# Safe Kernel Scheduler Development with Bossa

Gilles Muller

Obasco Group, Ecole des Mines de Nantes/INRIA, LINA

Julia L. Lawall

DIKU, University of Copenhagen

***http://www.emn.fr/x-info/bossa***

# Process scheduling is an old issue, but: there is no single perfect scheduler

- Application requirements:
  - Computation server: fairness
  - Number crunching: ASAP
  - Hard real-time: strict deadlines
  - Multimedia: user perception, relaxed deadlines
  - Embedded systems: energy
- Recent research work
  - OSDI (8), SOSP (4), RTSS (28), Usenix (5)

# Still …

- Very limited impact on commercial OSes
  - Round robin
  - Priority-based
  - Fifo
- Application needs known only by the application (or framework) programmer
  - The OS must be customized to application needs
  - Very few application programmers possess kernel expertise

# Bossa goals

Simplify scheduler development so that an application programmer can safely extend kernel behavior

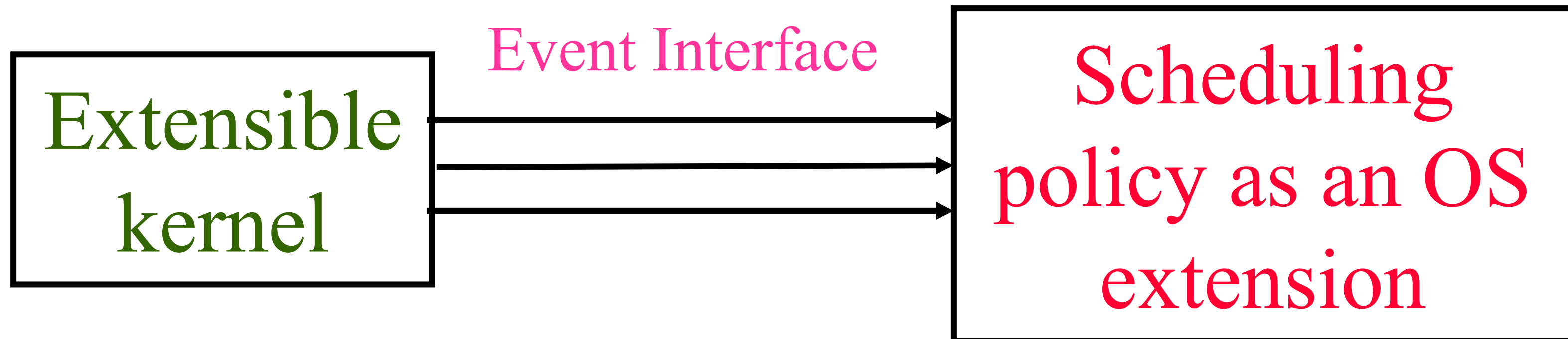
- Predictable development
- Safe development
- Integration within existing OSes

# Issues in customizing the OS

1. How to integrate a scheduling policy into the kernel?
2. How to write a policy?
3. How to verify policy correctness?

# 1- How to integrate new policies in the kernel

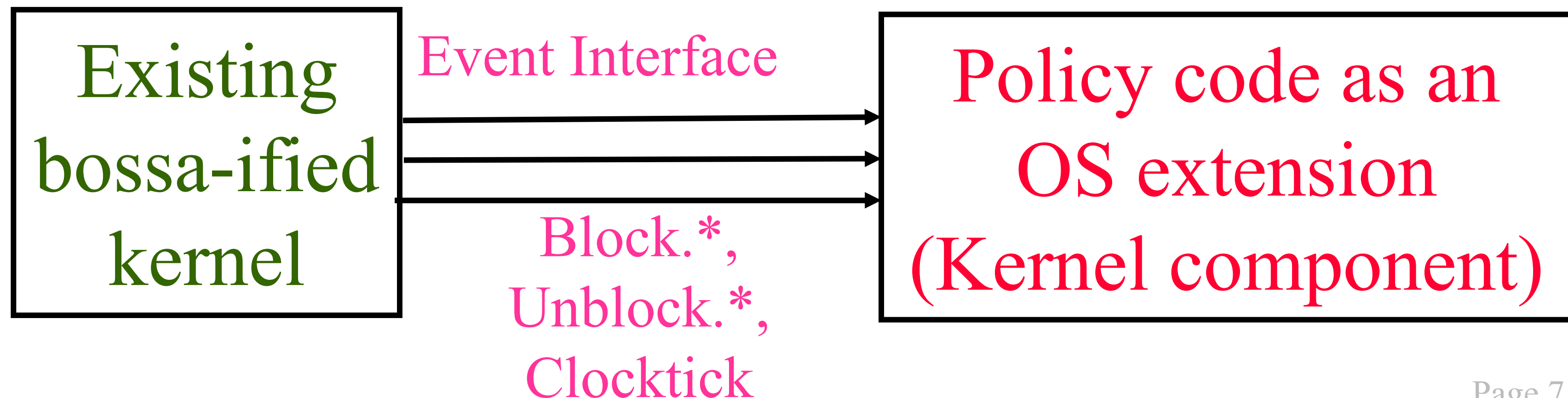
We need an extensible kernel (a la SPIN, exokernel)




- Complex to program
- Research prototypes, limited support (drivers, libraries)

# 1 - How to integrate new policies in the kernel – Bossa approach

- Enrich an existing kernel (Linux, Windows) with a scheduling-specific event interface
- Existing scheduling code removed
- Tool-assisted transformation process using AOP and temporal logic [ASE-2003, EW2004]



