Verification of System Calls in PikeOS

Christoph Baumann, Holger Blasum, Thorsten Bormer

Saarland Univ., SYSGO AG, Univ. of Koblenz-Landau

19.05.2009

Our Goal

The goal of our subproject is to ...

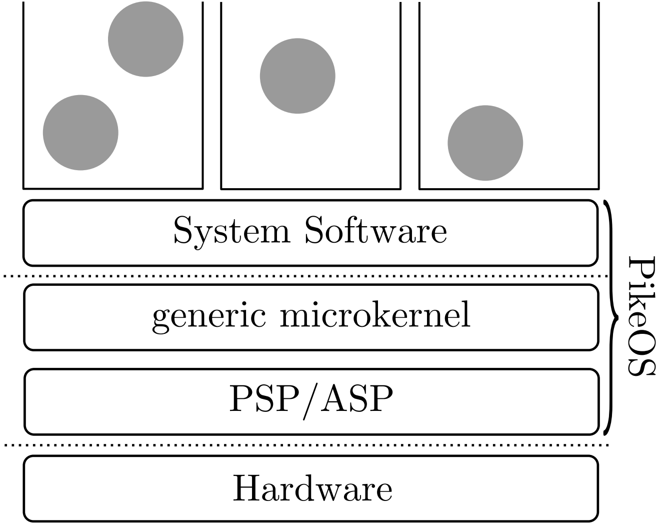
- verify a "real-world" microkernel (PikeOS)
- using the VCC toolchain

$\rightsquigarrow$ case study for large scale verification, VCC

PikeOS

- microkernel for use in safety and security-critical systems developed by SYSGO AG for multiple architectures
- verification target: PowerPC architecture, the snapshot under analysis is of compact size: 10% assembly language, rest in C

PikeOS – System Architecture



Verification of the PikeOS kernel

Verification Progress

- Done: helper functions only visible inside the kernel, sequential setting
- Goal: specify externally visible behavior of the kernel, concurrently executing

First target: system calls

Tasks

- abstract model of kernel state; prove refinement relation
- model and verify (inline) assembly instructions in VCC
- specify and verify sequential execution of system call
- adapt verification to concurrent setting

Verification of the PikeOS kernel

Prove that VCC methodology fits to concurrency model in PikeOS

- prove that scheduling operation is not visible to current thread (separation properties)
- prove functional correctness of the scheduler
- adapt specification to VCC technicalities

Verification of Hardware-related Layers

- model of PPC hardware as VCC ghost structure
- introduce one VCC spec function for each assembly instruction
- replace assembly instructions by C spec. functions

⇒ (inline) assembly can be verified as usual with VCC

Requirement Specification: p4_fast_set_prio

From the kernel reference manual:

"This function sets the current thread's priority to newprio. Invalid or too high priorities are limited to the caller's task MCP. Upon success, a call to this function returns the current thread's priority before setting it to newprio."

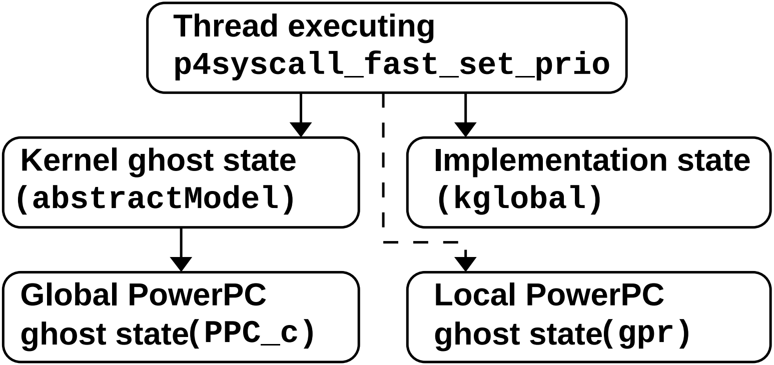
Implementation: p4_fast_set_prio(-helper)

```
P4_prio_t p4_runner_changeprio
(P4k_thrinfo_t *proc, P4_prio_t newprio)
{
  P4_prio_t oldprio; P4_cpureg_t oldstat;

  oldstat = p4arch_disable_int();
    oldprio = proc->userprio;
    proc->userprio = newprio;
    proc->schedprio = newprio;
    kglobal.kinfo->currprio = newprio;
  p4arch_restore_int(oldstat);

  return oldprio;
}
```

PikeOS Entities in our Verification Setup



Abstract Kernel Model

```
spec( struct absModel_str {
    bool interruptsEnabled;
    invariant(interruptsEnabled ==
      (PPC_c.msr.fld.EE == 1))

    struct P4k_thrinfo_t *currentThread;
    invariant(currentThread != NULL)

    invariant(keeps(currentThread, &PPC_c))
} abstractModel; )
```

