Kernels and Tracing

L41 Lecture 2

Dr Robert N. M. Watson

27 October 2017

Reminder: last time

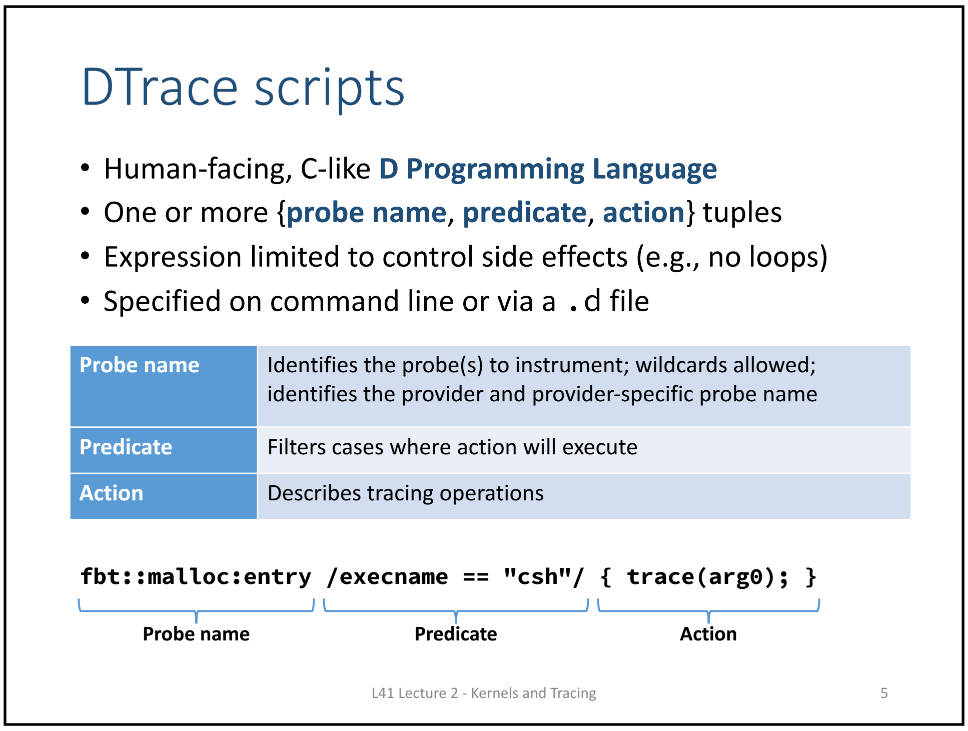
- What is an operating system?
- Systems research
- About the module
- Lab reports

This time: Tracing the kernel

- DTrace
- The **probe effect**
- The kernel: Just a C program?
- A little on kernel dynamics: How work happens

Dynamic tracing with DTrace

- Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. *Dynamic Instrumentation of Production Systems*, USENIX ATC 2004.
  - "Facility for dynamic instrumentation of production systems"
  - Unified and safe **instrumentation** of kernel and user space
  - Zero **probe effect** when not enabled
  - Dozens of **providers** representing different trace mechanisms
  - Tens (hundreds?) of thousands of **instrumentation probes**
  - **D language**: C-like scripting language with **predicates**, **actions**
  - Scalar variables, thread-local variables, associative arrays
  - **Data aggregation** and **speculative tracing**
- Solaris, Mac OS X, FreeBSD; Linux + Windows modules
- Wide influence – e.g., on Linux SystemTap, eBPF
- **Our tool of choice in this course**



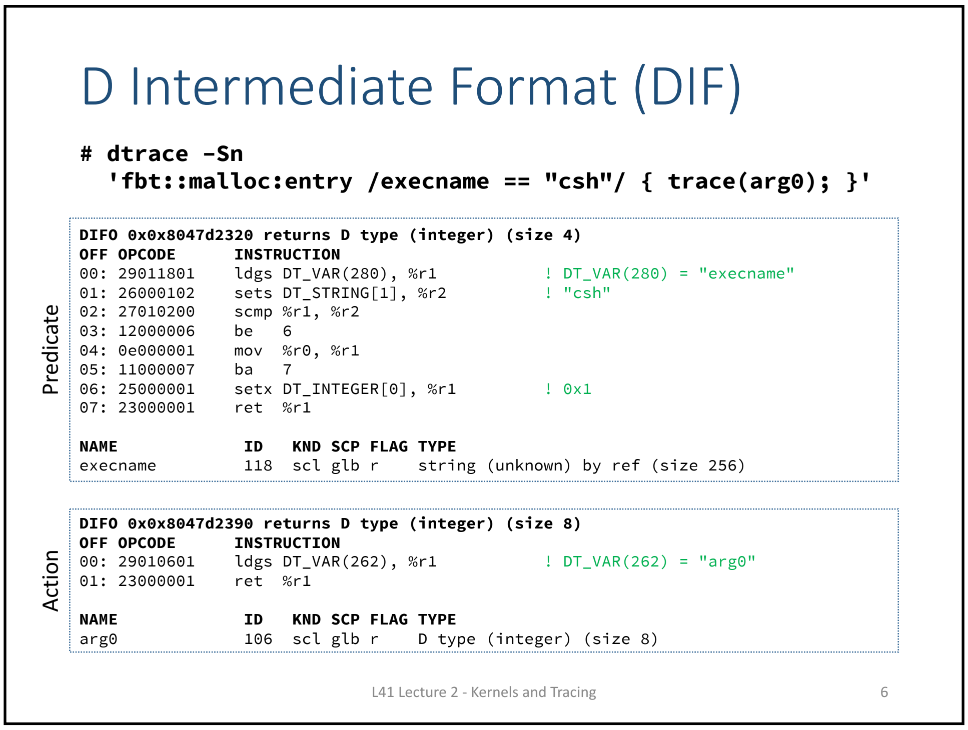
DTrace scripts

- Human-facing, C-like **D Programming Language**
- One or more {**probe name**, **predicate**, **action**} tuples
- Expression limited to control side effects (e.g., no loops)
- Specified on command line or via a `.d` file

