Measuring and Understanding Linux Kernel Tests

Tingxu Ren

Wentao Zhang, Jinghao Jia, Darko Marinov, Tianyin Xu

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Whoami

- An undergraduate student interested in Linux
- Working on a summer project on kernel testing at UIUC
- This is my first time giving a conference talk ☺

Motivation and Goals

- **Modernizing Linux kernel testing** (e.g., CI for Linux)
- **High-coverage, effective test cases**
  - Incremental changes can be well tested
- **Test selection and prioritization (and minimization)**
  - Only a small number of tests need to be run
  - First running tests that are more likely to find bugs
- **Bug localization and reproducibility**
  - Localizing bugs when tests fail
  - Reproducing the bug by rerunning the tests

How far are we?

What should be done?

How can we contribute?

Commonly Used Test Suites

- Kernel space
- User space
- Combo

- **KUnit**
  - Aiming at drivers and common data structures (list, string, sort)
- **Kselftest**
  - A set of developer unit and regression tests
- **LTP (Linux Test Project)**
  - A comprehensive suite of user-space tests
- **Module-specific tests**
  - e.g. xfstests, blktests, kvm-unit-tests, device tests
- **RHEL test suites**
  - Very large test suite including LTP , KUnit, stess-ng, xfstests, etc.
  - Many tests target on preinstalled packages in the distro

We focus on test suites used by KernelCI

- KernelCI native tests only contain kselftest and LTP
- We focus on studying the following test suites
  - Booting the kernel (linux v6.9.8)
  - KUnit (linux v6.9.8)
  - LTP (20240524)
  - Kselftest (linux v6.9.8)
- A future work is to add syscall fuzzers like Syzcallar
  - Didn’t get the chance to run
  - May not be suitable for CI testing

Setup and Configuration

- Kernel version 6.9.8
- defconfig for x86_64, along with:
  - LLVM coverage tool
  - KUnit tests
  - Ethernet driver
- LLVM 19.0.0
  - **Metrics:** Function, line, branch and MC/DC coverage
- Cloudlab c6420 machine
  - Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz
  - 251GB memory, 8GB swap.

Source-based Code Coverage (SCC)

- **Provides precise, source-based coverage reports**
  - Instrumentation happens at the frontend
  - Dedicated coverage mapping regions
- **Not susceptible to compiler optimization**
  - Optimization is enabled by default when building the kernel, which often confuses existing coverage tools
- **More informative when evaluating test suites**
- Kernel support is publicly requesting for comments

Source-based Code Coverage (SCC)

- **Provides precise, source-based coverage reports**
  - Instrumentation happens at the frontend
  - Dedicated coverage mapping regions
- **Not susceptible to compiler optimization**
  - Optimization is enabled by default when building the kernel, which often confuses existing coverage tools
- **More informative when evaluating test suites**
- Kernel support is publicly requesting for comments

"Source-based code coverage of Linux kernel" by Wentao Zhang
Safe Systems with Linux MC, **today 16:00, Hall N2** (Austria Center)

Overall Coverage (Boot + KUnit + Kselftest + LTP)

Module	Function	Line	Branch	MC/DC
arch/x86/	48.69% (2841/5835)	38.09% (26249/68913)	28.81% (11708/40634)	8.87% (276/3113)
block/	55.37% (810/1463)	42.63% (8357/19603)	31.26% (3324/10634)	13.24% (121/914)
certs/	57.14% (4/7)	50.00% (49/98)	36.67% (11/30)	0.00% (0/2)
crypto/	28.95% (264/912)	24.05% (3009/12512)	19.12% (825/4314)	3.41% (14/411)
drivers/	19.76% (6624/33524)	15.70% (93231/593941)	10.86% (31926/293880)	3.32% (919/27720)
fs/	56.65% (4798/8469)	46.93% (71736/152866)	38.20% (26443/69222)	19.90% (1260/6331)
include/	43.29% (5866/13550)	35.29% (24777/70214)	38.88% (5801/14920)	17.95% (300/1671)
init/	53.28% (65/122)	45.80% (775/1692)	31.60% (201/636)	9.30% (8/86)
io_uring/	34.55% (255/738)	23.24% (2738/11782)	14.09% (870/6174)	0.46% (3/657)
ipc/	79.94% (247/309)	73.24% (3898/5322)	61.10% (1233/2018)	37.58% (56/149)
kernel/	69.64% (5692/8173)	58.66% (64531/110009)	44.61% (24851/55712)	26.29% (1440/5477)
lib/	61.17% (1624/2655)	51.94% (23346/44949)	34.81% (11123/31954)	23.34% (345/1478)
mm/	76.42% (2483/3249)	65.82% (35656/54171)	52.13% (14463/27742)	30.11% (812/2697)
net/	36.89% (4876/13217)	25.87% (71023/274552)	17.58% (27168/154540)	6.13% (966/15747)
security/	54.58% (751/1376)	35.07% (8237/23485)	28.84% (3449/11958)	11.92% (98/822)
sound/	14.05% (235/1673)	11.09% (2972/26793)	7.56% (979/12956)	1.21% (15/1240)
Totals	**39.29% (37435/95272)**	**29.95% (440584/1470902)**	**22.29% (164375/737324)**	**9.68% (6633/68515)**

