` , `sbrk` , or other call.

Each memop operation is specific and details of each call can be found in the memop syscall documentation.

Return

- Dependent on the particular *memop* call.

6: Exit

The process signals the kernel that it has no more work to do and can be stopped or that it asks the kernel to restart it.

```
tock_exit(completion_code: u32)
tock_restart(completion_code: u32)
```

Return

None

2: Command

Command instructs the driver to perform a specific action.

```
command(driver: u32, command_number: u32, argument1: u32, argument2: u32) -> CommandReturn
```

Arguments

- `driver`: integer specifying which driver to use
- `command_number`: the requested command.
- `argument1`: a command-specific argument
- `argument2`: a command-specific argument

One Tock convention with the *Command* system call is that command number 0 will always return a value of 0 or greater if the driver is present.

Return

- three `u32` numbers
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `command_number`.
  - Other return codes based on the specific driver.

1: Subscribe

Subscribe assigns upcall functions to be executed in response to various events.

```
subscribe(driver: u32, subscribe_number: u32, upcall: u32, userdata: u32) -> Result<Upcall, (Upcall, ErrorCode)>
```

Arguments

- `driver` : integer specifying which driver to use
- `subscribe_number` : event number
- `upcall` : function's pointer to call upon event

```
void upcall(int arg1, int arg2, int arg3, void* userdata)
```

- `userdata` : value that will be passed back, usually a pointer

Return

- The previously registered upcall or `TOCK_NULL_UPCALL`
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `subscribe_number`.

0: Yield

Yield transitions the current process from the Running to the Yielded state.

```
// waits for the next upcall
// The process will not execute again until another upcall re-schedules the process.
yield()

// does not wait for the next upcall
// If a process has no enqueued upcalls, the process immediately re-enters the Running state.
yield_no_wait()

// waits for a specific upcall
// The process will not execute again until the desired upcall re-schedules the process.
yield_for(driver_number: u32, subscribe_number: u32);
```

Return

yield: None

yield_no_wait: `0` - there was no queued *upcall* function to execute / `1` - *upcall* ran

yield_for: None

3 and 4: AllowRead(Write/Only)

Allow shares memory buffers between the kernel and application.

```
allow_readwrite(driver: u32, allow_number: u32, pointer: usize, size: u32) -> Result<ReadWriteAppSlice, (ReadWriteAppSlice,
allow_readonly(driver: u32, allow_number: u32, pointer: usize, size: u32) -> Result<ReadWriteAppSlice, (ReadWriteAppSlice,
```

Arguments

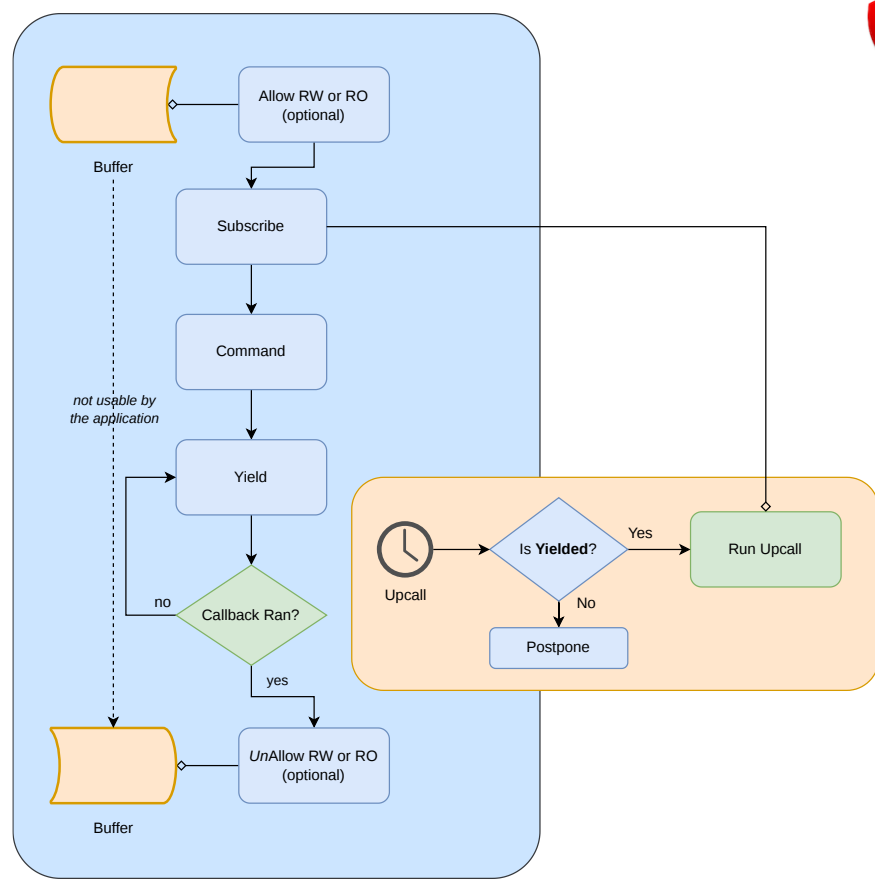
- `driver` : integer specifying which driver to use
- `allow_number` : driver-specific integer specifying the purpose of this buffer
- `pointer` : pointer to the buffer in the process memory space
  - null pointer revokes a previously shared buffer
- `size` : the length of the buffer

