Getting started

• What is an operating system?
• Operating systems research
• About the L41 module
• Laboratory reports
• Kernel tracing with DTrace
• The **probe effect**
• The kernel: Just a C program?
• A little on kernel dynamics: How work happens
What is an operating system?

(Whiteboarding exercise)
What is an operating system?

[An OS is] low-level software that supports a computer’s basic functions, such as scheduling tasks and controlling peripherals.

- Google hive mind
General-purpose operating systems

... are for general-purpose computers:
• Servers, workstations, mobile devices
• Run applications – i.e., software unknown at design time
• Abstract the hardware, provide ‘class libraries’
• E.g., Windows, Mac OS X, Android, iOS, Linux, BSD, ...

<table>
<thead>
<tr>
<th>Userspace</th>
<th>Local and remote shells, management tools, daemons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run-time linker, system libraries, logging and tracing facilities</td>
</tr>
<tr>
<td></td>
<td>– system-call layer –</td>
</tr>
<tr>
<td>Kernel</td>
<td>System calls, hypercalls, remote procedure call (RPC)*</td>
</tr>
<tr>
<td></td>
<td>Processes, filesystems, IPC, sockets, management</td>
</tr>
<tr>
<td></td>
<td>Drivers, packets/blacks, protocols, tracing, virtualisation</td>
</tr>
<tr>
<td></td>
<td>VM, malloc, linker, scheduler, threads, timers, tasks, locks</td>
</tr>
</tbody>
</table>

* Continuing disagreement on whether distributed-filesystem servers and window systems ‘belong’ in userspace or the kernel
Other kinds of operating systems

Specialise the OS for a specific application or environment:

- **Embedded, real-time operating systems**
  - Serve a single application in a specific context
  - E.g., WiFi access points, medical devices, washing machines, cars
  - Small code footprint, real-time scheduling
  - Might have virtual memory / process model
  - Microkernels or single-address space: VxWorks, RTEMS, L4
  - Now also: Linux, BSD (sometimes over a real-time kernel), etc.

- **Appliance operating systems**
  - Apply embedded model to higher-level devices/applications
  - File storage appliances, routers, firewalls, ...
  - E.g., Juniper JunOS, Cisco IOS, NetApp OnTap, EMC/Isilon
  - Under the hood, almost always Linux, BSD, etc.

Key concept: **Operating system as a reusable component**
Other kinds of operating systems?

What if we rearrange the boxes?

- **Microkernels, library operating systems, unikernels**
  - Shift code out of the kernel into userspace to reduce Trusted Computing Base (TCB); improve robustness/flexibility; ‘bare-metal’ apps
  - Early 1990s: Microkernels are king!
  - Late 1990s: Microkernels are too slow!
  - 2000s/2010s: Microkernels are back! But now ‘hypervisors’
  - Sometimes: programming-language runtime as OS
Other kinds of operating systems?

• **Hypervisors**
  • Kernels host applications; hypervisors host virtual machines
  • Virtualised hardware interface rather than POSIX
  • Paravirtualisation reintroduces OS-like interfaces for performance
  • A lot of microkernel ideas have found a home here
  • E.g., System/370, VMware, Xen, KVM, VirtualBox, bhyve, ...

• **Containers**
  • Host OS as hypervisor, but using the process model
  • Really more about code/ABI (Application Binary Interface) distribution and maintenance
What does an operating system do?

- Key hardware-software surface
- Low-level abstractions and services
  - **Operational model**: bootstrap, shutdown, watchdogs
  - **Process model, IPC**: processes, threads, IPC, program model
  - **Resource sharing**: scheduling, multiplexing, virtualisation
  - **I/O**: drivers, local/distributed filesystems, network stack
  - **Security**: authentication, encryption, ACLs, MAC, audit
  - **Local or remote access**: console, window system, SSH
  - **Libraries**: math, protocols, RPC, crypto, UI, multimedia
  - **Monitoring/debugging**: logs, profiling, tracing, debugging

Compiler? Text editor? E-mail package? Web browser?
Can an operating system be “distributed”?
Why study operating systems?

The OS plays a central role in whole-system design when building efficient, effective, and secure systems:

- Strong influence on whole-system performance
- Critical foundation for computer security
- Exciting programming techniques, algorithms, problems
  - Virtual memory; network stack; filesystem; run-time linker; ...
- Co-evolves with platforms, applications, users
- Multiple active research communities
- Reusable techniques for building complex systems
- Boatloads of fun (best text adventure ever)
Where is the OS research?