Overall Coverage (Boot + KUnit + Kselftest + LTP)

Module	Function	Line	Branch	MC/DC
arch/x86/	48.69% (2841/5835)	38.09% (26249/68913)	28.81% (11708/40634)	8.87% (276/3113)
block/	55.37% (810/1463)	42.63% (8357/19603)	31.26% (3324/10634)	13.24% (121/914)
certs/	57.14% (4/7)	50.00% (49/98)	36.67% (11/30)	0.00% (0/2)
crypto/	28.95% (264/912)	24.05% (3009/12512)	19.12% (825/4314)	3.41% (14/411)
drivers/	19.76% (6624/33524)	15.70% (93231/593941)	10.86% (31926/293880)	3.32% (919/27720)
fs/	56.65% (4798/8469)	46.93% (71736/152866)	38.20% (26443/69222)	19.90% (1260/6331)
include/	43.29% (5866/13550)	35.29% (24777/70214)	38.88% (5801/14920)	17.95% (300/1671)
init/	53.28% (65/122)	45.80% (775/1692)	31.60% (201/636)	9.30% (8/86)
io_uring/	34.55% (255/738)	23.24% (2738/11782)	14.09% (870/6174)	0.46% (3/657)
ipc/	79.94% (247/309)	73.24% (3898/5322)	61.10% (1233/2018)	37.58% (56/149)
kernel/	69.64% (5692/8173)	58.66% (64531/110009)	44.61% (24851/55712)	26.29% (1440/5477)
lib/	61.17% (1624/2655)	51.94% (23346/44949)	34.81% (11123/31954)	23.34% (345/1478)
mm/	76.42% (2483/3249)	65.82% (35656/54171)	52.13% (14463/27742)	30.11% (812/2697)
net/	36.89% (4876/13217)	25.87% (71023/274552)	17.58% (27168/154540)	6.13% (966/15747)
security/	54.58% (751/1376)	35.07% (8237/23485)	28.84% (3449/11958)	11.92% (98/822)
sound/	14.05% (235/1673)	11.09% (2972/26793)	7.56% (979/12956)	1.21% (15/1240)
Totals	**39.29% (37435/95272)**	**29.95% (440584/1470902)**	**22.29% (164375/737324)**	**9.68% (6633/68515)**

Overall Coverage (Boot + KUnit + Kselftest + LTP)

