Real-time system analysis using tracing and sampling data

François Rajotte

Laboratoire DORSAL
Département de génie informatique

POLYTECHNIQUE
MONTRÉAL

AFFILIÉE À
L'UNIVERSITÉ DE MONTRÉAL

UT TENSIO SIC VIS

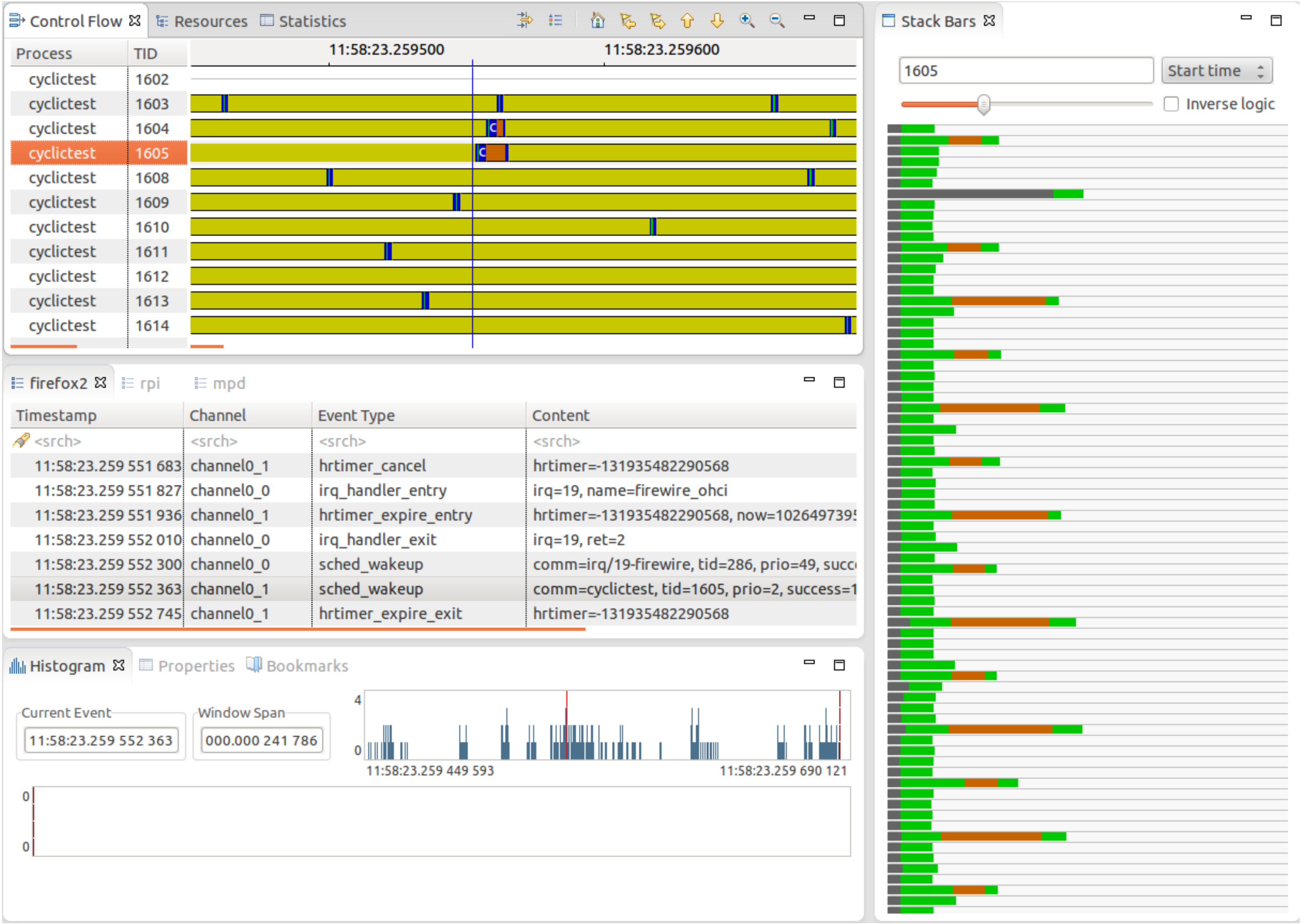
Outline

- Context
- Previous work
- Limitations and improvements
- Example

Context

- Tracing and real-time applications
  - Low-overhead system observation
  - Provides detailed information
- Challenges
  - Extracting meaningful data
    - Statistics, abstraction
  - Facilitate user exploration
    - Tools, viewers