Return

- The previous allowed buffer or NULL
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `allow_number` .
  - `INVAL` the buffer referred to by `pointer` and `size` lies completely or partially outside of the processes addressable RAM.

System Call Pattern

1. *allow*: if data exchange is required, share a buffer with a driver
2. *subscribe* to the *action done* event
3. send a *command* to ask the driver to start performing an action
4. *yield* to wait for the *action done* event
   - *the kernel calls a callback*
   - verify if the expected event was triggered, if not *yield*
5. *unallow*: get the buffer back from the driver



Making a system call

ARM Cortex-M

```
1   syscall_return_t command(uint32_t driver, uint32_t command,
2                            int arg1, int arg2) {
3     register uint32_t r0 __asm__ ("r0") = driver;
4     register uint32_t r1 __asm__ ("r1") = command;
5     register uint32_t r2 __asm__ ("r2") = arg1;
6     register uint32_t r3 __asm__ ("r3") = arg2;
7     register uint32_t rtype __asm__ ("r0");
8     register uint32_t rv1 __asm__ ("r1");
9     register uint32_t rv2 __asm__ ("r2");
10    register uint32_t rv3 __asm__ ("r3");
11    __asm__ volatile (
12      "svc 2"
13      : "=r" (rtype), "=r" (rv1), "=r" (rv2), "=r" (rv3)
14      : "r" (r0), "r" (r1), "r" (r2), "r" (r3)
15      : "memory"
16      );
17    syscall_return_t rval = {rtype, {rv1, rv2, rv3}};
18    return rval;
19  }
```

Making a system call

x86

```
1   push    0           # arg 4: unused
2   push    0           # arg 3: unused
3   push    0           # arg 2: unused
4   push    1           # arg 1: yield-wait
5   mov     eax, 0      # class: yield
6   int     0x40
7   add     esp, 16     # clean up stack
```

- performs a trap `int 04h`
- parameters are sent on the stack

> *In contrast with other embedded architectures like ARM or RISC-V, x86 does not have very many general purpose registers to spare. The ABI defined here draws heavily from the `cdecl` calling convention by using the stack instead of registers to pass data between user and kernel mode.*

