Why kselftest?

- Regression test suite
- Focuses on testing kernel from user-space
- User-space applications (Shell scripts, C programs)
  - Kernel Test modules used to exercise kernel code paths
- Allows for breadth and depth coverage (error paths etc.)
- Not for workload or application testing
Why kselftest?

- Perfect for feature, functional and regression testing
- Perfect for bug fix focused regression testing and subsystem testing
- Perfect for testing user APIs, system calls, critical user paths, common use cases
- Perfect for end to end regression testing
  - Provides assurance that “everything works”
- Combination of Open and Closed box testing
- For more information on Kselftest framework/run/write tests
  - Watch LF Live Mentorship webinar:
    - Kernel Validation With Kselftest
Why KUnit?

- Focuses on in-kernel testing
- Perfect for:
  - testing internal kernel APIs
  - libraries, drivers, …,
  - individual *units* of code
- Perfect for unit testing
  - Makes it tractable to test all the edge cases
McCabe’s Complexity

- Testing all edge cases?
  - Imagine trying to reach an arbitrary edge case in the kernel from a syscall
  - Reaching every state is intractable
- Solution: Call functions directly to test edge cases
McCabe’s Complexity

- Solution: Call functions directly to test edge cases
- McCabe’s complexity is a measure of the number of states, or branches a function can achieve
- If we have a function, A, call other functions, B₁, B₂, …, Bₙ, and we only test A
  - If we try to reach all branches from A, you can see that as the function depth increases, the total number of branches increases combinatorially
  - If we only reach all the states of each function individually, the branches increase linearly.
- KUnit is a really practical way to test the vast majority of edge cases.
McCabe’s Complexity

- Solution: Call functions directly to test edge cases
- McCabe’s complexity is a measure of the number of states, or branches a function can achieve
- If we have a function, A, call other functions, B1, B2, …, Bn, and we only test A
  - If we try to reach all branches from A, you can see that as the function depth increases, the total number of branches increases combinatorially
  - If we only reach all the states of each function individually, the branches increase linearly.
- KUnit is a really practical way to test the vast majority of edge cases.
- For more background info on KUnit like this please see LF Live Mentorship webinar: KUnit Testing Strategies
GCOV: How to coverage

- GCOV keeps track of code run during execution
- Generates reports
- Show what code ran, and what code did not
GCOV: How to coverage

- Shows directory level summaries

<table>
<thead>
<tr>
<th>Filename</th>
<th>Line Coverage</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>common.c</td>
<td>0.0 %</td>
<td>0 / 43</td>
</tr>
<tr>
<td>generic_gos.c</td>
<td>0.0 %</td>
<td>0 / 73</td>
</tr>
<tr>
<td>main.c</td>
<td>3.8 %</td>
<td>31 / 310</td>
</tr>
<tr>
<td>power.h</td>
<td>81.8 %</td>
<td>9 / 11</td>
</tr>
<tr>
<td>gos-test.c</td>
<td>100.0 %</td>
<td>49 / 49</td>
</tr>
<tr>
<td>gos.c</td>
<td>18.2 %</td>
<td>69 / 379</td>
</tr>
<tr>
<td>runtime.c</td>
<td>8.3 %</td>
<td>55 / 665</td>
</tr>
<tr>
<td>sysfs.c</td>
<td>5.2 %</td>
<td>13 / 248</td>
</tr>
<tr>
<td>waking.c</td>
<td>0.0 %</td>
<td>0 / 99</td>
</tr>
<tr>
<td>wakeup.c</td>
<td>1.0 %</td>
<td>3 / 302</td>
</tr>
<tr>
<td>wakeup_stats.c</td>
<td>6.8 %</td>
<td>5 / 73</td>
</tr>
</tbody>
</table>
GCOV: How to coverage

- Shows directory level summaries
GCOV: How to coverage

- Shows directory level summaries
- Shows overall summary as coverage number
Code Coverage IS

