Mechanized Verification of Preemptive OS Kernels

Xinyu Feng
University of Science and Technology of China
Why OS Kernel Verification?

Computer Systems
Why OS Kernel Verification?

Correctness of OS is crucial for safety and security of the whole system.
Why OS Kernel Verification?

• Fundamental, but also simpler to verify! (comparing to applications)
  • Less domain knowledge required
    – every programmer knows OS
  • Stable specifications
  • Slow evolution
  • Specs validated by application-level verification
OS Kernel Verification: Challenges

• Low-level programs
  • C + inline assembly, interrupts, task management, ...

• Larger code base (than algorithm verification)

• Code at different abstraction layers
  • E.g., threads vs. schedulers

• Involves both libraries (sys. calls) and runtime (scheduler)
  • What is a proper specification?

• Rich concurrency
  • Multi-tasking, multi-core, interrupts
Preemption and nested interrupts

Preemptions and multi-level interrupts are crucial for real-time systems. They also make system highly concurrent and complex.

Not fully supported in existing work
Concurrency & Preemption in Previous work

• seL4 [Klein et al. 2009 ...]
  • Mostly sequential
  • Limited support of interrupts at fixed program points

• Verisoft [Rieden et al. 2007 ...]
  • Kernel is sequential

• Verve [Yang & Hawblitzel. PLDI 2010]
  • Allows preemption, but no nested interrupts
  • Mostly about safety, limited functionality verification

• CertiKOS [Gu et al. 2015, Chen et al. 2016, Gu et al. 2016]
  • Evolving: sequential $\rightarrow$ limited interrupts $\rightarrow$ multicore
  • Still no preemption
Concurrency & Preemption in Previous work (2)

- **eChronos OS** [Andronick et al. 2015, 2016]
  - Supports preemption and nested interrupts
  - But verification at the model level only
  - Verifies scheduling invariants, no API correctness
Challenges for Verifying Preemptive OS Kernels

• Verifying concurrent programs is difficult
  • Non-deterministic interleaving

[Brookes & O’Hearn 2016], courtesy of Ilya Sergey
Challenges for Verifying Preemptive OS Kernels

• Verifying concurrent programs is difficult

• Verifying concurrent kernels is even more challenging
  • More difficult to establish refinement with concurrency
    • Theories not fully developed until recently [Turon et al. POPL’13, ICFP’13] [Liang et al. PLDI’13, CSL-LICS’14]

• Kernel-level preemption can be more complex than multi-tasking/multi-processor concurrency
Kernel-level preemption can be more complex than multi-tasking/multi-processor concurrency

Interrupt management is now a verification target: lower abstraction layer and non-uniform concurrency model

More low-level details:
  e.g., can context switch only when there are no nested interrupts
This talk

- Verification framework for preemptive OS kernels
  - Refinement reasoning about concurrent kernels
  - Multi-Level nested interrupts and preemption

- Verification of key modules of a commercial OS kernel \( \mu C/OS-II \) in Coq

The first mechanized verification of a commercial preemptive OS kernel.

[Xu et al. CAV’16]
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying μC/OS-II
OS Correctness

• **OS provides abstraction for programmers**
  • Hides details of the underlying hardware
  • Provides an abstract programming model

• **OS Correctness** : *refinement* between high-level abstraction and low-level concrete implementation
OS Correctness

High-Level Language

Applications

Low-Level Language

C + Abstract primitives

C + Assembly

High-Level Abstract Primitives

Low-Level Concrete Implementations
Refinement

Applications

High-Level Abstract Primitive

Low-Level Concrete Kernel Impl.

System API

System Call

IU

System Call
Contextual Refinement

For all applications

System Call

High-Level Abstract Primitive

Low-Level Concrete Kernel Impl.

