Controlled Owicki-Gries Concurrency:

Reasoning about the Preemptible eChronos Embedded Operating System

MARS 2015

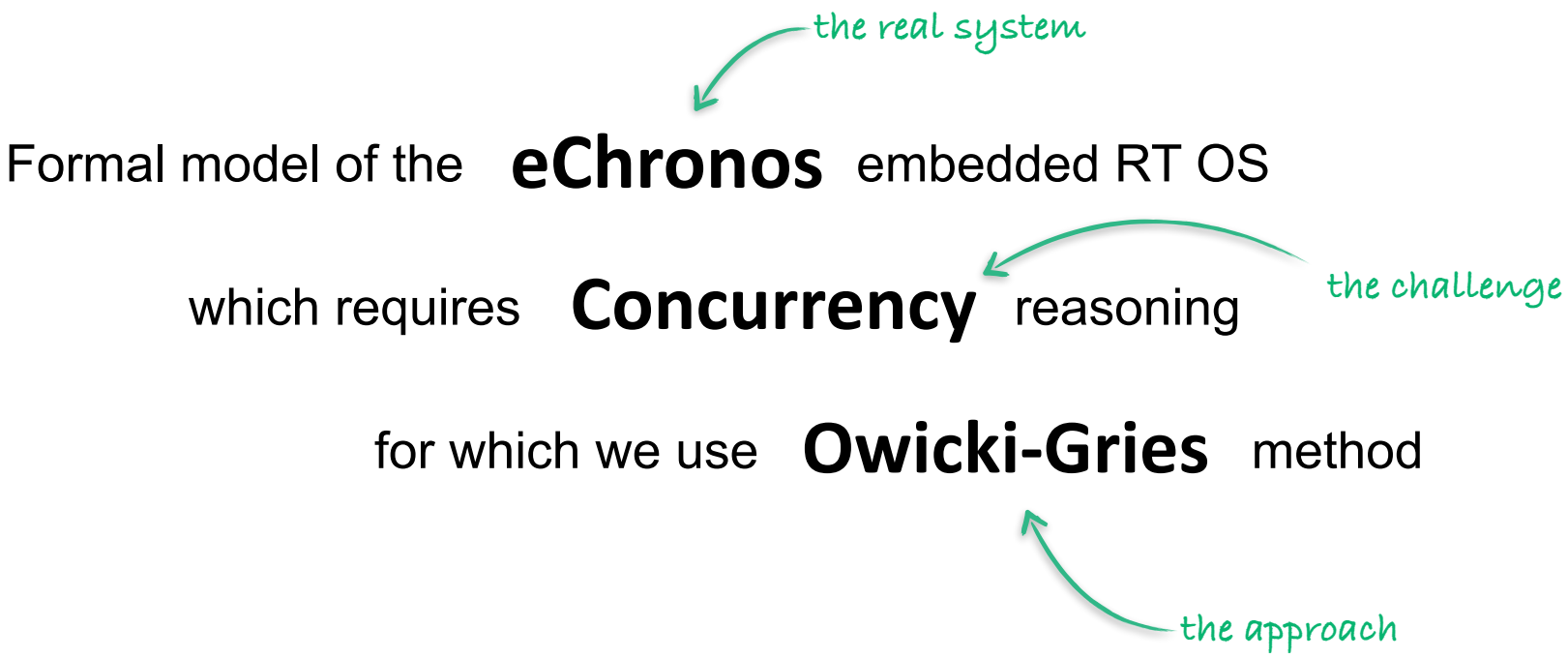
June Andronick, Corey Lewis, Carroll Morgan

November 2015

www.csiro.au

CSIRO

Aim


the real system
Formal model of the eChronos embedded RT OS
which requires Concurrency reasoning
the challenge
for which we use Owicki-Gries method
the approach

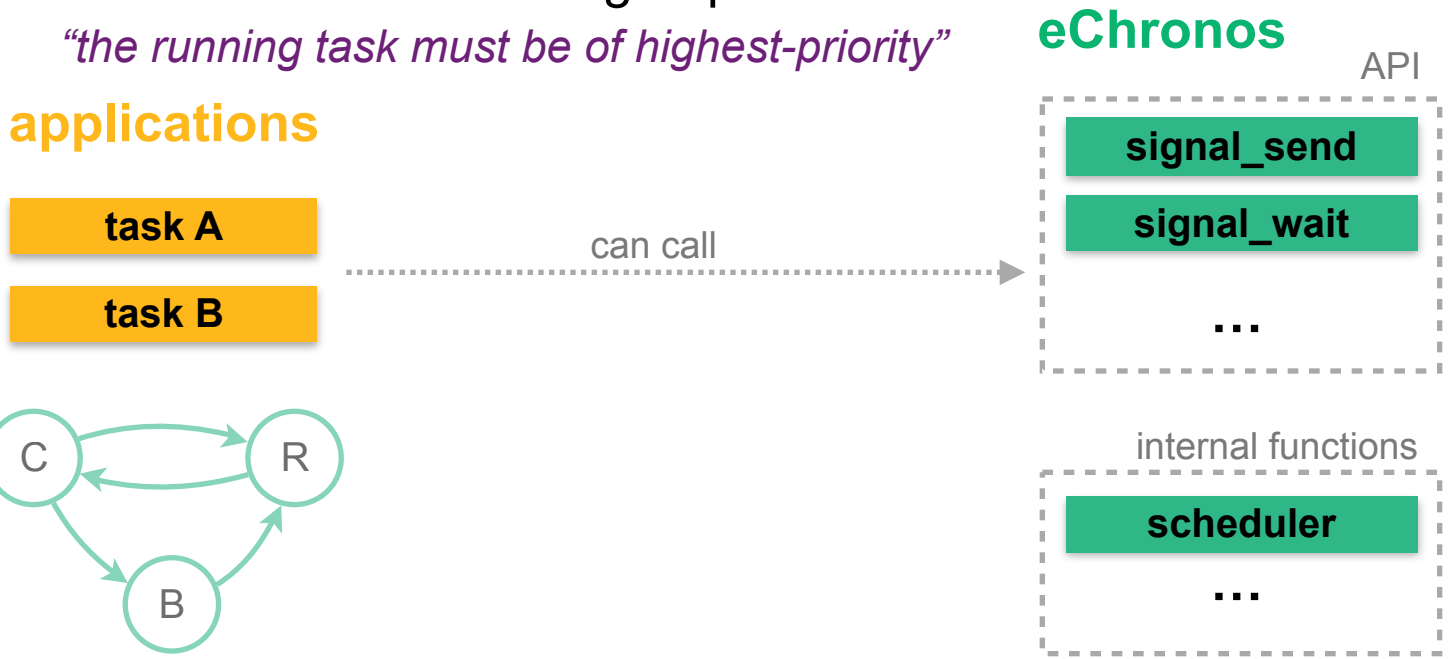
eChronos

What: Small real-time OS library (~500 SLOC)
(Joint development with Breakaway Consulting)

Where: Embedded devices, with limited resources, no memory-protection

Job:
- provides synchronisation primitives
- schedules tasks according to priorities

"the running task must be of highest-priority"

