Applying Formal Verification to Microkernel IPC at Meta

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter O'Hearn, and Francesco Zappa Nardelli

Certified Programs and Proofs
January 17, 2022

Introduction

- Complexity of modern software is growing extraordinarily fast – *how do we know if it works?*
- Verification toolchains are improving too – *are they ready for industry?*
- Project goal – answer the following questions:
  - Can formal verification be successfully applied in a *"move fast"* industrial setting?
  - What benefits can we achieve by using formal verification?

The XROS Operating System

Facebook is building an operating system so it can ditch Android

Josh Constine @joshconstine / 11:15 AM EST • December 19, 2019 Comment

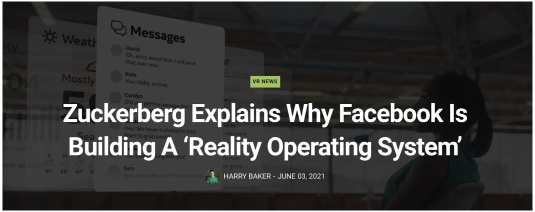
TECH \ FACEBOOK

Facebook is working on its own OS that could reduce its reliance on Android

Led by a co-author of Windows NT

By Jon Porter | @JonPorty | Dec 19, 2019, 12:54pm EST

SHARE



The XROS Operating System

- The success of the Metaverse relies on a new wave of wearable devices
- These devices have stringent power constraints
- XROS is a microkernel; inter-process communication (IPC) is the most crucial part of the OS
- OS components exchange messages via concurrent, non-blocking, multi-producer, multi-consumer queues
- These are our target for verification

Motivation

- Can formal verification be applied in industry?
- XROS IPC is a good fit
  - Easy to specify – we know exactly what it is supposed to do
  - Self-contained functionality
  - High leverage – entire OS relies on correctness of IPC
  - Algorithm is unlikely to change

Strategy

- Use off the shelf proof environment based on Concurrent Separation Logic (Coq + Iris)
- First verify the *algorithm*, not the actual C code
- Use our most valuable resource (human brain power) on the hardest problem (non-blocking concurrency)

Algorithm vs Code

The Algorithm

- 24 lines of pseudocode
- Simple and readable
- Only contains core logic
- Unlikely to change
- Changes require update to proof

The Code

- Several thousand lines of C
- Maximally performant
- Contains complex, low-level operations
- Changes frequently
- No updates to proof

Correspondence is certified by inspection of the OS engineers

Results

- We proved the correctness of two different queues (Generic Queue and Ports Queue)
- We found algorithmic simplifications (elimination of an atomic load and a conditional check)
- We found a bug in real OS device driver code

Primer on Concurrent Separation Logic

Hoare Logic

- Use pre- and post- conditions (Hoare Triples) the specify program behavior

$$\{P\}\ \mathbb{C}\ \{Q\}$$

- Triples are proven using a program logic
- For example, the following triple is valid

$$\{x \text{ is even}\}\ y := x + 2\ \{y \text{ is even}\}$$

Separation Logic

- A logic for reasoning about resources
- The *points-to* predicate specifies knowledge about a heap location

$$x \mapsto n$$

- The *separating conjunction* allows for local reasoning

$$P * Q$$

- Here, P and Q can only reference disjoint *heaplets*
- In the following example, it is impossible for x and y to alias each other

$$(x \mapsto n) * (y \mapsto m)$$

Hoare Logic – Concurrent Programming

- Specifications are more complicated in concurrent code
- For example, the following triple is no longer valid

$$\{\exists n, (x \mapsto n) * (n \text{ is even})\}\ y := !x + 2\ \{y \text{ is even}\}$$

- The value of x could be changed by another thread before we read it

Invariants

- Invariants are persistent assertions that are *always true*
- The following triple is valid:

$$\boxed{\exists n, (x \mapsto n) * (n \text{ is even})} \vdash \{\top\}\ y :=\, !x + 2\ \{y \text{ is even}\}$$

- Even so, the following triple is not valid for any pre- or post-condition

$$\boxed{\exists n, (x \mapsto n) * (n \text{ is even})} \vdash \{?\}\ \texttt{faa}(x, 1);\ \texttt{faa}(x, 1)\ \{?\}$$

- The invariant holds knowledge about the *physical state*

Specifying Concurrent Data Structures

- $\mathit{QueueContent}(q, \ell)$ – The *physical* queue q contains the elements in the *logical* list ℓ
- $\mathit{QueueInv}(q)$ – The physical structure of q is valid

$$\boxed{\mathit{QueueInv}(q)} \vdash \begin{array}{l} \{\mathit{QueueContent}(q, \ell)\} \\ \quad \mathtt{enqueue}(q, x) \\ \{\mathit{QueueContent}(q, \ell +\!\!+ x)\} \end{array}$$

