Formal modeling and analysis of advanced scheduling features in an avionics RTOS

Research supported by
NASA Langley Research Center and Honeywell
Cooperative agreement NCC-1-399

Darren Cofer
Murali Rangarajan

darren.cofer@honeywell.com
612-951-7279

Outline

- Deos™ real time operating system
  - Integrated Modular Avionics
  - time partitioning
  - capabilities
- Model
  - objectives
  - structure
  - components
  - abstractions
- Analysis results
  - time partitioning
  - function preconditions
  - non-progress cycles

Federated architecture

Key features –

Critical functional separation

- Low-criticality functions cannot corrupt critical functions

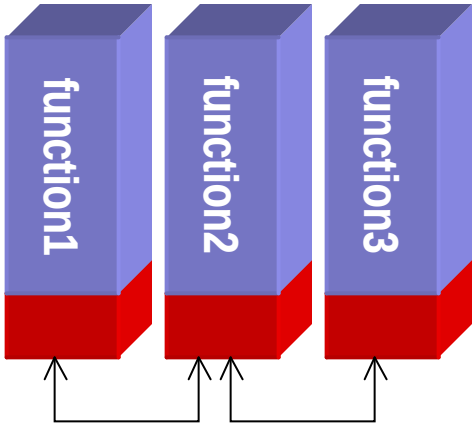
Guaranteed access

- to processing resources
- to I/O

Mixed software/hardware development levels

- customized to function

Data latency/synchronization



Drawbacks –

All functions that share a processor must certify to the highest criticality level of those functions

- Encourages many processors with lower utilization

Inefficiencies in use of resources

- Modern processors and memory devices have far more capability than a single critical function usually needs
- Multiple power supplies, multiple I/O systems

Proliferation of part types

- Each subsystem tends to design a point solution optimized for its performance and I/O needs

Increased weight, power and wiring

Integrated architecture

Key features –

Resources are shared

- Better approach for future growth - spare capacity can be used for function extensions, or for new applications of any criticality level

Hardware part types are reduced

- Reduces spares, wiring, labor, inventory,
- Simplifies obsolescence issues
- Increases reliability and common designs

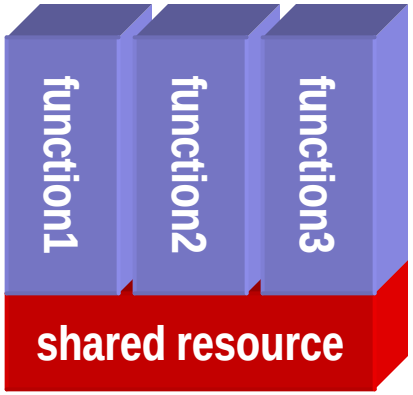
Partitioned software ensures that only the critical software is developed and validated to critical requirements

- Lowers risk by reducing the amount of critical software

Issues –

What could a non-critical function do to a critical one in a shared resource system?

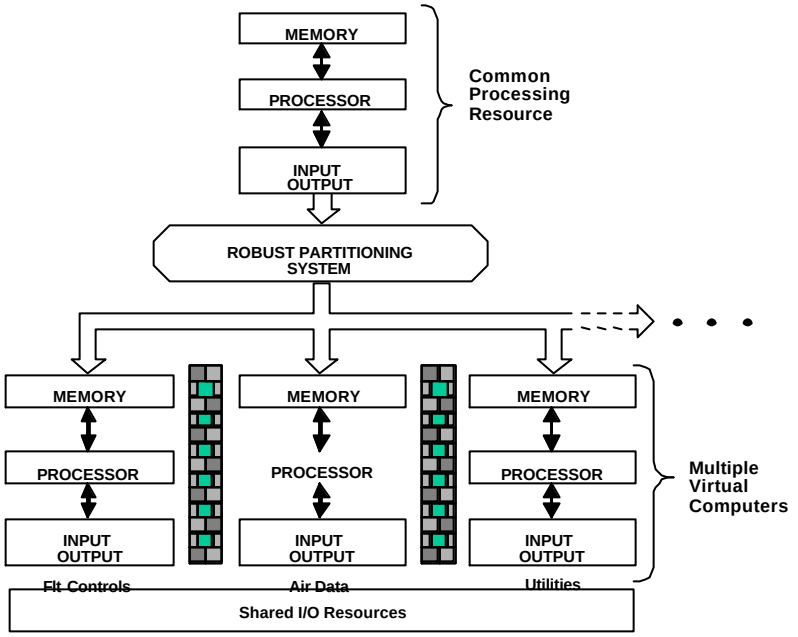
- Steal time/interrupt the processor
- Crash the processor
- Erroneously write data into wrong areas
- Corrupt I/O
- Hog internal communications paths (e.g. backplane)



Key concept: Partitioning

Space partitioning:

- Protection of program, data, registers, and dedicated I/O. Any persistent storage location (e.g., data memory) must only be writable by one partition. Any temporary storage location (e.g., processor registers) used by a partition must be saved when control is transferred.