# 2 - How to write policies: Kernel development is a nightmare

- C Development is error-prone
- Low-level C code

  => little help from the compiler
- Likely to crash the OS

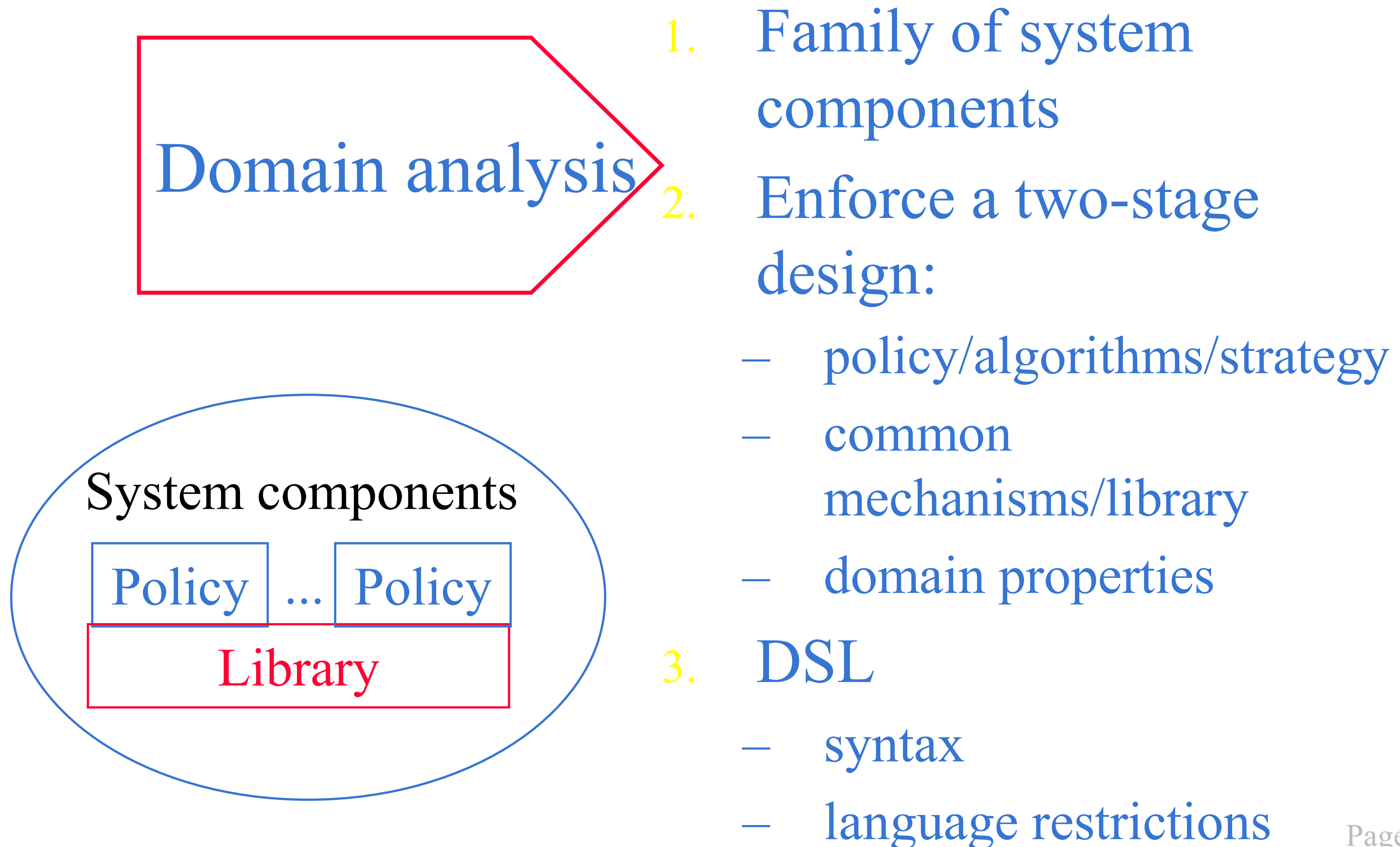
  => test and debug tedious

# 2 - How to write policies
## Capture kernel expertise into a DSL

**A programming language dedicated to a family of programs that offers specific abstractions and notations.**

- Trade expressiveness for expertise/knowledge:
  - Productivity : easier and safer programming
  - Robustness : (static) verification of properties
  - Performance : efficient compilation

# 2 - How to write policies
## Capture kernel expertise into a DSL

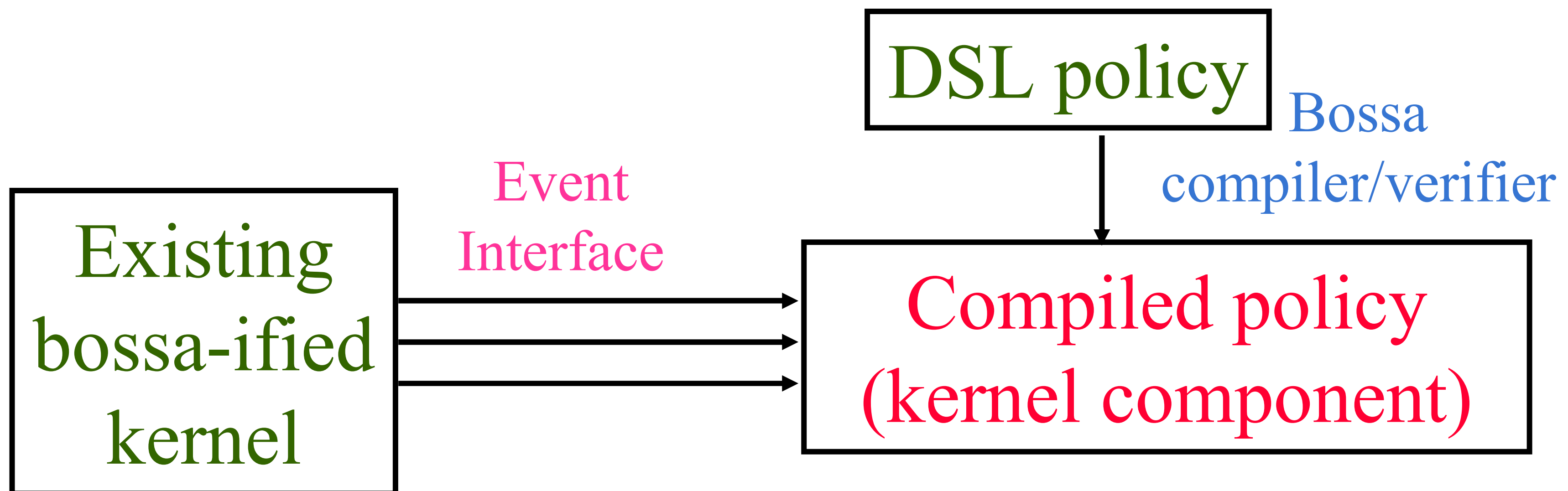


# 2 - How to write policies
# Capture kernel expertise into a DSL

## Benefits of Domain Specific Languages

- Expertise re-use
  - separate What/How
  - expertise repository in underlying kernel mechanisms
- Code re-use
  - well-identified basic mechanisms
  - enforced re-use of the mechanisms
- Program safety and robustness
  - property verification (predictable)
  - enforced correct usage of the mechanisms