System call dispatcher

```
match syscall {
  Syscall::Memop { operand, arg0 } => { /* ... */ }
  Syscall::Yield { which, param_a, param_b } => { /* ... */ }
  Syscall::Subscribe { driver_number, .. }
  | Syscall::Command { driver_number, .. }
  | Syscall::ReadWriteAllow { driver_number, .. }
  | Syscall::UserspaceReadableAllow { driver_number, .. }
  | Syscall::ReadOnlyAllow { driver_number, .. } => {
      resources
      .syscall_driver_lookup()
      .with_driver(driver_number, |driver| match syscall {
        Syscall::Subscribe {driver_number, subdriver_number, upcall_ptr, appdata} => { /* d.subscribe (...) */ }
        Syscall::Command {driver_number, subdriver_number, arg0, arg1} => { /* d.command(...) */ }
        Syscall::ReadWriteAllow {driver_number, subdriver_number, allow_address allow_size} => { /* d.read_write_allo
        Syscall::UserspaceReadableAllow {driver_number, subdriver_number, allow_address, allow_size} => { /* d.usersp
        Syscall::ReadOnlyAllow {driver_number, subdriver_number, allow_address, allow_size} => { /* d.read_only_allow
        Syscall::Yield { .. }
        | Syscall::Exit { .. }
        | Syscall::Memop { .. } => { debug_assert!(false, "Kernel system call handling invariant violated!"); },
      })
  }
  Syscall::Exit { which, completion_code } => { /* stop or restart process */ }
}
```

Address Verification

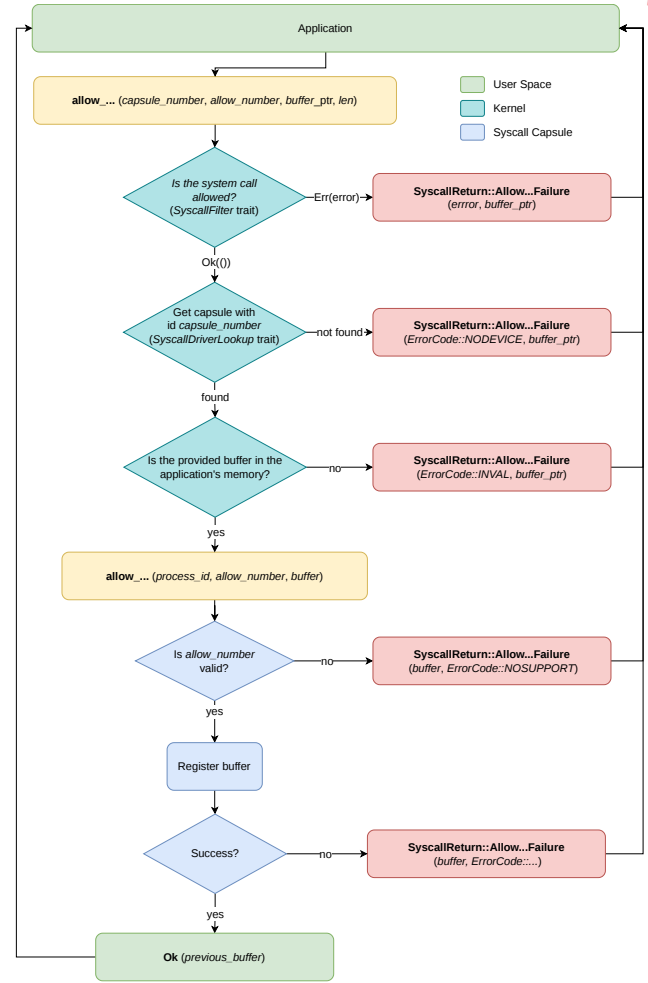
- memory is shared only through
  - *ReadOnlyAllow* / *ReadWriteAllow*
  - *YieldNoWait* - single point of use in kernel

Allow

- kernel verifies the buffer (address and length)
- build a *safe* `ProcessBuffer` that capsules use
- `memop` cannot reduce the process's memory
- the drivers needs to check if the buffer exists

