

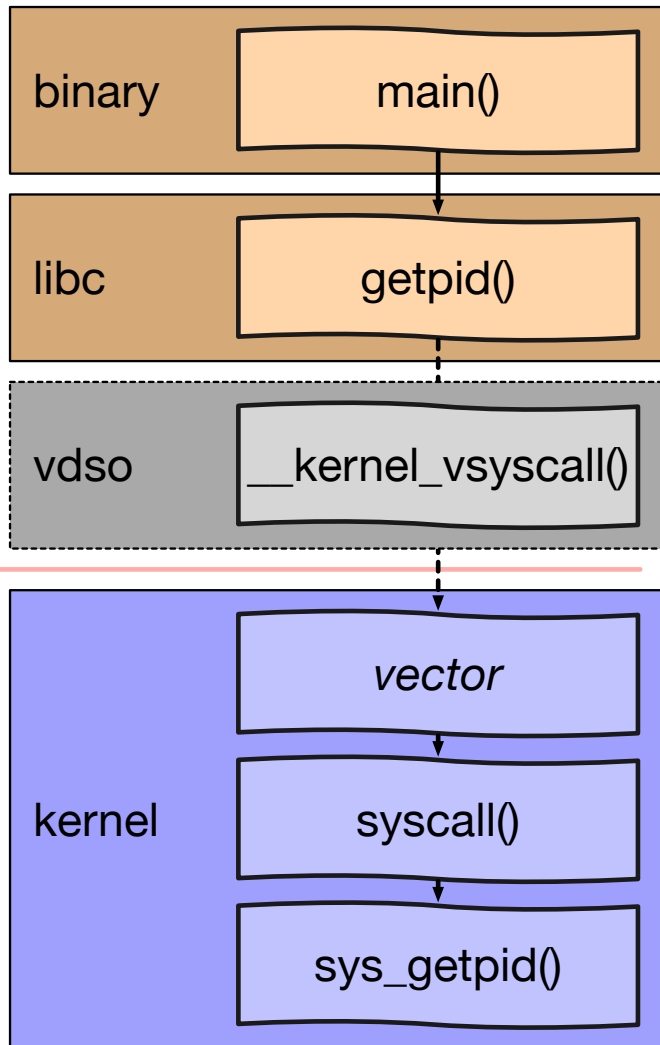
The Process Model (2)

Lecture 4, Part 2: Traps and Syscalls in Practice

Prof. Robert N. M. Watson

2022-2023

Reminder: System-call invocation



- libc system-call stubs provide linkable symbols
- Inline system-call instructions or dynamic implementations
 - Linux vdso
 - Xen hypercall page
- **Machine-dependent trap vector**
- **Machine-independent** function `syscall()`
 - Prologue (e.g., breakpoints, tracing)
 - Actual service invoked
 - Epilogue (e.g., tracing, signal delivery)

Note: This is something of a mashup of the system-call paths of different operating systems, to illustrate how the ideas compose

System-call entry – `syscallenter`

<code>cred_update_thread</code>	Update thread cred from process
<code>sv_fetch_syscall_args</code>	ABI-specific <code>copyin()</code> of arguments
<code>ktrsyscall</code>	ktrace syscall entry
<code>ptracestop</code>	ptrace syscall entry breakpoint
<code>IN_CAPABILITY_MODE</code>	Capsicum capability-mode check
<code>syscall_thread_enter</code>	Thread drain barrier (module unload)
<code>systrace_probe_func</code>	DTrace system-call entry probe
<code>AUDIT_SYSCALL_ENTER</code>	Security event auditing
<code>sa->callp->syscall</code>	System-call implementation! Woo!
<code>AUDIT_SYSCALL_EXIT</code>	Security event auditing
<code>systrace_probe_func</code>	DTrace system-call return probe
<code>syscall_thread_exit</code>	Thread drain barrier (module unload)
<code>sv_set_syscall_retval</code>	ABI-specific return value

- That's a lot of tracing hooks – why so many?

getaudit: return process audit ID

```
int
sys_getaudit(struct thread *td, struct getaudit_args *uap)
{
    int error;

    if (jailed(td->td_ucred))
        return (ENOSYS);
    error = priv_check(td, PRIV_AUDIT_GETAUDIT);
    if (error)
        return (error);
    return (copyout(&td->td_ucred->cr_audit.ai_audit, uap->audit,
        sizeof(td->td_ucred->cr_audit.ai_audit)));
}
```

- Arguments: **Current thread** pointer, system-call **argument struct**
- Security: **lightweight virtualisation, privilege check**
- Copy value to user address space – can't write to it directly!
- No explicit synchronisation as fields are thread-local
- Does it matter how fresh the credential pointer is?