Example



Process	TID
cyclictest	1602
cyclictest	1603
cyclictest	1604
cyclictest	1605
cyclictest	1608
cyclictest	1609
cyclictest	1610
cyclictest	1611
cyclictest	1612
cyclictest	1613
cyclictest	1614

Timestamp	Channel	Event Type	Content
<srch>	<srch>	<srch>	<srch>
11:58:23.259 551 683	channel0_1	hrtimer_cancel	hrtimer=-131935482290568
11:58:23.259 551 827	channel0_0	irq_handler_entry	irq=19, name=firewire_ohci
11:58:23.259 551 936	channel0_1	hrtimer_expire_entry	hrtimer=-131935482290568, now=102649739
11:58:23.259 552 010	channel0_0	irq_handler_exit	irq=19, ret=2
11:58:23.259 552 300	channel0_0	sched_wakeup	comm=irq/19-firewire, tid=286, prio=49, succ
11:58:23.259 552 363	channel0_1	sched_wakeup	comm=cyclictest, tid=1605, prio=2, success=1
11:58:23.259 552 745	channel0_1	hrtimer_expire_exit	hrtimer=-131935482290568

Current Event: 11:58:23.259 552 363

Window Span: 000.000 241 786

1605

☐ Inverse logic

Analysis

- Separating a process into individual tasks
  - Benefits
    - Extract statistics
    - Specialized display
  - Challenges
    - Reduce user input
    - Improve automatic detection

Process blocking and wakeup

- Basic approach
  - Wakeup event = start of a new task
- Limitations
  - Spurious wakeups
  - Blocking can have many causes
    - Resource sharing, synchronisation
    - Interleaved with the execution of a task

Real-time priority

- Special scheduling algorithm
  - Usually reserved for real-time tasks
  - Requires special privileges
  - Schedules tasks according to their absolute priority (0-100)
- Priority inheritance
  - Limits priority inversion scenarios
  - Implemented via POSIX mutexes
  - Choice between inheritance and ceiling

Blocking and preemption

- Blocking
  - Process stops executing and cannot resume until explicitly woken up by an external event
  - Happens only in system calls
- Preemption
  - Process stops executing because the kernel decides that another process should be executing instead
    - Fair share of CPU time
    - Higher priority process
  - Can happen in both kernel and user land

Blocking and preemption

- The highest priority runnable process is always executing
- Two events can change that
  - A higher priority process becomes runnable. The current process gets preempted.
  - The current process blocks. The next highest-priority runnable process starts executing.
    - With priority inheritance, this new process' priority is also boosted

Analysis

- With only the different processes' priority, we can tell whether:
  - A process has been preempted, if
    - The new executing process has high priority
  - A process has been blocked, if
    - The new executing process has lower or equal priority
    - No other process is executing
  - With the sched_pi_setprio event, it is also possible to see when a process' priority is boosted