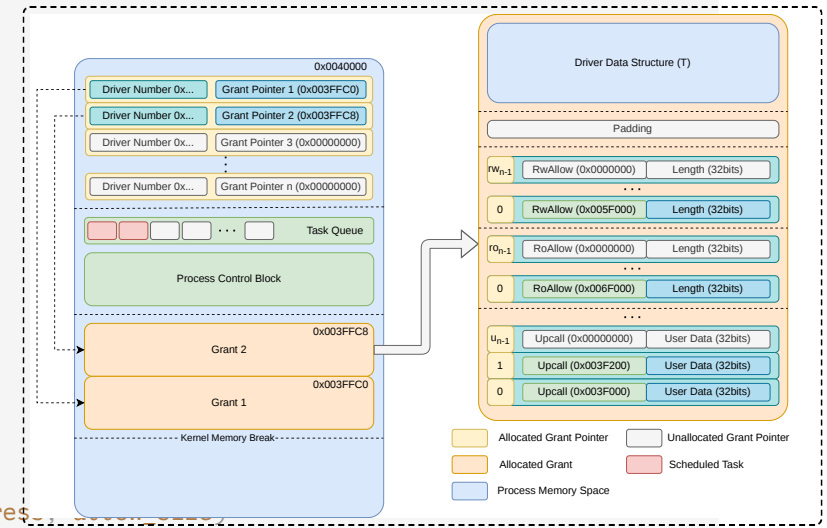
YieldNoWait

- kernel verifies the address every time
  - costly
  - works on non-MMU systems, not that slow



Allow System Calls (kernel)

```
1   match process.build_readwrite_process_buffer(allow_address, allow_size) {
2     Ok(rw_pbuf) => {
3         match crate::grant::allow_rw(process, driver_number, subdriver_number, rw_pbuf) {
4             Ok(rw_pbuf) => {
5                 let (ptr, len) = rw_pbuf.consume();
6                 SyscallReturn::AllowReadWriteSuccess(ptr, len)
7             }
8             Err((rw_pbuf, err @ ErrorCode::NOMEM)) => {
9               // simplified version
10              let (ptr, len) = rw_pbuf.consume();
11              SyscallReturn::AllowReadWriteFailure(err, ptr, len)
12            }
13            Err((rw_pbuf, err)) => {
14                let (ptr, len) = rw_pbuf.consume();
15                SyscallReturn::AllowReadWriteFailure(err, ptr, len)
16            }
17        }
18    }
19    Err(allow_error) => {
20        SyscallReturn::AllowReadWriteFailure(allow_error, allow_addres
21    }
22  }
```

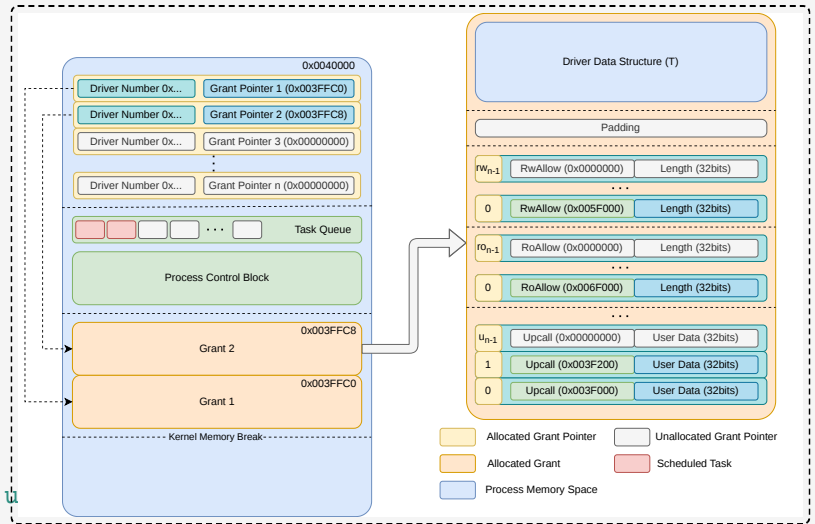


Allow System Call (driver)

```rust
struct App { /* per app stored driver data */ }

pub struct Console<'a> {
      apps: Grant<
          App,
          UpcallCount<{ upcall::COUNT }>,
          AllowRoCount<{ ro_allow::COUNT }>,
          AllowRwCount<{ rw_allow::COUNT }>
      >,
    // ...
}

impl SyscallDriver for Console<'_> {
    fn command(&self, cmd_num: usize, arg1: usize, arg2: u
        self.apps.enter(processid, |app, kernel_data| {
            if let Some(buffer) = kernel_data.get_readwrite_processbuffer(0 /* buffer_number */) {
                // access the `buffer`
            }
        }
    }
}
```