Module	Function	Line	Branch	MC/DC
arch/x86/	48.69% (2841/5835)	38.09% (26249/68913)	28.81% (11708/40634)	8.87% (276/3113)
block/	55.37% (810/1463)	42.63% (8357/19603)	31.26% (3324/10634)	13.24% (121/914)
certs/	57.14% (4/7)	50.00% (49/98)	36.67% (11/30)	0.00% (0/2)
crypto/	28.95% (264/912)	24.05% (3009/12512)	19.12% (825/4314)	3.41% (14/411)
drivers/	19.76% (6624/33524)	15.70% (93231/593941)	10.86% (31926/293880)	3.32% (919/27720)
fs/	56.65% (4798/8469)	46.93% (71736/152866)	38.20% (26443/69222)	19.90% (1260/6331)
include/	43.29% (5866/13550)	35.29% (24777/70214)	38.88% (5801/14920)	17.95% (300/1671)
init/	53.28% (65/122)	45.80% (775/1692)		9.30% (8/86)
io_uring/	34.55% (255/738)	23.24% (2738/11782)		0.46% (3/657)
ipc/	79.94% (247/309)	73.24% (3898/5322)	61.10% (1233/2018)	37.58% (56/149)
kernel/	69.64% (5692/8173)	58.66% (64531/110009)	44.61% (24851/55712)	26.29% (1440/5477)
lib/	61.17% (1624/2655)	51.94% (23346/44949)	34.81% (11123/31954)	23.34% (345/1478)
mm/	76.42% (2483/3249)	65.82% (35656/54171)	52.13% (14463/27742)	30.11% (812/2697)
net/	36.89% (4876/13217)	25.87% (71023/274552)	17.58% (27168/154540)	6.13% (966/15747)
security/	54.58% (751/1376)	35.07% (8237/23485)	28.84% (3449/11958)	11.92% (98/822)
sound/	14.05% (235/1673)	11.09% (2972/26793)	7.56% (979/12956)	1.21% (15/1240)
Totals	39.29% (37435/95272)	**29.95% (440584/1470902)**	22.29% (164375/737324)	9.68% (6633/68515)

Mostly ≤ 60%

Increase of Coverage (KUnit alone)

Module	Function	Line	Branch	MC/DC
arch/x86/	↑ 1.20% (↑ 70/ 5835)	↑ 1.17% (↑ 807/ 68913)	↑ 0.91% (↑ 369/ 40634)	↑ 0.13% (↑ 4/ 3113)
block/	↑ 0.61% (↑ 9/ 1463)	↑ 0.51% (↑ 100/ 19603)	↑ 0.58% (↑ 61/ 10634)	↑ 0.55% (↑ 5/ 914)
certs/	---% (---/ 7)	---% (---/ 98)	---% (---/ 30)	---% (---/ 2)
crypto/	---% (---/ 912)	---% (---/ 12512)	---% (---/ 4314)	---% (---/ 411)
drivers/	↑ 2.72% (↑ 912/ 33524)	↑ 2.31% (↑13694/ 593941)	↑ 1.77% (↑ 5207/ 293880)	↑ 0.56% (↑ 154/ 27720)
fs/	↑ 1.03% (↑ 88/ 8467)	↑ 0.94% (↑ 1477/ 152759)	↑ 0.76% (↑ 542/ 69178)	↑ 0.44% (↑ 28/ 6325)
include/	↑ 1.91% (↑ 259/ 13550)	↑ 1.88% (↑ 1320/ 70214)	↑ 2.64% (↑ 394/ 14920)	↑ 1.91% (↑ 32/ 1671)
init/	---% (---/ 122)	---% (---/ 1692)	---% (---/ 636)	---% (---/ 86)
io_uring/	---% (---/ 738)	---% (---/ 11782)	---% (---/ 6174)	---% (---/ 657)
ipc/	---% (---/ 309)	---% (---/ 5322)	---% (---/ 2018)	---% (---/ 149)
kernel/	↑ 3.79% (↑ 310/ 8173)	↑ 3.31% (↑ 3641/ 110009)	↑ 2.55% (↑ 1423/ 55712)	↑ 0.88% (↑ 48/ 5477)
lib/	↑19.62% (↑ 521/ 2655)	↑17.04% (↑ 7661/ 44949)	↑15.24% (↑ 4871/ 31954)	↑ 5.34% (↑ 79/ 1478)
mm/	↑ 1.65% (↑ 54/ 3248)	↑ 1.16% (↑ 636/ 54160)	↑ 0.95% (↑ 265/ 27738)	↑ 0.82% (↑ 22/ 2697)
net/	↑ 1.19% (↑ 158/ 13217)	↑ 1.28% (↑ 3539/ 274552)	↑ 0.98% (↑ 1508/ 154540)	↑ 0.40% (↑ 63/ 15747)
security/	↑ 0.44% (↑ 6/ 1376)	↑ 0.19% (↑ 43/ 23485)	↑ 0.23% (↑ 28/ 11958)	↑ 0.12% (↑ 1/ 822)
sound/	↑ 2.63% (↑ 44/ 1673)	↑ 1.75% (↑ 467/ 26793)	↑ 1.54% (↑ 200/ 12956)	↑ 0.65% (↑ 8/ 1240)
Totals	**↑ 2.55% (↑ 2431/ 95269)**	**↑ 2.26% (↑33385/1470784)**	**↑ 2.01% (↑14868/ 737276)**	**↑ 0.65% (↑ 444/ 68509)**