Time partitioning:

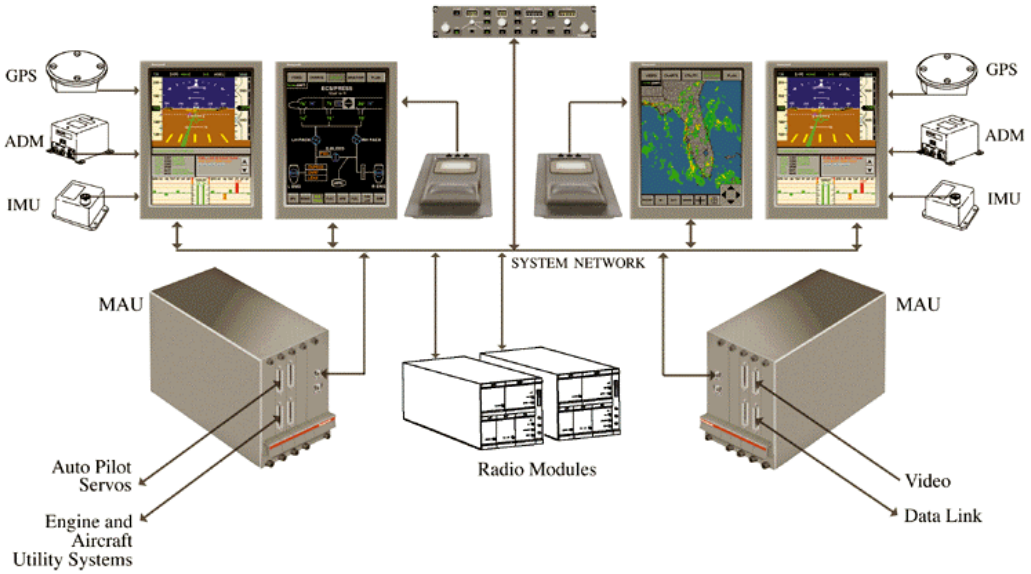
- Protection of processing and communications bandwidth assigned to a partition. A partition's access to a prescribed set of hardware resources for a prescribed period of time is guaranteed. The order of execution between communicating partitions is consistent each execution frame.

every thread gets its allotted CPU budget every period

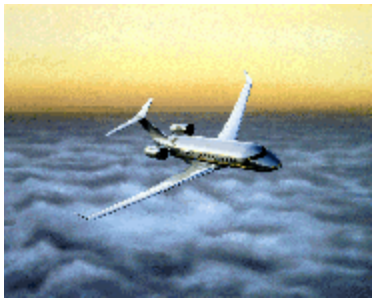
No Functional Partition may:

- Contaminate another's code, I/O, or data storage areas (Space Partitioning)
- Consume shared processor resources to the exclusion of any other partition (Time Partitioning)
- Consume I/O resources to the exclusion of any other partition
- Cause adverse affects to any other partition as a result of a hardware failure unique to that partition
- No single failure of common hardware will prevent continued safe flight and landing

Applications of formal verification in Primus Epic



- *Bombardier Global Express*
- *Raytheon Hawker Horizon*
- *Agusta-Bell AB-139*
- *Embraer 170*
- *Embraer 190*



Deos™ real-time OS

- Rate monotonic scheduling with priority inheritance
- Time & space partitioning, dynamic threads, slack scheduling, aperiodic interrupts, mutexes
- Model derived from operational flight code (C++)
- Analyzed/verified:
  - time partitioning for threads
  - function preconditions
  - overhead accounting in scheduler