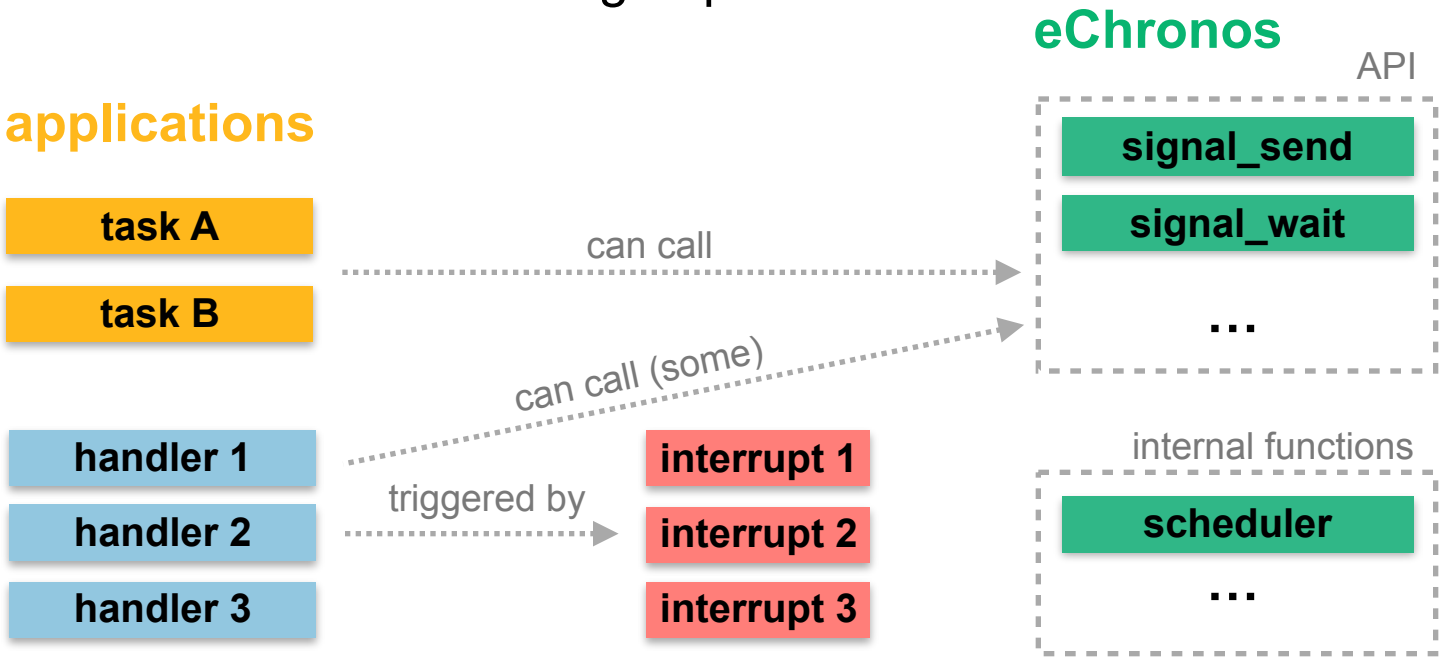


eChronos

What: Small real-time OS library (~500 SLOC)
(Joint development with Breakaway Consulting)

Where: Embedded devices, with limited resources, no memory-protection

Job:
- provides synchronisation primitives
- schedules tasks according to priorities



eChronos

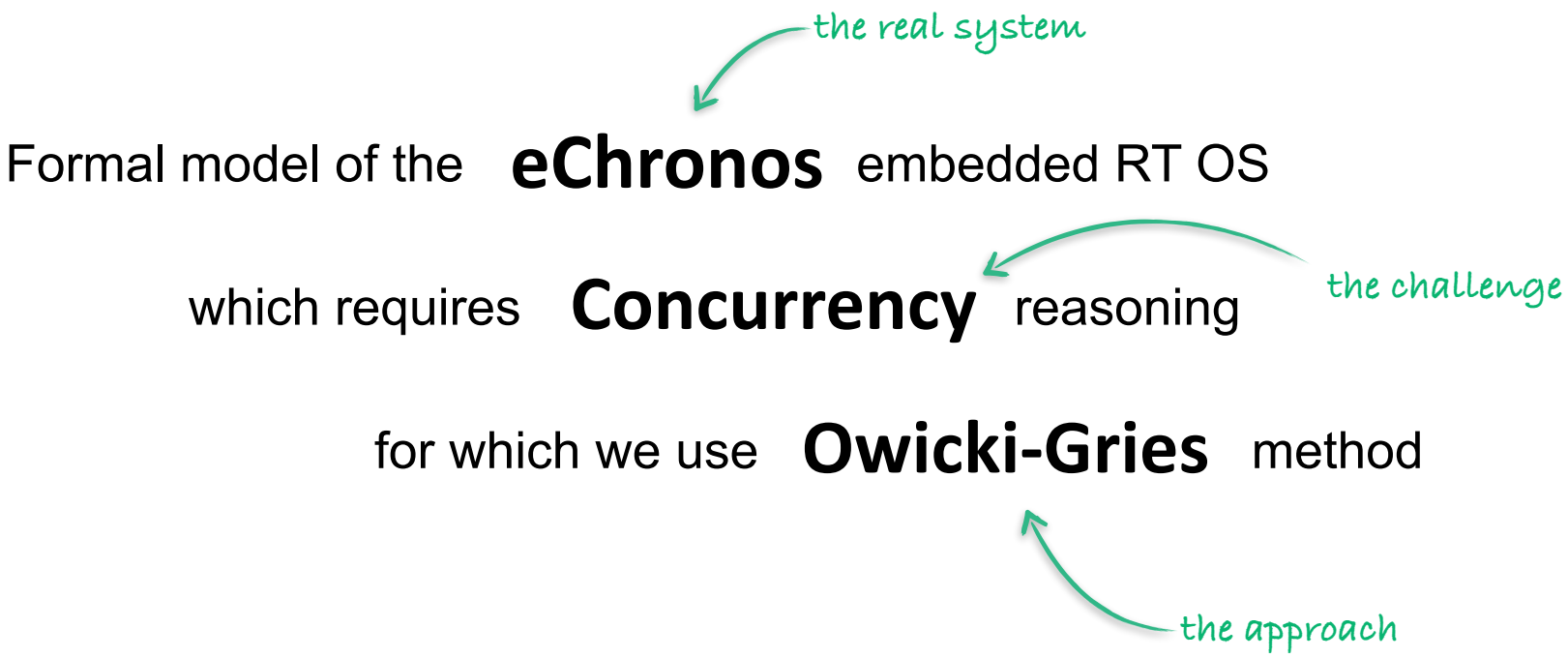
Characteristics:
- small, fast

Aim:
- **verified**

Challenges:
- tasks can be preempted by another task
- OS can be interrupted by external event

Our approach applies to OS systems with these characteristics:
- **tasks can be preempted by higher priority tasks**
- **OS code can be interrupted by external event**

Aim


the real system
Formal model of the eChronos embedded RT OS
which requires Concurrency reasoning
the challenge
for which we use Owicki-Gries method
the approach

Owicki-Gries

What: Extension of Hoare logic to shared-variable parallel programs

(Suzanne Owicki and David Gries,1976)

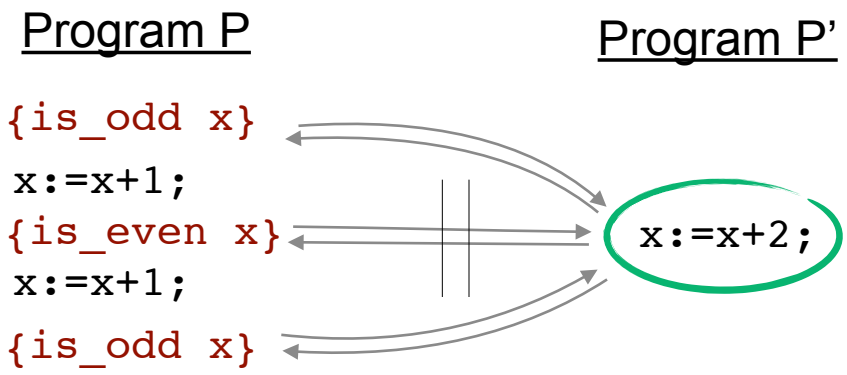
cobegin $S1$ // $S2$ // ... // Sn **coend**

await B **then** S

Why: Small, fast

Owicki-Gries

How:



P is (sequentially) correct
Local correctness

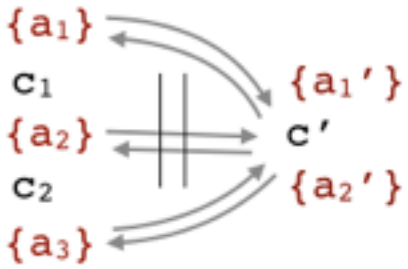
P' does not interfere with P
Interference freedom