Increase of Coverage (KUnit alone)

Module	Function	Line	Branch	MC/DC
arch/x86/	↑ 1.20% (↑ 70/ 5835)	↑ 1.17% (↑ 807/ 68913)	↑ 0.91% (↑ 369/ 40634)	↑ 0.13% (↑ 4/ 3113)
block/	↑ 0.61% (↑ 9/ 1463)	↑ 0.51% (↑ 100/ 19603)	↑ 0.58% (↑ 61/ 10634)	↑ 0.55% (↑ 5/ 914)
certs/	---% (---/ 7)	---% (---/ 98)	---% (---/ 30)	---% (---/ 2)
crypto/	---% (---/ 912)	---% (---/ 12512)	---% (---/ 4314)	---% (---/ 411)
drivers/	**↑ 2.72% (↑ 912/ 33524)**	**↑ 2.31% (↑13694/ 593941)**	**↑ 1.77% (↑ 5207/ 293880)**	**↑ 0.56% (↑ 154/ 27720)**
fs/	↑ 1.03% (↑ 88/ 8467)	↑ 0.94% (↑ 1477/ 152759)	↑ 0.76% (↑ 542/ 69178)	↑ 0.44% (↑ 28/ 6325)
include/	↑ 1.91% (↑ 259/ 13550)	↑ 1.88% (↑ 1320/ 70214)	↑ 2.64% (↑ 394/ 14920)	↑ 1.91% (↑ 32/ 1671)
init/	---% (---/ 122)	---% (---/ 1692)	---% (---/ 636)	---% (---/ 86)
io_uring/	---% (---/ 738)	---% (---/ 11782)	---% (---/ 6174)	---% (---/ 657)
ipc/	---% (---/ 309)	---% (---/ 5322)	---% (---/ 2018)	---% (---/ 149)
kernel/	↑ 3.79% (↑ 310/ 8173)	↑ 3.31% (↑ 3641/ 110009)	↑ 2.55% (↑ 1423/ 55712)	↑ 0.88% (↑ 48/ 5477)
lib/	**↑19.62% (↑ 521/ 2655)**	**↑17.04% (↑ 7661/ 44949)**	**↑15.24% (↑ 4871/ 31954)**	**↑ 5.34% (↑ 79/ 1478)**
mm/	↑ 1.65% (↑ 54/ 3248)	↑ 1.16% (↑ 636/ 54160)	↑ 0.95% (↑ 265/ 27738)	↑ 0.82% (↑ 22/ 2697)
net/	↑ 1.19% (↑ 158/ 13217)	↑ 1.28% (↑ 3539/ 274552)	↑ 0.98% (↑ 1508/ 154540)	↑ 0.40% (↑ 63/ 15747)
security/	↑ 0.44% (↑ 6/ 1376)	↑ 0.19% (↑ 43/ 23485)	↑ 0.23% (↑ 28/ 11958)	↑ 0.12% (↑ 1/ 822)
sound/	**↑ 2.63% (↑ 44/ 1673)**	**↑ 1.75% (↑ 467/ 26793)**	**↑ 1.54% (↑ 200/ 12956)**	**↑ 0.65% (↑ 8/ 1240)**
Totals	↑ 2.55% (↑ 2431/ 95269)	↑ 2.26% (↑33385/1470784)	↑ 2.01% (↑14868/ 737276)	↑ 0.65% (↑ 444/ 68509)

A Few Very High-level Observations

A Few Very High-level Observations

- **The coverage of kernel tests is far from adequate**
  - Especially compared with modern (user-space) software projects
  - MC/DC is particularly low (<10%) – long way to go as safe systems

A Few Very High-level Observations

- **The coverage of kernel tests is far from adequate**
  - Especially compared with modern (user-space) software projects
  - MC/DC is particularly low (<10%) – long way to go as safe systems
- **User-space tests (LTP/Kselftest) are ineffective in testing drivers**
  - LTP(2,440 tests) performs even weaker than KUnit (596 tests)