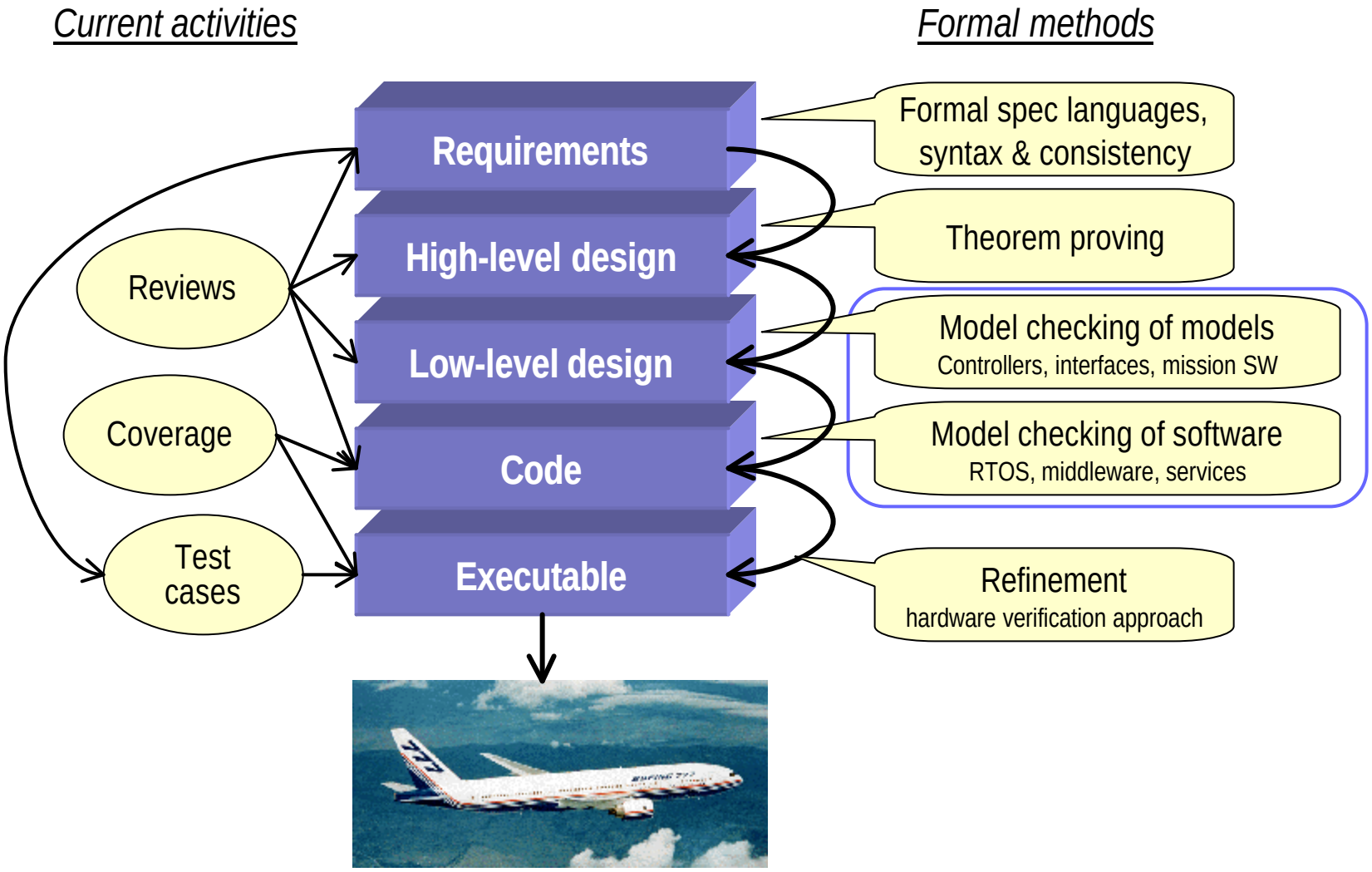
ASCB-D communications network

- Model derived from design specification (MS Word)
- Master node selection and time synchronization
- Analyzed/verified:
  - synchronization at start-up
  - handling of node faults, bus faults, clock drift
  - clarified many parts of specification

Deos features

- Dynamic thread creation and deletion
- Dynamic process creation and deletion
- Time partitioning at the thread level
  - every thread gets its allotted CPU budget every period
- Rate monotonic scheduling (harmonic periods)
- Space partitioning at the process level
- Dynamic enforcement of timing constraints and quotas
- Reuse of unused thread time – *slack scheduling*
  - allows delayed execution of high priority tasks while still meeting deadlines
  - usage: interrupt handling, incremental algorithms, variable QoS
- Aperiodic interrupt threads
- Bounded overhead (*O(1)*), charged to threads
- Mutexes with bounded blocking times – *highest locker protocol*
- Long and short duration waits, events, mailboxes and periodic IPC