A sub-genre of **systems research**:  
- Evolving hardware-software interfaces  
  - New computation models/architectures  
  - New kinds of peripheral devices  
- Integration with programming languages and runtimes  
- Concurrent/parallel programming models; scheduling  
- Security and virtualisation  
- Networking, storage, and distributed systems  
- Tracing and debugging techniques  
- Formal modeling and verification  
- As a platform for other research – e.g., mobile systems  

**Venues**: SOSP, OSDI; ATC; EuroSys; HotOS; FAST; NSDI; HotNets; ASPLOS; USENIX Sec.; ACM CCS; IEEE SSP; ...
What are the research questions?

Just a few examples: By changing the OS, can I...

• Create new abstractions for new hardware?
• Make my application run faster by...
  • Better masking latency?
  • Using parallelism more effectively?
  • Exploiting new storage mediums?
  • Adopting distributed-system ideas in local systems?
• Make my application more {reliable, energy efficient}
• Limit {security, privacy} impact of exploited programs?
• Use new language/analysis techniques in new ways?

Systems research focuses on **evaluation** with respect to **applications** or **workloads**: How can we measure whether it is {faster, better, ...}?
Teaching operating systems

• Two common teaching tropes:
  • **Trial by fire**: in micro, recreate classic elements of operating systems: microkernels with processes, filesystems, etc.
  • **Research readings course**: read, present, discuss, and write about classic works in systems research

• This module adopts elements of both styles while:
  • mitigating the risk of OS kernel hacking in a short course
  • working on real-world systems rather than toys; and
  • targeting research skills not just operating-system design

• Trace and analyse real systems driven by specially crafted benchmarks

• Possible only because of recent developments in tracing and hardware-based performance analysis tools
Aims of the module (1/2)

Teaching **methodology, skills, and knowledge** required to understand and perform research on contemporary operating systems by...

- Employing systems methodology and practice
- Exploring real-world systems artefacts through performance and functional evaluation/analysis
- Developing scientific writing skills
- Reading selected original systems research papers
Aims of the module (2/2)

On completion of this module, students should:

• Have an understanding of high-level OS kernel structure.

• Gained insight into hardware-software interactions for compute and I/O.

• Have practical skills in system tracing and performance analysis.

• Have been exposed to research ideas in system structure and behaviour.

• Have learned how to write systems-style performance evaluations.
Prerequisites

We will take for granted:

• **High-level knowledge of OS terminology** from an undergraduate course (or equivalent); e.g.:
  - What *schedulers* do
  - What *processes* are ... and how they differ from threads
  - What *Inter-Process Communication (IPC)* does
  - How might a simple *filesystem* might work

• Reasonable fluency in reading multithreaded C
• Working knowledge of Python (or R)
• Comfort with the UNIX command-line environment
• Undergraduate skills with statistics
  (mean/median/mode/stddev/t-tests/linear regression/boxplots/scatterplots ... )

You can pick up some of this as you go (e.g., IPC, Python, t-tests), but will struggle if you are missing several
Module structure – four complementary strands

- **3x two-hour lectures** in FS09
  - Theory, methodology, architecture, and practice

- **5x two-hour labs** in SW02
  - Start with 10-20-minute *lecturelets* on artefacts, practical skills
  - Remainder on hands-on measurement and experimentation – learn skills required to write assigned lab reports, start on experiments
  - Lab *experimental questions* must be answered in your lab reports

- **Assigned research and applied readings**
  - Selected portions of module texts – learn skills, methodology
  - Historic and contemporary research papers – research exposure

- **Marked lab reports**
  - Based on experiments done in (and out) of scheduled labs
  - Refine scientific writing style suitable for systems research
  - One ‘practice run’ marked but not assessed ➡ not optional!
  - Two assessed; 50% of final mark each
Outline of module schedule

• Submodule 1: Introduction to kernels and tracing/analysis
  • 1 lecture, 1 lab (I/O)
  • Introduction: OSes, Systems Research, and L41
  • The Kernel: Kernel and Tracing
  • First lab report due - 2019-02-11

• Submodule 2: The Process Model
  • 1 lecture, 2 labs (IPC, PMC)
  • The Process Model (1) – Binaries and Processes
  • The Process Model (2) – Traps, System Calls, and Virtual Memory
  • Second lab report due - 2019-03-19