P is globally correct

Owicki-Gries

From a parallel composition of fully annotated programs, generates correctness verification conditions

- local correctness — prove each $\{a_i\}\ c_i\ \{a_{i+1}\}$
- interference freedom — for each assertion a in P, and each command c' in P', prove that $\{a \wedge a'\}\ c'\ \{a\}$



! quadratic explosion of proof obligations, not compositional

➡ For our system, interleaving is more controlled
+
use automation power of modern theorem provers

Leonor Prensa Nieto

2002

Owicki-Gries

Aim:

- use OG to model interleaving between eChronos code, interrupt code, and tasks

Challenges:

- tasks are not 1st class citizens
- concurrency is uncontrolled

Contributions

Formal model of **eChronos** using **Owicki-Gries** method

Challenges:

- tasks can be preempted
- OS can be interrupted

Challenges:

- tasks are not 1st class citizens
- concurrency is uncontrolled

Contributions

1. Tasks as 1st class citizens in O-G
   "AWAIT-painting"
2. Controlled interleaving model
   Hardware API model
3. Formal model of eChronos
   to prove that the running task is always the highest-priority runnable task

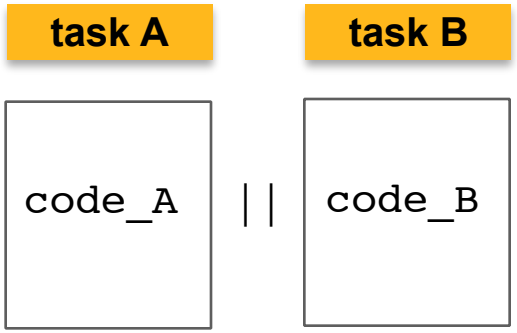
Formalised in Isabelle/HOL theorem prover

1. Tasks as 1st class citizens in O-G
   "AWAIT-painting"

2. Controlled interleaving model
   Hardware API model

3. Formal model of eChronos
   to prove that the running task is always the highest-priority runnable task

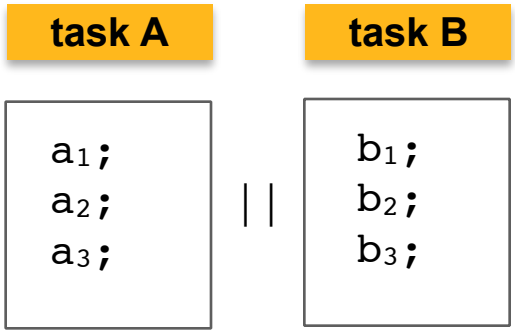
Context switching between tasks



No arbitrary concurrency between all these programs

Can only switch from A to B if B become the active task

Context switching between tasks



- We introduce: Variable AT (Active Task)
- We "AWAIT-paint" all statements:

```
AWAIT AT=A THEN a1;
AWAIT AT=A THEN a2;
AWAIT AT=A THEN a3;
||
AWAIT AT=B THEN b1;
AWAIT AT=B THEN b2;
AWAIT AT=B THEN b3;
```

- We automate this with an "`await_paint task-id code`" command:

```
await_paint A code_A;
||
await_paint B code_B;
```

- We model context switch: `context_switch task_id ≡ AT:=task_id;`

Tasks as 1st class citizens

Owicki-Gries

1975

Leonor Prensa Nieto

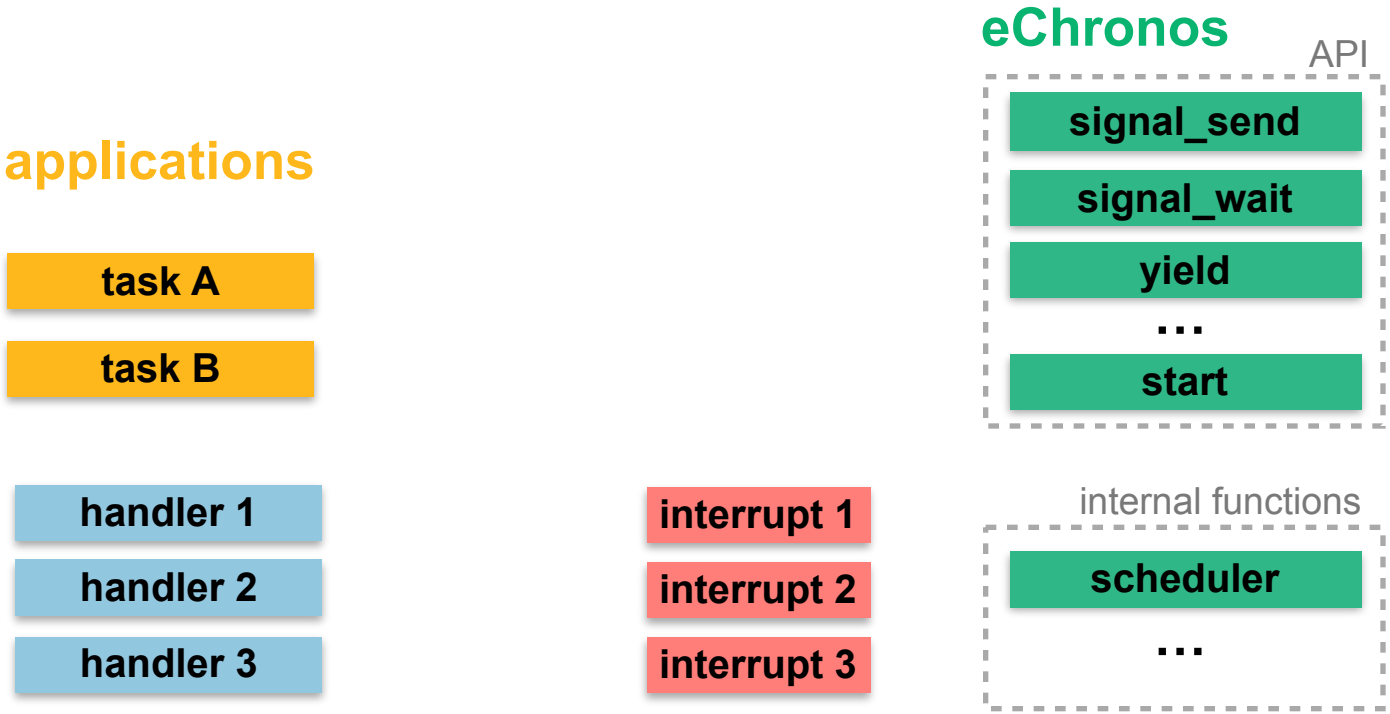
2002

+

```
AT
await_paint
context_switch
```

1. Tasks as 1st class citizens in O-G
   "AWAIT-painting"
2. Controlled interleaving model
   Hardware API model
3. Formal model of eChronos
   to prove that the running task is always the highest-priority runnable task

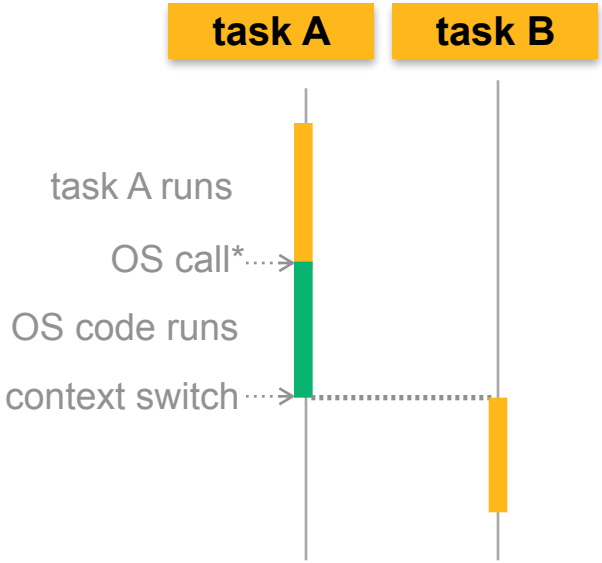
Recall eChronos



Execution model

A task runs until:

1. **calling** a OS API function that changes runnable tasks



eChronos

API

- signal_send
- signal_wait
- yield
- ...
- start

internal functions

- scheduler
- ...