A Few Very High-level Observations

- **The coverage of kernel tests is far from adequate**
  - Especially compared with modern (user-space) software projects
  - MC/DC is particularly low (<10%) – long way to go as safe systems
- **User-space tests (LTP/Kselftest) are ineffective in testing drivers**
  - LTP(2,440 tests) performs even weaker than KUnit (596 tests)
- **The higher covered modules are mostly less than 60% (line cov.)**
  - Branch coverage is much lower

A Few Very High-level Observations

- **The coverage of kernel tests is far from adequate**
  - Especially compared with modern (user-space) software projects
  - MC/DC is particularly low (<10%) – long way to go as safe systems
- **User-space tests (LTP/Kselftest) are ineffective in testing drivers**
  - LTP(2,440 tests) performs even weaker than KUnit (596 tests)
- **The higher covered modules are mostly less than 60% (line cov.)**
  - Branch coverage is much lower
- **Certain subsystems (e.g., `init` and `certs`) is rarely covered**
  - Makes boot-time bugs hard to detect early and causes trouble in debugging

Overall Coverage (Boot + KUnit + Kselftest + LTP)

Module	Function	Line	Branch	MC/DC
arch/x86/	48.69% (2841/5835)	38.09% (26249/68913)	28.81% (11708/40634)	8.87% (276/3113)
block/	55.37% (810/1463)	42.63% (8357/19603)	31.26% (3324/10634)	13.24% (121/914)
certs/	57.14% (4/7)	50.00% (49/98)	36.67% (11/30)	0.00% (0/2)
crypto/	28.95% (264/912)	24.05% (3009/12512)	19.12% (825/4314)	3.41% (14/411)
drivers/	19.76% (6624/33524)	15.70% (93231/593941)	10.86% (31926/293880)	3.32% (919/27720)
fs/	56.65% (4798/8469)	46.93% (71736/152866)	38.20% (26443/69222)	19.90% (1260/6331)
include/	43.29% (5866/13550)	35.29% (24777/70214)	38.88% (5801/14920)	17.95% (300/1671)
init/	53.28% (65/122)	45.80% (775/1692)	31.60% (201/636)	9.30% (8/86)
io_uring/	34.55% (255/738)	23.24% (2738/11782)	14.09% (870/6174)	0.46% (3/657)
ipc/	**79.94% (247/309)**	**73.24% (3898/5322)**	**61.10% (1233/2018)**	**37.58% (56/149)**
kernel/	69.64% (5692/8173)	58.66% (64531/110009)	44.61% (24851/55712)	26.29% (1440/5477)
lib/	61.17% (1624/2655)	51.94% (23346/44949)	34.81% (11123/31954)	23.34% (345/1478)
mm/	76.42% (2483/3249)	65.82% (35656/54171)	52.13% (14463/27742)	30.11% (812/2697)
net/	36.89% (4876/13217)	25.87% (71023/274552)	17.58% (27168/154540)	6.13% (966/15747)
security/	54.58% (751/1376)	35.07% (8237/23485)	28.84% (3449/11958)	11.92% (98/822)
sound/	14.05% (235/1673)	11.09% (2972/26793)	7.56% (979/12956)	1.21% (15/1240)
Totals	**39.29% (37435/95272)**	**29.95% (440584/1470902)**	**22.29% (164375/737324)**	**9.68% (6633/68515)**

ipc/

- **Small, simple subsystem (12 files, 5322 lines in total)**
- **Extensively exercised by existing tests (especially LTP)**
    - Line coverage: 73.2%
    - Branch coverage: 61.0%

ipc/

- **Small, simple subsystem (12 files, 5322 lines in total)**
- **Extensively exercised by existing tests (especially LTP)**
  - Line coverage: 73.2%
  - Branch coverage: 61.0%
- **What code are not covered by existing tests?**
  - Missing usages
  - Error path
  - Execution context (e.g., privileged or not)
  - 32-bit compatibility

Missing Usages

- **POSIX message queue messages can have different priorities**

```
int mq_send(mqd_t mqdes, const char msg_ptr[.msg_len],
            size_t msg_len, unsigned int msg_prio);
```

