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**: $\approx 3.4$M LoC in $\approx 6,450$ 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 $\approx 70$K lines of assembly over $\approx 6$ 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**: $\approx 3.0$M LoC in $\approx 3,500$ files
  - 415 device drivers (may support multiple devices)
Spelunking the kernel

% ls
Makefile ddb/ libkern/ nfs/ teken/
amd64/ dev/ mips/ nfscient/ 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

<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>COMM</th>
<th>TDNAME</th>
<th>CPU</th>
<th>PRI</th>
<th>STATE</th>
<th>WCHAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100000</td>
<td>kernel</td>
<td>swapper</td>
<td>-1</td>
<td>84</td>
<td>sleep</td>
<td>swapin</td>
</tr>
<tr>
<td>0</td>
<td>100006</td>
<td>kernel</td>
<td>dtrace_taskq</td>
<td>-1</td>
<td>84</td>
<td>sleep</td>
<td></td>
</tr>
</tbody>
</table>

... 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:
  • 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)