Advanced Operating Systems
Through tracing, analysis, and experimentation

ACS/Part III L41: Advanced Operating Systems
Part II: Advanced Operating Systems

Lecture 1, Part 3: Kernel dynamics
Prof. Robert N. M. Watson
2022-2023
The kernel: “Just a C program”?  

- I claimed that the kernel was mostly “just a C program”  
- This is indeed mostly true, especially in higher-level subsystems

<table>
<thead>
<tr>
<th>Userspace</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>crt/csu</td>
<td>locore</td>
</tr>
<tr>
<td>rtld</td>
<td>Kernel linker</td>
</tr>
<tr>
<td>Shared objects</td>
<td>Kernel modules</td>
</tr>
<tr>
<td>main()</td>
<td>main(),platform_start()</td>
</tr>
<tr>
<td>libc</td>
<td>libkern</td>
</tr>
<tr>
<td>POSIX threads API</td>
<td>kthread KPI</td>
</tr>
<tr>
<td>POSIX filesystem API</td>
<td>VFS KPI</td>
</tr>
<tr>
<td>POSIX sockets API</td>
<td>socket KPI</td>
</tr>
<tr>
<td>DTrace</td>
<td>DTrace</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
The kernel: not just *any* C program

• **Core kernel**: ≈3.4M LoC in ≈6,450 files
  - **Kernel runtime**: Run-time linker, object model, scheduler, memory allocator, threads, debugger, tracing, I/O routines, timekeeping
  - **Base kernel**: VM, process model, IPC, VFS w/20+ filesystems, network stack (IPv4/IPv6, 802.11, ATM, …), crypto framework
  - Includes roughly ≈70K lines of assembly over ≈6 architectures
• Alternative C runtime – e.g., SYSINIT, curthread
• Highly concurrent – really very, very concurrent
• Virtual memory makes pointers .. odd
• Debugging features – e.g., WITNESS lock-order verifier
• **Device drivers**: ≈3.0M LoC in ≈3,500 files
  - 415 device drivers (may support multiple devices)
Spelunking the kernel

% ls
Makefile            ddb/          libkern/          nfs/         teken/
amd64/             dev/          mips/            nfsclient/   tests/
arm/               dts/          modules/         nsfsvserver/  tools/
arm64/             fs/           net/             nlm/         ufs/
bsm/               gdb/          net80211/        ofed/        vm/
cam/               geom/         netgraph/        opencrypto/  x86/
cddl/              gnu/          netinet/        powerpc/     xdr/
compat/            i386/         netinet6/       riscv/        xen/
conf/              isa/          netipsec/        rpc/         
contrib/           kern/         netpfif/         security/    
crypto/            kgssapi/      netsmb/         sys/

% ls kern
Make.tags.inc      kern_sendfile.c     subr_prng.c
Makefile           kern_sharedpage.c    subr_prof.c
bus_if.m           kern_shutdown.c     subr_rangeset.c
capabilities.conf  kern_sig.c         subr_rman.c
clock_if.m         kern_switch.c       subr_RTC.c
cpuufreq_if.m      kern_sx.c          subr_sbuf.c
...

• Kernel source lives in /usr/src/sys:
  • kern/ – core kernel features
  • sys/ – core kernel headers
How work happens in the kernel

- Kernel code executes concurrently in multiple threads
  - User threads in the kernel (e.g., a system call)
  - Shared worker threads (e.g., callouts)
  - Subsystem worker threads (e.g., network-stack workers)
  - Interrupt threads (e.g., Ethernet interrupt handling)
  - Idle threads

```
# procstat -at
PID   TID   COMM          TDNAME     CPU PRI STATE   WCHAN
    0   100000  kernel      swapper    -1  84 sleep  swapin
    0   100006  kernel      dtrace_taskq -1  84 sleep    
... 10  100002  idle        -            -1 255 run     
11  100003  intr         swi3: vm     0  36 wait     
11  100004  intr         swi4: clock (0) -1  40 wait     
11  100005  intr         swi1: netisr 0 -1  28 wait     
... 11  100018  intr       intr16: ti_adc0 0  20 wait     
11  100019  intr       intr91: ti_wdt0 0  20 wait     
11  100020  intr       swi0: uart    -1  24 wait     
... 739  100064  login      -            -1 108 sleep  wait
740  100079  csh          -            -1 140 sleep  ttyin
751  100089  procstat     -            0  140 run     
```
Work processing and distribution

• Many operations begin with system calls in a user thread
• But may trigger work in many other threads; for example:
  • Triggering a callback in an interrupt thread when I/O is complete
  • Eventually writing back data to disk from the buffer cache
  • Delayed transmission if TCP isn’t able to send immediately
• We will need to be careful about these things, as not all work we are analysing will be in the obvious user thread
• Multiple mechanisms provide this asynchrony; e.g.:

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callout</td>
<td>Closure called after wall-clock delay</td>
</tr>
<tr>
<td>eventhandler</td>
<td>Closure called for key global events</td>
</tr>
<tr>
<td>task</td>
<td>Closure called .. eventually</td>
</tr>
<tr>
<td>SYSINIT</td>
<td>Function called when module loads/unloads</td>
</tr>
</tbody>
</table>

* Where closure in C means: function pointer, opaque data pointer
Wrapping up

• In this lecture, we have:
  • Explored the idea of an operating system
  • Detailed the structure of the course and its expectations
  • The dynamics of kernel execution (just a taster)

• Our next **prerecorded** lecture (intended to be watched before you start on Lab 1) will explore:
  • DTrace, the kernel tracing facility we will use
  • The *probe effect* and its impact
  • Our lab environment

• Readings for the next lecture:
  • Paper - Cantrill, et al. 2004
  • McKusick, et al. Chapter 3 (Kernel Subsystems)