Diagnosing Kernel Concurrency Failures with AITIA

Dae R. Jeong[1], Minkyu Jung[1], Yuchan Lee[1], Byoungyoung Lee[2], Insik Shin[1], Youngjin Kwon[1]

[1]Korea Advanced Institute of Science & Technology
[2]Seoul National University

KAIST 서울대학교 SEOUL NATIONAL UNIVERSITY

Concurrency failures

- Concurrency failures manifest depending on thread interleavings

Concurrency failures

- Concurrency failures manifest depending on thread interleavings

Execution 1

Syscall A	Syscall B
if (***ptr*** is not NULL)	
dereference ***ptr***	
	ptr = NULL

Concurrency failures

- Concurrency failures manifest depending on thread interleavings

Execution 1

Syscall A	Syscall B
if (***ptr*** is not NULL)	
dereference ***ptr***	
	ptr = NULL

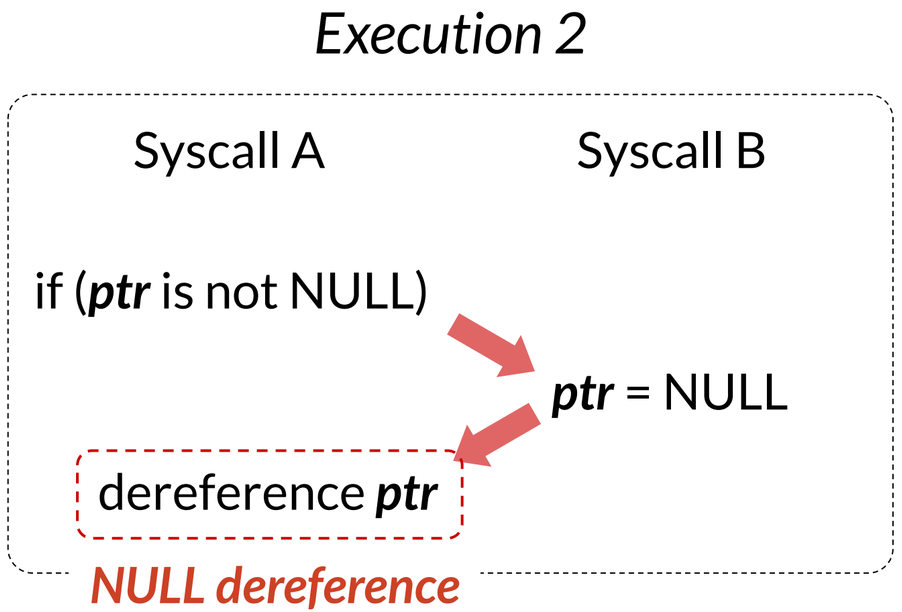
Execution 2

Syscall A	Syscall B
if (***ptr*** is not NULL)	
	ptr = NULL
dereference ***ptr***	

NULL dereference

Diagnosing the root causes of concurrency failures

- Developers need to understand the ***offending interleaving***



Diagnosing the root causes of concurrency failures

- Developers need to understand the ***offending interleaving***

- Developers fix the concurrency bug

  by preventing the ***offending interleaving***

Execution 2

Syscall A	Syscall B
lock(l)	
if (***ptr*** is not NULL)	
dereference ***ptr***	
unlock(l)	
	lock(l)
	ptr = NULL
	unlock(l)

Diagnosing the root causes of concurrency failures

- Developers need to understand the ***offending interleaving***

- Developers fix the concurrency bug

  by preventing the ***offending interleaving***

- However, developers often misunderstand the offending interleaving
  - *“In fact I was not aware of the possibility of such concurrent execution*

    and the current implementation would just make it …” - kernel developer

CVE-2017-15649: multi-variable concurrency bug

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A	Syscall B

if (***po->inactive***)
  return -EINVAL;
	if (***po->fanout***)
	  return -EINVAL;

po->fanout = match;	***po->inactive*** = true;

	if (***po->fanout*** &&
	  ! list_contain(sk, &***list***))
	  BUG();

list_add(sk, ***&list***);	

CVE-2017-15649: multi-variable concurrency bug

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A	Syscall B

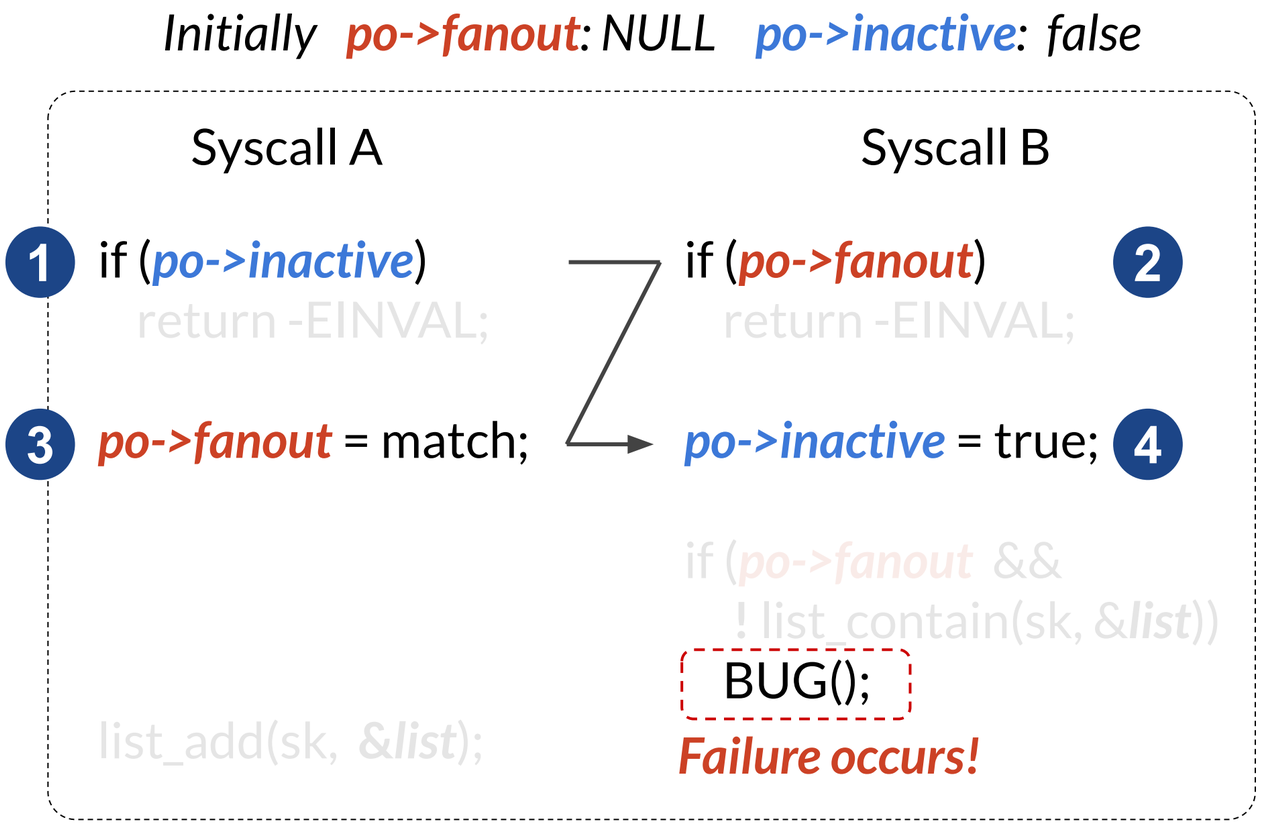
if (***po->inactive***)
  return -EINVAL;
 	
po->fanout = match;
	if (***po->fanout***)
	  return -EINVAL;
	
	po->inactive = true;

if (*po->fanout* &&
 ! list_contain(sk, &***list***))
 BUG();

list_add(sk, &***list***);

- One of *Syscall A* and *syscall B* should return with an error
  - Blue boxes should run atomically

CVE-2017-15649: multi-variable concurrency bug



- One of *Syscall A* and *syscall B* should return with an error
  - They should run atomically

- ***Multi-variable atomicity violation***
  - resulting in a failure

CVE-2017-15649: multi-variable concurrency bug

Initially ***po->fanout****: NULL* ***po->inactive****: false*

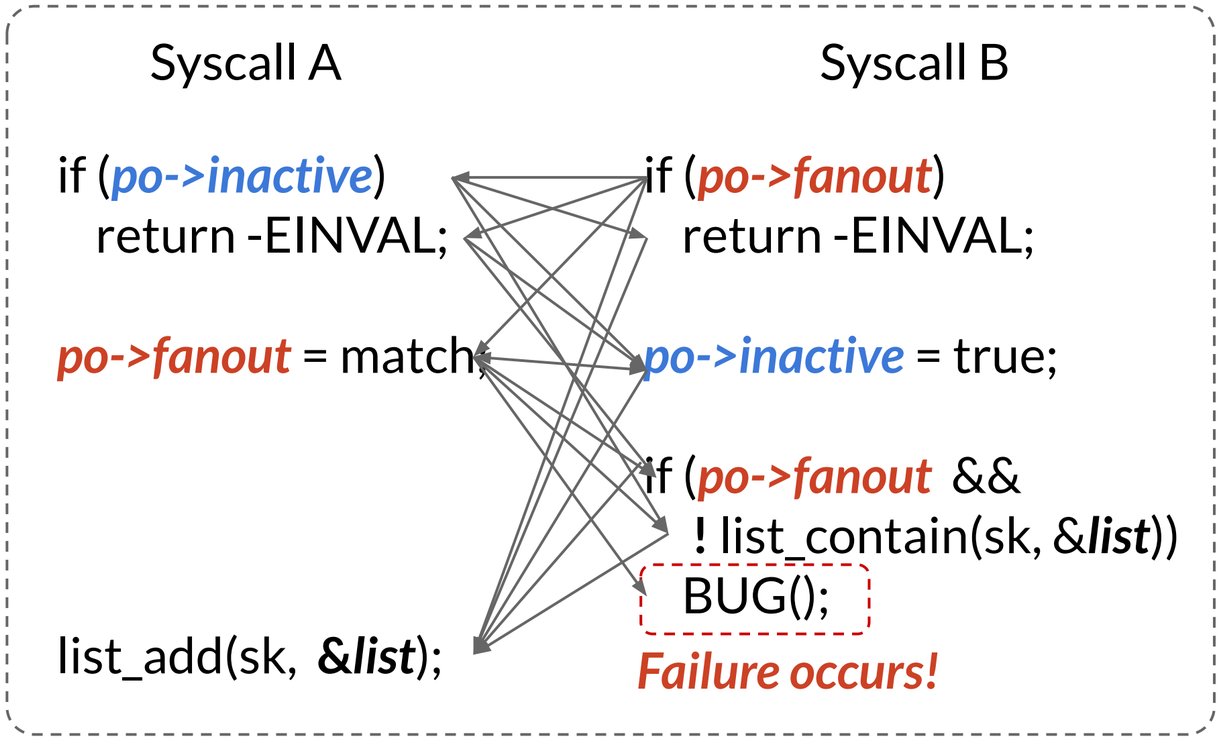
Syscall A	Syscall B
if (*po->inactive*) return -EINVAL;	if (*po->fanout*) return -EINVAL;
po->fa	
	! list_contain(sk, &*list*))
	BUG();
list_add(sk, &*list*);	***Failure occurs!***