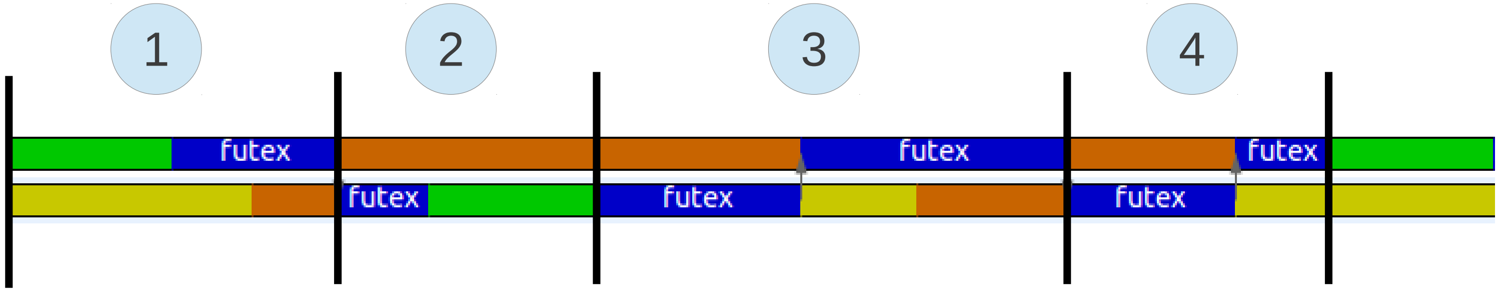
Producer-consumer example

- Implemented using semaphores
  - Does not use priority inheritance
  - Consumer is higher priorty
  - Producer is lower priority
- Expectations:
  - Buffer is always empty
  - The consumer is always preempting the producer

Producer-consumer example

- Four step process
  - Production (1)
  - Consumption (2)
  - 2 extra steps? (3-4)

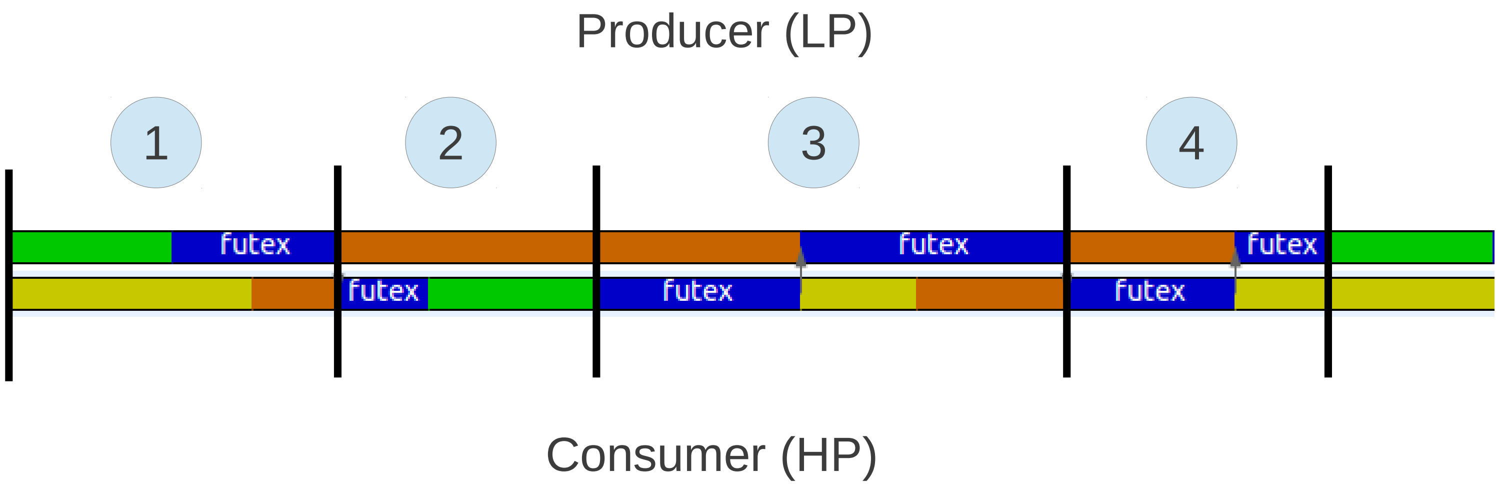
Producer (LP)



Consumer (HP)