Applications of FM in development process



Approach

Analysis of existing implementation

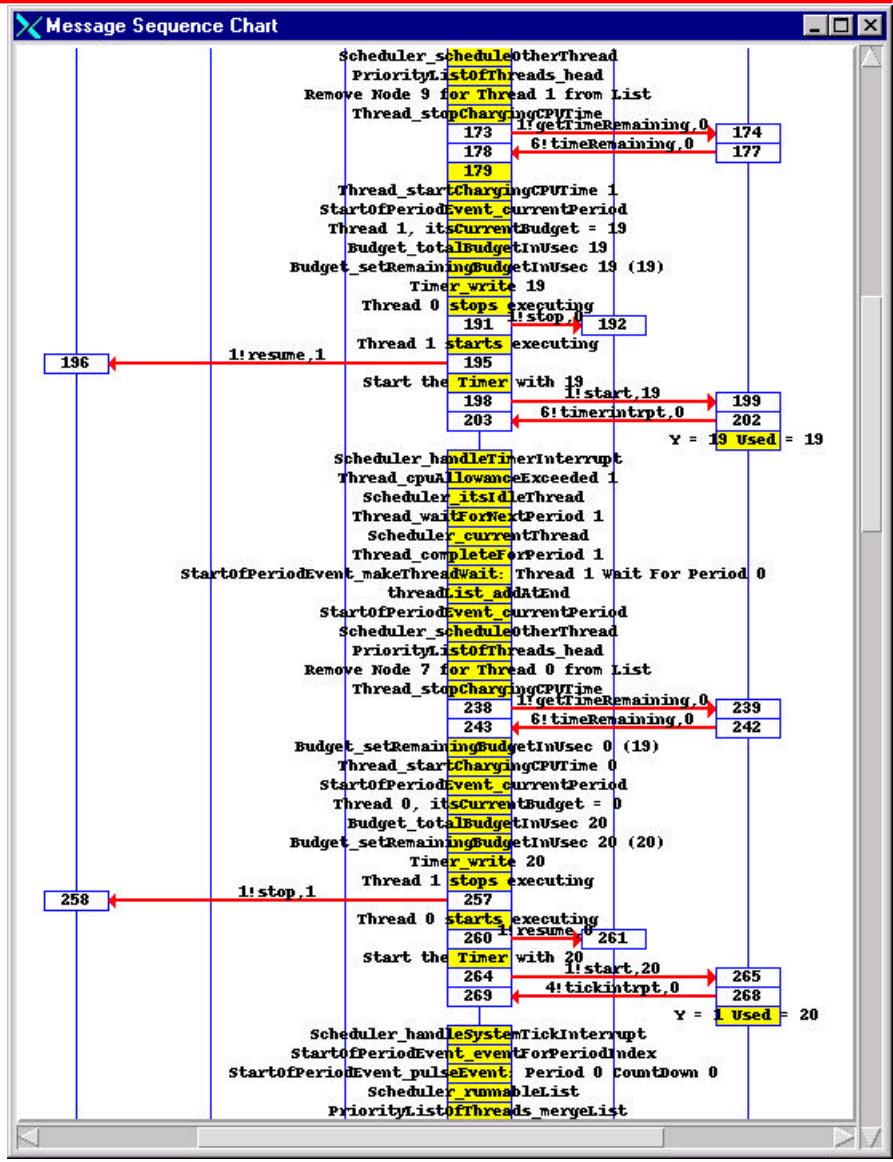
- Scheduler and elements that impact timing
- “Slice” of Deos scheduler extracted manually

Model checking

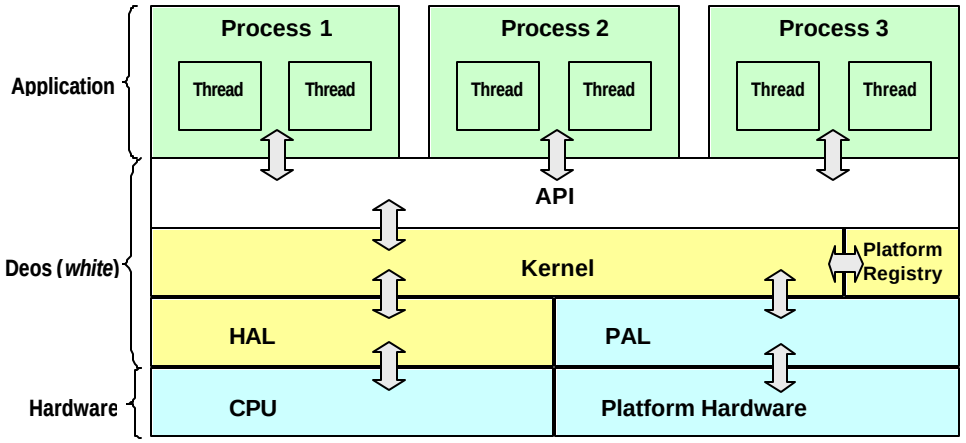
- Spin
- Verify time partitioning property, function preconditions, class invariants