Probe name	Identifies the probe(s) to instrument; wildcards allowed; identifies the provider and provider-specific probe name
Predicate	Filters cases where action will execute
Action	Describes tracing operations

```
fbt::malloc:entry /execname == "csh"/ { trace(arg0); }
```

$\underbrace{\texttt{fbt::malloc:entry}}_{\textbf{Probe name}}\ \underbrace{\texttt{/execname == "csh"/}}_{\textbf{Predicate}}\ \underbrace{\texttt{\{ trace(arg0); \}}}_{\textbf{Action}}$



D Intermediate Format (DIF)

```
# dtrace -Sn
  'fbt::malloc:entry /execname == "csh"/ { trace(arg0); }'
```

Predicate

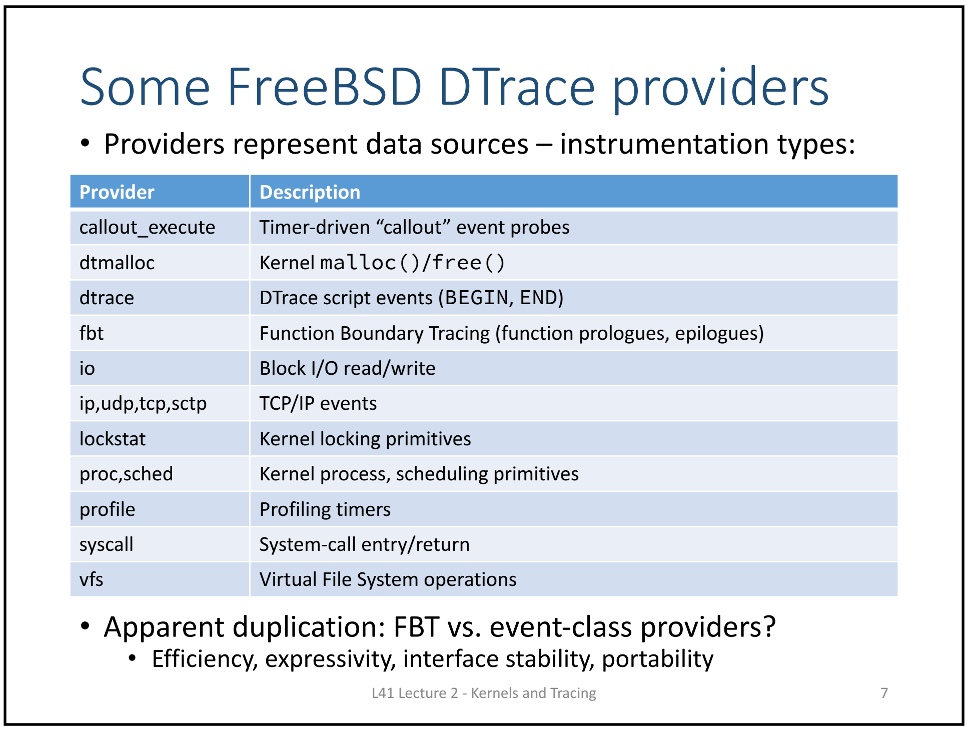
```
DIFO 0x0x8047d2320 returns D type (integer) (size 4)
OFF OPCODE      INSTRUCTION
00: 29011801    ldgs DT_VAR(280), %r1           ! DT_VAR(280) = "execname"
01: 26000102    sets DT_STRING[1], %r2          ! "csh"
02: 27010200    scmp %r1, %r2
03: 12000006    be   6
04: 0e000001    mov  %r0, %r1
05: 11000007    ba   7
06: 25000001    setx DT_INTEGER[0], %r1         ! 0x1
07: 23000001    ret  %r1

NAME             ID   KND SCP FLAG TYPE
execname         118  scl glb r    string (unknown) by ref (size 256)
```

Action

```
DIFO 0x0x8047d2390 returns D type (integer) (size 8)
OFF OPCODE      INSTRUCTION
00: 29010601    ldgs DT_VAR(262), %r1           ! DT_VAR(262) = "arg0"
01: 23000001    ret  %r1

NAME             ID   KND SCP FLAG TYPE
arg0             106  scl glb r    D type (integer) (size 8)
```

