Data-Race Detection for Interrupt-Driven Kernels

Nikita Chopra, Deepak D'Souza and Rekha Pai

Indian Institute of Science Bangalore

July 20, 2018

Overview

- Problem Definition
- Interrupt-Driven Programs
- Data-Races and Happens-Before Relation
- Analyzing FreeRTOS Kernel
- Conclusion

Problem Definition

Given an "interrupt-driven kernel program" detect data races in the program.

Interrupt-Driven Programs


```
Application:
                main()
                {
                     createTask(f1);
                         .
                         .
                     createTask(fn);
                     startScheduler();
                }

f1()            f2()                    fn()
{               {                       {
   ...              ...          ..         ...
}               }                       }
=============================================
Kernel:

startScheduler()    kAPI1()             kAPIn()
{                   {                   {
    ...                 ...     ..          ...          <==
}                   }                   }
```

Interrupt-Driven Programs

- finite number of threads
- `disableint-enableint`, `suspendsch-resumesch`, and synchronization flags
- "task" threads and "ISR" threads

Interrupt-Driven Programs


```
main:
 1. x := 0;
 2. y := 0;
 3. t := 0;
 4. create(t1);
 5. create(t2);
 6.

 t1:                    t2:
 7. x := x + 1;          9. disableint;
 8.                     10. y := t;
                        11. t := x;
                        12. if(t > 0) {
                        13.   y := y + 1;
                        14. }
                        15. else {
                        16.   t := t + 1;
                        17. }
                        18. enableint;
                        19.
```

Figure: Example program

Claim

Given an interrupt-driven program, we proposed a sound algorithm to detect data-races in the program.

- key insight is the notion of “disjoint blocks”

Data-Races

Motivation:

Give a definition of data-race based on the operational semantics of the class of interrupt-driven programs, that capture what a programmer typically tries to avoid.

Data-Races

Conflicting accesses:

> Two accesses are *conflicting accesses* if they are read/write accesses to the same variable, and at least one of them is a write.

Data-Races

Conflicting accesses:

> Two accesses are *conflicting accesses* if they are read/write accesses to the same variable, and at least one of them is a write.

Data-race:

> For classical concurrent programs, define a *race* as consecutive occurrences of conflicting accesses in an execution.

Data-Races


```
main:
 1. x := 0;
 2. y := 0;
 3. t := 0;
 4. create(t1);
 5. create(t2);
 6.

 t1:                    t2:
 7. x := x + 1;          9. disableint;
 8.                     10. y := t;
                        11. t := x;
                        12. if(t > 0) {
                        13.   y := y + 1;
                        14. }
                        15. else {
                        16.   t := t + 1;
                        17. }
                        18. enableint;
                        19.
```

Figure: Example program - race between Lines 7 and 11

Data-Races

Proposed Definition:

Two statements s and t in a program P are involved in a *data-race* if the following is true:

> Consider the program P' which is obtained from P by replacing the statement s with "`skip`; s; `skip`", and similarly for statement t. Then there is an execution of P' in which the two blocks containing s and t are involved in a high-level race.