- **Code about RB-tree search is not covered**
  - Implying that only one priority is used in each test case

```
202   90   if (likely(leaf->priority == msg->m_type))
203   90       goto insert_msg;
204   0        else if (msg->m_type < leaf->priority) {
             Branch (204:12): [True: 0, False: 0]
205   0        p = &(*p)->rb_left;
206   0        rightmost = false;
207   0    } else
208   0        p = &(*p)->rb_right;
```

/ipc/mqueue.c

Error Path

- **Error handling code spread all over the kernel code**
  - Validity check followed by cleanup code
  - Assignment of errno and return
- **Covering all errno of a syscall != covering all error paths**

```
556    13    if (!ipc_valid_object(&msq->q_perm)) {
               Branch (556:6): [True: 0, False: 13]
557     0        ipc_unlock_object(&msq->q_perm);
558     0        err = -EIDRM;
559     0        goto out_unlock;
560     0    }
```

/ipc/msg.c

Enhancing Existing Tests

- **Which tests to enhance (among hundreds)?**
  - Tests that already exercise the target functions/statements

- **Tooling for finding tests based on coverage data**
  1. Compile the kernel with only target modules instrumented
  2. Run test suite and record per-test profiles
  3. Select tests based on the coverage data (e.g., function and statements)

Enhancing Existing Tests

- **Which tests to enhance (among hundreds)?**
  - Tests that already exercise the target functions/statements

- **Tooling for finding tests based on coverage data**
  1. Compile the kernel with only target modules instrumented
  2. Run test suite and record per-test profiles
  3. Select tests based on the coverage data (e.g., function and statements)

- **Let's see a demo.**

Overall Coverage (Boot + KUnit + Kselftest + LTP)

Module	Function	Line	Branch	MC/DC
arch/x86/	48.69% (2841/5835)	38.09% (26249/68913)	28.81% (11708/40634)	8.87% (276/3113)
block/	55.37% (810/1463)	42.63% (8357/19603)	31.26% (3324/10634)	13.24% (121/914)
certs/	57.14% (4/7)	50.00% (49/98)	36.67% (11/30)	0.00% (0/2)
crypto/	28.95% (264/912)	24.05% (3009/12512)	19.12% (825/4314)	3.41% (14/411)
drivers/	19.76% (6624/33524)	15.70% (93231/593941)	10.86% (31926/293880)	3.32% (919/27720)
fs/	56.65% (4798/8469)	46.93% (71736/152866)	38.20% (26443/69222)	19.90% (1260/6331)
include/	43.29% (5866/13550)	35.29% (24777/70214)	38.88% (5801/14920)	17.95% (300/1671)
init/	53.28% (65/122)	45.80% (775/1692)	31.60% (201/636)	9.30% (8/86)
io_uring/	34.55% (255/738)	23.24% (2738/11782)	14.09% (870/6174)	0.46% (3/657)
ipc/	79.94% (247/309)	73.24% (3898/5322)	61.10% (1233/2018)	37.58% (56/149)
kernel/	69.64% (5692/8173)	58.66% (64531/110009)	44.61% (24851/55712)	26.29% (1440/5477)
lib/	61.17% (1624/2655)	51.94% (23346/44949)	34.81% (11123/31954)	23.34% (345/1478)
mm/	**76.42% (2483/3249)**	**65.82% (35656/54171)**	**52.13% (14463/27742)**	**30.11% (812/2697)**
net/	36.89% (4876/13217)	25.87% (71023/274552)	17.58% (27168/154540)	6.13% (966/15747)
security/	54.58% (751/1376)	35.07% (8237/23485)	28.84% (3449/11958)	11.92% (98/822)
sound/	14.05% (235/1673)	11.09% (2972/26793)	7.56% (979/12956)	1.21% (15/1240)
Totals	39.29% (37435/95272)	29.95% (440584/1470902)	22.29% (164375/737324)	9.68% (6633/68515)

mm/

- **10+ times bigger module than /ipc**
  - 54,171 statements and 3,249 functions
- **Many features cannot be directly invoked by system calls**
  - Transparent huge pages (THP)
- **Behaviors may depend on memory layouts**
- **Complex multi-threading concurrency**

mm/

- **10+ times bigger module than /ipc**
  - 54,171 statements and 3,249 functions
- **Many features cannot be directly invoked by system calls**
  - Transparent huge pages (THP)
- **Behaviors may depend on memory layouts**
- **Complex multi-threading concurrency**
- **Few existing tests targeting THP**
  - 0 in KUnit
  - 1 in Kselftest as a stress test to exhaust physical memory
  - 4 regression tests to prevent old bugs