System API
Contextual Refinement as OS API Correctness

\[ O \subseteq_{\text{ctxt}} S \text{ iff } \forall A. \ ObsBeh(A[O]) \subseteq ObsBeh(A[S]) \]

The set of observable behaviors

With some assumptions about A

A: Application  O: Concrete Impl. of OS API  S: Abstract Prim.
Contextual Refinement as OS API Correctness

But OS correctness is more than API correctness:

Correctness of runtime services, e.g., scheduler
(not exported as an API)

Whole system properties,
e.g., isolation and security, real-time properties, ...

*Cannot be specified as abstract API primitives!*

*How to specify their correctness?*
Runtime services and Sys. Props

**Runtime**: specified as part of the high-level language semantics (e.g., scheduling)

Verified through refinement
Runtime services and Sys. Props

Whole system properties: specified as trace properties of all apps (with high-level views)

Proved at high-level only, propagated to low-level through contextual refinement!
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying μC/OS-II
Our Verification Framework

A. Modeling of OS Kernels

B. Refinement-Based Verification

- CSL-Style Refinement-Based Program Logic
- Contextual Refinement

C. Coq Tactics

Relational Assertion Entailment

Verification Condition Generator

Domain-Specific Solvers

...
OS Correctness

High-Level Language

Applications

High-Level Language

IU

Low-Level Language

High-Level Abstract Primitives

Low-Level Concrete Implementations
Our Verification Framework

A. Modeling of OS Kernels

B. Refinement-Based Verification

- CSL-Style Refinement-Based Program Logic
- Contextual Refinement
- Low-Level Operational Semantics with Context Switch and Interrupts
- High-Level Operational Semantics with Configurable Schedulers

C. Coq Tactics

Relational Assertion Entailment
Verification Condition Generator
Domain-Specific Solvers

...
The Low-Level Language

\[ L ::= C \mid Pr \mid L;L \mid \cdots \]

\[ C ::= \text{while} \ e \{ C \} \mid \text{if} \ e \{ C_1 \} \{ C_2 \} \mid f(e) \mid e=e \mid \cdots \]

\[ e ::= &e \mid *e \mid e[e] \mid e.id \mid \cdots \]
The Low-Level Language

OSCtxSw:

```
pushfl
pushal  # Save current task's context
mov 0STCBCur,%ebx
mov %esp,(%ebx)  # 0STCBCur->OSTCBStkPtr = ESP
call OSTaskSwHook  # Call user defined task switch hook
mov OSTCBHighRdy,%eax  # OSTCBCur <= OSTCBHighRdy
mov %eax,OSTCBCur
mov OSPrioHighRdy,%al  # OSPrioCur <= OSPrioHighRdy
mov %al,OSPrioCur
mov OSTCBHighRdy,%ebx  # ESP = OSTCBHighRdy->OSTCBStkPtr
mov (%ebx),%esp
popal
```

`Pr ::= encrt | excrt | switch | ...`

Explicit interrupts management and context switch

```c
#define OS_ENTER_CRITICAL() __asm__ ("pushf \n\t cli") /* Disable interrupts*/
#define OS_EXIT_CRITICAL() __asm__ ("popf")     /* Enable interrupts*/
```
Semantics

Small-step, even for expressions:
Try to be faithful to the granularity of machine-code

Interrupt handler:
\[ y = y + 2 \]

Semantics similar to CompCertTSO [Sevcik et al. 2011]
(but is interleaving semantics instead of TSO model)
Our Verification Framework

A. Modeling of OS Kernels

B. Refinement-Based Verification

C. Coq Tactics

Relational Assertion Entailment

Verification Condition Generator

Domain-Specific Solvers

CSL-Style Refinement-Based Program Logic

Contextual Refinement

High-Level Language

High-Level Operational Semantics with Configurable Schedulers

Low-Level Operational Semantics with Context Switch and Interrupts

High-Level Spec. Language

C Subset

Assembly Primitives

Low-Level Language

Relevance of assertion entailment: relational refinement

Refinement-Based Verification Framework
High-Level Language

\[
H ::= C \mid S \mid H;H \mid \ldots
\]

\[
S ::= \text{sched} \mid \gamma(v) \mid S;S \mid S+S \mid \ldots
\]