Incorporate features that impact timing

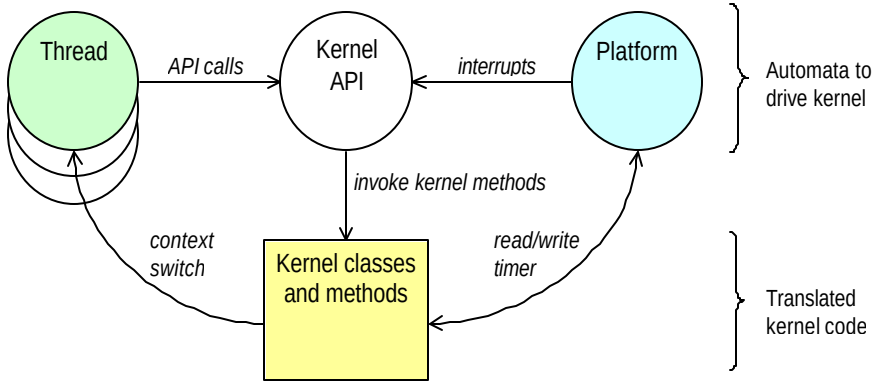
- Thread creation/deletion
- Slack stealing
- Aperiodic interrupts
- Overhead accounting
- Mutexes, HLP



Deos scheduler model in Spin



Real system



Model

Scheduler is a set of API functions that must be called in response to certain events from environment

- Threads
  - waitUntilNextPeriod()
  - waitUntilNextInterrupt()
  - deleteThread()
- Platform HW
  - handleSystemTickInterrupt()
  - handleTimerInterrupt()
  - raisePlatformInterrupt()

Scheduler reads/writes HW timer and initiates context switch to start/stop thread execution

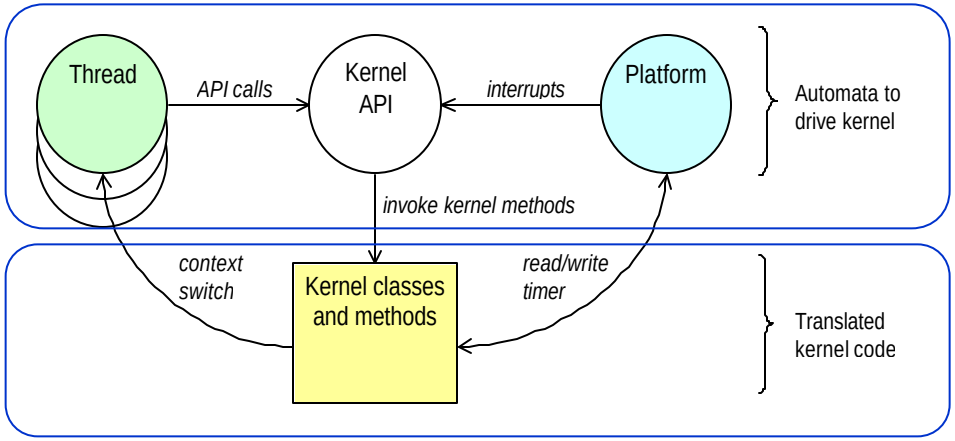
Modeled/analyzed using *Spin* model checker developed at Bell Labs

Structure of model

- Separation of system and environment
  - compare controller/plant
- System modeled with high fidelity, traceability to real system
  - certification
  - implementation, autocoding
- Environment
  - abstract, only model necessary parts
  - fault injection

Details:

- **Single process which has a Main thread**
- **Two user threads receive their budgets from Main upon creation and return it to Main upon deletion**
- **Threads can complete early, be forced to complete if they exceed their allotted time budget, wait of slack, or be preempted by higher priority thread**
- **Platform models all hardware interactions: system tick, thread timer, hardware interrupts**
- **Threads may be ISRs**



Environment

System to be verified

High fidelity model of software

Model derived from kernel code (rather than specification)

- Accuracy (design = code)
- Utility in verification/certification
- Level of detail required to capture subtle timing behaviors

```
void Scheduler::initializeInterruptEvents()
{
  interruptEvents =
   new InterruptEvent*[ numInterrupts ];

  for ( platformInterruptNumber i = 0;
                i < numInterrupts; i++ )
  {
    interruptEvents[i] = new InterruptEvent( i );
    ignorePlatformInterrupt[i] = false;
  }
}
```