- A great way to quickly find what code IS tested and what code IS NOT tested.
- Allows you to quickly identify problem areas, and drill down into a report.
- Identify missed branches.
- Identify unused code.
Imagine we are testing some code:

```c
static int _dev_pm_qos_add_request(struct device *dev,
    struct dev_pm_qos_request *req,
    enum dev_pm_qos_req_type type, s32 value)
{
    int ret = 0;

    if (!dev || !req || dev_pm_qos_invalid_req_type(dev, type))
        return -EINVAL;

    if (WARN(dev_pm_qos_request_active(req),
        "%s() called for already added request\n", __func__))
        return -EINVAL;

    if (IS_ERR(dev->power_qos))
        ret = -ENODEV;
    else if (!dev->power_qos)
        ret = dev_pm_qos_constraints_allocate(dev);
    trace_dev_pm_qos_add_request(dev_name(dev), type, value);
    if (ret)
        return ret;

    req->dev = dev;
    req->type = type;
    if (req->type == DEV_PM_QOS_MIN_FREQUENCY)
        ret = freq_qos_add_request(dev->power_qos->freq,
            &req->data.freq, FREQ_QOS_MIN, value);
    else if (req->type == DEV_PM_QOS_MAX_FREQUENCY)
        ret = freq_qos_add_request(dev->power_qos->freq,
            &req->data.freq, FREQ_QOS_MAX, value);
    else
        ret = apply_constraint(req, PM_QOS_ADD_REQ, value);

    return ret;
}
```
Code Coverage is: Example

Imagine we are testing some code:

- We can see that we have edge cases for
  - DEV_PM_QOS_MIN_FREQUENCY
  - DEV_PM_QOS_MAX_FREQUENCY

```c
static int dev_pm_qos_add_request(struct device *dev,
                                  struct dev_pm_qos_request *req,
                                  enum dev_pm_qos_req_type type, s32 value)
{
    int ret = 0;

    if (!dev || !req || !dev_pm_qos_invalid_req_type(dev, type))
        return -EINVAL;

    if (WARN(dev_pm_qos_request_active(req),
             "%s() called for already added request\n", __func__))
        return -EINVAL;

    if (IS_ERR(dev->power_qos))
        ret = -ENODEV;
    else if (!dev->power_qos)
        ret = dev_pm_qos_constraints_allocate(dev);

    trace_dev_pm_qos_add_request(dev_name(dev), type, value);
    if (ret)
        return ret;

    req->dev    = dev;
    req->type   = type;
    if (req->type == DEV_PM_QOS_MIN_FREQUENCY)
        ret = freq_qos_add_request(dev->power_qos->freq,
                                    &req->data.freq,
                                    FREQUENCY_MIN, value);
    else if (req->type == DEV_PM_QOS_MAX_FREQUENCY)
        ret = freq_qos_add_request(dev->power_qos->freq,
                                    &req->data.freq,
                                    FREQUENCY_MAX, value);
    else
        ret = apply_constraint(req, PM_QOS_ADD_REQ, value);

    return ret;
}
```
Imagine we are testing some code:

- We can see that we have edge cases for
  - `DEV_PM_QOS_MIN_FREQUENCY`
  - `DEV_PM_QOS_MAX_FREQUENCY`
- The report shows us that our tests do not cover these edge cases.
Imagine we are testing some code:

- We can see that we have edge cases for
  - `DEV_PM_QOS_MIN_FREQUENCY`
  - `DEV_PM_QOS_MAX_FREQUENCY`

The report shows us that our tests do not cover these edge cases.

This shows the power of KUnit with coverage:

- We can (and do) call this function directly in tests
Code Coverage IS NOT

- Code coverage is a tool, not a panacea
- Code coverage helps quickly identify and prioritize problem areas
- Code coverage summaries do not tell you whether your testing is good or bad
  - What is the right amount of line coverage?
    - 50%?
    - 70%?
    - 90%?
    - 100%?
What’s the right coverage?

- How do we measure coverage?
  - % of lines?
  - % of functions?
  - % of branches?
- What about absolute vs incremental?
What’s the right coverage?

- Absolute coverage:
  - What you expect.
  - Everything in the entire codebase at some point in time.
- Incremental coverage:
  - The test coverage of the Δ in a change
Absolute vs. Incremental Coverage

- Incremental Coverage is usually more interesting
  - It's much easier to achieve high incremental coverage immediately
  - Helps prioritize code more likely to be buggy
  - More actionable by developers
  - Code that has not changed in a long time is *more* likely to be fine
Absolute vs. Incremental Coverage

- Absolute Coverage is still important, just less important
  - Old code may be less likely to contain bugs…
  - ...but it’s often worse when it does
- Often easier for comparing coverage health of subsystems
- Easier to compute
Kselftest & KUnit

- **Kselftest**
  - Good for depth testing covering deeper code paths
  - Good for testing commonly used code paths
  - A good test could test some error paths

- **KUnit**
  - Good for targeting error paths & edge cases
  - Easier and faster for zeroing in on a kernel area
● Code coverage important for Safety?
● Kselftest & KUnit
  ○ Improvements that could be made?
  ○ More tests for coverage?
  ○ More tests for regression?
  ○ ????