• Submodule 3: The Network Stack (TCP/IP)
  • 1 lecture, 2 labs (TCP state machine, congestion control)
  • The Network Stack (1) – Sockets, NICs, and Work Distribution
  • The Network Stack (2) – TCP protocol
  • Final lab report due - 2019-04-24
The platform

TI BeagleBone Black

- 1GHz ARM Cortex-A8 32-bit CPU
- Superscalar pipeline, MMU, L1/L2 caches
- FreeBSD operating system (13-CURRENT) + DTrace
- Bespoke “potted benchmarks”
- Jupyter notebook measurement and analysis environment
Labs and lab reports

Lab reports document an experiment and analyse its results – typically using one or more hypotheses.

Our lab reports will contain the following sections (see notes, template):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Title + abstract (1 page)</td>
<td>5. Conclusion (1-2 para)</td>
</tr>
<tr>
<td>2. Introduction (1-2 para)</td>
<td>6. References</td>
</tr>
<tr>
<td>3. Experimental setup and methodology (1-2 pages)</td>
<td>7. Appendices</td>
</tr>
<tr>
<td>4. Results and discussion (3-4 pages)</td>
<td></td>
</tr>
</tbody>
</table>

Some formats break out (e.g.) experimental setup vs. methodology, and results vs. discussion. The combined format seems to work better for systems experimentation as compared to (e.g.) biology.

• The target length is 10 pages excluding appendices, references
• **Over-length reports** will be assessed within page limit
• **Appendices** may not be read if too long, and should not be essential to understanding the core content of the report
Module texts – core material

You will need to make frequent reference to these books both in the labs and outside of the classroom:


Module texts – additional material

If your OS recollections feel a bit hazy:

**Operating systems**: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne


If you want to learn a bit more about architecture and measurement:

Dynamic tracing with DTrace

  • “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

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

**Table:**

<table>
<thead>
<tr>
<th>Probe name</th>
<th>Predicate</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifies the probe(s) to instrument; wildcards allowed; identifies the provider and provider-specific probe name</td>
<td>Filters cases where action will execute</td>
<td>Describes tracing operations</td>
</tr>
</tbody>
</table>
Some FreeBSD DTrace providers

- Providers represent data sources – instrumentation types:

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

- 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

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

<table>
<thead>
<tr>
<th>Probe</th>
<th>Use FBT to instrument malloc() function prologue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td>Limit actions to processes executing csh</td>
</tr>
<tr>
<td>Action</td>
<td>Trace the first argument (arg0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>64</td>
</tr>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>2748</td>
</tr>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>8408</td>
<td>malloc:entry</td>
<td>392</td>
</tr>
</tbody>
</table>

^C
Aggregations – summarising traces

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

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

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>Number of times called</td>
</tr>
<tr>
<td>sum()</td>
<td>Sum of arguments</td>
</tr>
<tr>
<td>avg()</td>
<td>Average of arguments</td>
</tr>
<tr>
<td>min()</td>
<td>Minimum of arguments</td>
</tr>
<tr>
<td>max()</td>
<td>Maximum of arguments</td>
</tr>
<tr>
<td>stddev()</td>
<td>Standard deviation of arguments</td>
</tr>
<tr>
<td>lquantize()</td>
<td>Linear frequency distribution (histogram)</td>
</tr>
<tr>
<td>quantize()</td>
<td>Log frequency distribution (histogram)</td>
</tr>
</tbody>
</table>
Profiling kernel malloc() calls by csh