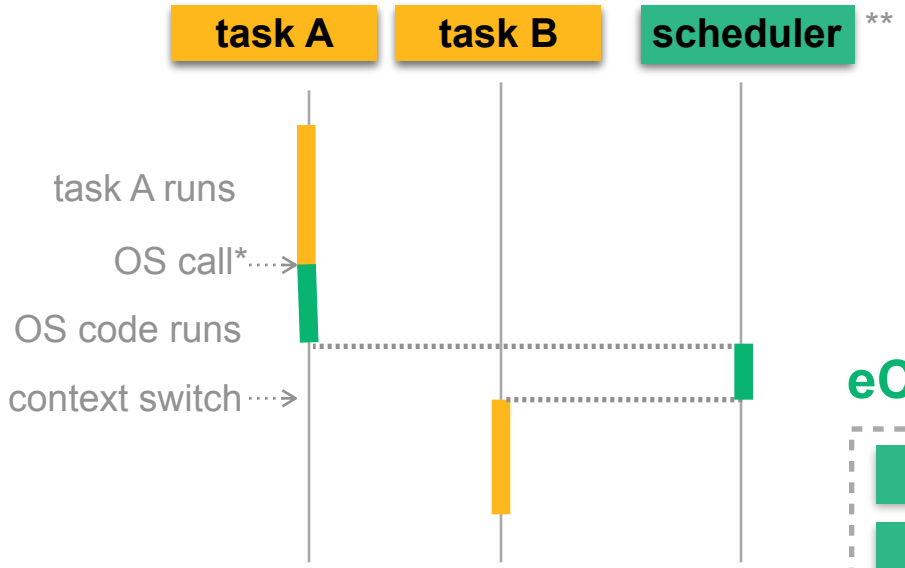
- handler 1
- handler 2
- handler 3

* Note: no mode switch between OS and task in such constrained hardware.

Execution model

A task runs until:

1. **calling** a OS API function that changes runnable tasks



eChronos API

- signal_send
- signal_wait
- yield
- ...
- start

internal functions

- scheduler
- ...

- handler 1
- handler 2
- handler 3

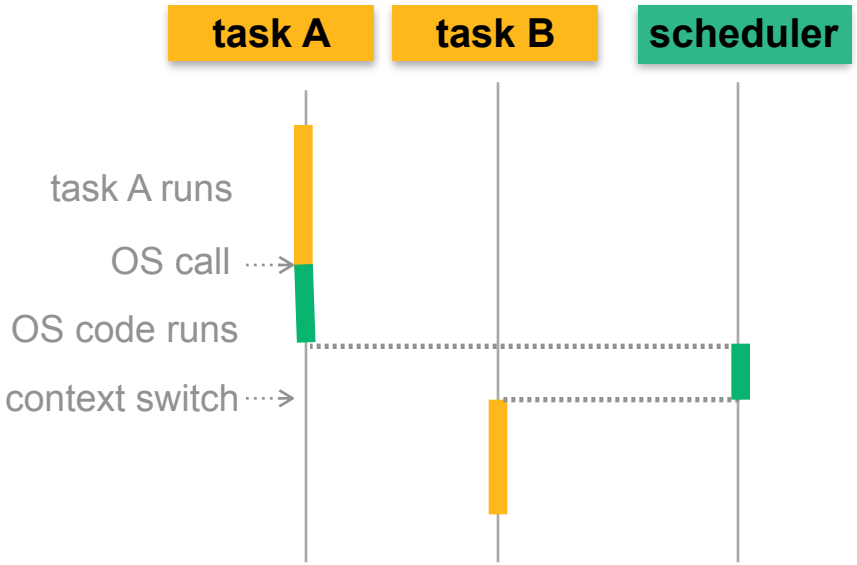
* Note: no mode switch between OS and task in such constrained hardware.

** Note: using ARM "supervisor call" (svc) mechanism

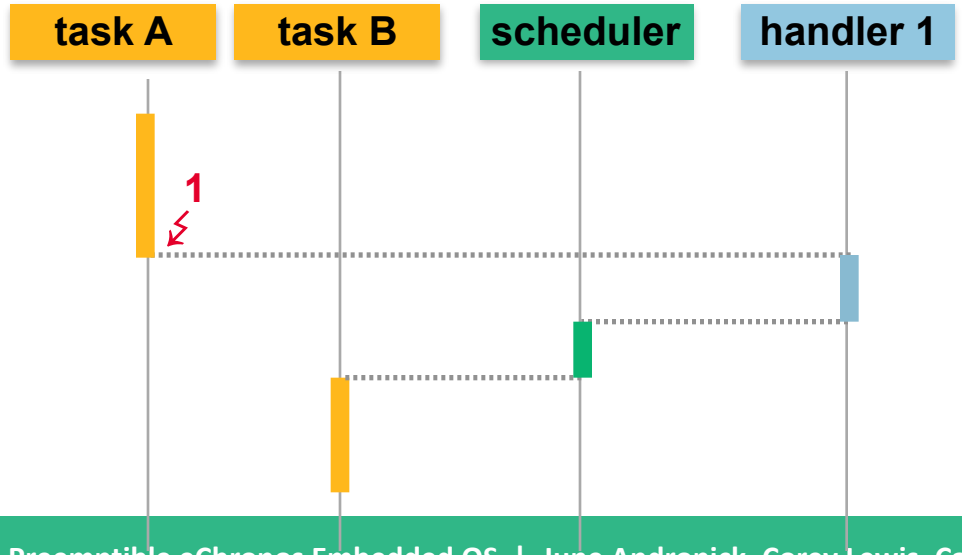
Execution model

A task runs until:

1. **calling** a OS API function that changes runnable tasks



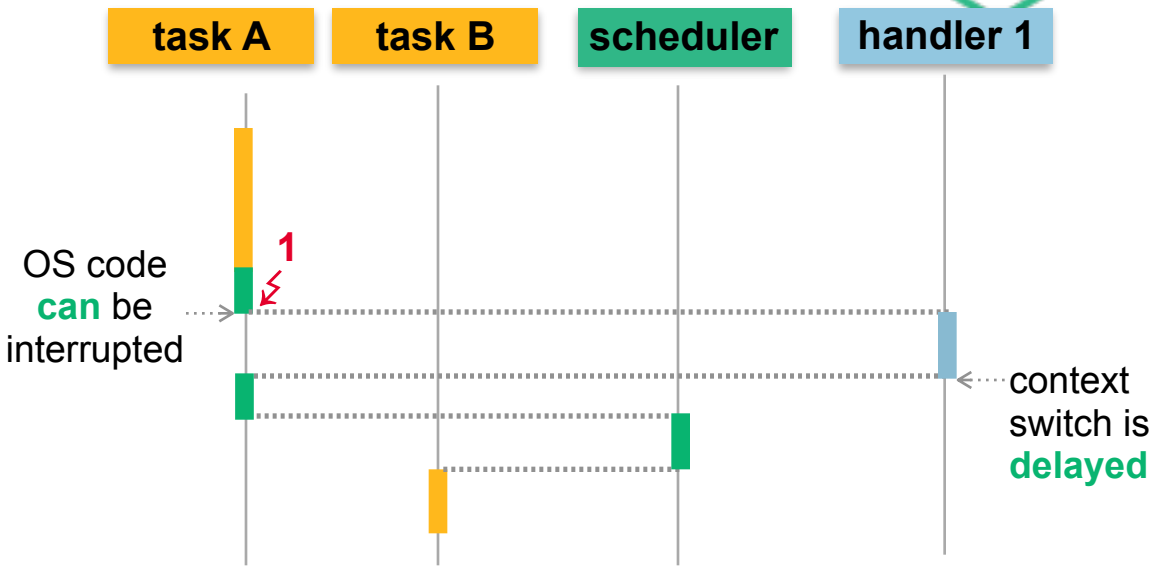
2. being **interrupted** by a handler that changes runnable tasks