YieldNoWait System Call

```
match which.try_into() {
    Ok(YieldCall::NoWait) => {
        let has_tasks = process.has_tasks();

        // Set the "did I trigger upcalls" flag.
        // If address is invalid does nothing.
        unsafe {
            let address = param_a as *mut u8;
            process.set_byte(address, has_tasks as u8);
        }

        if has_tasks {
            process.set_yielded_state();
        }
    }

    Ok(YieldCall::Wait) => { /* ... */ }

    Ok(YieldCall::WaitFor) => { /* ... */ }

    _ => { /* return to process */ }
}
```

System Call for Linux

CISC-like system calls

Linux's System Call

for 32 bit x86 processors

Number	Syscall Name	Description	Arguments (eax, ebx, ecx, edx, esi, edi)
1	sys_exit	Exit a process	(exit_code)
2	sys_fork	Create a child process	(none)
3	sys_read	Read from file descriptor	(fd, buf, count)
4	sys_write	Write to file descriptor	(fd, buf, count)
5	sys_open	Open a file	(filename, flags, mode)
6	sys_close	Close a file descriptor	(fd)
7	sys_waitpid	Wait for child process	(pid, status, options)
8	sys_creat	Create a file	(filename, mode)
9	sys_link	Create a hard link	(oldpath, newpath)
10	sys_unlink	Remove a file	(filename)
11	sys_execve	Execute a program	(filename, argv, envp)
12	sys_chdir	Change working directory	(path)
13	sys_time	Get system time	(tloc)
14	sys_mknod	Create a special file	(filename, mode, dev)
15	sys_chmod	Change file permissions	(filename, mode)
16	sys_lchown	Change owner of a file (symbolic)	(filename, owner, group)
19	sys_lseek	Move file read/write pointer	(fd, offset, whence)
20	sys_getpid	Get process ID	(none)
29	sys_pause	Wait for signal	(none)
37	sys_kill	Send signal to a process	(pid, signal)
45	sys_brk	Change data segment size	(addr)
54	sys_ioctl	Device-specific I/O operations	(fd, request, argp)
78	sys_gettimeofday	Get current time	(tv, tz)
90	sys_mmap	Map memory	(addr, length, prot, flags, fd, offset)
91	sys_munmap	Unmap memory	(addr, length)
102	sys_socketcall	Socket system calls wrapper	(call, args)
120	sys_clone	Create a new process (thread)	(flags, child_stack, ptid, tls, ctid)
122	sys_uname	Get system information	(buf)
140	sys_llseek	Large file seek	(fd, offset_high, offset_low, result, whence)
162	sys_nanosleep	Sleep for a given time	(rqtp, rmtp)
168	sys_poll	Wait for I/O events	(fds, nfds, timeout)
183	sys_getcwd	Get current working directory	(buf, size)
252	sys_exit_group	Exit all threads in process	(exit_code)

Linux's System Calls

for 64 bit x86 processors

Syscall Number	Syscall Name	Description	Arguments (rax, rdi, rsi, rdx, r10, r8, r9)
60	sys_exit	Exit a process	(exit_code)
39	sys_fork	Create a child process	(none)
0	sys_read	Read from file descriptor	(fd, buf, count)
1	sys_write	Write to file descriptor	(fd, buf, count)
2	sys_open	Open a file	(filename, flags, mode)
3	sys_close	Close a file descriptor	(fd)
9	sys_mmap	Memory mapping	(addr, length, prot, flags, fd, offset)
11	sys_execve	Execute a program	(filename, argv, envp)
16	sys_lseek	Change file offset	(fd, offset, whence)
20	sys_getpid	Get process ID	(none)
23	sys_getppid	Get parent process ID	(none)
26	sys_kill	Send a signal to a process	(pid, signal)
33	sys_nanosleep	Sleep for a given time	(rqtp, rmtp)
41	sys_socket	Create a socket	(domain, type, protocol)
42	sys_connect	Connect a socket to a remote address	(fd, addr, addrlen)
57	sys_clone	Create a new process (thread)	(flags, child_stack, ptid, tls, ctid)
59	sys_wait4	Wait for process to change state	(pid, status, options, rusage)
72	sys_fstatat	Get file status	(dirfd, pathname, statbuf, flags)
87	sys_munmap	Unmap memory	(addr, length)
93	sys_ioctl	Device-specific I/O operations	(fd, request, argp)
104	sys_set_tid_address	Set thread ID address	(tid)
115	sys_fadvise64	Advise on file I/O operations	(fd, offset, len, advice)
156	sys_prlimit64	Get or set resource limits	(pid, resource, new_limit, old_limit)
183	sys_getcwd	Get current working directory	(buf, size)
231	sys_uname	Get system information	(buf)
263	sys_exit_group	Exit all threads in the process group	(exit_code)