System-call return – `syscallret`

`userret`

- `KTRUSERRET`
- `g_waitidle`
- `addupc_task`
- `sched_userret`

`p_throttled`

`ktrsysret`

`ptracestop`

`thread_suspend_check`

`P_PPWAIT`

Complicated things, like signals

`ktrace syscall return`

Wait for disk probing to complete

System-time profiling charge

Scheduler adjusts priorities

... various debugging assertions...

`racct resource throttling`

Kernel tracing: `syscall return`

`ptrace syscall return breakpoint`

Single-threading check

`vfork wait`

- That is a lot of stuff that largely **never happens**
- The trick is making all of this nothing fast – e.g., via per-thread flags and globals that remain in the data cache

System calls in practice: dd (1)

```
root@rpi4-000:/data # time dd if=/dev/zero of=/dev/null bs=10m count=1
status=none
0.000u 0.035s 0:00.03 100.0%          26+176k 0+0io 0pf+0w
```

```
__sysctl          0
cap_enter         0
cap_fcntls_limit 0
cap_ioctls_limit 0
cap_rights_limit 0
close            0
fstat            0
fstatat         0
ioctl           0
issetugid      0
munmap          0
open            0
openat          0
pread          0
readlink        0
sigaction       0
sigfastblock   0
write           0
mmap            997784
mprotect       1017154
read           25010967
```

27025905

Zero execution times probably reflect coarse-grained DTrace timer granularity on FreeBSD/arm64

System calls in practice: dd (2)

```
root@rpi4-000:/data # time dd if=/dev/zero of=/dev/null bs=1000m count=1
status=none
0.000u 2.838s 0:02.83 100.0% 23+154k 0+0io 0pf+0w
```

```
profile::profile-997 /execname == "dd"/ {
    @traces[stack()] = count();
}
```

The two most frequent
kernel stack traces

```
kernel`uiomove_faultflag+0x14c
kernel`uiomove_faultflag+0x148
kernel`zero_read+0x3c
kernel`devfs_read_f+0xd0
kernel`dofileread+0x7c
kernel`sys_read+0xbc
kernel`do_el0_sync+0x448
kernel`handle_el0_sync+0x90
527
```

Trace taken
while copying
zeros from
kernel to user
buffer

```
kernel`vm_fault+0xb64
kernel`vm_fault+0xb60
kernel`vm_fault_trap+0x60
kernel`data_abort+0xf4
kernel`handle_e11h_sync+0x78
kernel`uiomove_faultflag+0x14c
kernel`zero_read+0x3c
kernel`devfs_read_f+0xd0
kernel`dofileread+0x7c
kernel`sys_read+0xbc
kernel`do_el0_sync+0x448
kernel`handle_el0_sync+0x90
783
```

Trap from kernel to kernel

Trace taken
while processing
a VM fault
during memory
copy to
userspace

Trap from user to kernel

```
static void
vm_fault_zerofill(struct faultstate *fs)
{
    /*
     * If there's no object left, fill the page in the top
     * object with zeros.
     */
    if (fs->object != fs->first_object) {
        vm_object_pip_wakeup(fs->object);
        fs->object = fs->first_object;
        fs->pinindex = fs->first_pinindex;
    }
    MPASS(fs->first_m != NULL);
    MPASS(fs->m == NULL);
    fs->m = fs->first_m;
    fs->first_m = NULL;

    /*
     * Zero the page if necessary and mark it valid.
     */
    if (((fs->m->flags & PG_ZERO) == 0) {
        pmap_zero_page(fs->m);
    } else {
        VM_CNT_INC(v_ozfod);
    }
    VM_CNT_INC(v_zfod);
    vm_page_valid(fs->m);
}
```

What have we learned?

- Our benchmark was synthetic (and quite artificial):
 - Read 1GB of zeros from /dev/zero
 - Write 1GB of read zeroes to /dev/null
- Observations:
 - The read(2) system call dominates kernel tracing
 - Zeroes are really copied into user memory
 - The write(2) system call doesn't appear at all
 - The /dev/null implementation elides its memory copy
 - Much of the read(2) time was spent in nested traps
 - The VM system was zeroing the 1GB buffer as it was copied to
 - We were zeroing all the memory twice!
- The security and reliability properties of the process model come with a real cost
- To prevent confused deputies, the process abstraction is also maintained for kernel access to user memory
- The VM system performed most of its work lazily