# 2 - How to write policies
## Capture kernel expertise into a DSL



# The Bossa DSL

Looks like C but:

- Provides high level abstractions
  - Process attributes
  - Ordering criteria
  - Process states
  - Event handlers
  - Interface functions
- Typical well known scheduling policies are under 200 lines

# Process attributes and priorities

```
scheduler Linux = {

  type policy_t = enum {
      SCHED_FIFO,SCHED_RR,      // RT policies
      SCHED_OTHER               // Round Robin
  };

  process = {
      policy_t policy,
      int      rt_priority,     // 0 for round robin
      int      priority,        // initial time slice
      int      ticks            // current time slice
  };

  ordering_criteria = {
      highest rt_priority, highest ticks
  };
```

# Process states

- Class of state + Process storage

```
states = {
    RUNNING running : process;

    READY   ready   : fifo select queue;
    READY   expired : queue;
    READY   yield   : process;

    BLOCKED blocked : queue;

    TERMINATED terminated;
}
```

# Event handlers

```
handler (event e) {
 …
 On block.* {
  e.target => blocked;
 }

 On unblock.preemptive {
  if (e.target in blocked) {
    e.target => ready;
    if (!empty(running) &&
        (e.target > running))
     running => ready;
  }
 }
…
```

# Event handlers - Schedule

```
On bossa.schedule {
  if (empty(ready)) {
   foreach (p in blocked) { p.ticks = p.ticks/2 +
   (((p.priority)>>2)+1); }
   if (!empty(yield)) {
     yield.ticks = yield.ticks/2 +
   (((yield.priority)>>2)+1);
   }

   if ( empty (expired)) { yield => ready; }
   else {  foreach (p in expired) { p => ready;  } }
    }

    select() => running;

    if  ( !empty(yield)) { yield => ready; }
}
```

# Properties of the Bossa DSL

- Termination
  - Bounded loops
- Complete set of event handlers
- No loss of a process
- Kernel protection w.r.t. crashes

# 3 -How to verify policy correctness?

- Is the implementation consistent?
  - DSL properties

- Does the implementation interact correctly with the target OS?
  - Extensible system development
    - » Kernel expert
    - » Policy programmer
  - Example: do not elect a blocked process

# Event types

For each event, describe:

- Event notification context.
- Expected handler effect.
- block.*: [tgt in RUNNING] -> [tgt in BLOCKED]

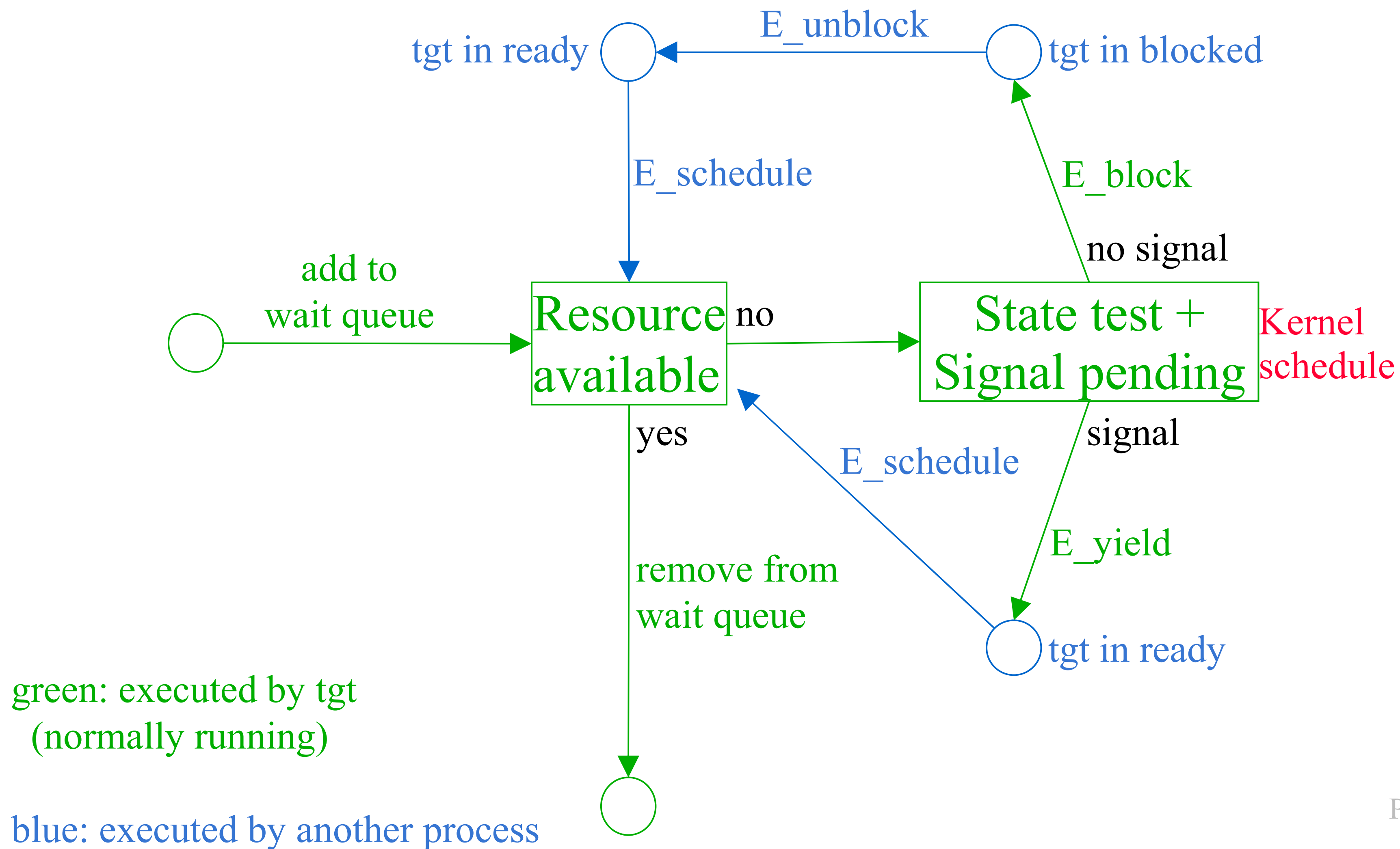
Usage:

- Check that kernel expectations are satisfied at compile time
- Document these expectations.

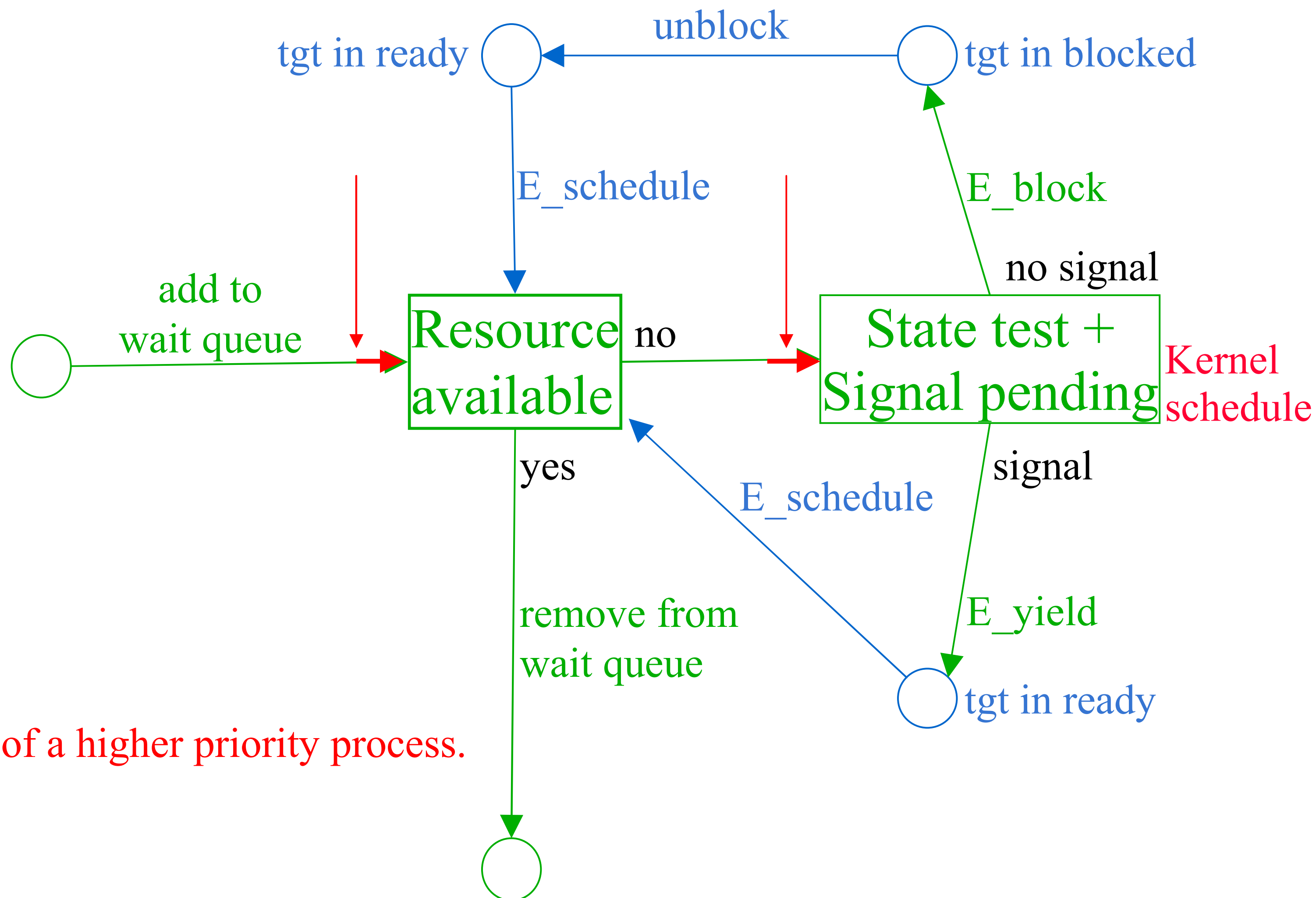
Event types are kernel-specific.

- Written once by the kernel expert.