Deos scheduler
- 55 files
- 800K C++ files
- ~15K LOC

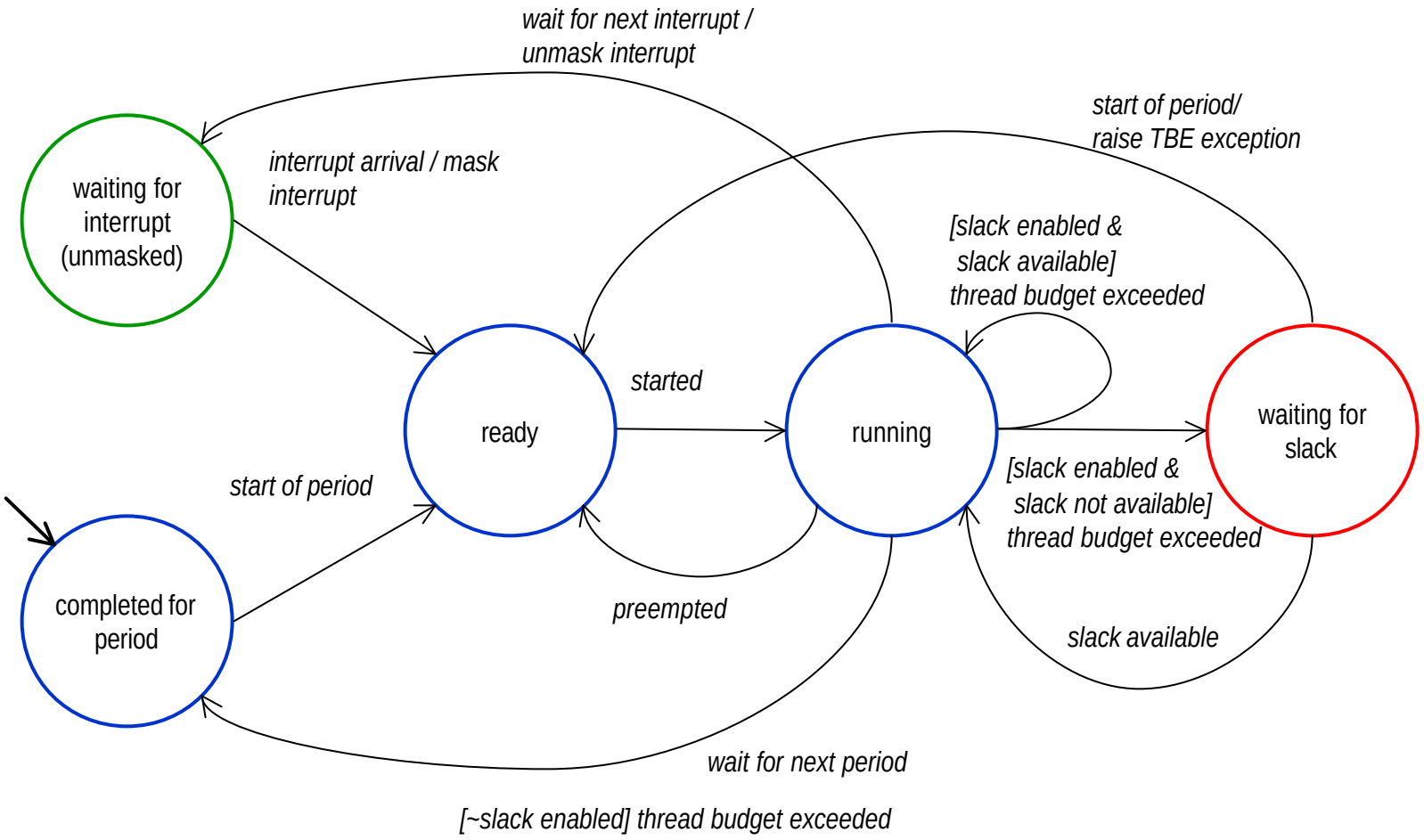
```
inline Scheduler_initializeInterruptEvents()
{
  InterruptEvent_PTR
   Scheduler_interruptEvents[InterruptEvent_MAX];

  byte i = 0;
  do
  :: (i < InterruptEvent_MAX) ->
     NEW_InterruptEvent(Scheduler_interruptEvents[i]);
     i++;
  :: else -> break;
  od;
}
```

Spin model
- 20 classes
- 120 methods
- >4000 LOC (Promela)

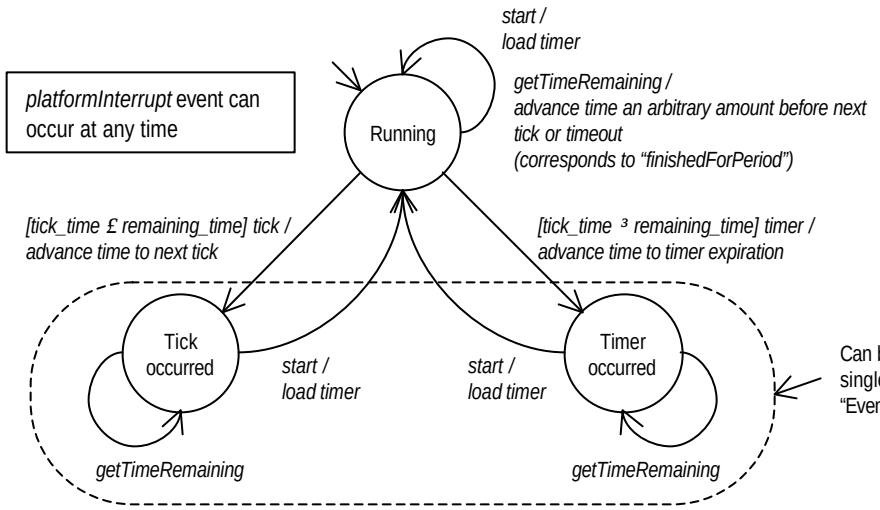
```
inline Thread_waitForNextInterrupt() {
assert(currentBudget.remainingBudget >= 2*contextSwitchPlusDelta+cacheBonus);
```