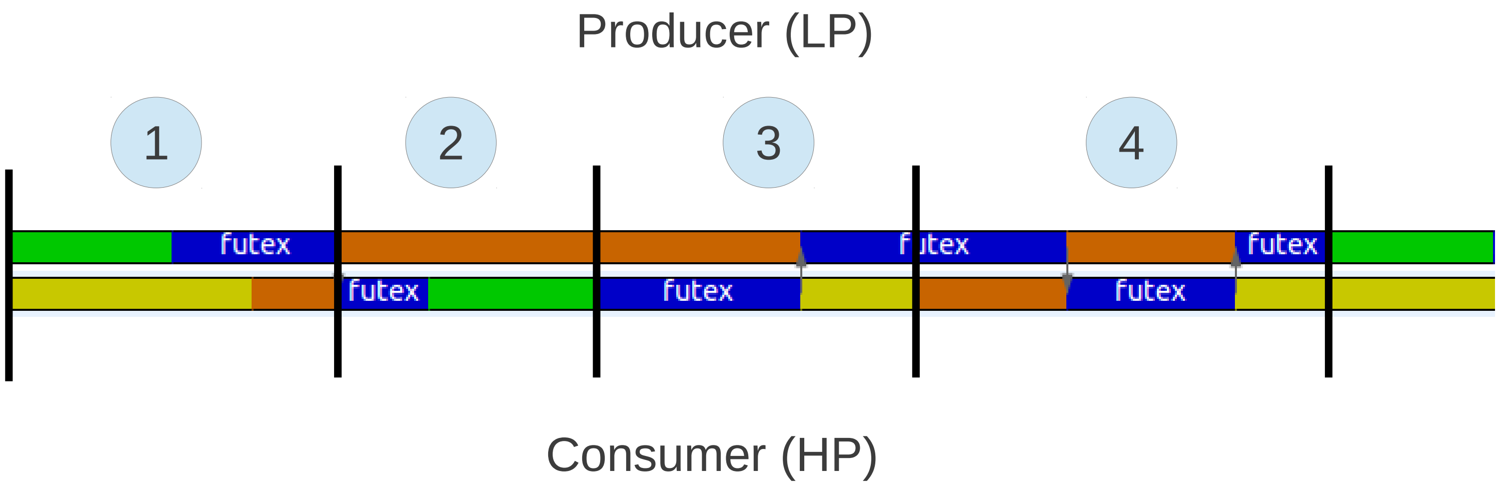
Producer-consumer example

- Step 1
  - Producer is filling buffer
  - Producer wakes up consumer because data is ready



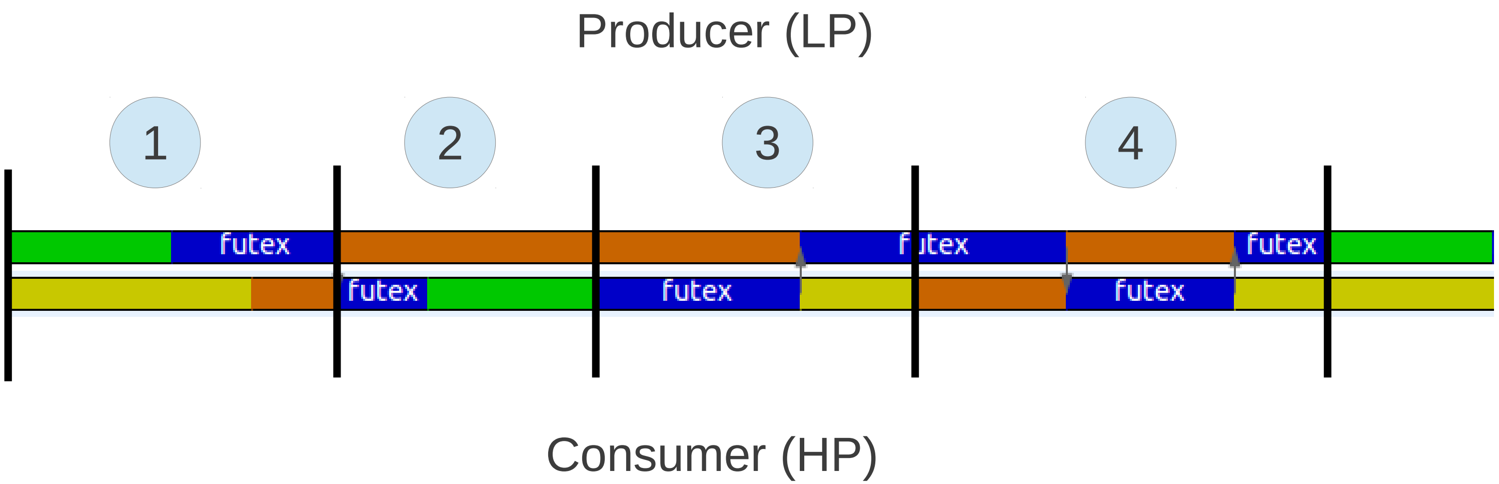
Producer-consumer example

- Step 2
  - Consumer grabs the CPU and starts consuming
  - Producer is preempted