Specification: p4_runner_changeprio

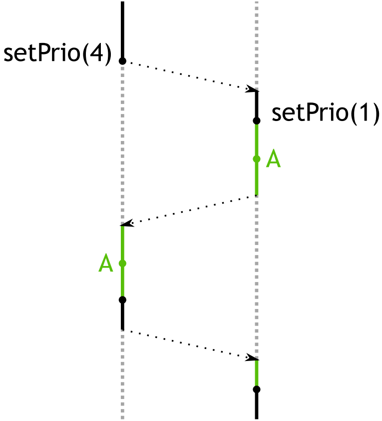
```
P4_prio_t p4_runner_changeprio
(P4k_thrinfo_t *proc, P4_prio_t newprio)
  requires(proc ==
             abstractModel.currentThread)
  ensures(proc->schedprio == newprio && ...)
  returns(old(proc->userprio))

  maintains(wrapped(...))
  writes(...)
{
 ...
}
```

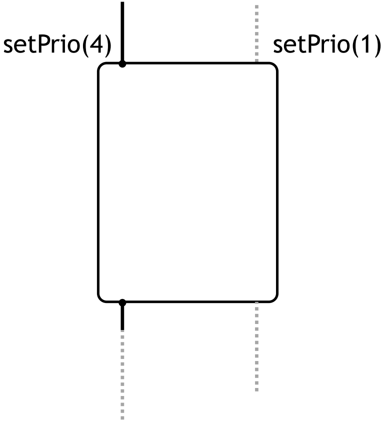
Results – Sequential Setting

- abstract model of PikeOS
- proof of refinement relation between abstract model and concrete state
- proof of sequential behavior of first system calls in terms of abstract model

Concurrent Setting



prio = 4



prio ∈ {1, 4}

Consequences for the Specification of setprio

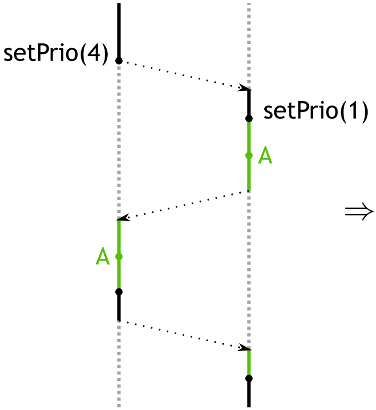
In the concrete implementation, other threads may interfere after atomic block

- one of the threads wins, but we don't know which
- only a rather weak invariant can be shown without further information

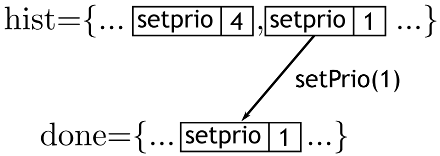
$\Rightarrow$ introduce history for system calls

Concurrent Specification of setprio

Idea: each invocation of a system call is recorded in a history



$\Rightarrow$



$\text{done} \subseteq \text{hist}$

$\text{done}_i \subseteq \text{done}_{i+1} \quad \text{hist}_i \subseteq \text{hist}_{i+1}$

Definition of thread data structure

```
typedef struct update {
    int id;
    int value;
} update, *pUpdate;

typedef struct thread {
    volatile int prio;

    spec(volatile ptrset done;)
    spec(volatile ptrset hist;)
} thread, *pThread;
```

Invariants of thread

```
//hist contains only updates
invariant(forall(obj_t o; set_in(o, hist)
==> is(o, update) && set_in(o, owns(this))))

//D is a subset of H
invariant(set_subset(done,hist))

//H only increases
invariant(set_subset(old(hist), hist))

//D only increases
invariant(set_subset(old(done), done))
```

Invariants of thread

We execute each update from hist, but only once:

```
invariant(unchanged(prio) ||
 exists(update *u; set_in((obj_t) u, done)
          && !set_in((obj_t) u, old(done))
          && prio == u->value))
```

Implementation of setprio for VCC

```
void setPrio(pThread t, int v)
{
     atomic(...) {
          t->prio = v;
     }

     atomic(...) {}
}
```

Concurrent Specification of setprio

```
void setPrio(pThread t, int v
     spec(update *up))
    maintains(up->value == v)

    requires(set_in((obj_t) up, t->hist)
        && !set_in((obj_t) up, t->done))

    ensures(exists(update *u;
    set_in((obj_t) u, t->done)
       && !set_in((obj_t) u, old(t->done))
       && t->prio == u->value))

    ensures(set_in((obj_t) up, t->done))
```

Conclusion

Results

- verification of concurrent system call with histories
- analysis of PikeOS concurrency model w.r.t VCC model

Further Work

- solve remaining technicalities
- extend verification to other system calls
- prove concurrency model