- C subset
- High-level API specification language
- explicit scheduling points
High-Level Language

\[ H ::= C \mid S \mid H;H \mid \ldots \]

\[ S ::= \text{sched} \mid \gamma(v) \mid S;S \mid S+S \mid \ldots \]

abstract \textbf{atomic} transitions
(over the abstract kernel states)
void OSTimeDly (INT16U ticks) {
    if (ticks > 0) {
        OS_ENTER_CRITICAL();
        // Suspend the current thread, and remove it from the READY thread queue
        __asm__ ("pushf \n\t cli"); /*Disable interrupts*/
        OS_ENTER_CRITICAL();
        ......
        OS_EXIT_CRITICAL();
        __asm__ ("popf"); /*Enable interrupts*/
        OS_Sched();
        return;
    }
}

Suspend the current thread, and remove it from the READY thread queue

call scheduler
Low-level Code VS. High-Level Spec

void OSTimeDly (INT16U ticks) {
    if (ticks > 0) {
        OS_ENTER_CRITICAL();
        ......
        OS_EXIT_CRITICAL();
        OS_Sched();
    }
    return;
}
System Model

- **Low-level impl. O**: $(\eta_a, \theta, \eta_i)$
  - $\eta_a$: API implementations
  - $\theta$: Interrupt handlers
  - $\eta_i$: Internal functions

- **High-level spec. S**: $(\varphi, \varepsilon, \chi)$
  - $\varphi$: API specs. (high-level primitives for APIs)
  - $\varepsilon$: Abstract events (high-level primitives for int. handlers)
  - $\chi$: Abstract scheduler
    - Scheduling policy can be customized by instantiating $\chi$
System Model

Runtime services and Sys. Props

- **Low-level impl.**: \((\eta_a, \theta, \eta_i)\)
  - \(\eta_a\): API implementations
  - \(\theta\): Interrupt handlers
  - \(\eta_i\): Internal functions

- **High-level spec.**: \((\phi, \epsilon, \chi)\)
  - \(\phi\): API specs. (high-level primitives for APIs)
  - \(\epsilon\): Abstract events (high-level primitives for int. handlers)
  - \(\chi\): Abstract scheduler
    - Scheduling policy can be customized by instantiating \(\chi\)
    - Shows abstractions for runtime
System Model

- Low-level impl. O: \((\eta_a, \theta, \eta_i)\)
  - \(\eta_a\): API implementations
  - \(\theta\): Interrupt handlers
  - \(\eta_i\): Internal functions

- High-level spec. S: \((\varphi, \varepsilon, \chi)\)
  - \(\varphi\): API specs. (high-level primitives for APIs)
  - \(\varepsilon\): Abstract events (high-level primitives for int. handlers)
  - \(\chi\): Abstract scheduler
    - Scheduling policy can be customized by instantiating \(\chi\)
    - Shows abstractions for runtime

Verification goal:
\((\eta_a, \theta, \eta_i) \subseteq_{\text{ctxt}} (\varphi, \varepsilon, \chi)\)
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
    • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying μC/OS-II
Program Logic for Refinement and Multi-Level Interrupts

• Relational program logic for simulation/refinements
  [Liang et al. PLDI’13, CSL-LICS’14]

• Ownership-Transfer semantics for interrupts
  [Feng et al. PLDI’08]

• Combining the two: CSL-R for refinement reasoning with multi-level interrupts
Refinement Verification via Simulation

High ($A[S]$):


$S$
Simulation with Interrupts & Multitasking

High ($A[S]$):


Interrupt handler:

Another task:

How to do compositional verification?
Simulation with Interrupts & Multitasking

High (A[S]):

Low (A[O]):

Interrupt handler:

Another task:

Use invariant “I” to specify non-deterministic interference
Simulation with Interrupts & Multitasking

High ($A[S]$):


Adapted from RGSim [Liang et al. POPL’12] and the relational program logic [Liang et al. PLDI’13, CSL-LICS’14]
Program Logic for Simulation

High-Level abstract primitive \( S \) \( \rightarrow \) High-Level abstract states

Low-Level concrete code \( C \) \( \rightarrow \) Low-Level concrete states

\( x \rightarrow \ldots \rightarrow z \)
Program Logic for Simulation

- Judgement

\[ I \vdash \{ p^{*} [\mid S \mid] \} \ C \{ q^{*} [\mid \text{end} \mid] \} \]

Remaining high-level code that needs to be refined
Program Logic for Simulation

- Judgement

\[ I \mid \{ p \ast [|S|] \} C \{ q \ast [|\text{end}|] \} \]

No remaining high-level code (refinement is done)
Program Logic for Simulation

- Judgement

\[ I \models \{ p^* [ |S| ] \} C \{ q^* [ |\text{end}| ] \} \]

Relational assertions for pre-/post-condition

<table>
<thead>
<tr>
<th>High-Level abstract primitive</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Level concrete code</td>
<td>( C )</td>
</tr>
</tbody>
</table>

- \( S \) represents high-level abstract primitive states.
- \( C \) represents low-level concrete code states.

- \( I \) models initial conditions.
- \( \models \) represents satisfaction.
- \( \{ \} \) represents guard conditions.
- \( \succcurlyeq \) represents transition relations.
Program Logic for Simulation

- Judgement

Relational Invariants

\[ I \models \{ p \cdot [S] \} \subseteq \{ q \cdot [\text{end}] \} \]

<table>
<thead>
<tr>
<th>High-Level abstract primitive</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Level concrete code</td>
<td>C</td>
</tr>
</tbody>
</table>

High-Level abstract states

Low-Level concrete states

Relational Invariants
Soundness

If

\[ I \models \{ p^* [ |S| ] \} \preceq \{ q^* [ |\text{end}| ] \} \]

then $C$ is simulated by $S$, ...
An Example

```c
void Add() {
    OS_ENTER_CRITICAL();
    Count ++;
    OS_EXIT_CRITICAL();
}
```

$I ::= \exists v. \text{Count} \rightarrow v \quad \ast \quad \text{CNT}=v$
An Example

```c
OS_ENTER_CRITICAL();

Count ++;

{
  [< CNT++ >] * I
}

OS_EXIT_CRITICAL();
```
An Example

OS_ENTER_CRITICAL();

Count ++ ;

{ [<CNT++ >] }

The code refines <CNT++>

OS_EXIT_CRITICAL();

{ [<end>] }
An Example

```c
{ [<CNT++>] }

OS_ENTER_CRITICAL();

{ [<CNT++>] } *
Count → v *
CNT = v

Count ++ ;

{ [<CNT++>] } *
Count → v + 1 *
CNT = v

{ [<end>] } *
Count → v + 1 *
CNT = v + 1

Execute high-level code

OS_EXIT_CRITICAL();

{ [<end>] }
```
An Example

\{ [<\text{CNT++}>] \}  
\text{OS\_ENTER\_CRITICAL();}

\text{Count ++ ;}

\{ [<\text{CNT++}>] * \text{Count} \rightarrow v+1 * \text{CNT=v}\}
\rightarrow
\{ [<\text{end}>] * \text{Count} \rightarrow v+1 * \text{CNT=v+1}\}

\text{Execute high-level code}

\text{OS\_EXIT\_CRITICAL();}

\{ [<\text{end}>] \}

\text{Abstract consequence rule:}
\begin{align*}
p^*[S] & \implies r^*[S'] & & \vdash \{ r^*[S'] \} C \{ q \} \\
\text{}\qquad \vdash \{ p^*[S] \} C \{ q \}
\end{align*}