Making a system call

for 32 bit x86 processors

- put the arguments in registers
- switch to *supervisor mode*
  - make a *trap* - `int 80h` (Linux) / `int 20h` (Windows)
  - call a specific instruction like `sysenter` (Intel) or `syscall` (AMD)

```
1   mov eax, syscall_number
2   mov ebx, arg1
3   mov ecx, arg2
4   mov edx, arg3
5   mov esi, arg4
6   mov edi, arg5
7   ; trap
8   int 0x80
```

- the single return value is placed in `eax`

System Call Dispatcher

```
#define __SYSCALL_I386(nr, sym, qual) [nr] = sym,

const sys_call_ptr_t ia32_sys_call_table[] = {
  [0 ... __NR_syscall_compat_max] = &sys_ni_syscall,
  #include <asm/syscalls_32.h>
};

__SYSCALL_I386(0, sys_restart_syscall)
__SYSCALL_I386(1, sys_exit)
__SYSCALL_I386(2, sys_fork)
__SYSCALL_I386(3, sys_read)
__SYSCALL_I386(4, sys_write)
#ifdef CONFIG_X86_32
__SYSCALL_I386(5, sys_open)
#else
__SYSCALL_I386(5, compat_sys_open)
#endif
__SYSCALL_I386(6, sys_close)
```

```
/* Handles int $0x80 */
void do_int80_syscall_32(struct pt_regs *regs)
{
    enter_from_user_mode();
    local_irq_enable();
    do_syscall_32_irqs_on(regs);
}

/* simplified version of the Linux x86 32bit System Call
   Dispatcher */
static void do_syscall_32_irqs_on(struct pt_regs *regs)
{
    unsigned int nr = regs->orig_ax;

    if (nr < IA32_NR_syscalls)
        regs->ax = ia32_sys_call_table[nr]
            (regs->bx, regs->cx,
             regs->dx, regs->si,
             regs->di, regs->bp);
    syscall_return_slowpath(regs);
}
```

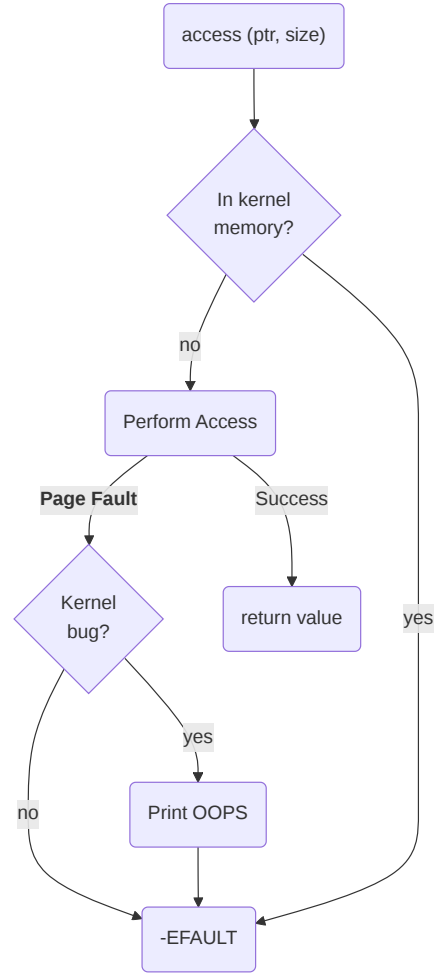
Accessing memory from userspace

pointers from userspace have to be validated

1. Is the address in the kernel's memory? - `if`
2. Is the address in the process' address space?
   - difficult with `if` - takes time
   - use MMU faults
3. Access the memory
   - works -> return the value
   - faults -> figure out why?