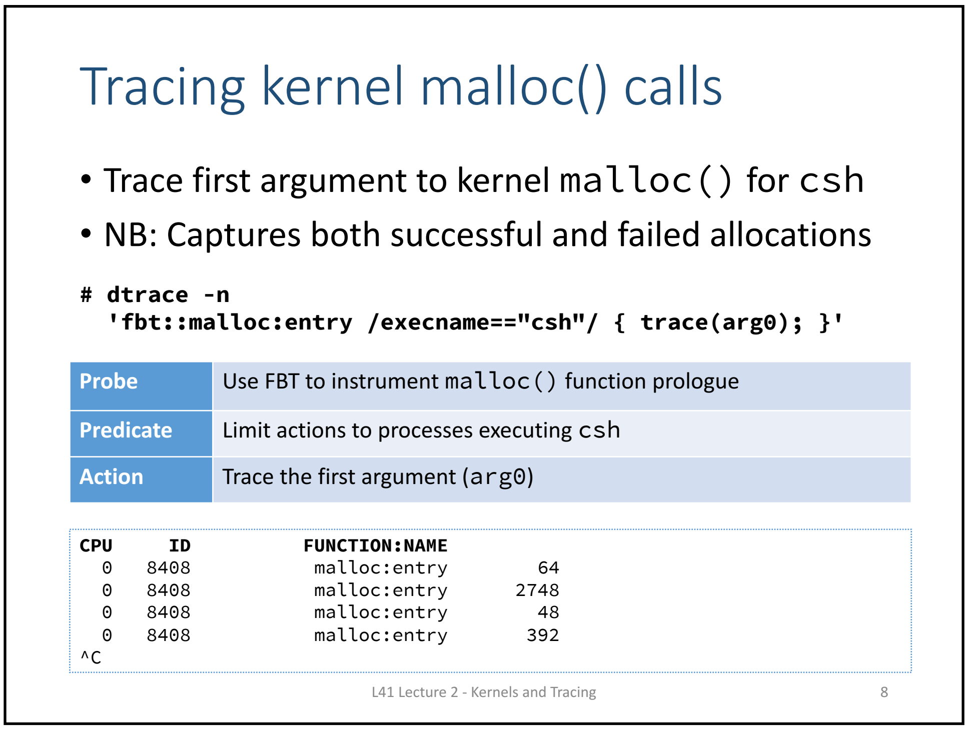


Some FreeBSD DTrace providers

- Providers represent data sources – instrumentation types:

Provider	Description
callout_execute	Timer-driven "callout" event probes
dtmalloc	Kernel `malloc()`/`free()`
dtrace	DTrace script events (`BEGIN`, `END`)
fbt	Function Boundary Tracing (function prologues, epilogues)
io	Block I/O read/write
ip,udp,tcp,sctp	TCP/IP events
lockstat	Kernel locking primitives
proc,sched	Kernel process, scheduling primitives
profile	Profiling timers
syscall	System-call entry/return
vfs	Virtual File System operations

- Apparent duplication: FBT vs. event-class providers?
  - Efficiency, expressivity, interface stability, portability



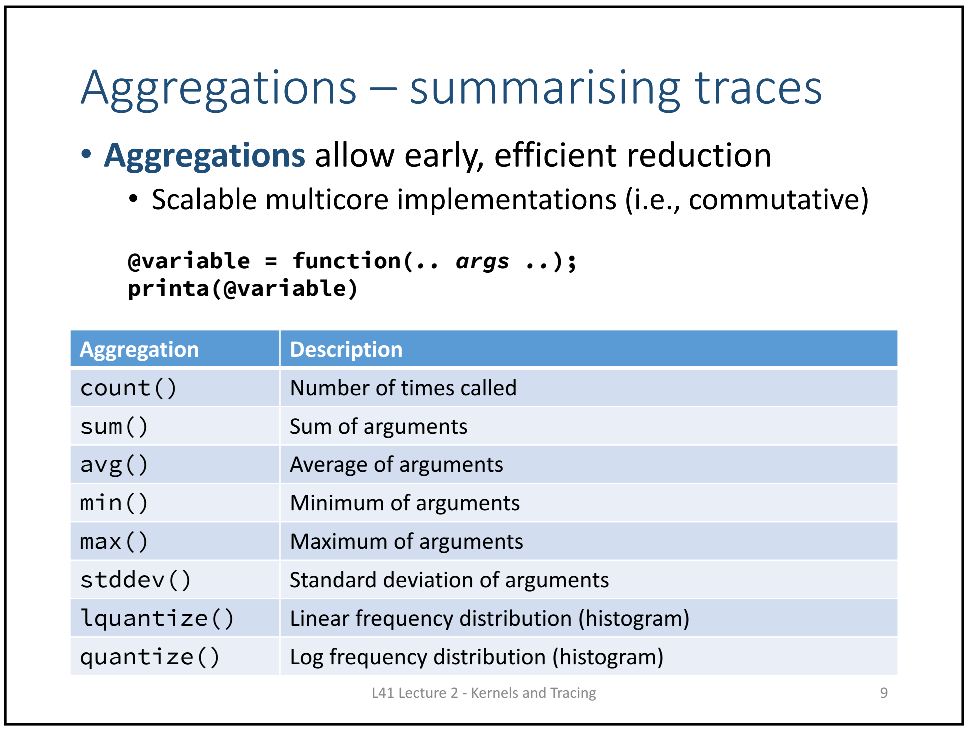
Tracing kernel malloc() calls

- Trace first argument to kernel `malloc()` for `csh`
- NB: Captures both successful and failed allocations

```
# dtrace -n
  'fbt::malloc:entry /execname=="csh"/ { trace(arg0); }'
```

Probe	Use FBT to instrument `malloc()` function prologue
Predicate	Limit actions to processes executing `csh`
Action	Trace the first argument (`arg0`)

```
CPU     ID                    FUNCTION:NAME
  0   8408                    malloc:entry        64
  0   8408                    malloc:entry      2748
  0   8408                    malloc:entry        48
  0   8408                    malloc:entry       392
^C
```