Execution model

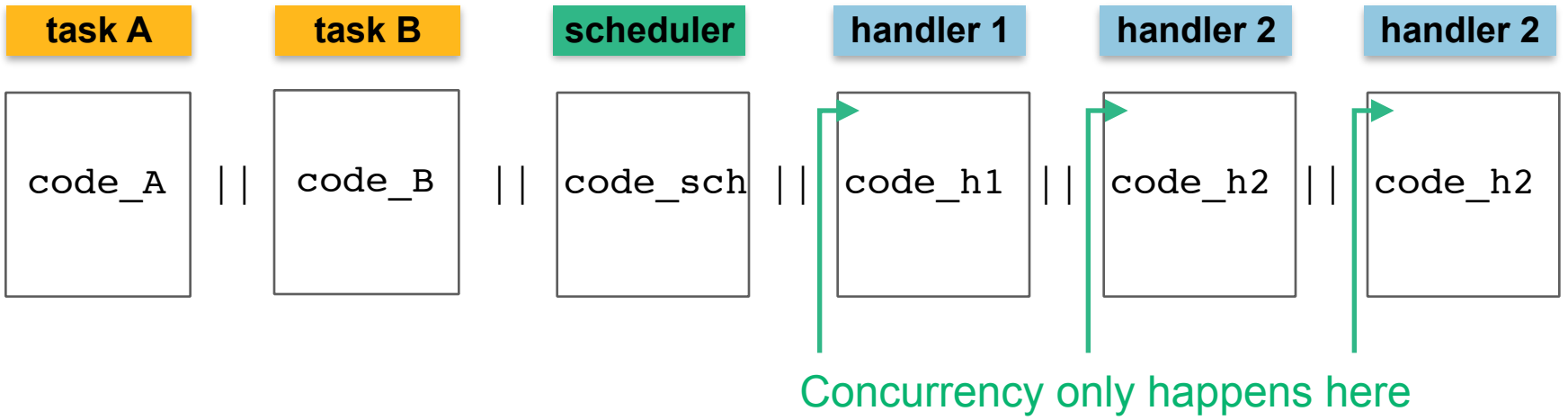
A task runs until:

3. **calling** a OS API function that gets **interrupted** by a handler that changes runnable tasks



Our approach applies to OS systems with these characteristics:
- **tasks can be preempted by higher priority tasks**
- **OS code can be interrupted by external event**

Controlled Interleaving in OG

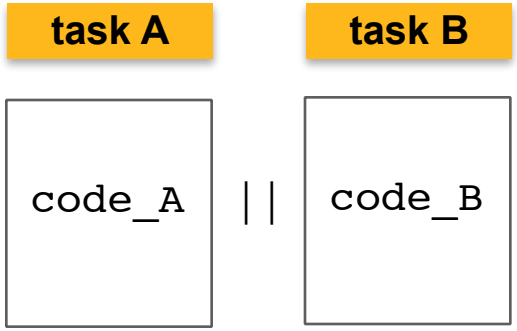


We do not have arbitrary concurrency between all these programs

Other interleaving is controlled by hardware instructions

context-switch
return-from-interrupt
interrupt masking

Modelling application tasks



➡ We add the "await-painting"

```
WHILE True DO
  await_paint A code_A ||
END
```

```
WHILE True DO
  await_paint B code_B
END
```

```
WHILE True DO
  AWAIT AT=A THEN a1;
  AWAIT AT=A THEN a2;
  ...
END
```

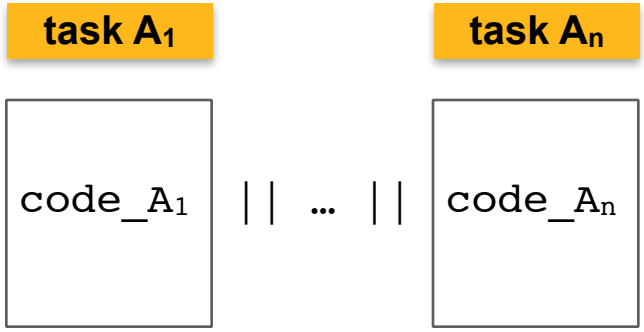
||

```
WHILE True DO
  AWAIT AT=B THEN b1;
  AWAIT AT=B THEN b2;
  ...
OD
```

Interleaving can only happen if one instruction is a call to an OS function calling the scheduler calling `context_switch`

…
or if an interrupt happens

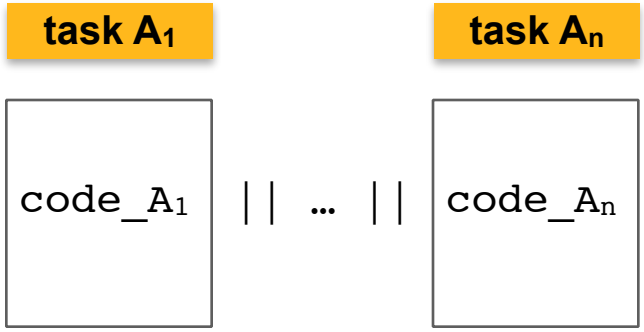
Modelling application tasks

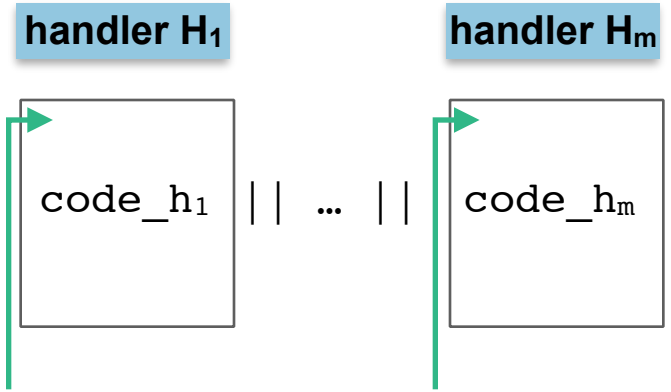


➡ We can generalise to n tasks

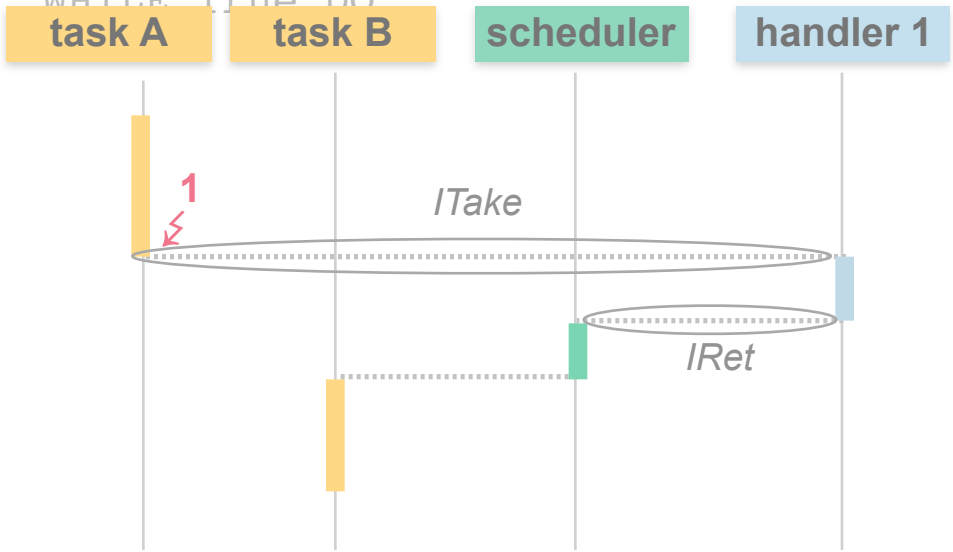
```
SCHEME [0≤i<n]
   WHILE True DO
     await_paint Ai code_Ai
   END
```