Thread state in scheduler

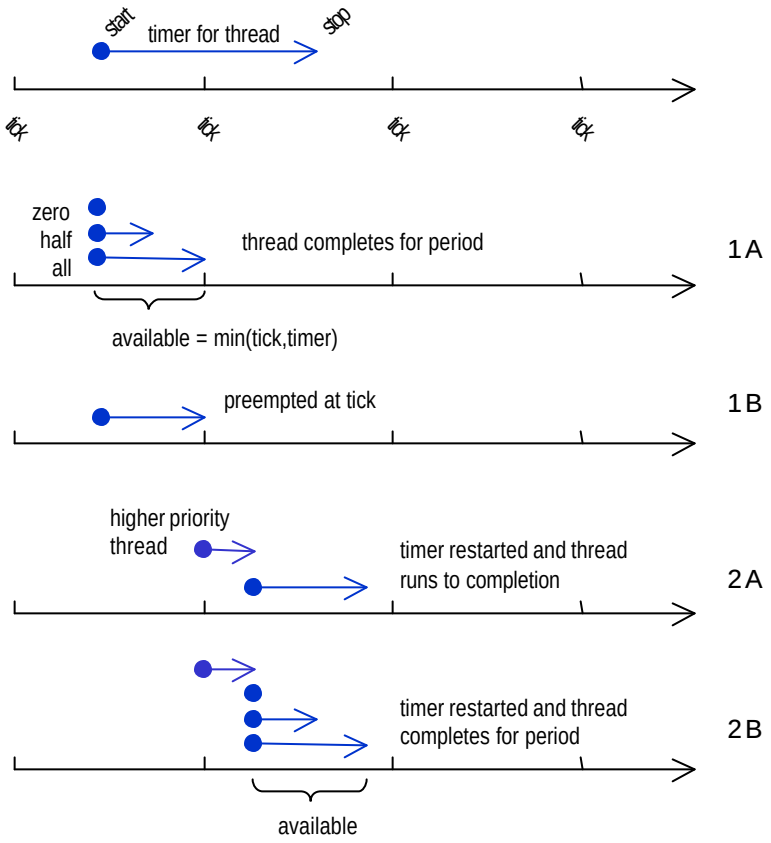


Platform automaton (timer)

- HW timers & interrupts
  - system tick
  - thread timer (preemption)
- Platform interrupts
  - physical I/O
- Discrete event model of time
  - Non-deterministically select next event from eligible set
  - Advance clock to time of selected event
  - Update eligible event set