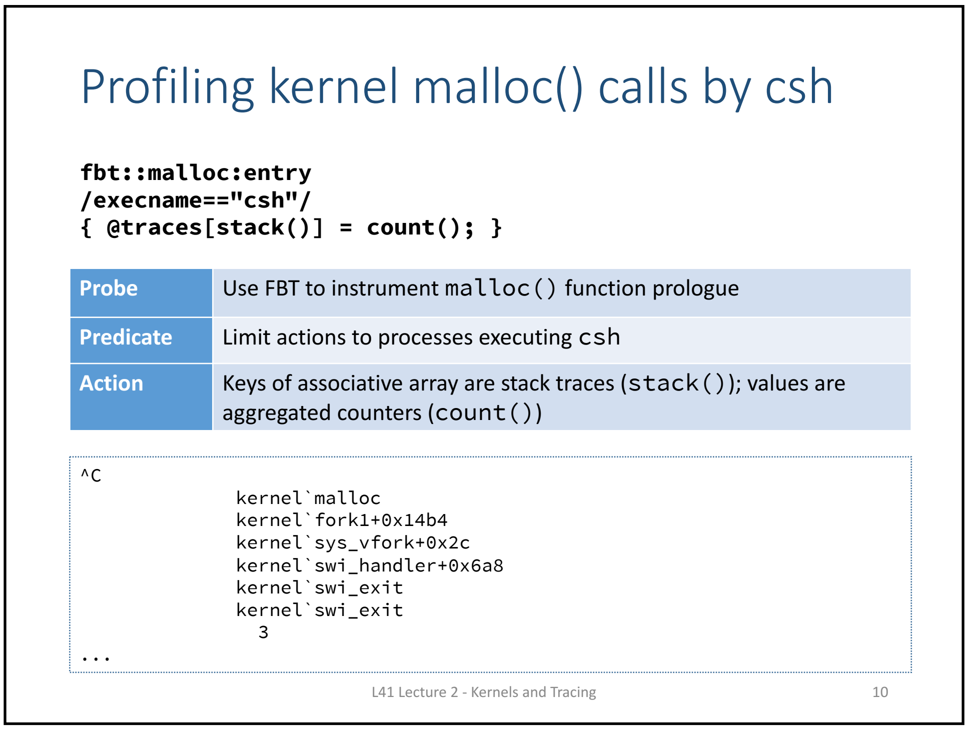


Aggregations – summarising traces

- **Aggregations** allow early, efficient reduction
  - Scalable multicore implementations (i.e., commutative)

```
@variable = function(.. args ..);
printa(@variable)
```

Aggregation	Description
`count()`	Number of times called
`sum()`	Sum of arguments
`avg()`	Average of arguments
`min()`	Minimum of arguments
`max()`	Maximum of arguments
`stddev()`	Standard deviation of arguments
`lquantize()`	Linear frequency distribution (histogram)
`quantize()`	Log frequency distribution (histogram)



Profiling kernel malloc() calls by csh

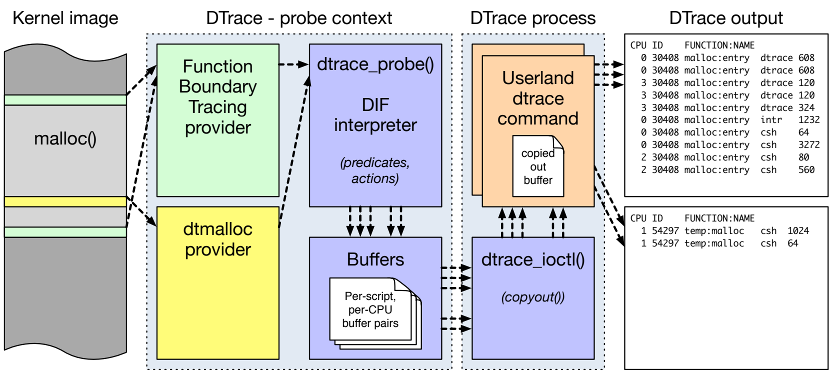
```
fbt::malloc:entry
/execname=="csh"/
{ @traces[stack()] = count(); }
```

Probe	Use FBT to instrument `malloc()` function prologue
Predicate	Limit actions to processes executing `csh`
Action	Keys of associative array are stack traces (`stack()`); values are aggregated counters (`count()`)

```
^C
              kernel`malloc
              kernel`fork1+0x14b4
              kernel`sys_vfork+0x2c
              kernel`swi_handler+0x6a8
              kernel`swi_exit
              kernel`swi_exit
                3
...
```

DTrace: Implementation

```
dtrace -n 'fbt::malloc:entry { trace(execname); trace(arg0); }'
```



```
CPU ID    FUNCTION:NAME
  0 30408 malloc:entry  dtrace 608
  0 30408 malloc:entry  dtrace 608
  3 30408 malloc:entry  dtrace 120
  3 30408 malloc:entry  dtrace 120
  3 30408 malloc:entry  dtrace 324
  0 30408 malloc:entry  intr   1232
  0 30408 malloc:entry  csh    64
  0 30408 malloc:entry  csh    3272
  2 30408 malloc:entry  csh    80
  2 30408 malloc:entry  csh    560
```

```
CPU ID    FUNCTION:NAME
  1 54297 temp:malloc   csh  1024
  1 54297 temp:malloc   csh  64
```

```
dtrace -n 'dtmalloc::temp:malloc /execname="csh"/ { trace(execname); trace(arg3); }'
```

The *Probe Effect*

- The **probe effect** is the unintended alteration of system behaviour that arises from measurement
  - Software instrumentation is **active**: execution is changed
- DTrace minimises probe effect when not being used…
  - … but has a very significant impact when it is used
  - Disproportionate effect on probed events
- Potential perturbations:
  - Speed relative to other cores (e.g., lock hold times)
  - Speed relative to external events (e.g., timer ticks)
  - Microarchitectural effects (e.g., cache, branch predictor)
- What does this mean for us?
  - Don't benchmark while running Dtrace …
  - … unless **measuring probe effect**
  - Be aware that traced applications may behave differently
  - E.g., more timer ticks will fire, I/O will "seem faster"