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

<table>
<thead>
<tr>
<th><strong>Probe</strong></th>
<th>Use FBT to instrument malloc() function prologue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Predicate</strong></td>
<td>Limit actions to processes executing csh</td>
</tr>
<tr>
<td><strong>Action</strong></td>
<td>Keys of associative array are stack traces (stack()); values are aggregated counters (count())</td>
</tr>
</tbody>
</table>

```
^C

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

L41 Lecture 1 - Advanced Operating Systems 28
D Intermediate Format (DIF)

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

### Predicate

DIFO 0x8047d2320 returns D type (integer) (size 4)

<table>
<thead>
<tr>
<th>OFF</th>
<th>OPCODE</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:</td>
<td>29011801</td>
<td>ldgs DT_VAR(280), %r1</td>
</tr>
<tr>
<td>01:</td>
<td>26000102</td>
<td>sets DT_STRING[1], %r2</td>
</tr>
<tr>
<td>02:</td>
<td>27010200</td>
<td>scmp %r1, %r2</td>
</tr>
<tr>
<td>03:</td>
<td>12000006</td>
<td>be 6</td>
</tr>
<tr>
<td>04:</td>
<td>0e000001</td>
<td>mov %r0, %r1</td>
</tr>
<tr>
<td>05:</td>
<td>11000007</td>
<td>ba 7</td>
</tr>
<tr>
<td>06:</td>
<td>25000001</td>
<td>setx DT_INTEGER[0], %r1</td>
</tr>
<tr>
<td>07:</td>
<td>23000001</td>
<td>ret %r1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>ID</th>
<th>KND</th>
<th>SCP</th>
<th>FLAG</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>execname</td>
<td>118</td>
<td>scl</td>
<td>glb</td>
<td>r</td>
<td>string (unknown) by ref (size 256)</td>
</tr>
</tbody>
</table>

### Action

DIFO 0x8047d2390 returns D type (integer) (size 8)

<table>
<thead>
<tr>
<th>OFF</th>
<th>OPCODE</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:</td>
<td>29010601</td>
<td>ldgs DT_VAR(262), %r1</td>
</tr>
<tr>
<td>01:</td>
<td>23000001</td>
<td>ret %r1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>ID</th>
<th>KND</th>
<th>SCP</th>
<th>FLAG</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg0</td>
<td>106</td>
<td>scl</td>
<td>glb</td>
<td>r</td>
<td>D type (integer) (size 8)</td>
</tr>
</tbody>
</table>
DTrace: Implementation

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

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

Kernel image | DTrace - probe context | DTrace process | DTrace output
---|---|---|---
malloc() | dtrace_probe() | Userland dtrace command |
| Function Boundary Tracing provider | DIF interpreter | dtrace_ioctl() |
| dtmalloc provider | (predicates, actions) | (copyout()) |

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

CPU ID | FUNCTION:NAME |
---|---|
1 | temp:malloc csh 1024 |
1 | temp:malloc csh 64 |

L41 Lecture 1 - Advanced Operating Systems
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:

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

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Avg</th>
<th>Stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0.2</td>
<td>0.22</td>
<td>0.21</td>
<td>0.20818182</td>
<td>0.0060302269</td>
</tr>
<tr>
<td>+</td>
<td>0.2</td>
<td>0.22</td>
<td>0.21</td>
<td>0.21272727</td>
<td>0.0064666979</td>
</tr>
</tbody>
</table>

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:

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

```plaintext
x no-dtrace
+ lockstat-count
```

<table>
<thead>
<tr>
<th>N</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Avg</th>
<th>Stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>11</td>
<td>0.2</td>
<td>0.22</td>
<td>0.21</td>
<td>0.20818182</td>
</tr>
<tr>
<td>+</td>
<td>11</td>
<td>0.42</td>
<td>0.44</td>
<td>0.44</td>
<td>0.43454545</td>
</tr>
</tbody>
</table>

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(); }
  ```

- 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(); }
```

<table>
<thead>
<tr>
<th>x</th>
<th>no-dtrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>lockstat-stack</td>
</tr>
<tr>
<td>*</td>
<td>lockstat-stack-dd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>11</th>
<th>0.2</th>
<th>0.22</th>
<th>0.21</th>
<th>0.20818182</th>
<th>0.0060302269</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>11</td>
<td>1.38</td>
<td>1.57</td>
<td>1.44</td>
<td>1.4618182</td>
<td>0.058449668</td>
</tr>
</tbody>
</table>

1.25364 +/- 0.0369572
602.183% +/- 17.7524%

<table>
<thead>
<tr>
<th>*</th>
<th>11</th>
<th>1.5</th>
<th>1.55</th>
<th>1.51</th>
<th>1.5127273</th>
<th>0.014206273</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.30455 +/- 0.00970671</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
626.638% +/- 4.66261%
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/                  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.:

<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
For next time

- McKusick, et al. – Chapter 3
- Cantrill, et al. 2004 – full article
- Read Ellard and Seltzer, *NFS Tricks and Benchmarking Traps*
- Skim the handout, *L41: DTrace Quick Start* (available from L41 module website)
- Be prepared to try out DTrace on a real system