Modelling interrupt handlers



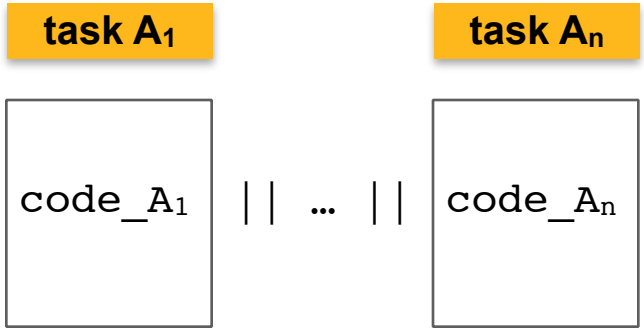


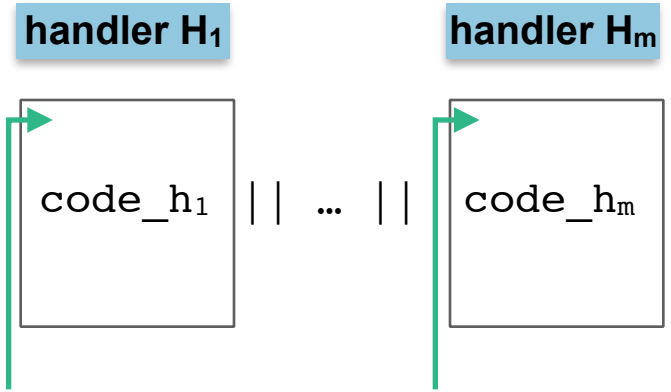
SCHEME [0≤i<n]
WHILE True DO




```
SCHEME [0≤j<m]
    WHILE True DO
      ITake(Hj)
      await_paint Hj code_hj
      await_paint Hj IRet()
    END
```

Modelling interrupt handlers

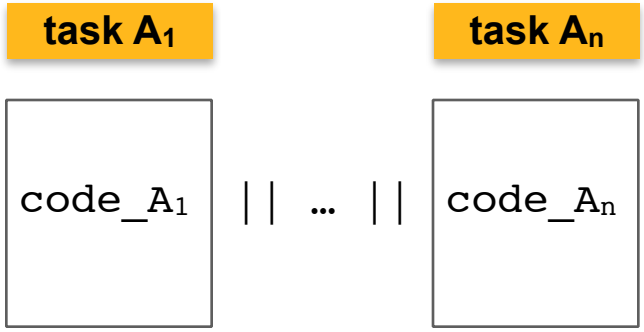


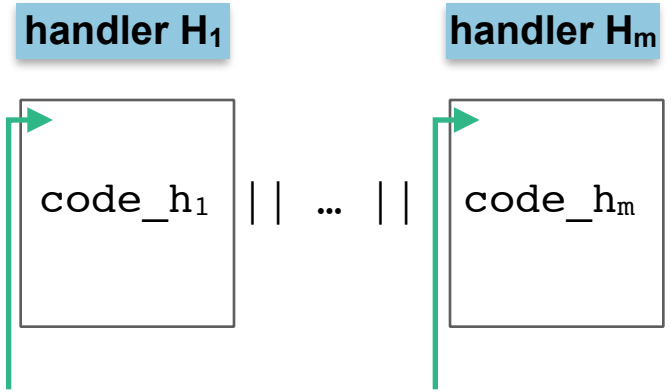


ITake(X) ≡ AT:= X

```
SCHEME [0≤j<m]
    WHILE True DO
      ITake(Hj)
       await_paint Hj code_hj
       await_paint Hj IRet()
      END
```

Modelling interrupt handlers





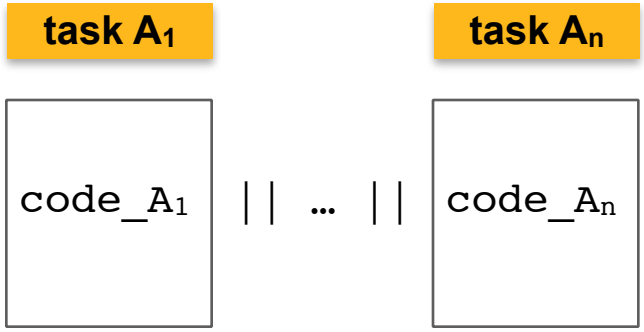

```
ITake(X) ≡ AWAIT X ∈ EI
           THEN
              AT:= X
```

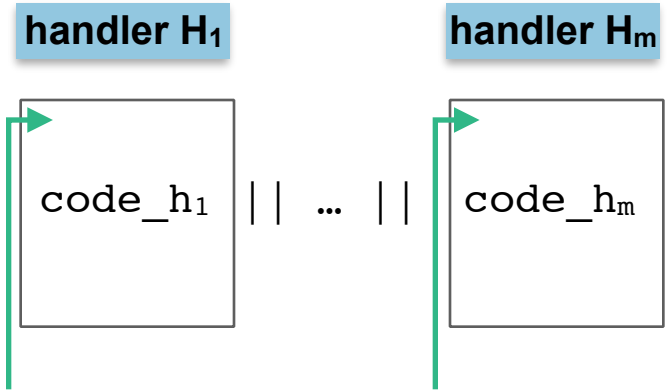
```
SCHEME [0≤j<m]
    WHILE True DO
      ITake(Hj)
       await_paint Hj code_hj
       await_paint Hj IRet()
    END
```

- What if X is masked?
- New variable EI (Enabled Interrupts)
- New hardware functions

```
IntDisable(X) ≡  EI := EI − X
IntEnable(X)  ≡  EI := EI ∪ X
```

Modelling interrupt handlers



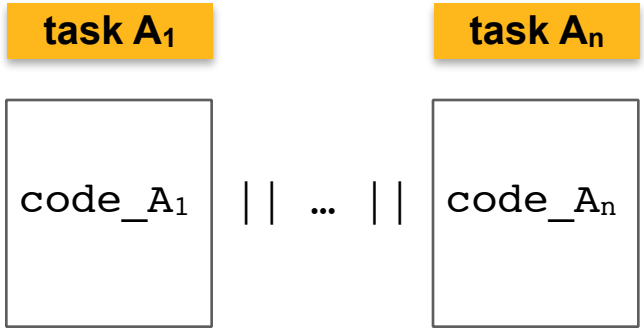


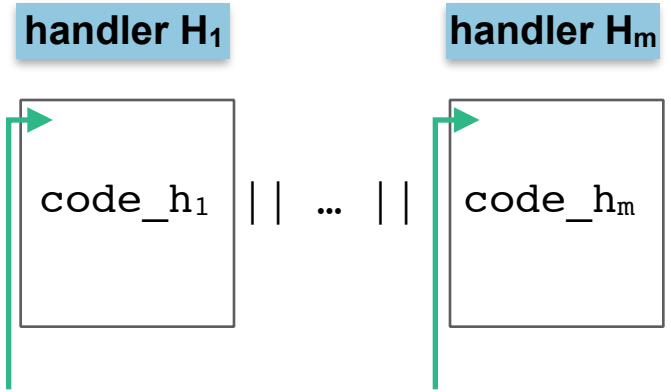

```
ITake(X) ≡  AWAIT X ∈ EI
            THEN
              push AT ATstack;
              AT:= X
```

```
SCHEME [0≤j<m]
   WHILE True DO
     ITake(Hj)
      await_paint Hj codehj
      await_paint Hj IRet()
   END
```

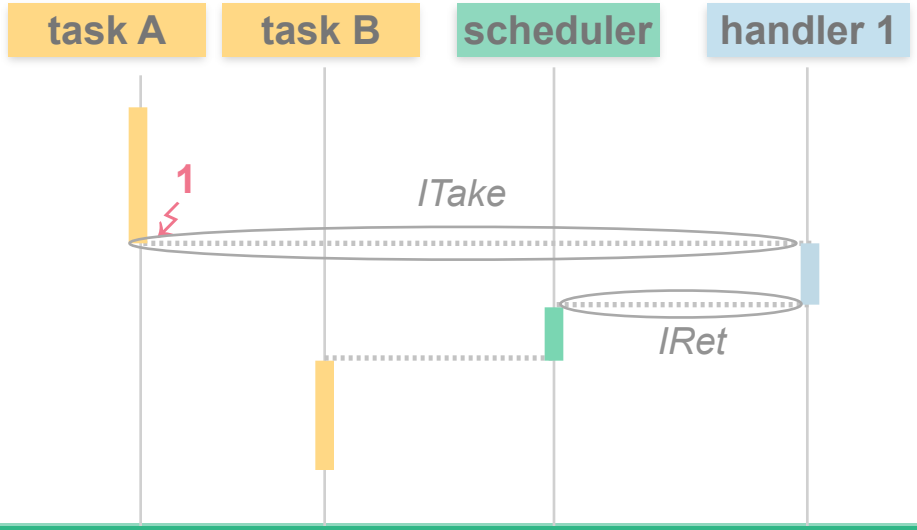
- ➡ How to return to previsouly running task eventually?
- ➡ New variable ATstack