Probe effect example: dd(1) execution time

- Simple (naïve) microbenchmark – `dd(1)`
  - dd copies blocks from input to output
  - Copy 10M buffer from `/dev/zero` to `/dev/null`
  - Execution time measured with `/usr/bin/time`

```
# dd if=/dev/zero of=/dev/null bs=10m count=1 status=none
```

- Simultaneously, run various DTrace scripts
  - Compare resulting execution times using `ministat`
  - Difference is probe effect (+/- measurement error)

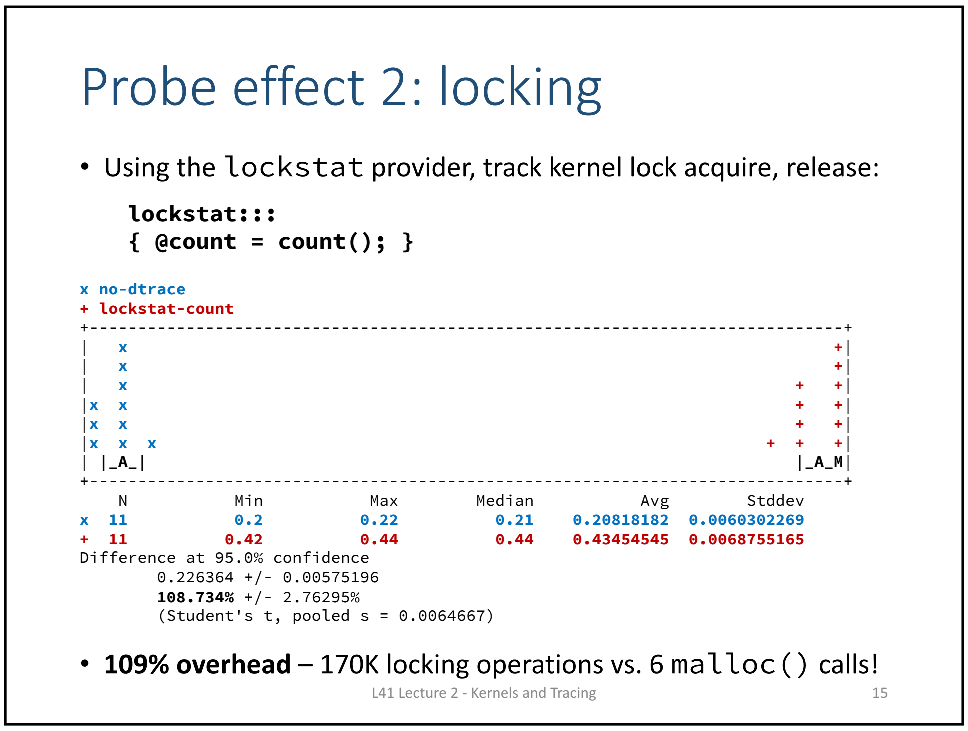
Probe effect 1: memory allocation

- Using the `dtmalloc` provider, count kernel memory allocations:

```
dtmalloc:::
{ @count = count(); }
```

```
x no-dtrace
+ dtmalloc-count
+------------------------------------------------------------------------------+
|                                      *                                       |
|                                      *                                       |
|                                      *                                      +|
|x                                     *                                      +|
|x                                     *                                      +|
|*                                     *                                      *|
|        |_______________|______A______M__________A_____|__________________|   |
+------------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x  11           0.2          0.22          0.21    0.20818182  0.0060302269
+  11           0.2          0.22          0.21    0.21272727  0.0064666979
No difference proven at 95.0% confidence
```