\text{p \implies q \quad iff \quad } \forall (\sigma, \Sigma, S) \models p, \quad \\
\exists (\Sigma', S'). (\Sigma, S) \rightarrow^* (\Sigma', S') \quad \\
\land (\sigma, \Sigma', S') \models q,
An Example

```
{ [<CNT++>] }
OS_ENTER_CRITICAL();
{ [<CNT++>] * I }

Count ++ ;

{ [<end>] * I }
OS_EXIT_CRITICAL();
{ [<end>] }
```

Ownership transfer

Interrupt reasoning

Ownership transfer
Interrupt Reasoning

Program invariant  [O'Hearn CONCUR’04]

There is always a partition of resource among concurrent entities, and each concurrent entity only accesses its own part.

But note:

The partition is dynamic: ownership of resource can be dynamically transferred.

Interrupt operations can be modeled as operations that trigger resource ownership transfer. [Feng et al. PLDI’08]
Ownership-Transfer Semantics for Single-Level Interrupt

Resource

IF = 1

IF = 0

B1

B0

Task

Handler 0

Interrupt enabled

Interrupt disabled

Resource

{p} cli {p * I0}

{p * I0} sti {p}
Memory Model for Multi-Level Interrupts

• Higher-priority handler has priority to select its required resource

• N blocks are assigned to N interrupt handlers

• Each well-formed resource block is specified by a resource invariant
Ownership-Transfer Semantics for Multi-Level Interrupts

[8259A interrupt controller]
Inference Rules for Interrupt Operations

\[ I \vdash \{ [\text{ISR}, 1, k] \ast p \} \textbf{cli} \{ [\text{ISR}, 0,k] \ast p \ast \} I[0\ldots k-1] \]
Top Rule for Proving \( \mathcal{O} \subseteq_{\text{ctxt}} \mathcal{S} \)

- Verifying internal functions
- Verifying kernel APIs
- Verifying interrupt handlers

\[
\begin{align*}
\chi, I \vdash \eta_i : \Gamma \\
\Gamma, \chi, I \vdash \eta_a : \varphi \\
\Gamma, \chi, I \vdash \theta : \epsilon
\end{align*}
\]

Side conditions

- abstract primitives for kernel APIs
- abstract primitives for abstract scheduler
- kernel APIs
- interrupt handlers
- internal functions
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
    • Coq tactics

• Verifying μC/OS-II
Our Verification Framework

A. Modeling of OS Kernels
   - High-Level Language
     - High-Level Operational Semantics with ConfigurableSchedulers
   - Domain-Specific Language
     - C Subset
   - Low-Level Language
     - Low-Level Operational Semantics with Context Switch and Interrupts

B. Refinement-Based Verification
   - CSL-Style Refinement-Based Program Logic
   - Contextual Refinement

C. Coq Tactics

Relational Assertion Entailment
Verification Condition Generator
Domain-Specific Solvers

...
Coq Tactics for Automation Support

• Verification condition generator: **hoare forward**
  • Automatically select and apply the inference rules

• Assertion entailment prover: **sep auto**
  • Automatically prove “p => q”

• Domain specific solvers: **mauto** ...

∀x.x < 64 → x ≥ 3 < 8 ;  ∀x.x < 8 → (x ≤ 3) & 7 = 0
Coq Tactics for Automation Support

• To reduce the proof efforts

• To hide the underlying details of the verification framework

• To prove domain specific propositions

The ratio of Coq scripts to the verified C is around 27:1

lots of space for improvement
Outline

• OS Correctness Specification

• Verification Framework
  • System modeling
  • CSL-R: Program logic for refinement & multi-level interrupts
  • Coq tactics

• Verifying μC/OS-II
μC/OS-II

• A commercial preemptive real-time multitasking OS kernel developed by Micrium.

• 6,316 lines of C & 316 lines of assembly code.