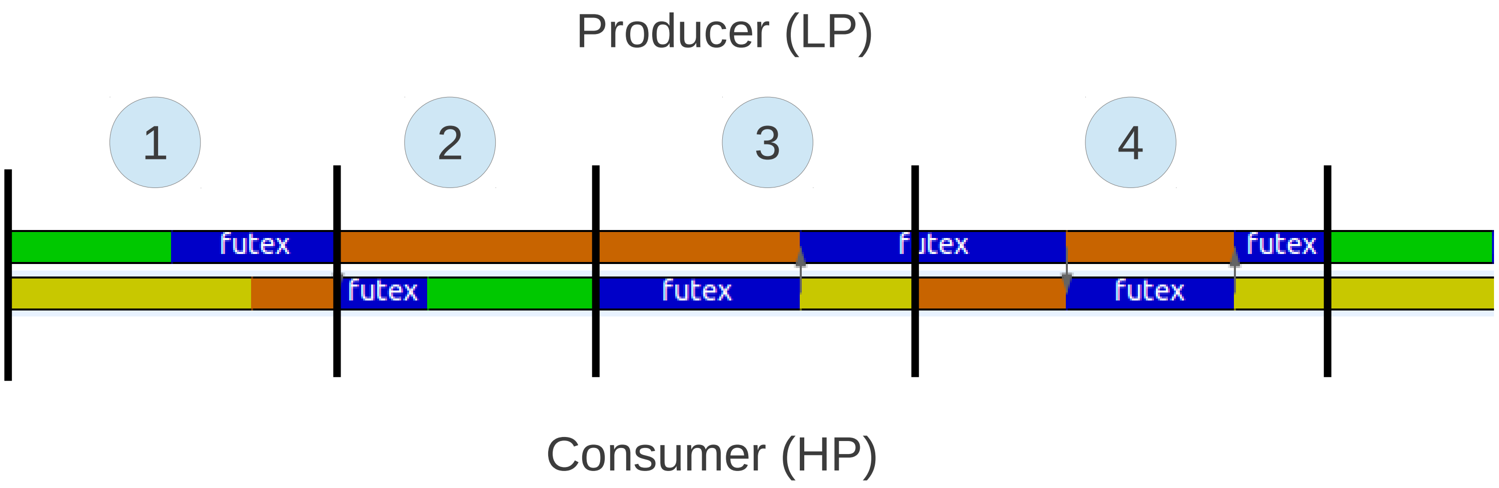
Producer-consumer example

- Step 3
  - Consumer has consumed everything and tries to wait
  - Producer is still holding an internal kernel lock from its futex call preventing the consumer from completing its call
  - Consumer boosts producer's priority to help it complete its call



Producer-consumer example

- Step 4
  - Producer releases its lock and wakes up consumer
  - Consumer is executed and can finally block
  - Producer completes its futex call and starts the cycle again