Let's diagnose this failure

- One of *Syscall A* and *syscall B* should return with an error
  - They should run atomically
  - ***…ation***
  - resulting in a failure

CVE-2017-15649: multi-variable concurrency bug

Initially ***po->fanout****: NULL* ***po->inactive****: false*



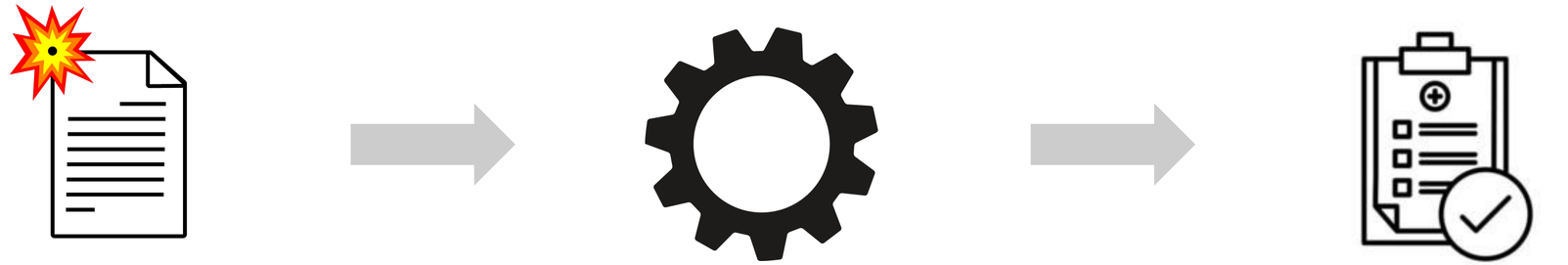
- A failure occurred
  - We want to understand the failure
- Too complicated to understand the offending interleaving
  - Enormous number of possible thread interleavings

AITIA: a root cause diagnosis for kernel concurrency bugs

- ***AITIA***
  - Automatically identifying the offending interleaving of the given concurrency failure

AITIA: a root cause diagnosis for kernel concurrency bugs

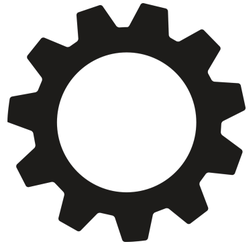
- ***AITIA***
  - Automatically identifying the offending interleaving of the given concurrency failure



AITIA: a root cause diagnosis for kernel concurrency bugs

- ***AITIA***
  - Automatically identifying the offending interleaving of the given concurrency failure

Coredump



AITIA

Root cause
as causality chain

AITIA: a root cause diagnosis for kernel concurrency bugs

- ***AITIA***
  - Automatically identifying the offending interleaving of the given concurrency failure

- ***Causality chain***
  - Explaining how the given failure eventually occurred

Causality chain

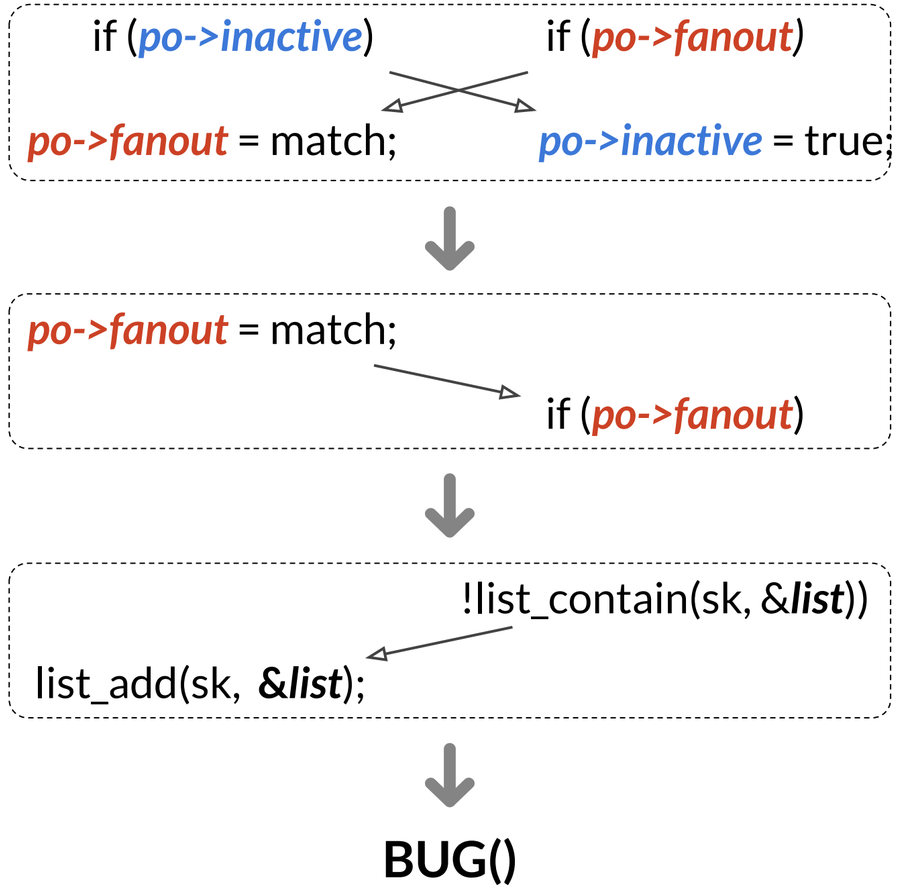
- Explaining how the given failure eventually occurred

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A	Syscall B

if (*po->inactive*)
  return -EINVAL;
	if (*po->fanout*)
	  return -EINVAL;

po->fanout = match;	*po->inactive* = true;
	if (*po->fanout* && ! list_contain(sk, &*list*))
	BUG();
list_add(sk, &*list*);	***Failure occurs!***



Causality chain

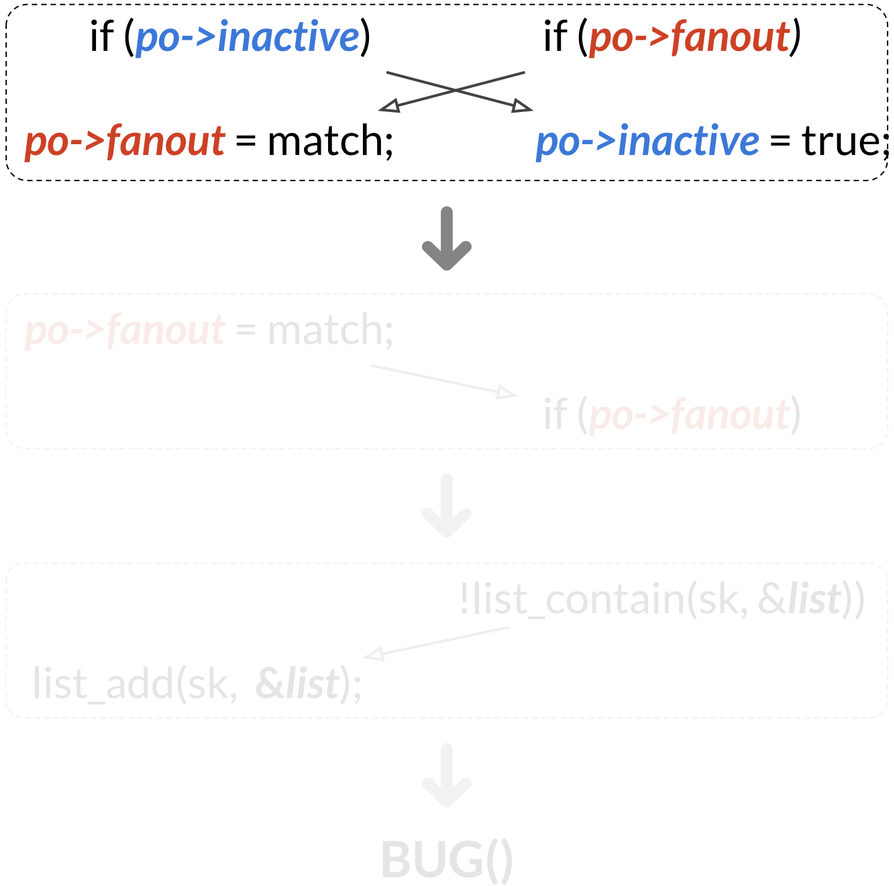
- Explaining how the given failure eventually occurred

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A	Syscall B

if (***po->inactive***)
  return -EINVAL;
	if (***po->fanout***)
	  return -EINVAL;

po->fanout = match;	***po->inactive*** = true;
	if (***po->fanout*** && ! list_contain(sk, &***list***))
	BUG();
list_add(sk, &***list***);	***Failure occurs!***