• Multitasking & Multi-Level interrupts & Preemptive priority-based scheduling & Synchronization mechanism

• Deployed in many real-world safety critical applications
  • Avionics and medical equipments, etc.
Verifying μC/OS-II

A. Modeling of OS Kernels

B. Refinement-Based Verification

C. Coq Tactics

D. Verifying key modules of μC/OS-II

- Multi-level Interrupts
- Priority-Based Scheduler
- Task Mana.
- Message Queue
- Mutex
- Semaphore
- Mail Box

Synchronization Mechanisms

Refinement-Based Verification Framework
Proving Priority Inversion Freedom

Runtime services and Sys. Props

Whole system properties: specified as trace properties of all apps (with high-level views)

Proved at high-level only, propagated to low-level through contextual refinement!

Priority inversion freedom of mutex in μC/OS-II

Assembly Primitives

Operational Semantics with Context Switch and Interrupts
Bugs found in μC/OS-II

- Priority Inversion Freedom in Mutex
  - Use a simplified priority ceiling protocol
Limitation of Mutex

- Mutual exclusion semaphores with built-in priority ceiling protocol to prevent priority inversions

- Delivered with complete, clean, consistent, 100% ANSI C source code with in-depth documentation.

- Mutual exclusion semaphores with built-in priority ceiling protocol to prevent priority inversions

- Timeouts on ‘pend’ calls to prevent deadlocks

- Up to 254 application tasks (1 task per priority level), and unlimited number of kernel objects

- Highly scalable (6K to 24K bytes code space, 1K+ bytes data space)

- Very low interrupt disable time

- Third party certifiable
Bugs found in \( \mu \)C/OS-II

- Priority Inversion Freedom in Mutex
  - Use a simplified priority ceiling protocol
  - **May cause priority inversion with nested use of mutex!**
  - Fixed in \( \mu \)C/OS-III

- Concurrency bug (atomicity violation)
  - \texttt{INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio)}
  - **May lead to access of invalid pointers**
  - Found in \( \mu \)C/OS-II v2.52 (the version we verified)
  - Fixed in \( \mu \)C/OS-II v2.9
Coq Implementations

- CertiOS
  - framework
    - machine
    - simulation
    - logic
    - theory
  - tactics
  - certiucos
    - code
    - spec
    - proofs

- D. Verifying key modules of uC/OS-II
- C. Coq Tactics
- B. Refinement-Based Verification
- A. Modeling of OS Kernels

Refinement-Based Verification Framework

- Multi-level Interrupts
- Priority-Based Scheduler
- Task Management
- Message Queue
- Mutex
- Semaphore
- Mail Box
Time cost: 6 person-years

<table>
<thead>
<tr>
<th>Components</th>
<th>Cost (py)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic framework design and impl.</td>
<td>4</td>
</tr>
<tr>
<td>First module: message queue (750 lines of C)</td>
<td>1</td>
</tr>
<tr>
<td>the other modules (3000 loc)</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
</tr>
</tbody>
</table>

Debugging and fixing framework, specifications, tactics, etc.

Verification can be much faster with stable framework, tools and libraries

http://staff.ustc.edu.cn/~fuming/research/certiucos/
Conclusion

• Contextual refinements: a natural correctness formulation for OS kernels

• Verification framework for preemptive kernels
  • CSL-R: Concurrency refinement + hardware interrupts

• Verification of μC/OS-II
  • Commercial system independently developed by third-party
Limitations & Future Work

• No termination proofs
  • Relatively simple, can be done in logic or using tools

• Assembly and compiler are not verified
  • Ongoing work

• No separate addr. space and isolation

• No real-time properties

• More whole-system properties, in addition to PIF

• Improvements for automation (better tools and libs)
Acknowledgments: Group Members

Ming Fu  Fengwei Xu  Xiaoran Zhang  Zhaohui Li

Hui Zhang  Ding Ma  Haibo Gu

Alumni: Jingyuan Cao, Jiebo Ma
Thank you!