Data-Races

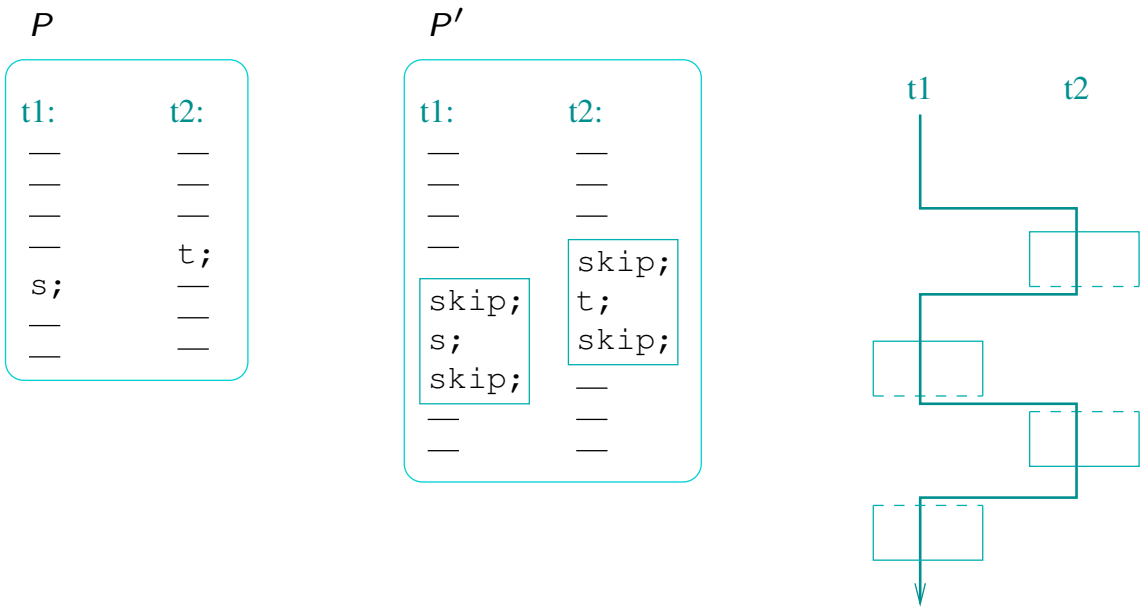


Figure: Illustrating the definition of a data-race on statements s and t. A program P, its transformation P', and an execution of P' in which the blocks overlap.

Happens-Before Relation

In the classical setting of lock-based synchronization,

> *happens-before* relation is a partial order on the instructions in an execution, that is the union of
> the *program-order* relation between two instructions in the same thread, and
> the *synchronizes-with* relation which relates a release of a lock in a thread to the next acquire of the same lock in another thread.

Happens-Before Relation

In the classical setting of lock-based synchronization,

> *happens-before* relation is a partial order on the instructions in an execution, that is the union of
> the *program-order* relation between two instructions in the same thread, and
> the *synchronizes-with* relation which relates a release of a lock in a thread to the next acquire of the same lock in another thread.

How does one define *synchronizes-with* relation in interrupt-driven programs?

Happens-Before Relation

Disjoint blocks:

Disjoint blocks are syntactically identifiable pairs of code blocks in different threads, which are guaranteed by the execution semantics of the class of programs never to *overlap* in any execution of a program.

Happens-Before Relation

Proposed Definitions

Synchronizes-with relation:

> for every pair (A, B) of disjoint blocks in the program, the end of block A *synchronizes-with* the beginning of the succeeding occurrence of block B in the execution; and vice-versa.

Happens-Before Relation

Proposed Definitions

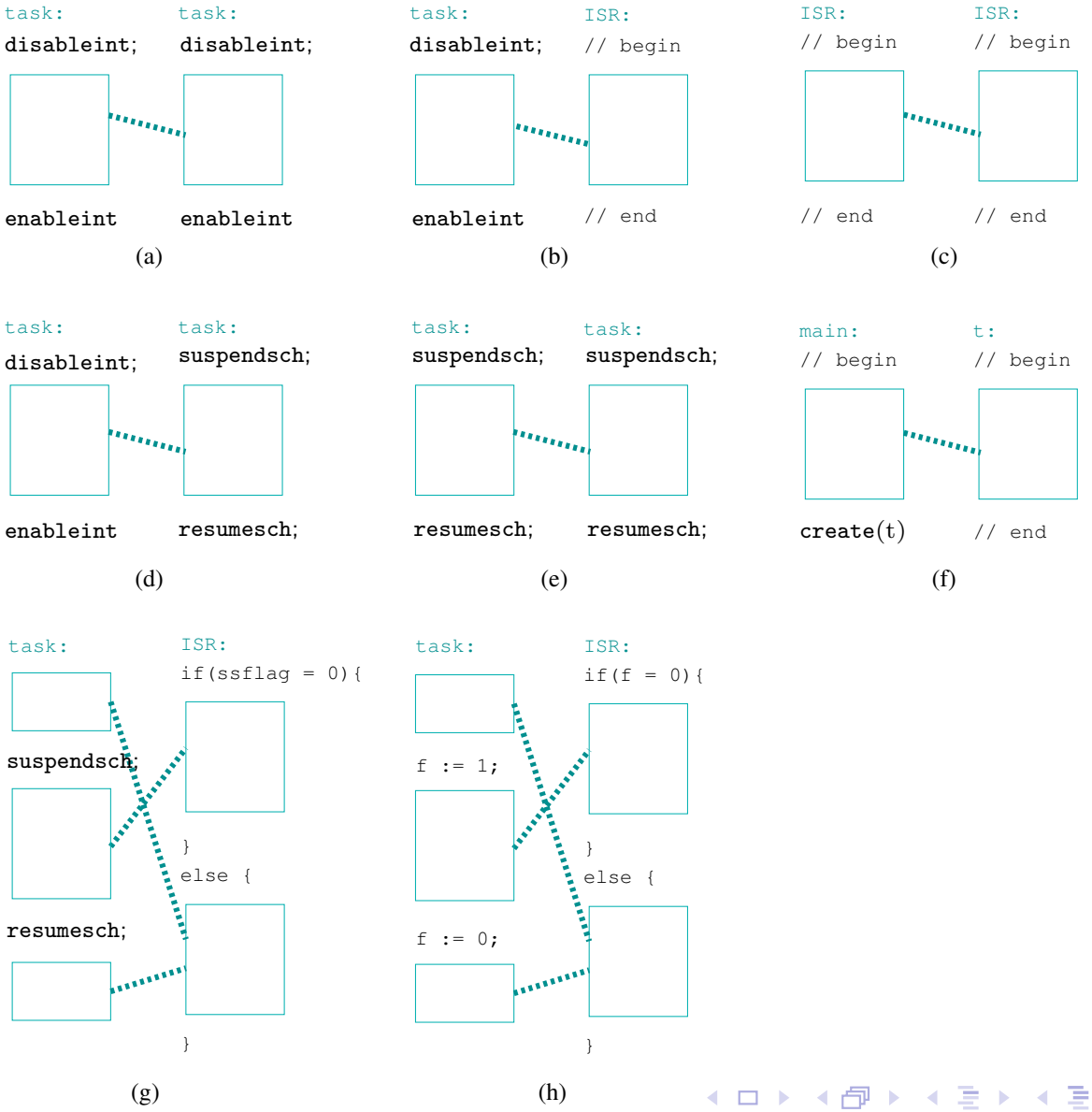
Synchronizes-with relation:

> for every pair (A, B) of disjoint blocks in the program, the end of block A *synchronizes-with* the beginning of the succeeding occurrence of block B in the execution; and vice-versa.

Happens-Before relation:

> defined, as before, in terms of the program order and the synchronizes-with order.

Happens-Before Relation



Analyzing FreeRTOS Kernel Library

Table: Potential Races

	Variable	Functions	Remark
1	`pcQueueName`	vQueueDelete(w) vQueueDelete(w)	Two tasks attempting to write into `pcQueueName` while unregistering a queue in vQueueDelete
2	`pcQueueName`	vQueueDelete(w) vQueueAddToRegistry(r)	A read of `pcQueueName` in vQueueAddToRegistry is attempted while it is written into during unregistering a queue in vQueueDelete
3	`pcQueueName`	vQueueDelete(w) vQueueAddToRegistry(w)	The queue name is reset while unregistering a queue in vQueueDelete and a queue name is written by vQueueAddToRegistry for a new queue
4	`pcQueueName`	vQueueDelete(w) pcQueueGetName(r)	A read of `pcQueueName` in pcQueueGetName is attempted while it is written into during unregistering a queue in vQueueDelete
5	`pcQueueName`	vQueueAddToRegistry(r) vQueueAddToRegistry(w)	A read of `pcQueueName` in vQueueAddToRegistry is attempted while it is set in vQueueAddToRegistry by another task
6	`pcQueueName`	vQueueAddToRegistry(w) vQueueAddToRegistry(w)	Simultaneous writes to `pcQueueName` is attempted
7	`pcQueueName`	vQueueAddToRegistry(w) pcQueueGetName(r)	The write in vQueueAddToRegistry happens simultaneously with read in pcQueueGetName
8	`xHandle`	vQueueDelete(r) vQueueDelete(w)	An attempt to read the `xHandle` while it is written into simultaneously
9	`xHandle`	vQueueDelete(r) vQueueAddToRegistry(w)	The read in vQueueDelete during unregistering a queue happens simultaneously with write in vQueueAddToRegistry
10	`xHandle`	vQueueDelete(w) vQueueAddToRegistry(w)	A task attempting to delete a queue while another tries to register a new queue simultaneously
11	`xHandle`	vQueueDelete(w) pcQueueGetName(r)	A task attempting to delete a queue while another tries to get the queue name
12	`xHandle`	vQueueAddToRegistry(w) vQueueAddToRegistry(w)	Two tasks attempting to register queues simultaneously
13	`xHandle`	vQueueAddToRegistry(w) pcQueueGetName(r)	A task attempts to read queue name while another tries to register a new queue
14	`xHandle`	vQueueDelete(w) vQueueDelete(w)	Two tasks attempting to delete a queue

Conclusion

- proposed definition of data-races
- proposed definition for synchronizes-with relation, based on disjoint blocks
- proposed a sound algorithm to detect data-races in the program
- detected 14 real races in the FreeRTOS kernel library