Causality chain

- Explaining how the given failure eventually occurred

Initially ***po->fanout****: NULL* ***po->inactive****: false*

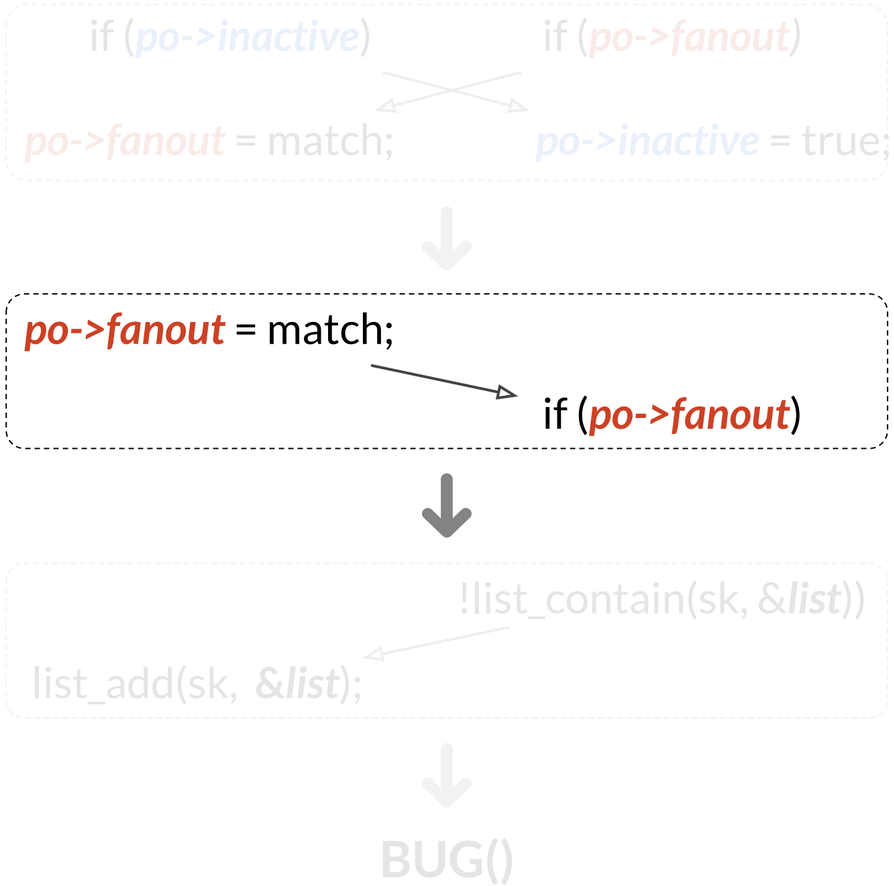
Syscall A	Syscall B

if (*po->inactive*)
  return -EINVAL;
	if (*po->fanout*)
	  return -EINVAL;

po->fanout = match;	*po->inactive* = true;

	if (***po->fanout*** &&
	  ! list_contain(sk, &***list***))
	  BUG();

list_add(sk, &***list***);	***Failure occurs!***



Causality chain

- Explaining how the given failure eventually occurred

Initially ***po->fanout***: *NULL* ***po->inactive***: *false*

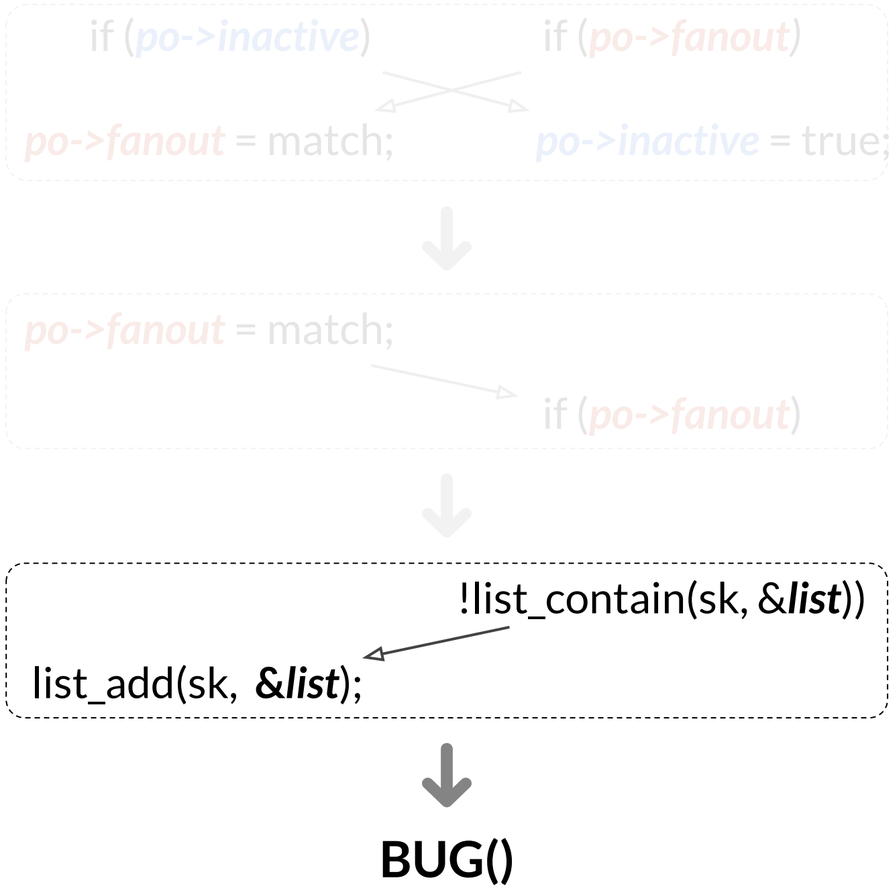
Syscall A	Syscall B

if (*po->inactive*)
  return -EINVAL;
	if (*po->fanout*)
	  return -EINVAL;

po->fanout = match;	*po->inactive* = true;

	if (*po->fanout* &&
	if (! list_contain(sk, &***list***))
	  BUG();

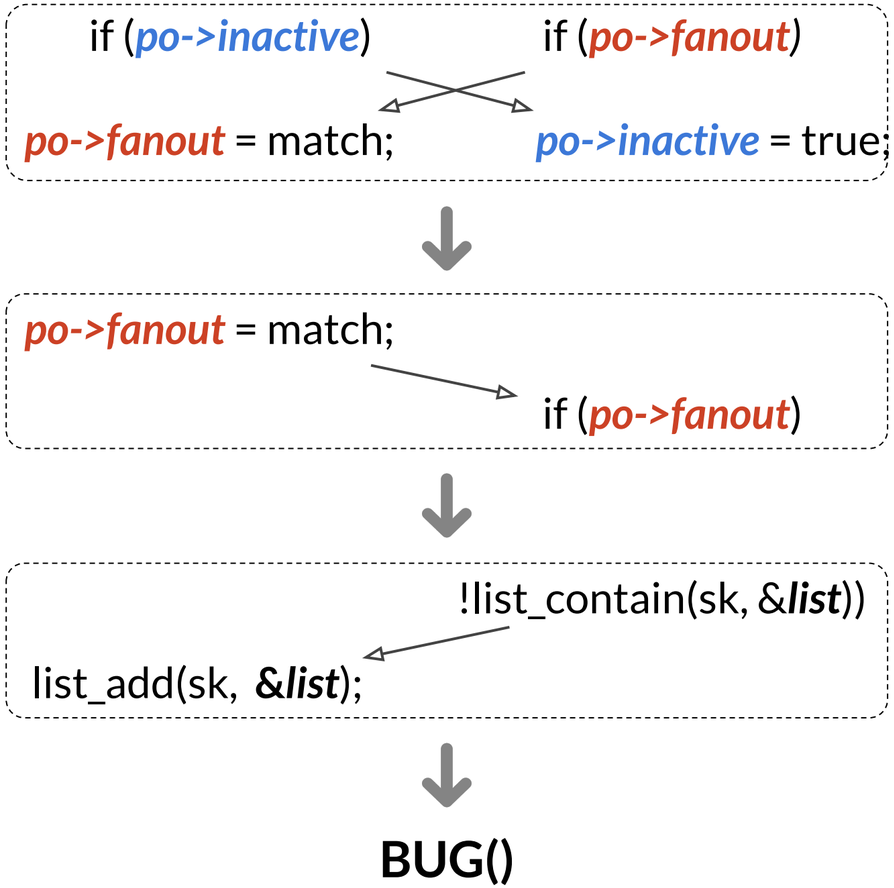
list_add(sk, ***&list***);	***Failure occurs!***



Causality chain

- Explaining how the given failure eventually occurred

- Causality chain
  - ***Comprehensively*** explain how the failure eventually occurred
  - ***Concise*** presentation excluding failure-irrelevant information