Proof Sketch

The XROS Generic Queue

- The generic queue is used in the XROS kernel to exchange messages between threads
- Based on a fixed-size ring buffer
- All operations are non-blocking
- Enqueues and dequeues happen in two phases

The Code

```
start_enqueue(q):
  while true:
    pc    = atomic_load(q.pc)
    i, k  = pc / q.cap, pc % q.cap
    ik    = atomic_load(q.itr[k])
    ok    = atomic_load(q.own[k])
    if ik == i-1 && ok == PROD :
      if CAS(q.pc, pc, pc+1) :
        return (k, &q.dat[k])
```

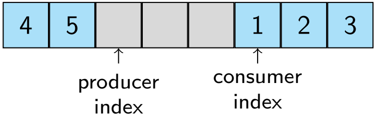
```
start_dequeue(q):
  while true:
    cc    = atomic_load(q.cc)
    i, k  = cc / cap, cc % cap
    ok    = atomic_load(q.own[k])
    ik    = atomic_load(q.itr[k])
    if ik == i  && ok == CONS:
      if CAS(q.cc, cc, cc+1) :
        return (k, &q.dat[k])
```

```
mark_ready(q, k):
  atomic_store(q.own[k], CONS)
  atomic_incr(q.itr[k])
```

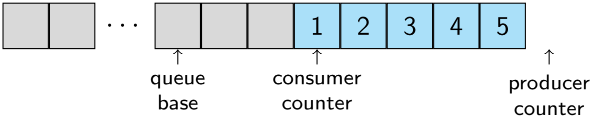
```
mark_free(q, k):
  atomic_store(q.own[k], PROD)
```

The Endless Ribbon

- Physically, the queue data is stored in a ring buffer



- Reasoning about modular arithmetic is hard, so logically we unfold the ring into an endless ribbon



The Ribbon State

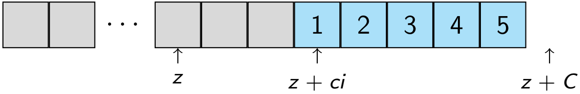
- The queue invariant tracks a mapping from ribbon locations to logical states

$$
\begin{aligned}
\textit{state} ::= \; & \text{Empty} \\
\mid \; & \text{ClaimEnq} \\
\mid \; & \text{OwnerSet(v)} \\
\mid \; & \text{Ready(v)} \\
\mid \; & \text{ClaimDeq} \\
\mid \; & \text{Free}
\end{aligned}
$$

- The logical state says *more* about a cell than just its physical value

Queue Invariant

$$
\begin{aligned}
&\mathit{QueueInv}(\langle C, \ell_{pc}, \ell_{cc}, \ell_d, \ell_o, \ell_i \rangle) = \exists z, ci, \mathbf{r}. \\
&\quad (\ell_{pc} \mapsto z + C) \ * \\
&\quad (\ell_{cc} \mapsto z + ci) \ * \\
&\quad \mathop{\Huge\ast}_{z \,\leq\, i \,<\, z+C} (\ell_d +_l (i \bmod C) \mapsto \mathit{Data}(r_i)) \ * \\
&\quad \ldots
\end{aligned}
$$



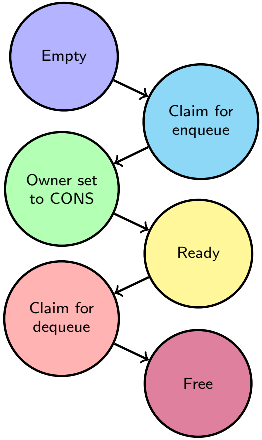
Proof Sketch – Logical States

```
start_enqueue(q):
  while true:
    pc    = atomic_load(q.pc)
    i, k = pc / q.cap, pc % q.cap
    ik    = atomic_load(q.itr[k])
    ok    = atomic_load(q.own[k])
    if ik == i-1 && ok == PROD :
      if CAS(q.pc, pc, pc+1) :
        return (k, &q.dat[k])
```

```
mark_ready(q, k):
   atomic_store(q.own[k], CONS)
   atomic_incr(q.itr[k])
```

```
start_dequeue(q):
  while true:
    cc    = atomic_load(q.cc)
    i, k = cc / cap, cc % cap
    ok = atomic_load(q.own[k])
    ik    = atomic_load(q.itr[k])
    if ik == i  && ok == CONS:
      if CAS(q.cc, cc, cc+1) :
        return (k, &q.dat[k])
```

```
mark_free(q, k):
   atomic_store(q.own[k], PROD)
```



Conclusion

- This project is evidence that it is practical and useful to apply formal methods in industry
- Proofs were completed quickly, especially after ramp up
- Reception was good, especially after simplifications and bugs were found
- Concurrent Separation Logic makes you a better programmer!

Thank You!