Allocating a Huge Page

1. Read lock
2. Scan
    - check availability
3. Read unlock
4. Allocate a huge page
5. Write lock
6. Isolate
    - recheck availability
7. Copy and other work
8. Write unlock

Allocating a Huge Page

1. Read lock
2. **Scan**
   - **check availability**
3. Read unlock
4. Allocate a huge page
5. Write lock
6. Isolate
   - recheck availability
7. Copy and other work
8. Write unlock

```
          static int hpage_collapse_scan_pmd(...)
          {
            ...
            for (_address = address, _pte = pte; _pte < pte + HPAGE_PMD_NR;
                    _pte++, _address += PAGE_SIZE) {
              ...
1.57M         if (pte_none(pteval) || is_zero_pfn(pte_pfn(pteval))) {
                ++none_or_zero;
133k            if(... none_or_zero <= khugepaged_max_ptes_none)) {
133k              continue;
133k            } else {
238               result = SCAN_EXCEED_NONE_PTE;
238               count_vm_event(THP_SCAN_EXCEED_NONE_PTE);
238               goto out;
238             }
```

Allocating a Huge Page

1. Read lock
2. **Scan**
   - **check availability**
3. Read unlock
4. Allocate a huge page
5. Write lock
6. Isolate
   - recheck availability
7. Copy and other work
8. Write unlock

```
        static int hpage_collapse_scan_pmd(...)
        {
          ...
          for (_address = address, _pte = pte; _pte < pte + HPAGE_PMD_NR;
                _pte++, _address += PAGE_SIZE) {
            ...
1.57M       if (pte_none(pteval) || is_zero_pfn(pte_pfn(pteval))) {
              ++none_or_zero;
133k          if(... none_or_zero <= khugepaged_max_ptes_none)) {
133k            continue;
133k          } else {
238             result = SCAN_EXCEED_NONE_PTE;
238             count_vm_event(THP_SCAN_EXCEED_NONE_PTE);
238             goto out;
238           }
```

Allocating a Huge Page

1. Read lock
2. **Scan**
   - **check availability**
3. Read unlock
4. Allocate a huge page
5. Write lock
6. Isolate
   - recheck availability
7. Copy and other work
8. Write unlock

```
static int hpage_collapse_scan_pmd(...)
            result = SCAN_EXCEED_NONE_PTE;
            count_vm_event(THP_SCAN_EXCEED_NONE_PTE);
            goto out;
        }
```

Allocating a Huge Page

1. Read lock
2. Scan
   - check availability
3. Read unlock
4. Allocate a huge page
5. Write lock
6. **Isolate**
   - **recheck availability**
7. Copy and other work
8. Write unlock

```
          static int hpage_collapse_scan_pmd(...)
 238              result = SCAN_EXCEED_NONE_PTE;
 238              count_vm_event(THP_SCAN_EXCEED_NONE_PTE);
 238              goto out;
 238          }
```

```
          static int __collapse_huge_page_isolate(...)
          {
            ...
1.19M       for (_address = address, _pte = pte; _pte < pte + HPAGE_PMD_NR;
1.19M            _pte++, _address += PAGE_SIZE) {
              ...
1.19M         if (pte_none(pteval) || is_zero_pfn(pte_pfn(pteval))) {
1.18M           ++none_or_zero;
2.60k           if(... none_or_zero <= khugepaged_max_ptes_none)) {
2.60k             continue;
2.60k           } else {
  0               result = SCAN_EXCEED_NONE_PTE;
  0               count_vm_event(THP_SCAN_EXCEED_NONE_PTE);
  0               goto out;
  0             }
```