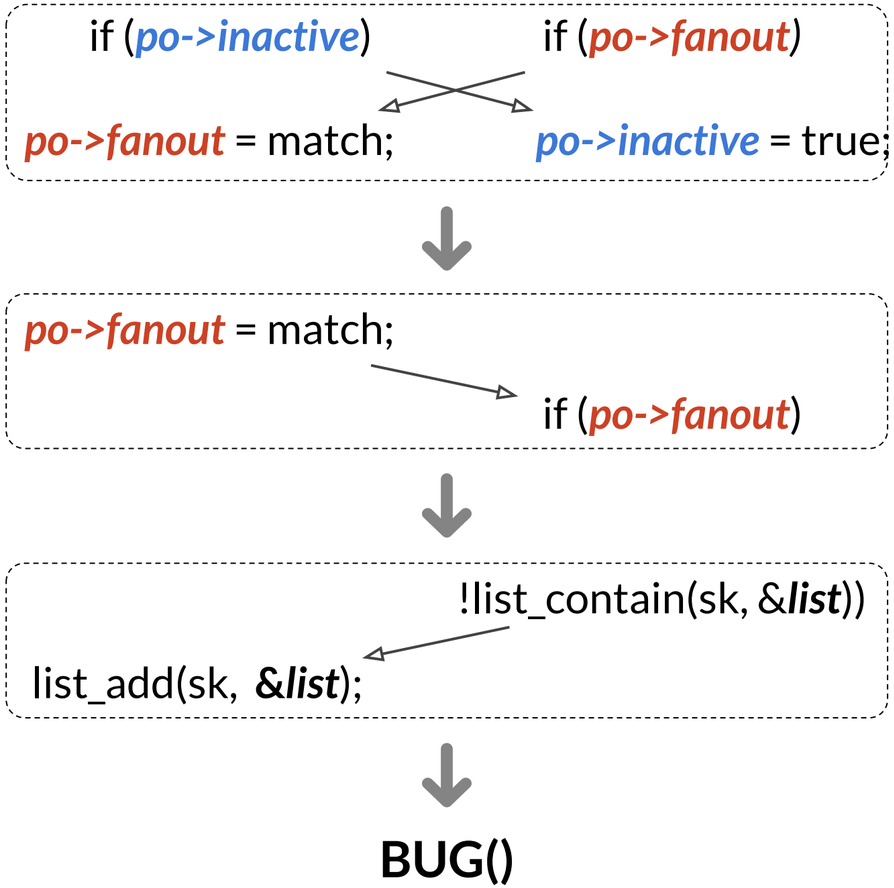


Causality chain

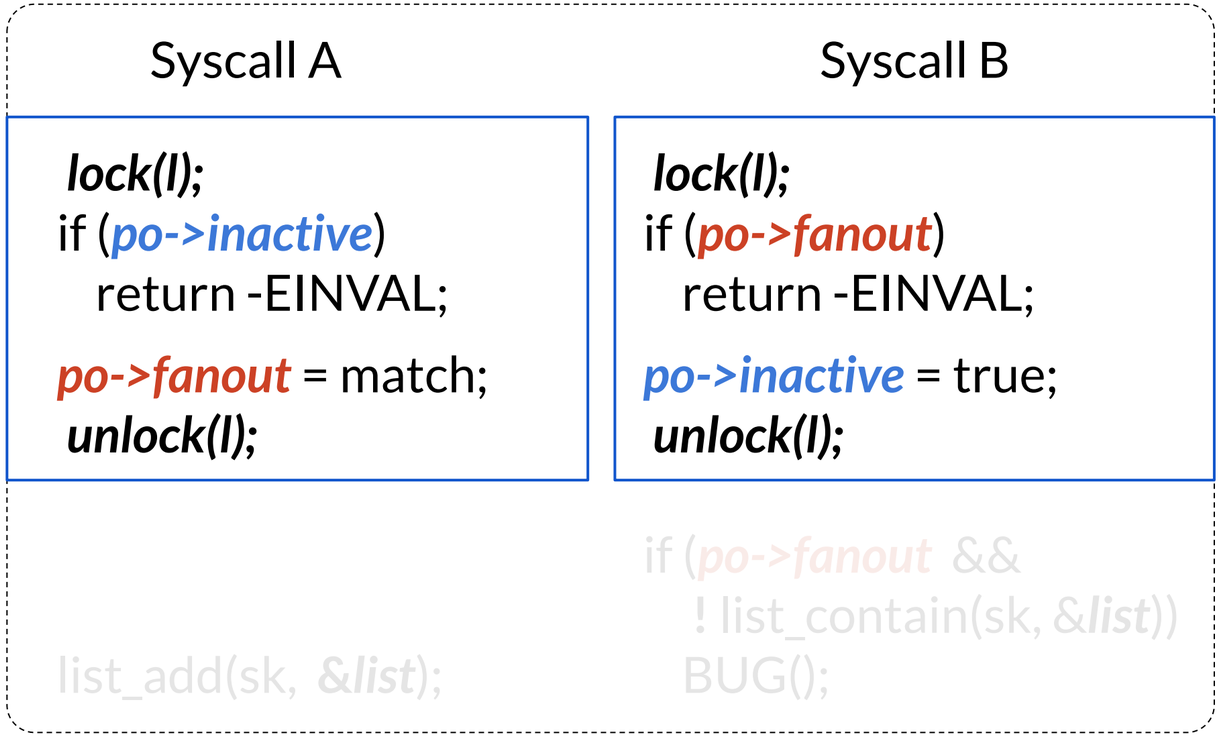
- Explaining how the given failure eventually occurred

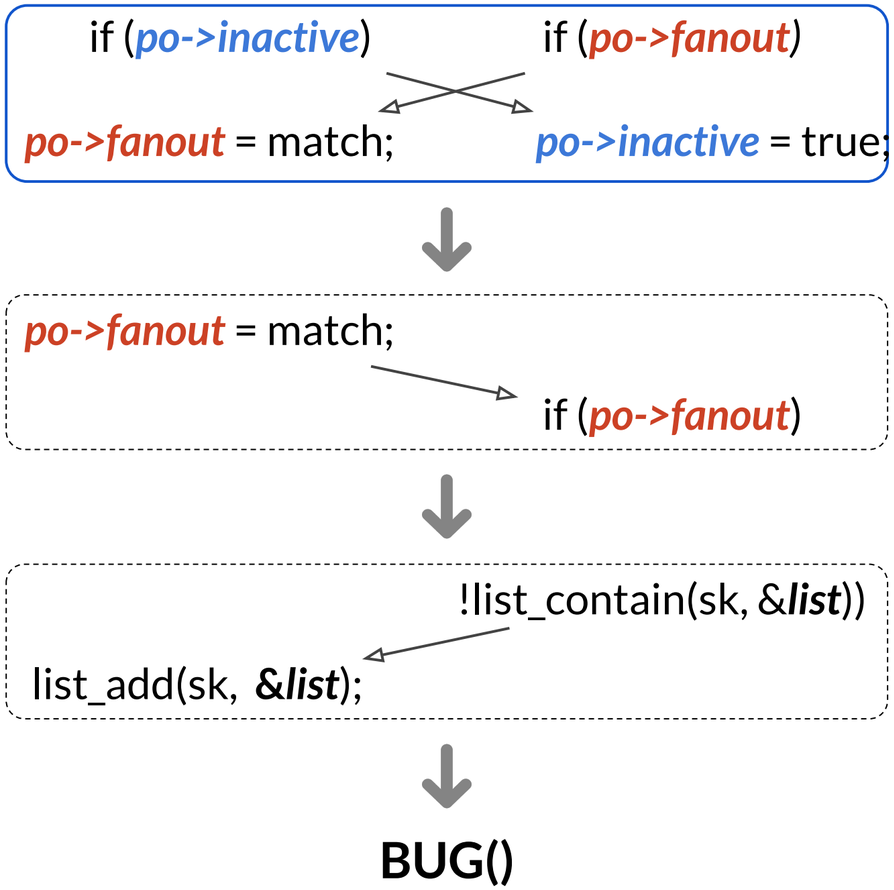
- Causality chain
  - ***Comprehensively*** explain how the failure eventually occurred
  - ***Concise*** presentation excluding failure-irrelevant information

- *"If a fix does not allow some interleaving orders in the chain, it prevents the failure"*



Causality chain





AITIA: a root cause diagnosis for kernel concurrency bugs

- ***AITIA***
  - Automatically identifying the offending interleaving of the given concurrency failure

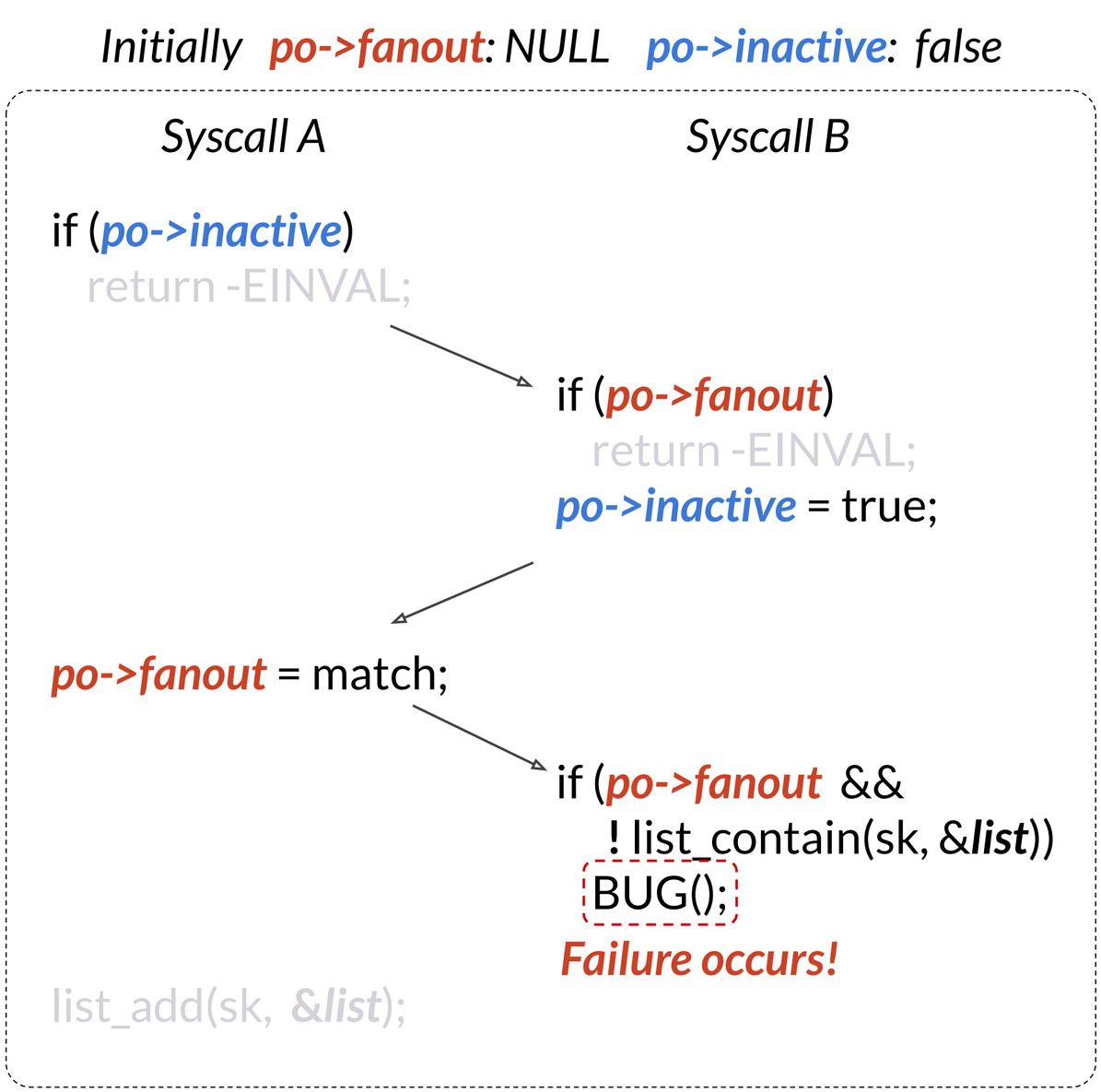
- ***Causality chain***
  - Explaining how the given failure eventually occurred

- Two steps to build a causality chain
  - ***Step 1:*** Constructing a totally-ordered instruction sequence
  - ***Step 2:*** Constructing the causality chain

Constructing the causality chain

- Key idea: **Flipping one interleaving order** of two instructions
  - Assume a totally-ordered failure-causing instruction sequence

Constructing the causality chain



From a large number of instructions, find the interleaving that directly contributes to the failure

Constructing the causality chain

Initially ***po->fanout***: NULL ***po->inactive***: *false*

Syscall A

```
if (po->inactive)
    return -EINVAL;
```

Syscall B

```
if (po->fanout)
    return -EINVAL;
po->inactive = true;
```

Flipping the interleaving order of instructions accessing ***list***

```
if (po->fanout && !po->inactive)
    list_add(sk, &list);
```

```
if (! list_contain(sk, &list))
    BUG();
```

Failure occurs!