- **No statistically significant overhead** at 95% confidence level



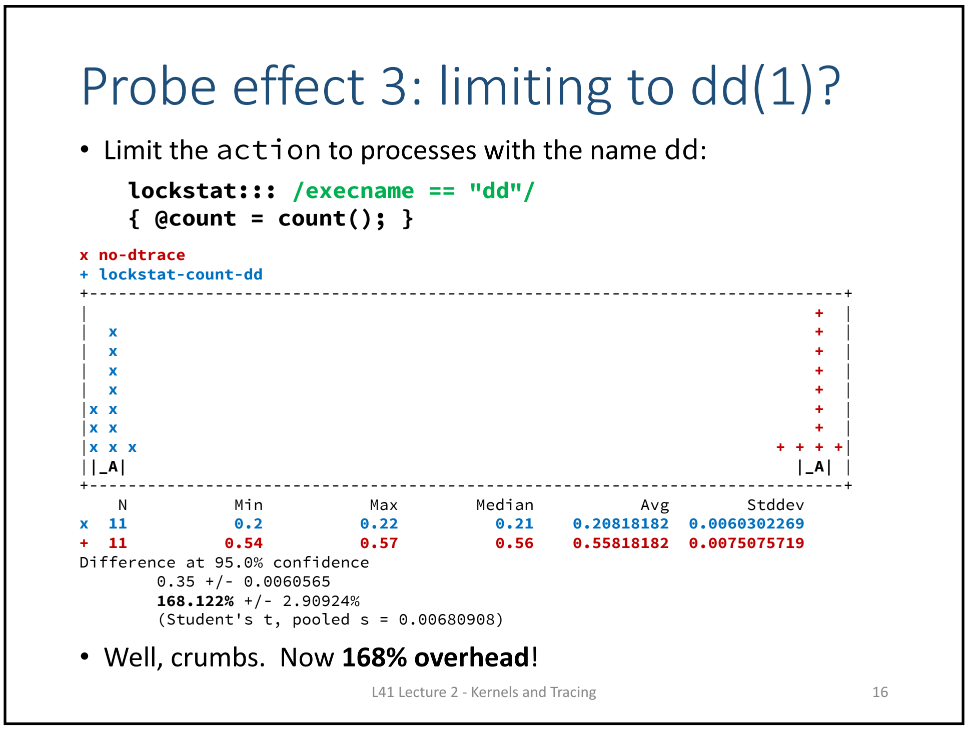
Probe effect 2: locking

- Using the `lockstat` provider, track kernel lock acquire, release:

```
lockstat:::
{ @count = count(); }
```

```
x no-dtrace
+ lockstat-count
+-------------------------------------------------------------------------------+
|   x                                                                          +|
|   x                                                                          +|
|   x                                                                     +    +|
|x  x                                                                     +    +|
|x  x                                                                     +    +|
|x  x  x                                                               +  +    +|
| |_A_|                                                                   |_A_M|
+-------------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x  11           0.2          0.22          0.21    0.20818182  0.0060302269
+  11          0.42          0.44          0.44    0.43454545  0.0068755165
Difference at 95.0% confidence
        0.226364 +/- 0.00575196
        108.734% +/- 2.76295%
        (Student's t, pooled s = 0.0064667)
```

- **109% overhead** – 170K locking operations vs. 6 `malloc()` calls!



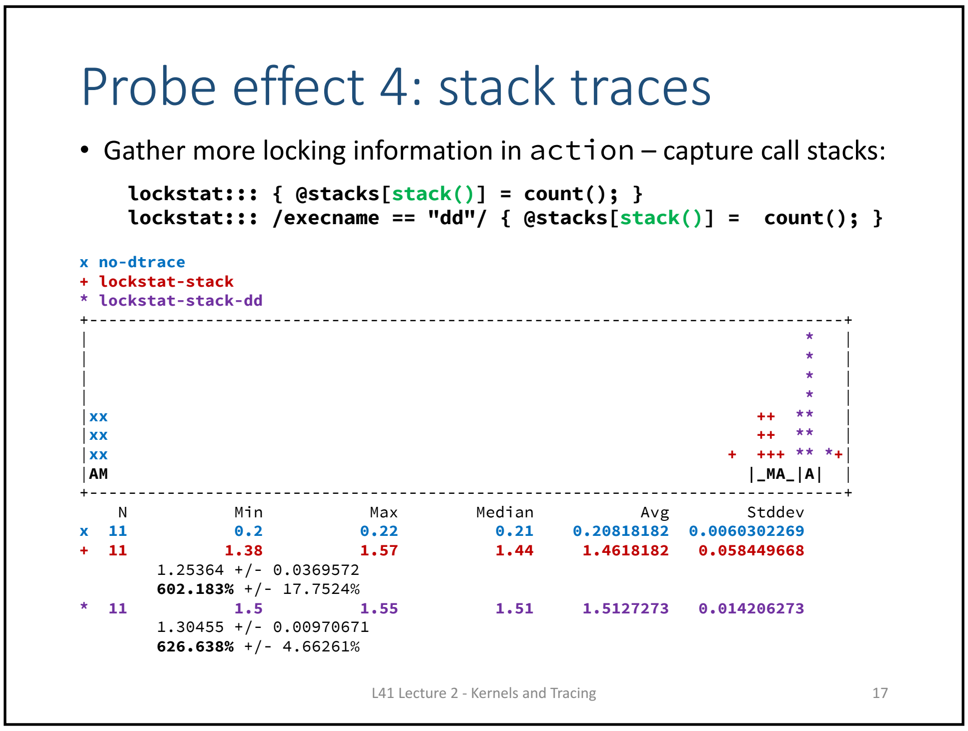
Probe effect 3: limiting to dd(1)?

- Limit the `action` to processes with the name dd:

```
lockstat::: /execname == "dd"/
{ @count = count(); }
```

```
x no-dtrace
+ lockstat-count-dd
+-------------------------------------------------------------------------------+
|                                                                           +  |
|  x                                                                        +  |
|  x                                                                        +  |
|  x                                                                        +  |
|  x                                                                        +  |
|x x                                                                        +  |
|x x                                                                        +  |
|x x x                                                                  + + + +|
||_A|                                                                   |_A| |
+-------------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x  11           0.2          0.22          0.21    0.20818182  0.0060302269
+  11          0.54          0.57          0.56    0.55818182  0.0075075719
Difference at 95.0% confidence
        0.35 +/- 0.0060565
        168.122% +/- 2.90924%
        (Student's t, pooled s = 0.00680908)
```

- Well, crumbs. Now **168% overhead**!