Modelling interrupt handlers



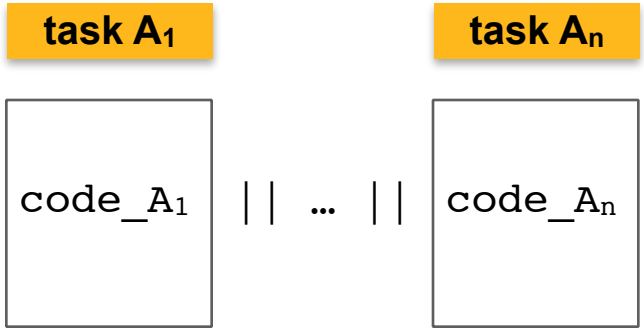


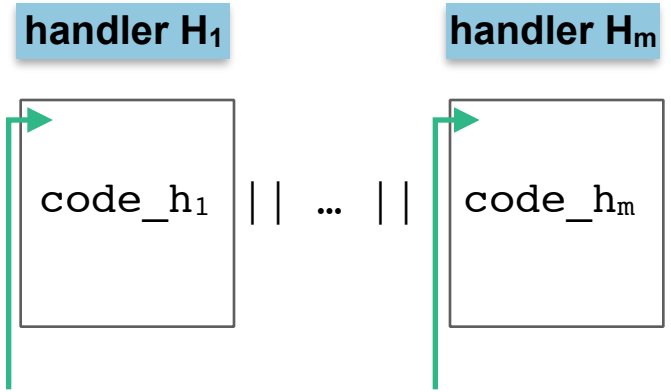
IRet() ≡ AT := sched;



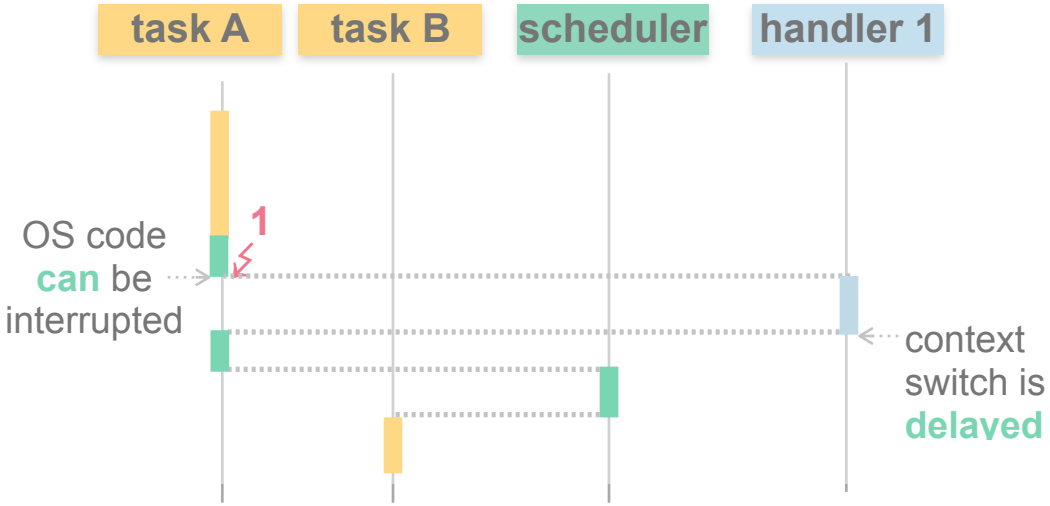

```
SCHEME [0≤j<m]
    WHILE True DO
     ITake(Hj)
     await_paint Hj code_hj
     await_paint Hj IRet()
    END
```

Modelling interrupt handlers





IRet() ≡ AT := sched;




```
SCHEME [0≤j<m]
    WHILE True DO
     ITake(Hj)
     await_paint Hj code_hj
     await_paint Hj IRet()
    END
```

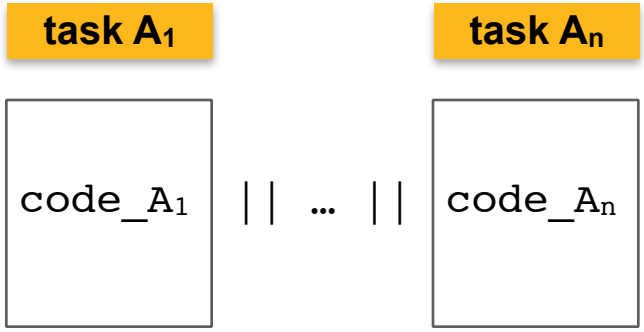
➡ New hardware functions

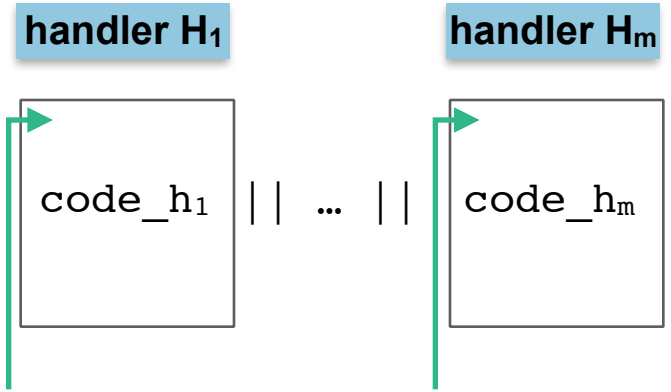
SchedDisable()≡ EI:=EI−sched

SchedEnable() ≡ EI:=EI∪sched

➡ New flag schedReq

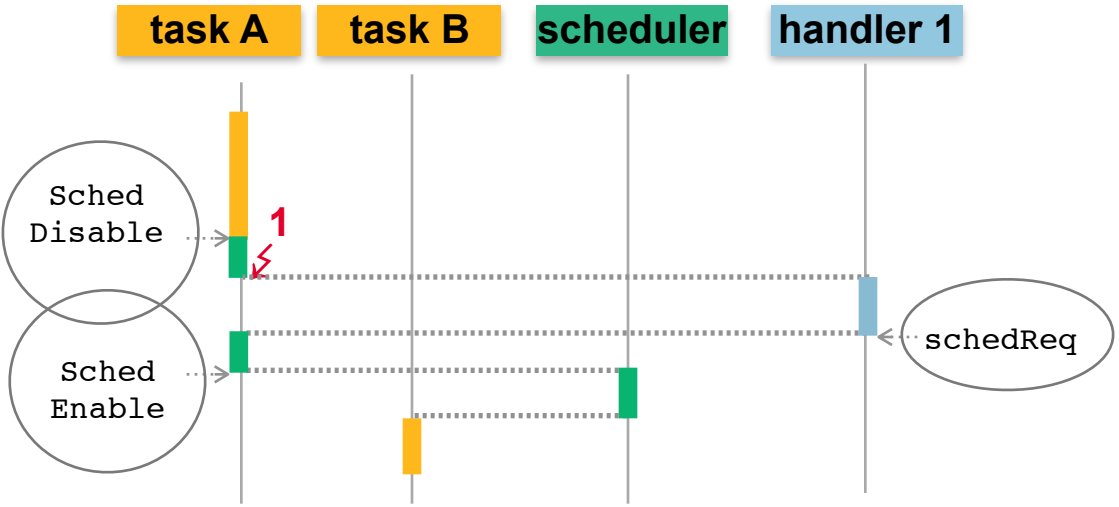
Modelling interrupt handlers





IRet() ≡ AT := sched;

```
SCHEME [0≤j<m]
    WHILE True DO
     ITake(Hj)
     await_paint Hj code_hj
     await_paint Hj IRet()
    END
```