Constructing the causality chain

Initially ***po->fanout***: NULL ***po->inactive***: *false*

Syscall A

if (***po->inactive***)
  return -EINVAL;

Syscall B

if (***po->fanout***)
  return -EINVAL;
po->inactive = true;

Flipping the interleaving order of instructions accessing ***list***

list_add(sk, ***&list***);

if (**!** list_contain(sk, &***list***))

Failure does not occur

Constructing the causality chain

Initially ***po->fanout***: NULL ***po->inactive***: *false*

Syscall A

if (*po->inactive*)
  return -EINVAL;

po->fanout = match;

list_add(sk, ***&list***);

Syscall B

if (*po->fanout*)
  return -EINVAL;
po->inactive = true;

if (*po->fanout* &&
if (**!** list_contain(sk, &***list***))
  BUG();

Failure occurs!

!list_contain(sk, &***list***))

list_add(sk, ***&list***);

BUG()

Constructing the causality chain

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A

```
if (po->inactive)
  return -EINVAL;
```

Syscall B

```
if (po->fanout)
  return -EINVAL;
po->inactive = true;
```

```
po->fanout = match;
```

```
if (po->fanout  &&
   ! list_contain(sk, &list))
  BUG();
```

Failure occurs!

```
list_add(sk,  &list);
```

```
!list_contain(sk, &list))
list_add(sk,  &list);
```

BUG()

Constructing the causality chain

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A

if (*po->inactive*)
  return -EINVAL;

Syscall B

if (*po->fanout*)

Flipping the interleaving order of instructions accessing ***po->fanout***

po->fanout = match;

if (***po->fanout*** &&
  ! list_contain(sk, &***list***))
  BUG();

Failure occurs!

list_add(sk, &***list***);

!list_contain(sk, &***list***))

list_add(sk, &***list***);

BUG()

Constructing the causality chain

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A

if (***po->inactive***)
return -EINVAL;

Syscall B

if (***po->fanout***)

Flipping the interleaving order of instructions accessing ***po->fanout***

if (***po->fanout*** &&
! list_contain(sk, &***list***))

po->fanout = match;

Failure does not occur

list_add(sk, &***list***);

!list_contain(sk, &***list***))

list_add(sk, &***list***);

BUG()

Constructing the causality chain

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A

Syscall B

if (*po->inactive*)
  return -EINVAL;

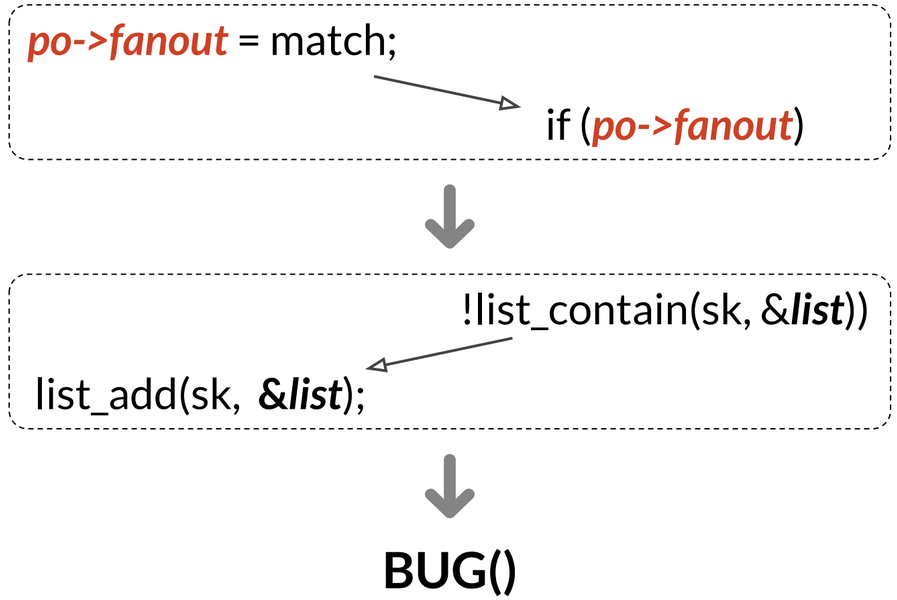
if (*po->fanout*)
  return -EINVAL;
po->inactive = true;

po->fanout = match;

if (***po->fanout*** &&
  ! list_contain(sk, &***list***))
  BUG();

Failure occurs!

list_add(sk, &***list***);



Constructing the causality chain

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A

```
if (po->inactive)
  return -EINVAL;
```

Syscall B

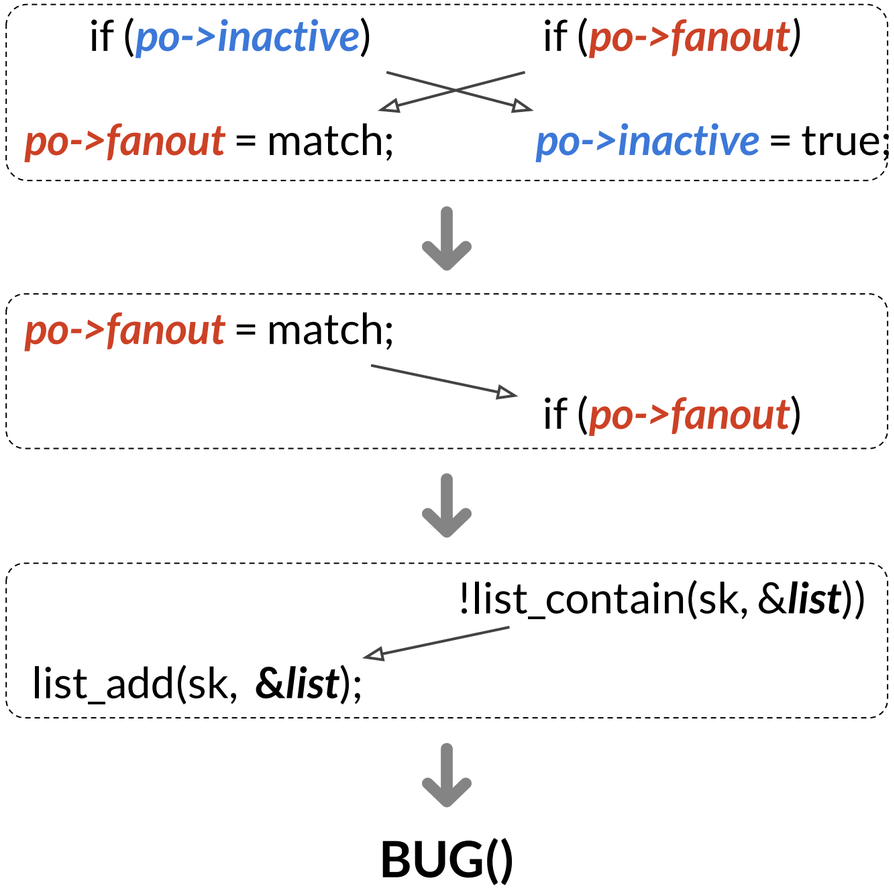
```
if (po->fanout)
  return -EINVAL;
po->inactive = true;
```

```
po->fanout = match;
```

```
if (po->fanout  &&
   ! list_contain(sk, &list))
  BUG();
```