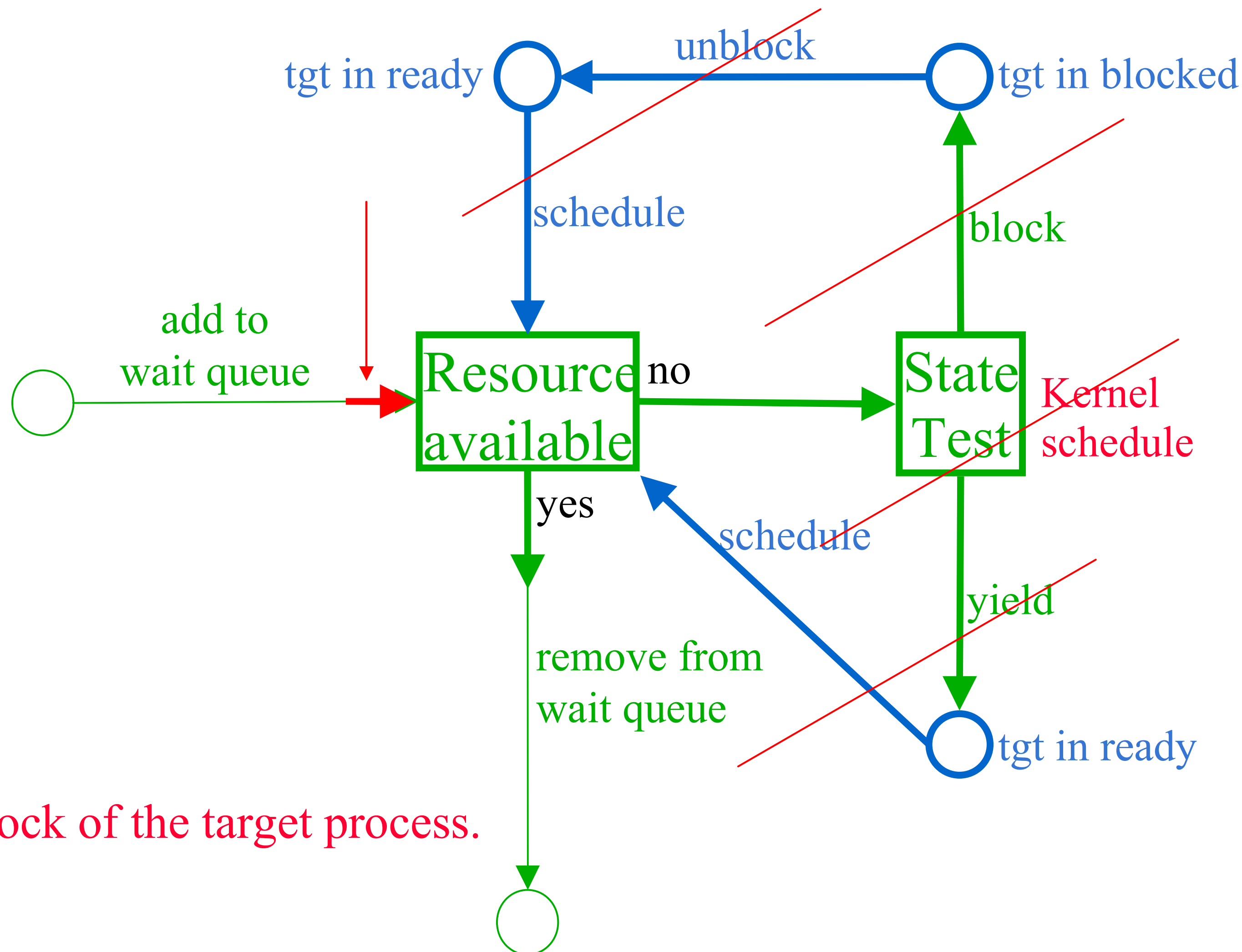
# Blocking in Linux (Tout ce que vous avez toujours voulu savoir… sans oser le demander)