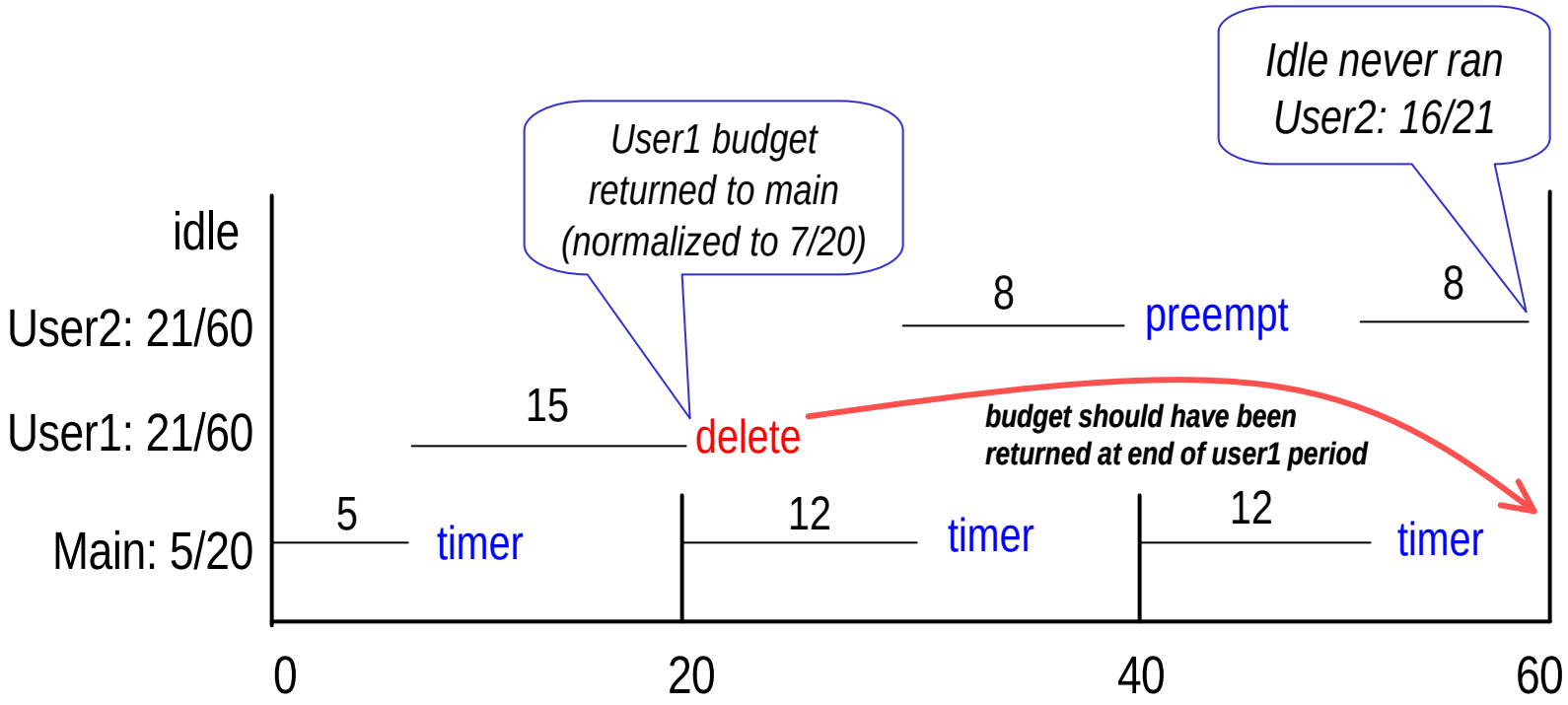
Abstraction: thread time consumption



Scheduler operations

- System tick interrupt
  - Generated by platform hardware
  - Tick handler may cause currently running thread to be preempted
- Timer interrupt
  - Generated by platform hardware
  - Produced when thread timer runs down to zero
- User interrupts
  - Generated asynchronously by I/O or other hardware
  - May cause currently running thread to be preempted by ISR thread
- Traps (SW interrupts)
  - Triggered when user thread attempts to execute kernel service
  - Services may include critical sections (interrupts disabled)

Time partitioning error trace (previous work)




```
LTL property -- time partitioning
[](startPeriod -> (!endPeriod U idleRun))
```

Penix, Visser, Engstrom, Larson, Weininger, "Verification of Time Partitioning in the DEOS Scheduler Kernel," ICSE 2000

Time partitioning

With slack and ISRs, idle thread may never run: need new approach

- Assertion checked in the tick interrupt handler
- Detects the problem after scheduler has over-committed
- Consists of 2 parts –
  - one for currently running thread
  - another for threads in all periods ending at this tick
- Disjunction of 3 conditions –
  - thread is the Idle thread (no deadline)
  - thread received its full budget (remaining budget = 0)
  - thread voluntarily completed for period

No errors found

- Doesn't hold for ISR threads
  - can't guarantee interrupt occurs early enough in period
  - but interrupts don't interfere with other threads

Preconditions

- Deos coding standards require that preconditions be explicitly stated for each function
  - compare Design by Contract, Eiffel
  - but captured as comments
  - manually reviewed during current verification process
  - also class invariants
- For preconditions relevant to our model
  - captured as assertions
  - verified by Spin
- Examples

```
// Invariants:
// - Idle always appears on this list, except when idle is itsRunningThread.
// - Every thread on this list must have at least contextSwitchPlusDelta
//   CPU time remaining in its budget.
static PriorityListOfThreads*    itsRunnableList;

// Preconditions:
//  - Must be called from within a critical section.
//  - The thread consuming slack must be the Scheduler::currentThread().
//    This is necessary for proper tracking of each thread's slack
//    consumption.
static void updateSlackConsumed(cpuTimeInMicroseconds slackConsumed);
```

Non-progress cycles

- Previous work involved assertions and LTL properties, and checks for dead locks
- Added checks for live locks (non-progress cycles)
  - Involves placing “progress” labels in strategic locations in model
  - If model includes a cycle that does not contain a progress label, it signifies a livelock
- Check that systemTick event always occurs
- Detected two problems:
  - Detected the live lock when overhead = 0 in updated model
  - This technique proved its usefulness on non-slack model by detecting a bug with the 0/1 abstraction for period ID
  - Both cases are artifacts of the model cannot occur in the actual code

Summary

- Results
  - high fidelity model of Deos scheduler
  - focus on features impacting time partitioning guarantees
  - verified time partitioning, various preconditions, absence of non-progress cycles
- Other work
  - overhead accounting
  - mutexes, HLP
  - proofs of abstractions