Allocating a Huge Page

1. Read lock
2. Scan
   - check availability
3. Read unlock
4. Allocate a huge page
5. Write lock
6. **Isolate**
   - **recheck availability**
7. Copy and other work
8. Write unlock

```
         static int hpage_collapse_scan_pmd(...)
238              result = SCAN_EXCEED_NONE_PTE;
238              count_vm_event(THP_SCAN_EXCEED_NONE_PTE);
238              goto out;
238          }
```

```
         static int __collapse_huge_page_isolate(...)
         {
           ...
1.19M      for (_address = address, _pte = pte; _pte < pte + HPAGE_PMD_NR;
1.19M          _pte++, _address += PAGE_SIZE) {
             ...
1.19M        if (pte_none(pteval) || is_zero_pfn(pte_pfn(pteval))) {
1.18M          ++none_or_zero;
2.60k          if(... none_or_zero <= khugepaged_max_ptes_none)) {
2.60k            continue;
2.60k          } else {
  0              result = SCAN_EXCEED_NONE_PTE;
  0              count_vm_event(THP_SCAN_EXCEED_NONE_PTE);
  0              goto out;
  0            }
```

Use Kprobe to change memory states

1. Read lock
2. Scan
    - check availability
3. Read unlock
4. **Allocate a huge page**
5. Write lock
6. Isolate
    - recheck availability
7. Copy and other work
8. Write unlock

- We can change the memory status during step 4

```
static int collapse_huge_page(...)
{
    ...
    mmap_read_unlock(mm); ...
    result = alloc_charge_hpage(&hpage, mm, cc);
    if (result != SCAN_SUCCEED)
        goto out_nolock;
    ...
    mmap_read_lock(mm);
```

Use Kprobe to change memory states

1. Read lock
2. Scan
   - check availability
3. Read unlock
4. **Allocate a huge page**
5. Write lock
6. Isolate
   - recheck availability
7. Copy and other work
8. Write unlock

- We can change the memory status during step 4

```
1087    static int collapse_huge_page(...)
1090    {
            ...
1110        mmap_read_unlock(mm); ...
1112        result = alloc_charge_hpage(&hpage, mm, cc);
1113        if (result != SCAN_SUCCEED)
1114            goto out_nolock;
1115        ...
1116        mmap_read_lock(mm);
```

- Use kprobe to insert code before line 1112

Summary and Implications

- **Existing kernel tests need to be enhanced**
- **Many opportunities to enhancing existing tests**
  - Coverage measure can guide the enhancement
- **New mechanisms are needed beyond system calls**
  - e.g., THP and other event-triggering code
- **Existing tooling is quite rudimentary**
  - No test selection or analysis
  - Debugging is nontrivial and often challenging

Test Coverage and Bugs

	Module	Description	Type
1	mm/vmalloc	Fix return value of vb_alloc if size is 0	**Uncovered (function)**
2	mm/hugetlb	Fix missing hugetlb_lock for resv uncharge	**Uncovered (function)**
3	mm/madvise	Make MADV_POPULATE_(READ|WRITE) handle VM_FAULT_RETRY properly	**Uncovered (function)**
4	x86/mm/pat	Fix VM_PAT handling in COW mappings	**Uncovered (function)**
5	mm/vmalloc	Fix vmalloc which may return null if called with __GFP_NOFAIL	**Uncovered (branch)**
6	mm/hugetlb	Fix DEBUG_LOCKS_WARN_ON(1) when dissolve_free_hugetlb_folio()	**Uncovered (branch)**
7	mm/hugetlb	Check for anon_vma prior to folio allocation	**Uncovered (line)**
8	maple_tree	Fix mas_empty_area_rev() null pointer dereference	**Uncovered (execution)**
9	mm	Use memalloc_nofs_save() in page_cache_ra_order()	Concurrency
10	mm	Turn folio_test_hugetlb into a PageType	Concurrency
11	fork	Defer linking file vma until vma is fully initialized	Concurrency

Gaps and Opportunities

	Linux kernel tests	Advanced software tests
Unit tests	596 in KUnit with low coverage	Hundreds to thousands with high code coverage
Test selection	None	Widely used and many regression test selection algorithms
Test prioritization	None	Many algorithms but not too widely used.
Continuous integration	KernelCI, continuously running a few test suites	Frequent, incremental testing on every diff with test selection
Bug localization and repair	Manual, often difficult	Many algorithms and advances, leveraging ML/LLMs

Discussion

- How can we (from academia) help and contribute?
- What are the important, urgent (research) problems?

“Making Linux Fly: Towards a Certified Linux Kernel”
Refereed Track

“Measuring and Understanding Linux Kernel Tests”
Kernel Testing & Dependability MC

“Source-based code coverage of Linux kernel”
*Safe Systems with Linux MC, **today 16:00, Hall N2** (Austria Center)*