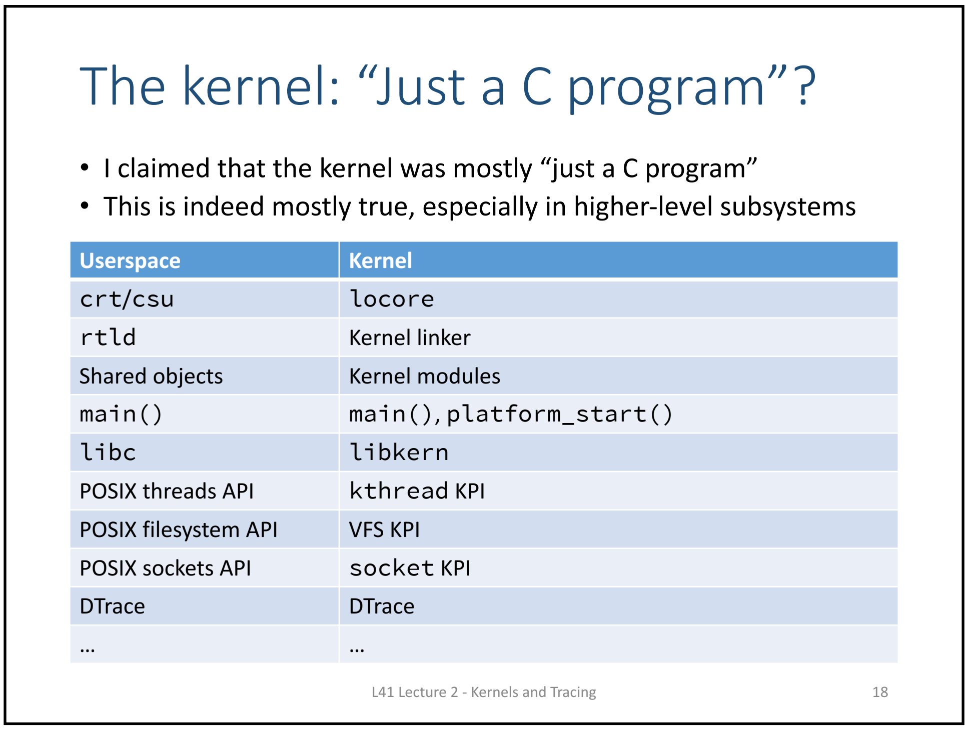
Probe effect 4: stack traces

- Gather more locking information in `action` – capture call stacks:

```
lockstat::: { @stacks[stack()] = count(); }
lockstat::: /execname == "dd"/ { @stacks[stack()] =  count(); }
```

```
x no-dtrace
+ lockstat-stack
* lockstat-stack-dd
+------------------------------------------------------------------------------+
|                                                                          *   |
|                                                                          *   |
|                                                                          *   |
|                                                                          *   |
|xx                                                                   ++  **   |
|xx                                                                   ++  **   |
|xx                                                                +  +++ ** *+|
|AM                                                                  |_MA_|A|  |
+------------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x  11           0.2          0.22          0.21    0.20818182  0.0060302269
+  11          1.38          1.57          1.44     1.4618182   0.058449668
        1.25364 +/- 0.0369572
        602.183% +/- 17.7524%
*  11           1.5          1.55          1.51     1.5127273   0.014206273
        1.30455 +/- 0.00970671
        626.638% +/- 4.66261%
```

L41 Lecture 2 - Kernels and Tracing 17

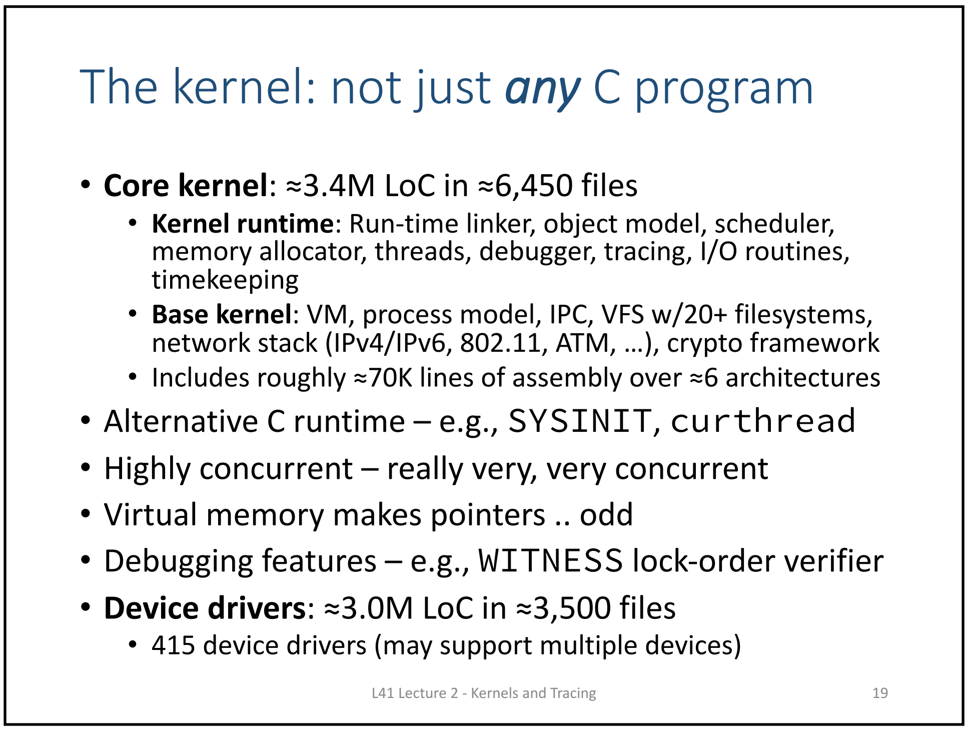


The kernel: "Just a C program"?

