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

cred_update_thread Update thread cred from process
sv_fetch_syscall_args ABI-specific copyin() of arguments
ktrsyscall ktrace syscall entry
ptracestop ptrace syscall entry breakpoint
IN_CAPABILITY_MODE Capsicum capability-mode check
syscall_thread_enter Thread drain barrier (module unload)
systrace_probe_func DTrace system-call entry probe
AUDIT_SYSCALL_ENTER Security event auditing

sa->callp->sy_call System-call implementation! Woo!
AUDIT_SYSCALL_EXIT Security event auditing
systrace_probe_func DTrace system-call return probe
syscall_thread_exit Thread drain barrier (module unload)
sv_set_syscall_retval ABI-specific return value

• That’s a lot of tracing hooks – why so many?
getauid: return process audit ID

```c
int sys_getauid(struct thread *td, struct getauid_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_auid, uap->auid, sizeof(td->td_ucred->cr_audit.ai_auid)));
}
```

- 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
```

<table>
<thead>
<tr>
<th>System Call</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>__sysctl</td>
<td>0</td>
</tr>
<tr>
<td>cap_enter</td>
<td>0</td>
</tr>
<tr>
<td>cap_fcntls_limit</td>
<td>0</td>
</tr>
<tr>
<td>cap_ioctl_limit</td>
<td>0</td>
</tr>
<tr>
<td>cap_rights_limit</td>
<td>0</td>
</tr>
<tr>
<td>close</td>
<td>0</td>
</tr>
<tr>
<td>fstat</td>
<td>0</td>
</tr>
<tr>
<td>fstatat</td>
<td>0</td>
</tr>
<tr>
<td>ioctl</td>
<td>0</td>
</tr>
<tr>
<td>issetugid</td>
<td>0</td>
</tr>
<tr>
<td>munmap</td>
<td>0</td>
</tr>
<tr>
<td>open</td>
<td>0</td>
</tr>
<tr>
<td>openat</td>
<td>0</td>
</tr>
<tr>
<td>pread</td>
<td>0</td>
</tr>
<tr>
<td>readlink</td>
<td>0</td>
</tr>
<tr>
<td>sigaction</td>
<td>0</td>
</tr>
<tr>
<td>sigfastblock</td>
<td>0</td>
</tr>
<tr>
<td>write</td>
<td>0</td>
</tr>
<tr>
<td>mmap</td>
<td>997784</td>
</tr>
<tr>
<td>mprotect</td>
<td>1017154</td>
</tr>
<tr>
<td>read</td>
<td>25010967</td>
</tr>
</tbody>
</table>

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:
- `vm_fault_zerofill` (527 times)
- `handle_el0_sync` (783 times)

Trace taken while copying zeros from kernel to user buffer.

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

```c
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->pindex = fs->first_pindex;
  }
  MPASS(fs->first_m != NULL);
  MPASS((fs->m == NULL) || (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