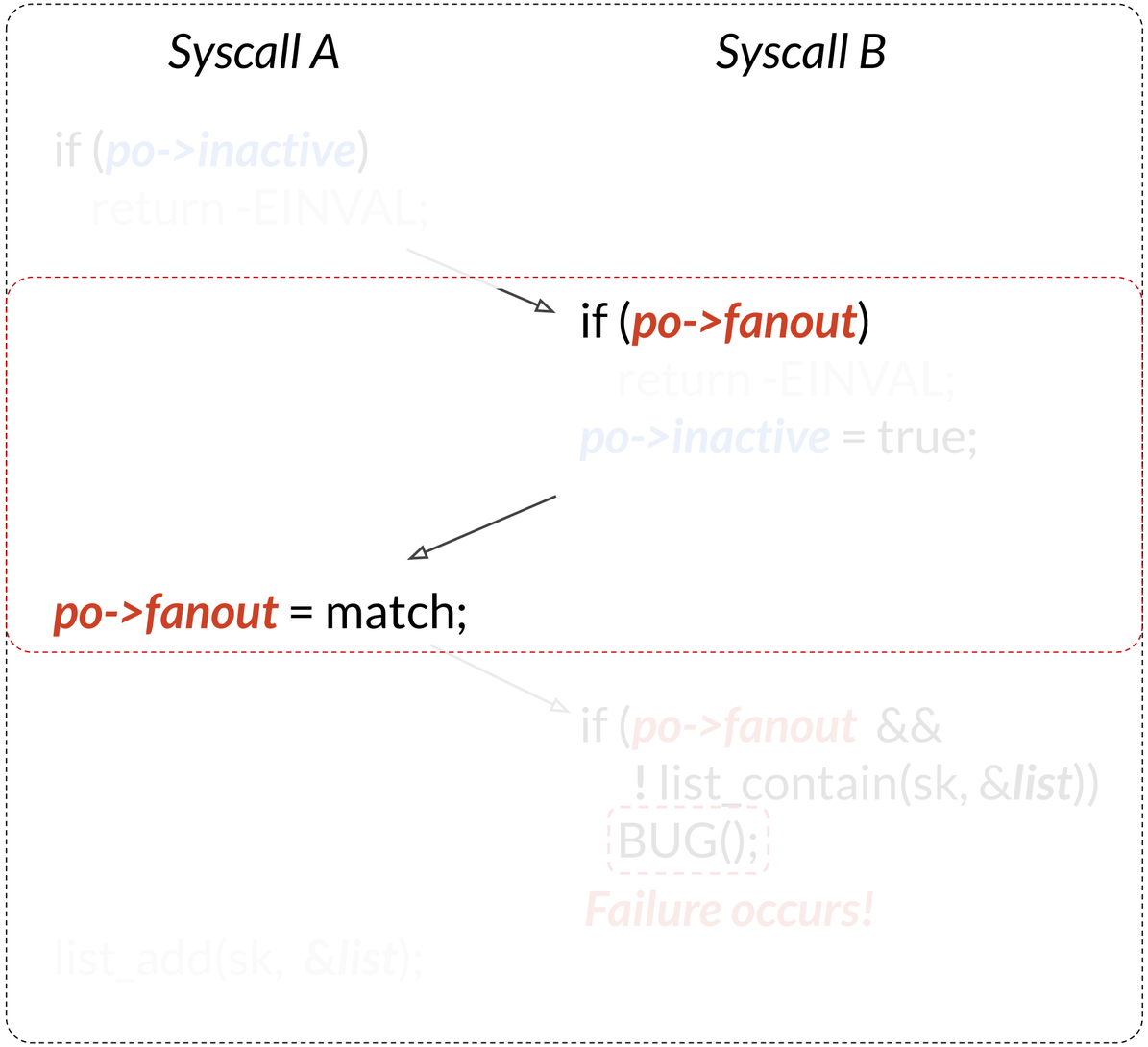
Failure occurs!

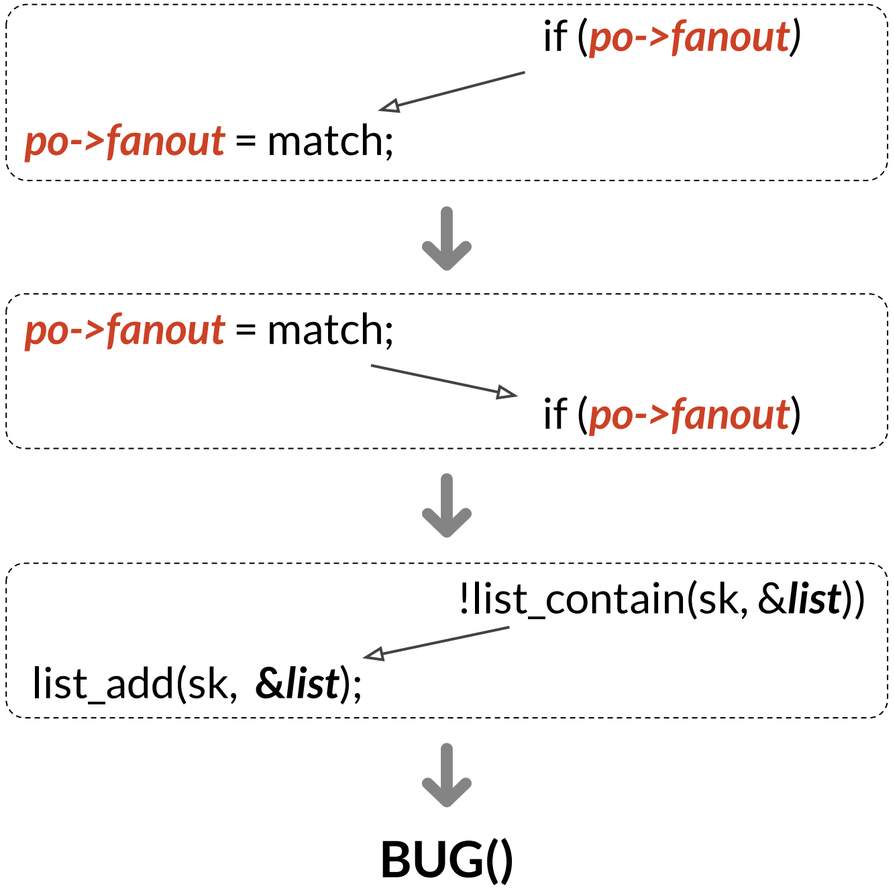
```
list_add(sk,  &list);
```



Constructing the causality chain

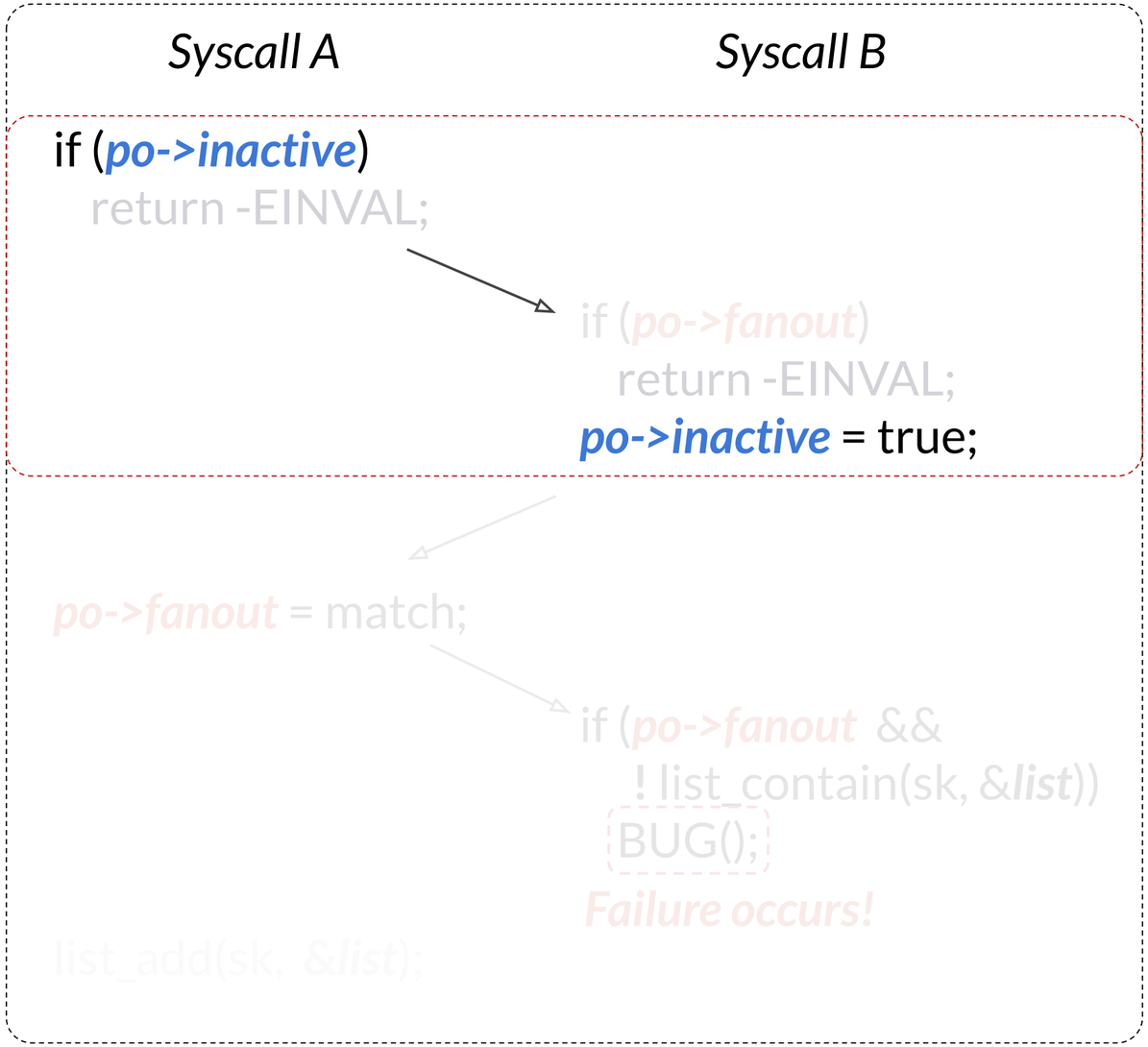
Initially ***po->fanout****: NULL* ***po->inactive****: false*

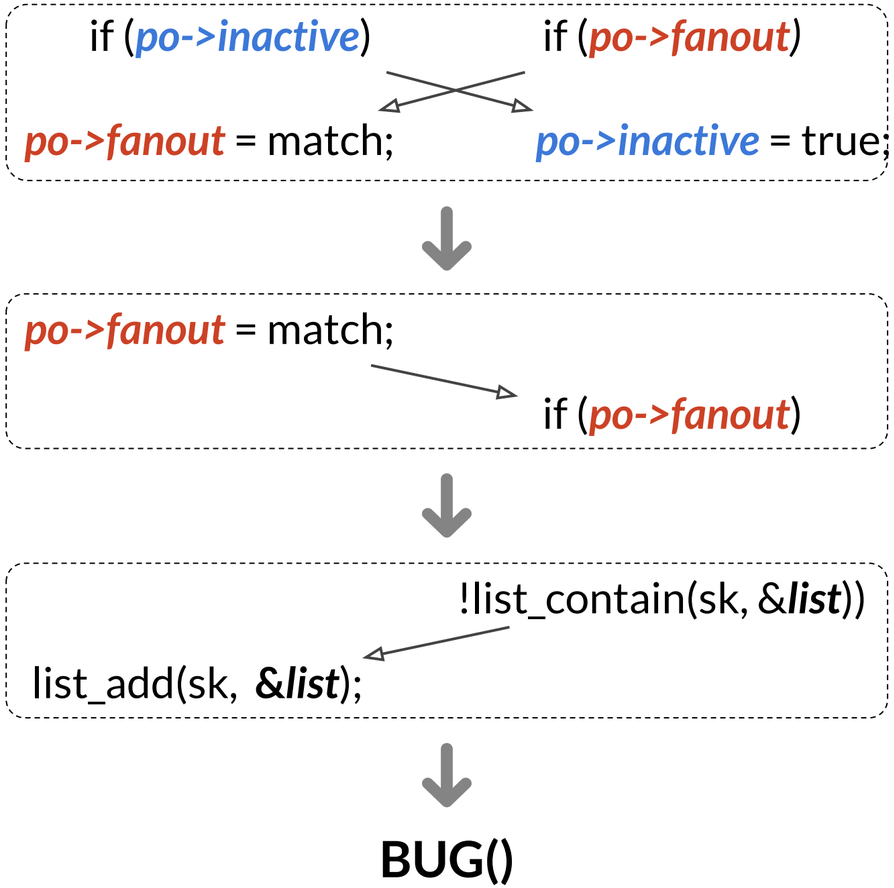




Constructing the causality chain

Initially ***po->fanout***: NULL ***po->inactive***: *false*





Constructing the causality chain

Initially ***po->fanout****: NULL* ***po->inactive****: false*

Syscall A

if (***po->inactive***)
  return -EINVAL;