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

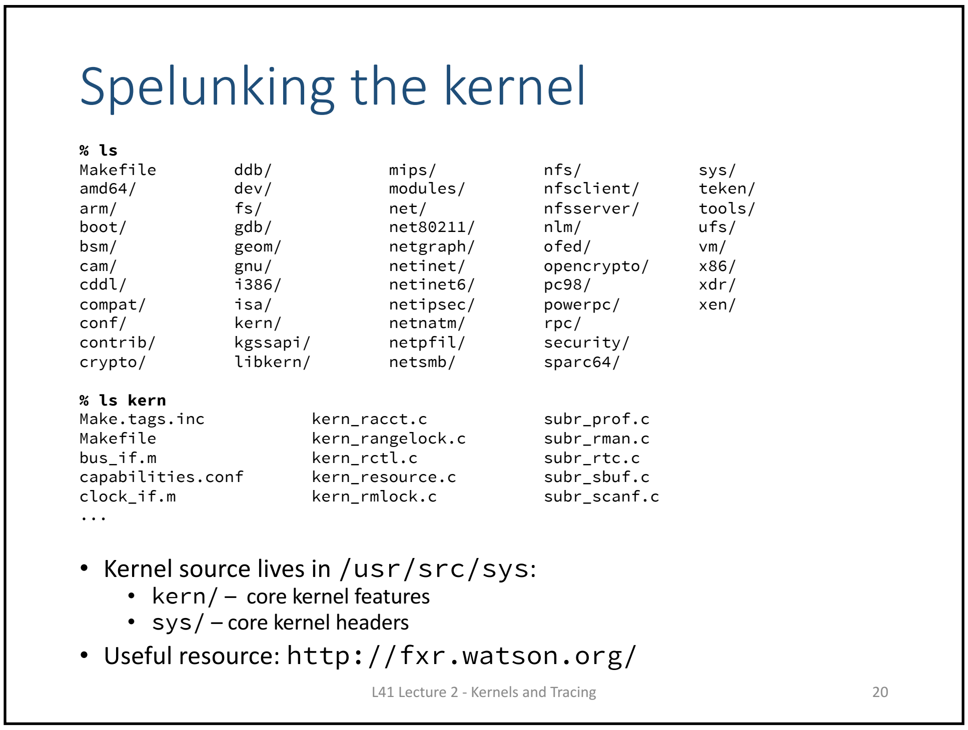
Userspace	Kernel
`crt/csu`	`locore`
`rtld`	Kernel linker
Shared objects	Kernel modules
`main()`	`main()`, `platform_start()`
`libc`	`libkern`
POSIX threads API	`kthread` KPI
POSIX filesystem API	VFS KPI
POSIX sockets API	`socket` KPI
DTrace	DTrace
…	…

L41 Lecture 2 - Kernels and Tracing 18



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/            mips/           nfs/            sys/
amd64/          dev/            modules/        nfsclient/      teken/
arm/            fs/             net/            nfsserver/      tools/
boot/           gdb/            net80211/       nlm/            ufs/
bsm/            geom/           netgraph/       ofed/           vm/
cam/            gnu/            netinet/        opencrypto/     x86/
cddl/           i386/           netinet6/       pc98/           xdr/
compat/         isa/            netipsec/       powerpc/        xen/
conf/           kern/           netnatm/        rpc/
contrib/        kgssapi/        netpfil/        security/
crypto/         libkern/        netsmb/         sparc64/

% ls kern
Make.tags.inc           kern_racct.c            subr_prof.c
Makefile                kern_rangelock.c        subr_rman.c
bus_if.m                kern_rctl.c             subr_rtc.c
capabilities.conf       kern_resource.c         subr_sbuf.c
clock_if.m              kern_rmlock.c           subr_scanf.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

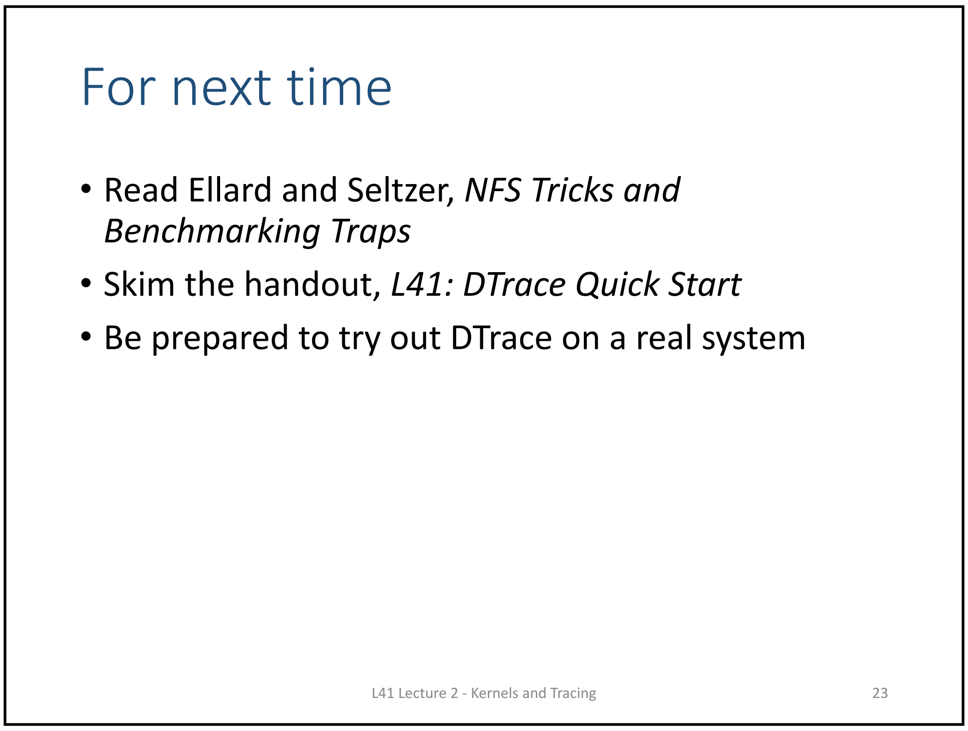
- Many operations begin with system calls in a user thread
- But may trigger work in many other threads; for example:
  - Triggering a callback in an interrupt thread when I/O is complete
  - Eventually writing back data to disk from the buffer cache
  - Delayed transmission if TCP isn't able to send 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 *closer* in C means: function pointer, opaque data pointer



For next time

- Read Ellard and Seltzer, *NFS Tricks and Benchmarking Traps*
- Skim the handout, *L41: DTrace Quick Start*
- Be prepared to try out DTrace on a real system