# Taking into account interrupts: Target of block might not be running

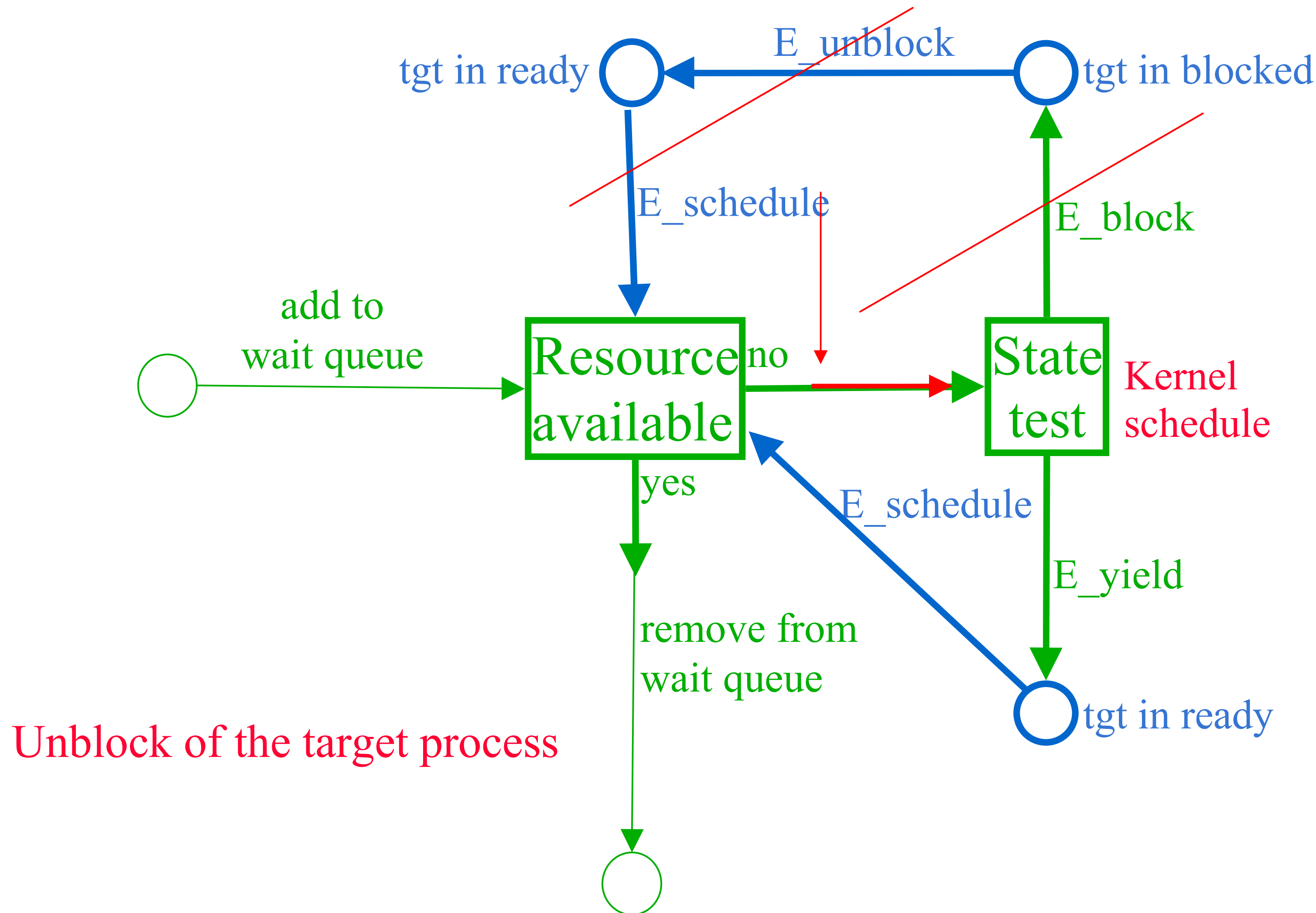


# Taking into account interrupts:
# Target of unblock might not be blocked




Unblock of the target process.

# Unblock of the target process:
# Target of unblock might not be blocked



# Linux event types (kernel expert)

- unblock.preemptive:
  - [[] = RUNNING, tgt in BLOCKED] ->
    [[] = RUNNING, tgt in READY]
  - [p in RUNNING, tgt in BLOCKED] ->
    {[p in RUNNING, tgt in READY],
    [[p, tgt] in READY]}
  - [tgt in RUNNING] -> [tgt in RUNNING]
  - [tgt in READY] -> [tgt in READY]
- block.*:
  - [tgt in RUNNING] -> [tgt in BLOCKED]
  - [[] = RUNNING, tgt in READY] -> [tgt in BLOCKED]

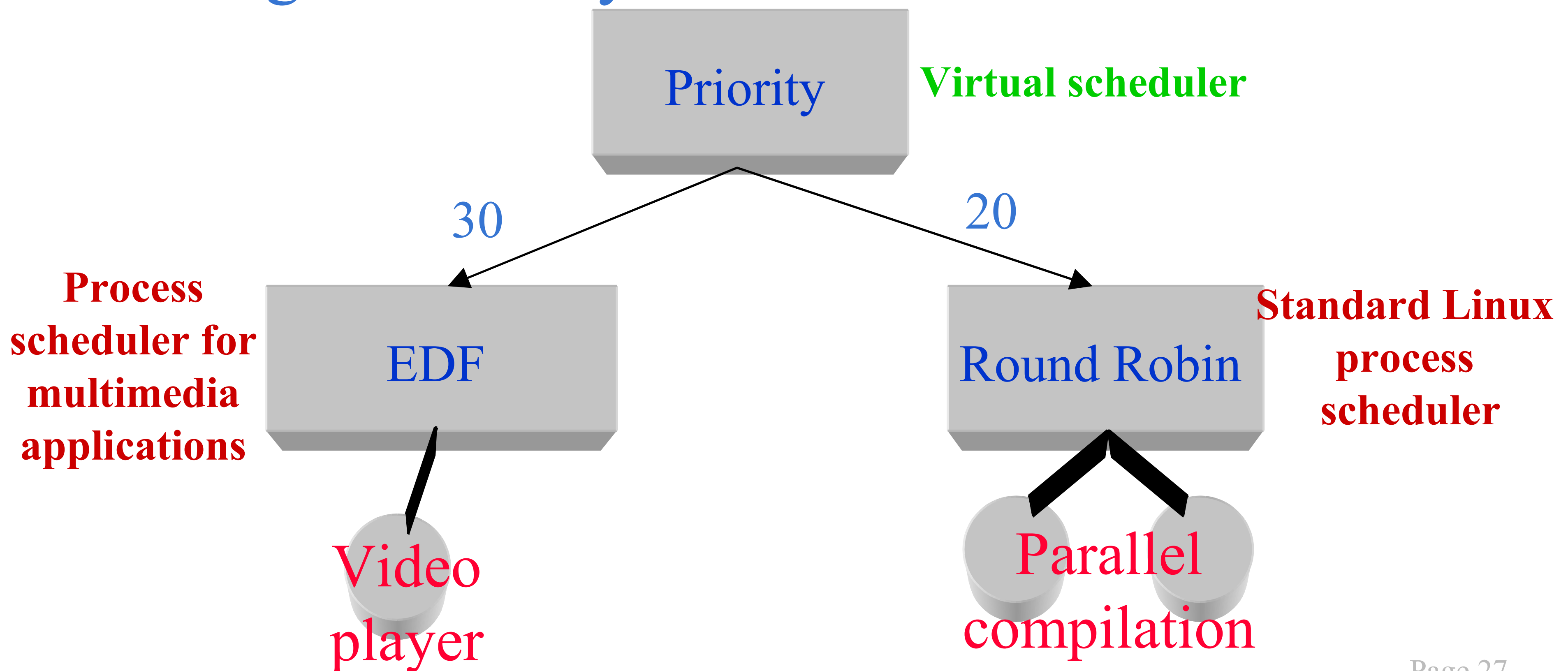
# Bossa evaluation

- Benefit of new policies
  - QoS for a video player on a highly loaded machine
  - Precise control of CPU usage for legacy applications (web servers)

- Performance overhead w.r.t. the original Linux kernel
  - LMbench micro-benchmark
- Impact of context switches on legacy applications
  - Web server - Apache