Possible Faults

1. copy-on-write, demand paging or reserved but not committed page
2. faulty address
3. kernel bug



Memory Access API

- The kernel API provides special userspace memory access *functions / macros*
- Drivers and kernel code **have to** access userspace memory only through these

```c
/* OK: return -EFAULT if user_ptr is invalid */
if (copy_from_user(&kernel_buffer, user_ptr, size))
    return -EFAULT;

/* Not OK: only works if user_ptr is valid
   otherwise crashes kernel */
memcpy(&kernel_buffer, user_ptr, size);
```

```c
// Is the address in the kernel's memory?
int access_ok(const void * addr, unsigned long size) {
  unsigned long a = (unsigned long) addr;
  if (a + size < a ||
    a + size > current_thread_info()->addr_limit.seg)
    return 0;
  return 1;
}
```

Function / Macro	Description
`get_user()`	Safely retrieves a single value from user-space memory and copies it into kernel-space.
`put_user()`	Safely stores a single value from kernel-space into user-space memory.
`copy_from_user()`	Copies a block of memory from user-space to kernel-space.
`copy_to_user()`	Copies a block of memory from kernel-space to user-space.
`access_ok()`	Checks if the user-space address is valid and accessible.
`clear_user()`	Clears a region of memory in user-space (sets bytes to zero).

Fault cause?

Is it a wrong address or a kernel bug?

The `get_user` functions

```
__get_user_1:       ; get 1 byte
    1: movzx   edx, byte ptr [eax]
    ...
__get_user_2:       ; get 2 bytes
    2: movzx   edx, word ptr [eax - 1]
    ...
__get_user_4:       ; get 4 bytes
    3: mov     edx, dword ptr [eax - 3]
    ...
bad_get_user:
    xor     edx, edx
    mov     eax, -EFAULT
    ret

.section __ex_table, "a"    ; Exception table
    .long   1b, bad_get_user, ex_handler_default
    .long   2b, bad_get_user, ex_handler_default
    .long   3b, bad_get_user, ex_handler_default
.previous
```

Simulates a the behaviour of the `cmp` instruction

```
// Called by the page fault handler
int fixup_exception(struct pt_regs *regs, int trapnr)
{
    const struct exception_table_entry *e;
    ex_handler_t handler;

    e = search_exception_tables(regs->ip);
    if (!e)
        // no handler, this is a kernel bug
        return 0;

    handler = ex_fixup_handler(e);
    return handler(e, regs, trapnr);
}

bool ex_handler_default(const struct exception_table_entry *f
                        struct pt_regs *regs, int trapnr)
{
    // jump to the `if-fault address`
    regs->ip = ex_fixup_addr(fixup);
    return true;
}
```

System call instruction?

`int80h`, `sysenter` or `syscall`

- depends on the processor version
- `sysenter` and `syscall` are faster but not always available
- the kernel and the `libc` must use the same instruction

`vsyscall` vDSO object

- `ysenter_setup()` generates an ELF shared object that exports `vsyscall` that performs the system call
- libc calls `vsyscall` instead of an actual instruction

without `sysenter` - up to Pentium

```
1   __kernel_vsyscall:
2       int 80h
3       ret
```

with `syseneter` - starting with Pentium II

```
1   __kernel_vsyscall:
2       push    ecx
3       push    edx
4       push    ebp
```

Conclusion

we talked about

- What is a system call?
- What is vDSO?
- How a system call is performed?
- Tock system calls
- Linux system calls