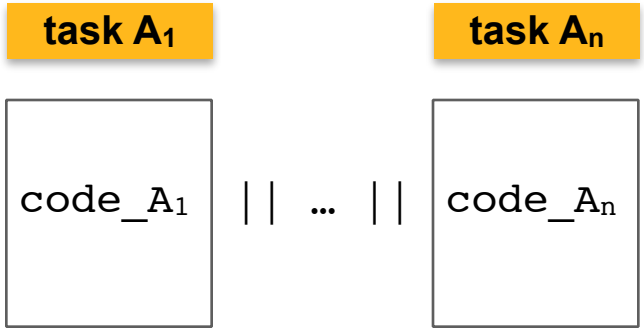
➡ New hardware functions

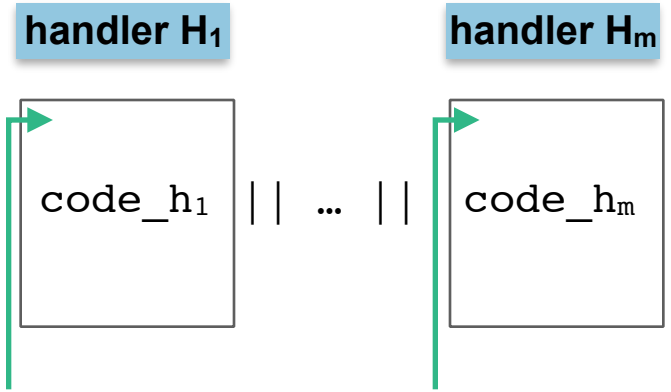
SchedDisable()≡ EI:=EI−sched

SchedEnable() ≡ EI:=EI∪sched

➡ New flag schedReq

Modelling interrupt handlers

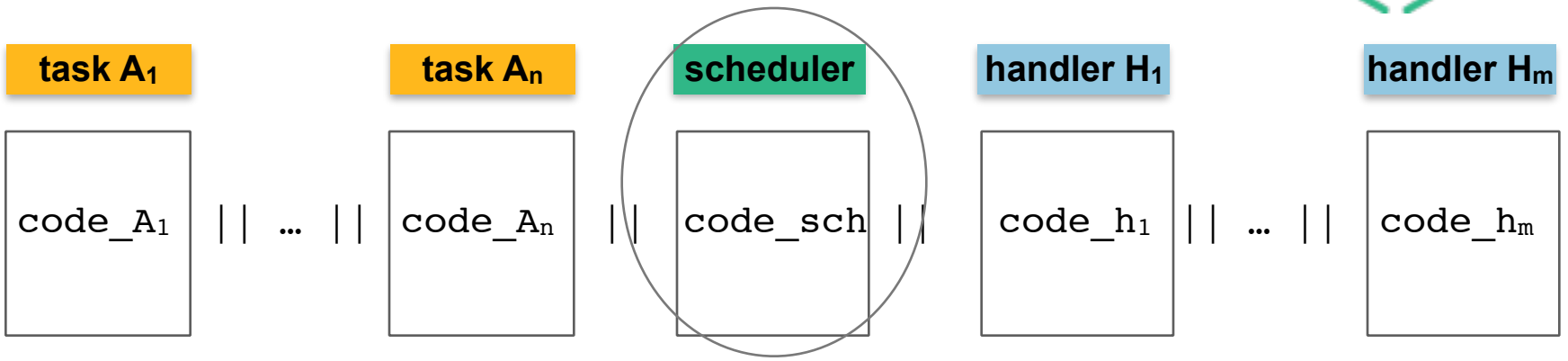





```
IRet() ≡
  IF  schedReq  ∧ sched ∈ EI
  THEN  AT := sched;
  ELSE  AT:= pop ATstack;
```

```
SCHEME [0≤j<m]
    WHILE True DO
      ITake(Hj)
      await_paint Hj code_hj
      await_paint Hj IRet()
    END
```

Modelling interrupt handlers




```
WHILE True DO *
  ITakeSched()
  await_paint sched code_sch
  await_paint sched IRet()
END
```

```
ITakeSched() ≡
  AWAIT schedReq  ∧ sched ∈ EI
  THEN
    schedReq:=False;
    push AT ATStack;
    AT:=shed;
END
```

*simplified version, full details in paper

Our model of interleaving and HW API

task A_1 ⋯ **task A_n**

```
SCHEME [0≤i<n]
   WHILE True DO
     await-paint Aᵢ code-Aᵢ
   END
```

||

scheduler

```
WHILE True DO
  ITakeSched()
  await_paint sched code_sch
  await_paint sched IRet()
END
```

||

handler H_1 ⋯ **handler H_m**

```
SCHEME [0≤j<m]
   WHILE True DO
    ITake(Hⱼ)
    await_paint Hⱼ code_hⱼ
    await_paint Hⱼ IRet()
   END
```

Variables `AT, ATStack, EI, schedReq`

+

```
IntEnable(X)
IntDisable(X)
SchedEnable()
SchedDisable()

ITake(X)
IRet()
ITakeSched()
```

1. Tasks as 1st class citizens in O-G
   "AWAIT-painting"
2. Controlled interleaving model
   Hardware API model
3. Formal model of eChronos
   to prove that the running task is always the highest-priority runnable task

Instantiation to eChronos

task A_1 … **task A_n** | **scheduler** | **handler H_1** … **handler H_m**

```
SCHEME [0≤i<n]
   WHILE True DO
     await-paint Ai code-Ai
   END
```

||

```
WHILE True DO
  ITakeSched()
  await-paint sched code-shed
  await-paint sched IRet()
END
```

||

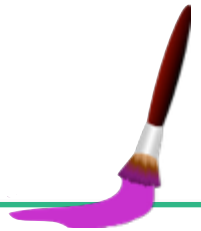
```
SCHEME [0≤j<m]
   WHILE True DO
    ITake(Hj)
    await-paint Hj code_hj
    await-paint Hj IRet()
   END
```

```
code_Ai ≡
  SchedDisable();
  R := changeRunnable(R);
  SchedEnable();
```

```
code_hi ≡
  E := changeEvents();
  schedReq:=True;
```

```
code-sch ≡
    nextT:=None;
    WHILE nextT=None
      DO
        Etmp := E;
        R := handleEvents Etmp R;
        E := E - Etmp;
        nextT:= schedPolicy(R);
      OD;
    context_switch(nextT);
```

Summary



1. Tasks as 1st class citizens in O-G
   "AWAIT-painting"

   `await_paint task_id code`

2. Controlled interleaving model
   Hardware API model

   `AT, ATStack, EI, schedReq`

   ```
   IntEnable(X)
   IntDisable(X)
   SchedEnable()
   SchedDisable()

   ITake(X)
   IRet()
   ITakeSched()
   ```

3. Formal model of eChronos
   to prove that the running task is always the highest-priority runnable task

   $code\_A_i \equiv \ldots$
   $code\_h_i \equiv \ldots$
   $code\_sch \equiv \ldots$

Future work

Tasks as 1st class citizens in O-G

"AWAIT-painting"

Controlled interleaving model

Hardware API model

Formal model of eChronos

to prove that the running task is always the highest-priority runnable task

Prove the model correctly abstracts eChronos code

Prove this property holds on our eChronos model

Informal arguments of validity

AWAIT statements do not exist in the implementation!

➡ only use them to represent atomicity enforced by hardware

Introduced variables do no exist in the implementation!

➡ only modified by hardware API functions

Preliminary arguments of practicality

We have started proving the correctness of the scheduling behaviour

- ➡ proof done* for an initial version of the model
- ➡ ~10,000 verification conditions generated
- ➡ down to ~500 by removing redundant conditions automatically
- ➡ down to ~10 after automatic discharge by Isabelle/HOL

*almost ;-)

Thank you