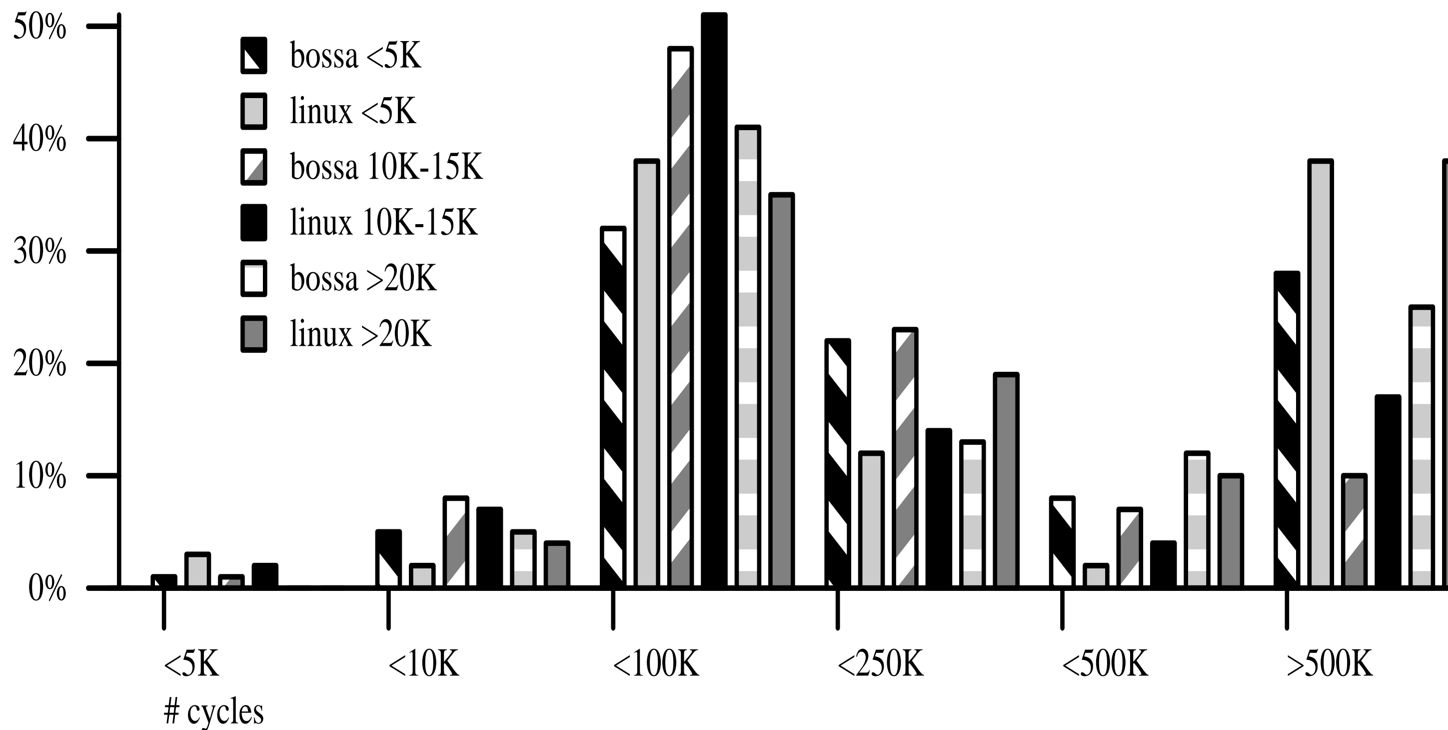
# QoS for multimedia applications

Managing several classes of applications using a hierarchy of schedulers

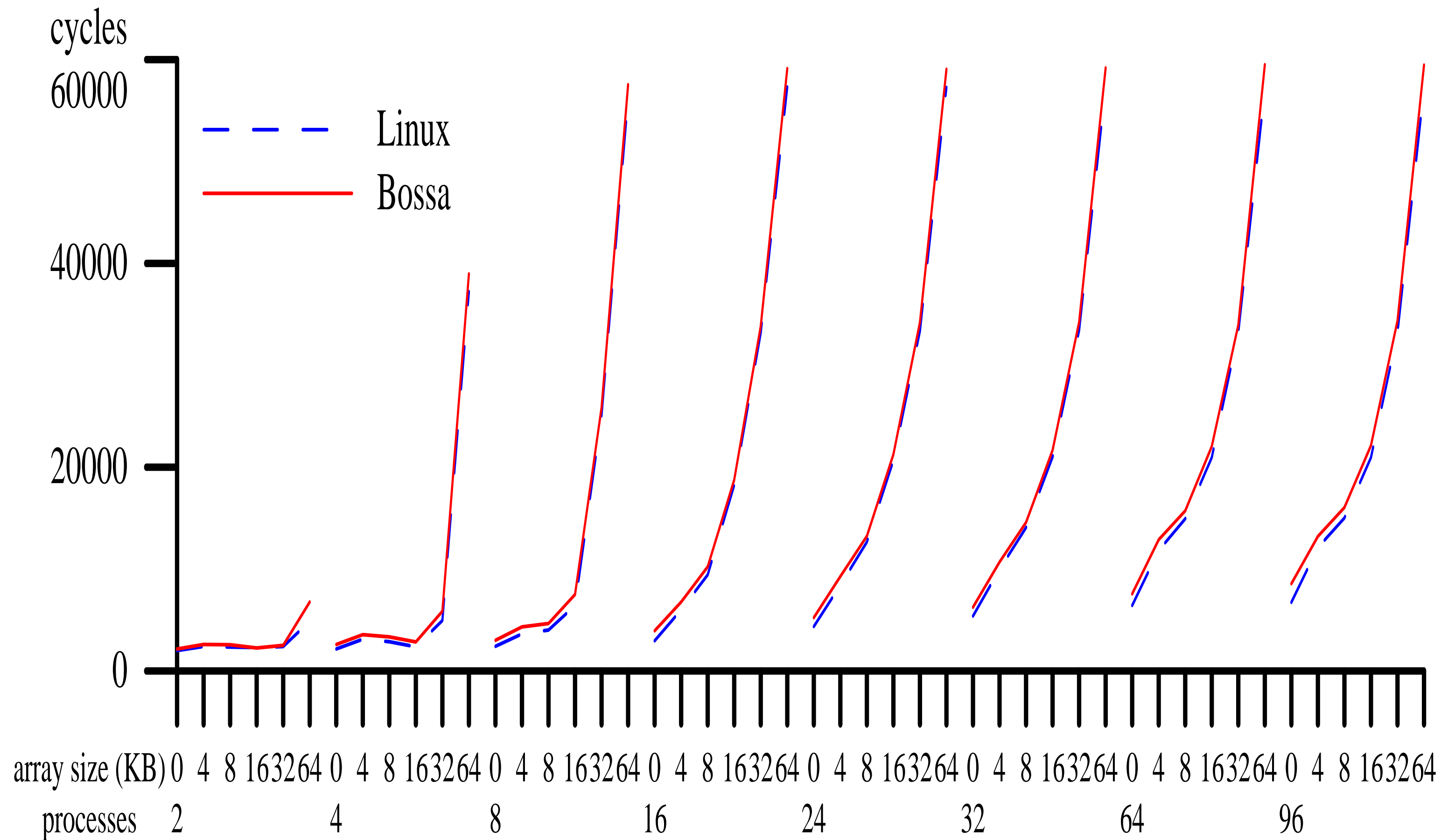


# Impact of context switches on Apache

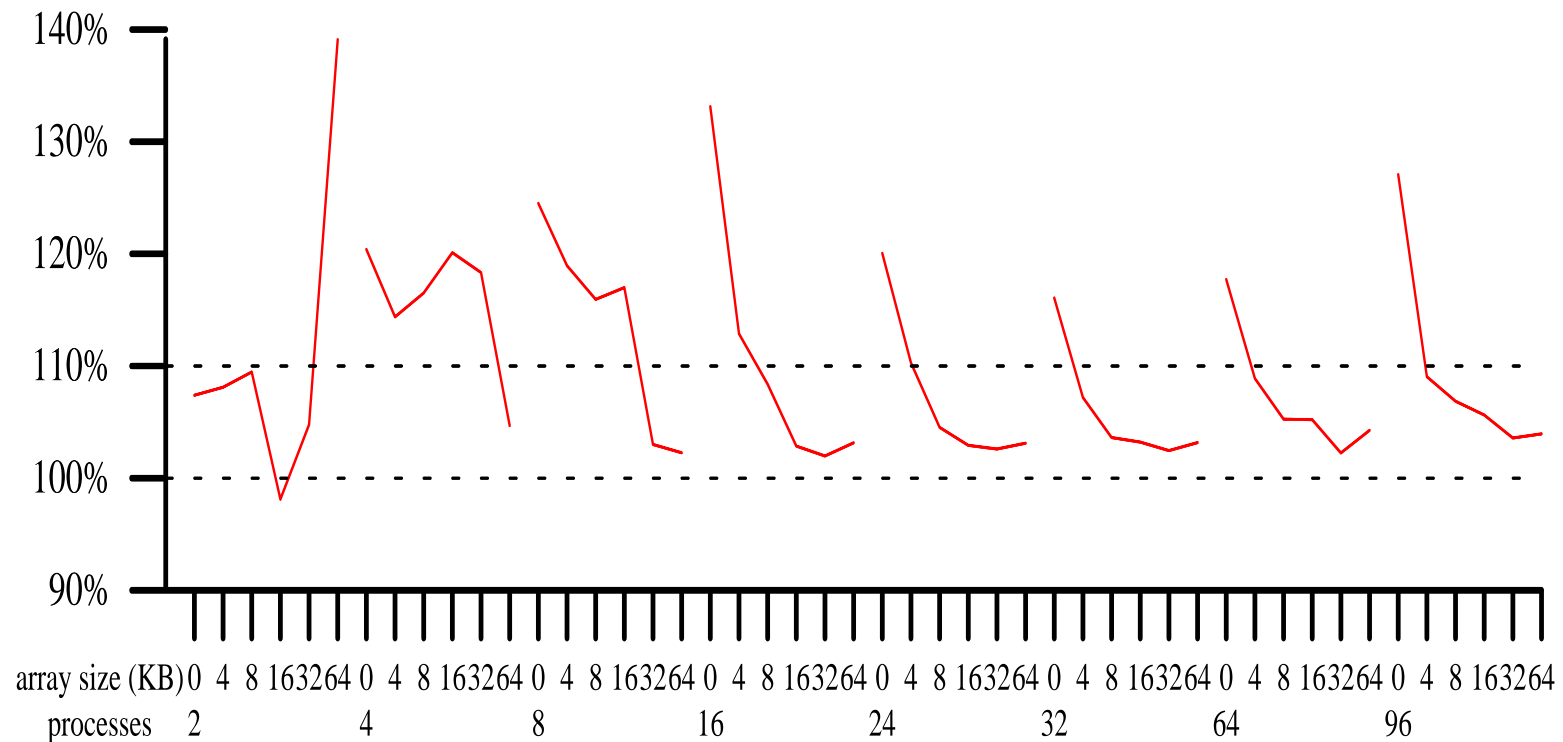
Same number of req/s on Linux & Bossa (1160 req/s, 5kb pages)



# LMbench - Absolute overhead Bossa2.4/Linux 2.4



# LMbench - relative overhead Bossa2.4/Linux 2.4



# On-going work

- Encyclopedia of scheduling policies (Bossa Nova)
- Bossa-Box: Personal Video Recorder with QOS
- Generalization to other resources
  - Energy management (R. Urunuela)
  - Multi-OS generalized approach (C. Augier)

- Port to Windows XP…
- Port to Jaluna/Chorus (RT kernel)
- Port to the 2.6 linux kernel

# Conclusion

Programming scheduling policies is now possible for non kernel experts

- Dissemination of research work
- Nice support for teaching scheduling
- Verification of safety properties
  - Confidence in system behavior
- Event types document kernel behavior

# PUB !

2.4/2.6 bossa-linux kernel,
Teaching lab,
Bossa-Knoppix,

**http://www.emn.fr/x-info/bossa**

# Re- PUB !

- 8 Octobre – EuroSys

- 23-26 Octobre – SOSP, Brighton
- 28-30 Novembre – Middleware, Grenoble

- 3-7 Juillet 2006 ECOOP, Nantes – 20 ans