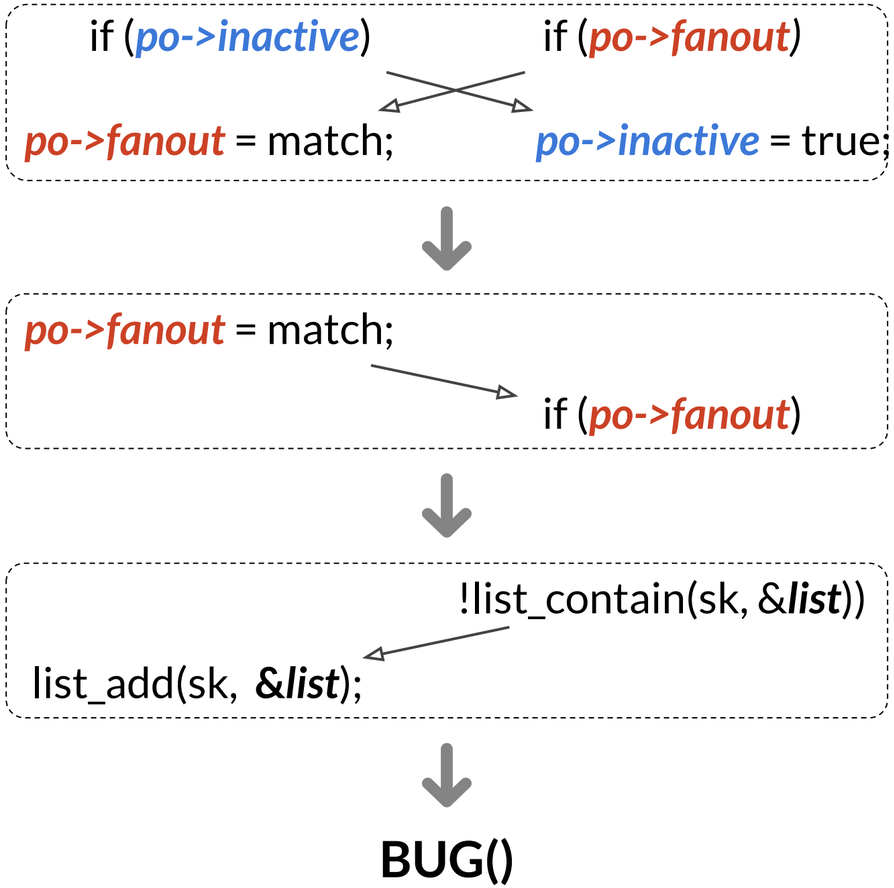
po->fanout = match;

list_add(sk, ***&list***);

Syscall B

if (***po->fanout***)
  return -EINVAL;
po->inactive = true;

if (***po->fanout*** &&
  ! list_contain(sk, &***list***))
  BUG();
Failure occurs!



Implementation of AITIA

Implementation of AITIA

- AITIA needs to know system calls and kernel background threads to run concurrently
  - Using ftrace and tracing system calls and kernel background thread invocations during the failed execution

Implementation of AITIA

- AITIA needs to know system calls and kernel background threads to run concurrently
  - Using ftrace and tracing system calls and kernel background thread invocations during the failed execution

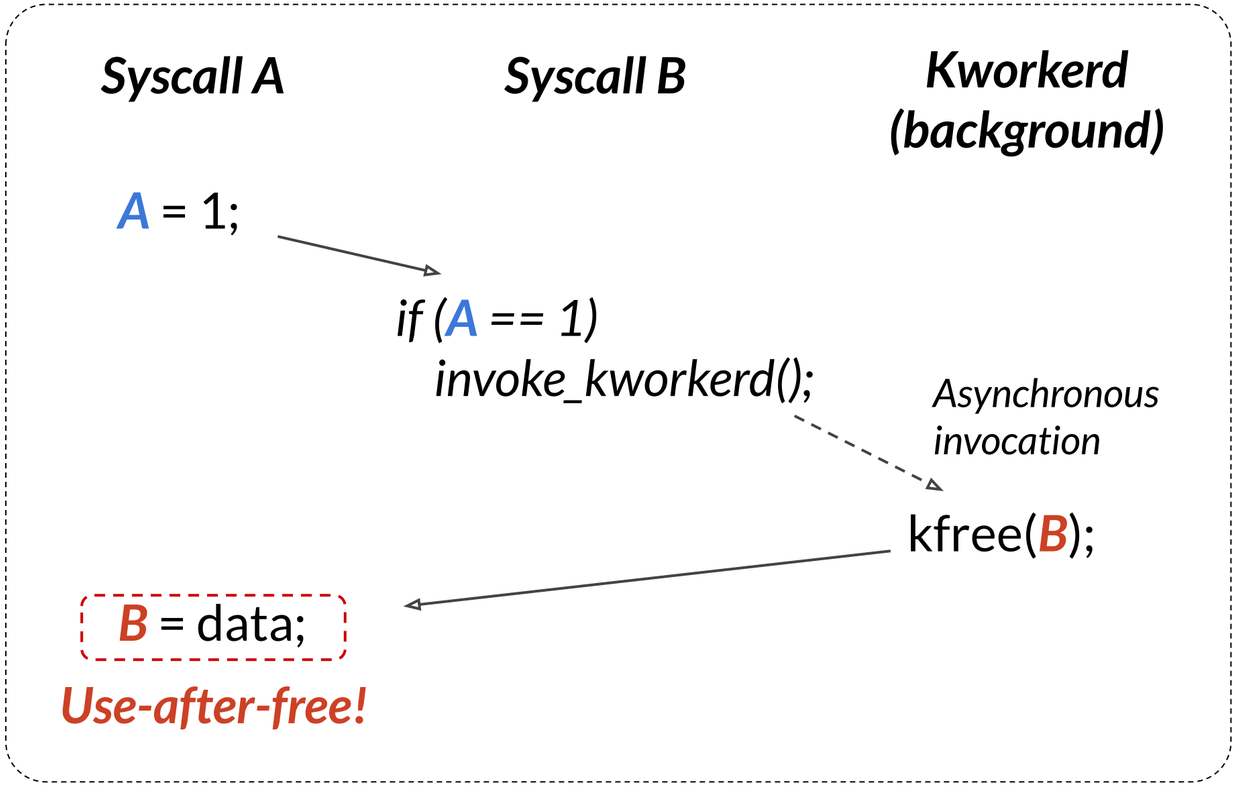
- The AITIA's hypervisor controls thread scheduling without modifying the kernel
  - Using ***breakpoints*** to suspend and resume the execution of threads
  - Using ***watchpoints*** to monitor memory accesses

Evaluation

- Evaluated AITIA with 22 kernel concurrency failures
  - 10 well-known concurrency failures from the CVE database
  - 12 concurrency failures found by Syzkaller
    - Including 6 unfixed concurrency bugs

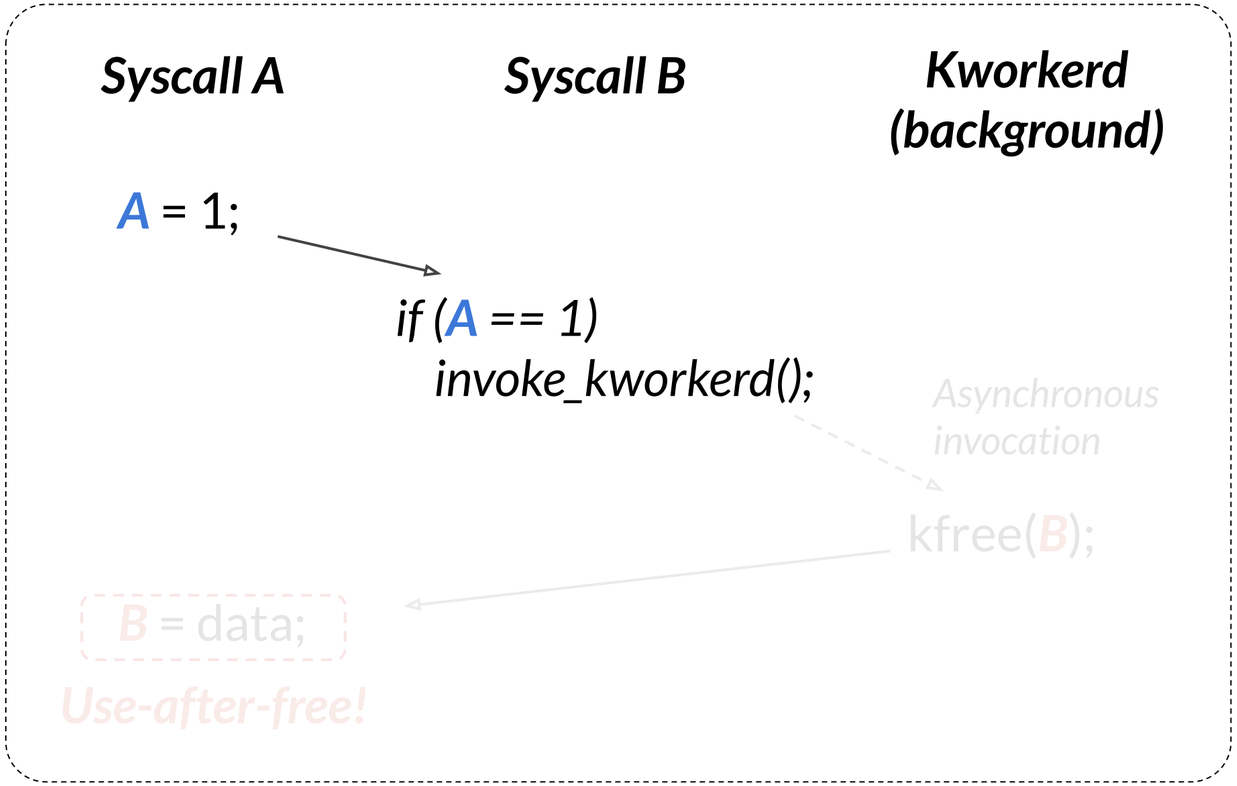
- AITIA can construct the causality chains for all 22 kernel concurrency failures
  - On average, AITIA takes 2059 seconds to generate a causality chain
  - For those unfixed concurrency failures, we were confirmed by kernel developers

