Kernels and Tracing

Lecture 2, Part 3: Kernel Dynamics

Prof. Robert N. M. Watson

2021-2022

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

Userspace	Kernel
`crt/csu`	`locore`
`rtld`	Kernel linker
Shared objects	Kernel modules
`main()`	`main()`, `platform_start()`
`libc`	`libkern`
POSIX threads API	`kthread` KPI
POSIX filesystem API	VFS KPI
POSIX sockets API	`socket` KPI
DTrace	DTrace
…	…

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/        nfsserver/      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/           netpfil/        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
cpufreq_if.m            kern_sx.c               subr_sbuf.c
...
```

- Kernel source lives in `/usr/src/sys`:
  - `kern/` – core kernel features
  - `sys/` – core kernel headers
- Useful resource: `http://fxr.watson.org/`

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.:

callout	Closure called after wall-clock delay
eventhandler	Closure called for key global events
task	Closure called .. eventually
`SYSINIT`	Function called when module loads/unloads

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

Wrapping up

- In this lecture, we have:
  - DTrace, the kernel tracing facility we will use
  - The *probe effect* and its impact
  - The dynamics of kernel execution (just a taster)

- Our next lecture will explore:
  - The *process model*
  - The practical implications of the process model

- Readings for the next lecture:
  - McKusick, et al: Chapter 4 (Process Management)
  - Anderson, et al. 1992. (**L41 only**)