Case study: use-after-free in the KVM submodule



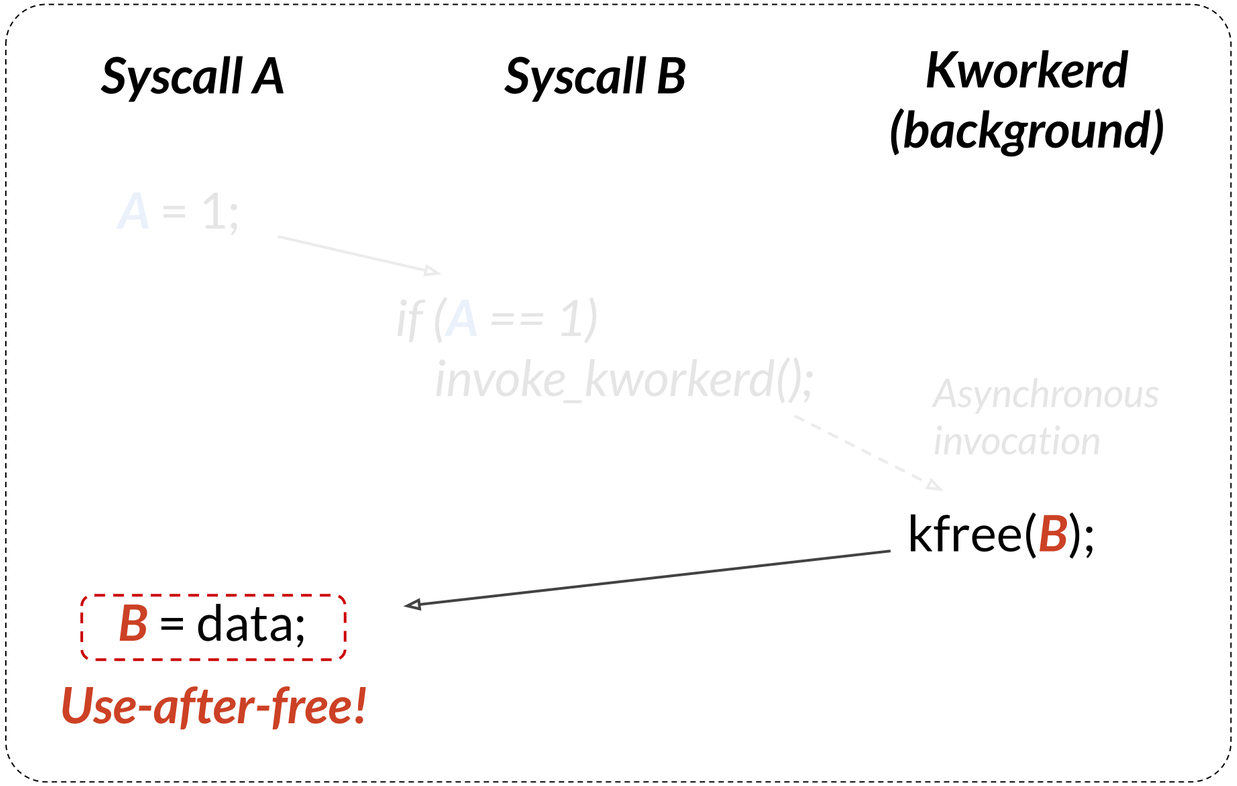
***Syscall A** and **kworkerd** should be executed atomically*

Case study: use-after-free in the KVM submodule



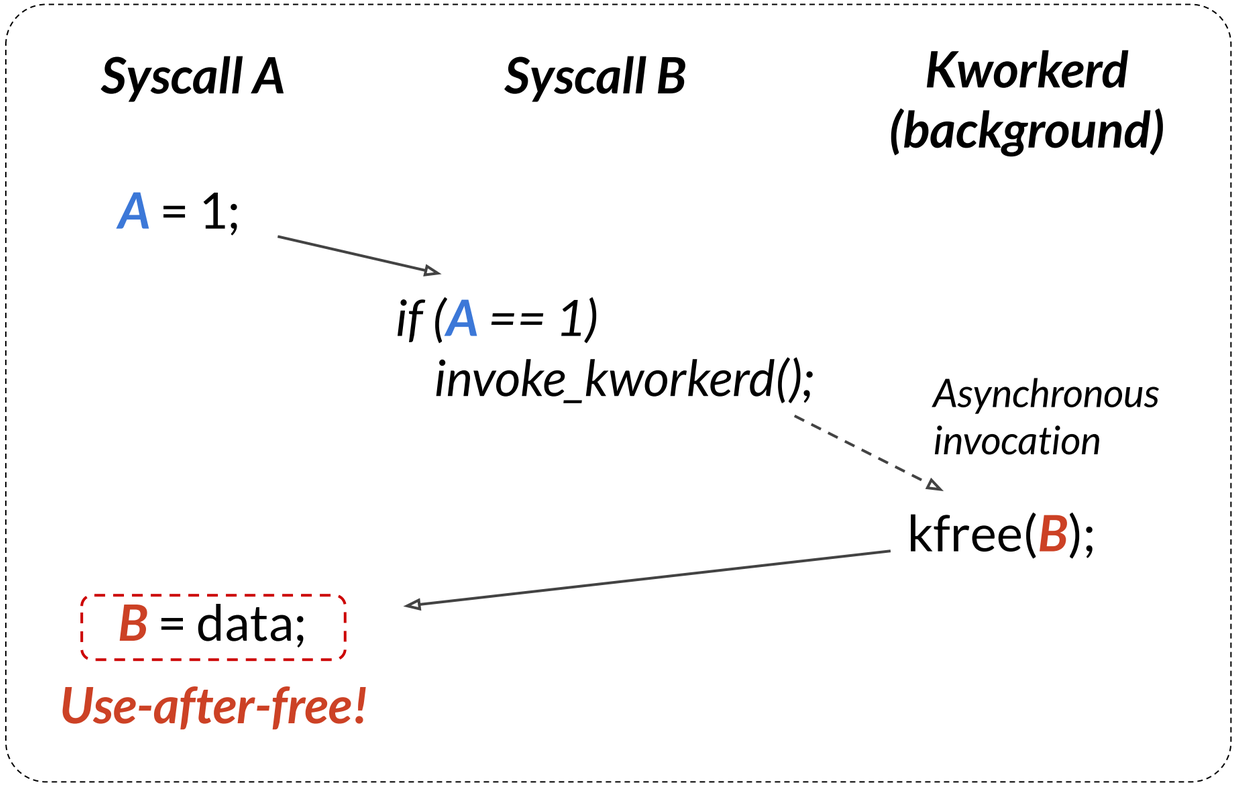
***Syscall A** and **kworkerd** should be executed atomically*

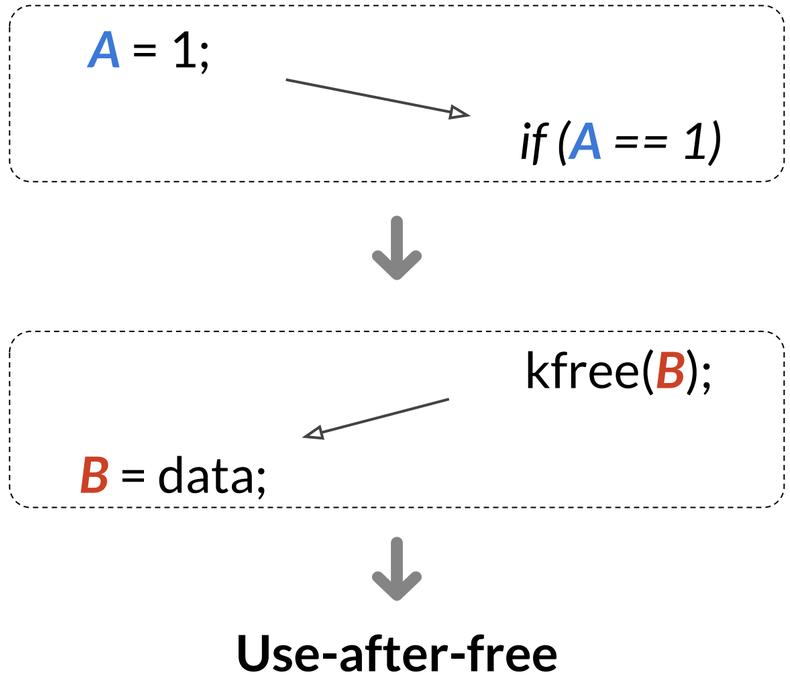
Case study: use-after-free in the KVM submodule



***Syscall A** and **kworkerd** should be executed atomically*

Case study: use-after-free in the KVM submodule





***Syscall A** and **kworkerd** should be executed atomically*

Conclusion

- ***Causality chain***
  - A root cause form of concurrency bugs explaining how a failure eventually occurred

- ***AITIA***, an automated root cause diagnosis tool for concurrency bugs
  - ***Step 1:*** Constructing a totally-ordered instruction sequence
  - ***Step 2:*** Flipping interleaving orders one at a time

- AITIA can diagnose 22 concurrency failures in the kernel
  - Including 6 unknown concurrency failures


Thank You!