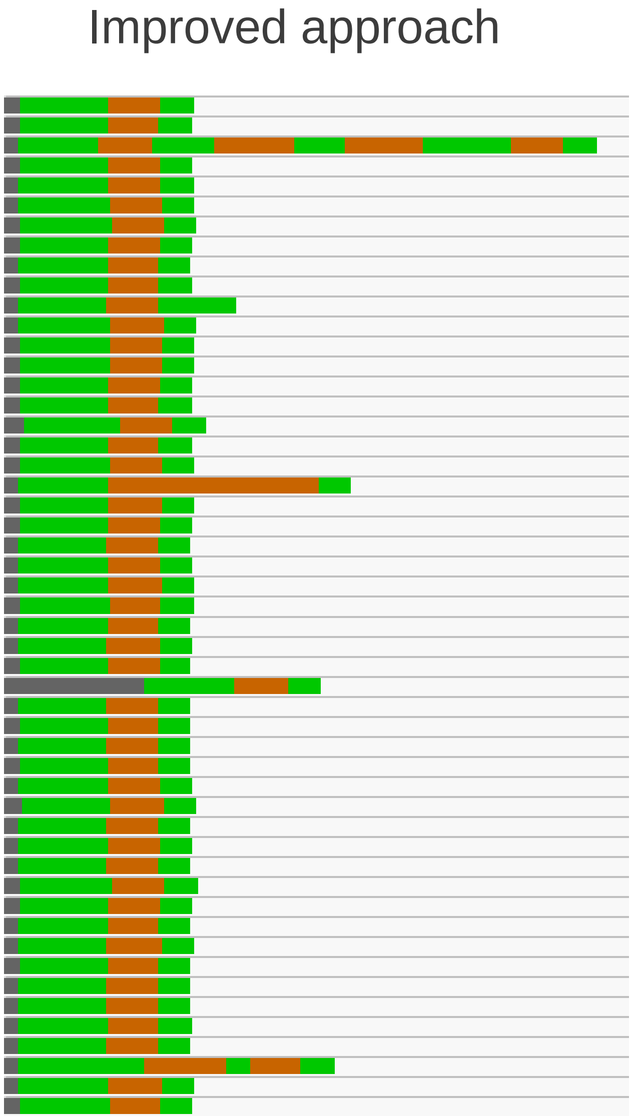
Notes

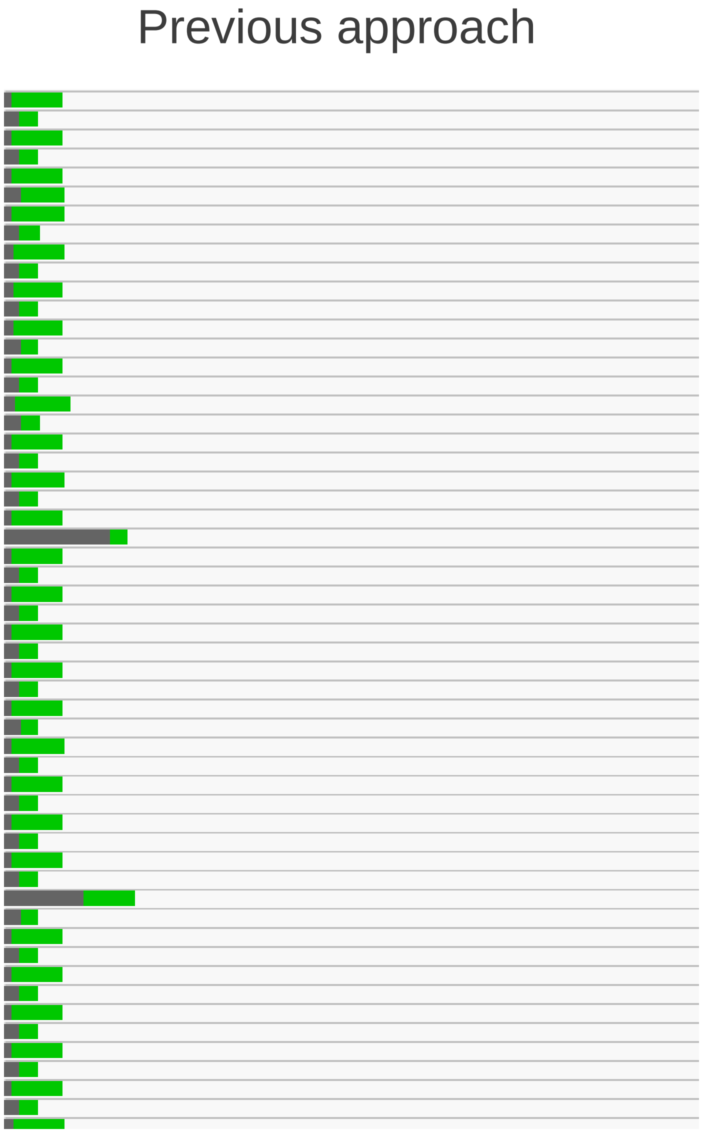
- Producer is never blocked, only preempted
- Consumer is blocked twice per period
- Priority boosting can happen without explicit user consent

Lessons

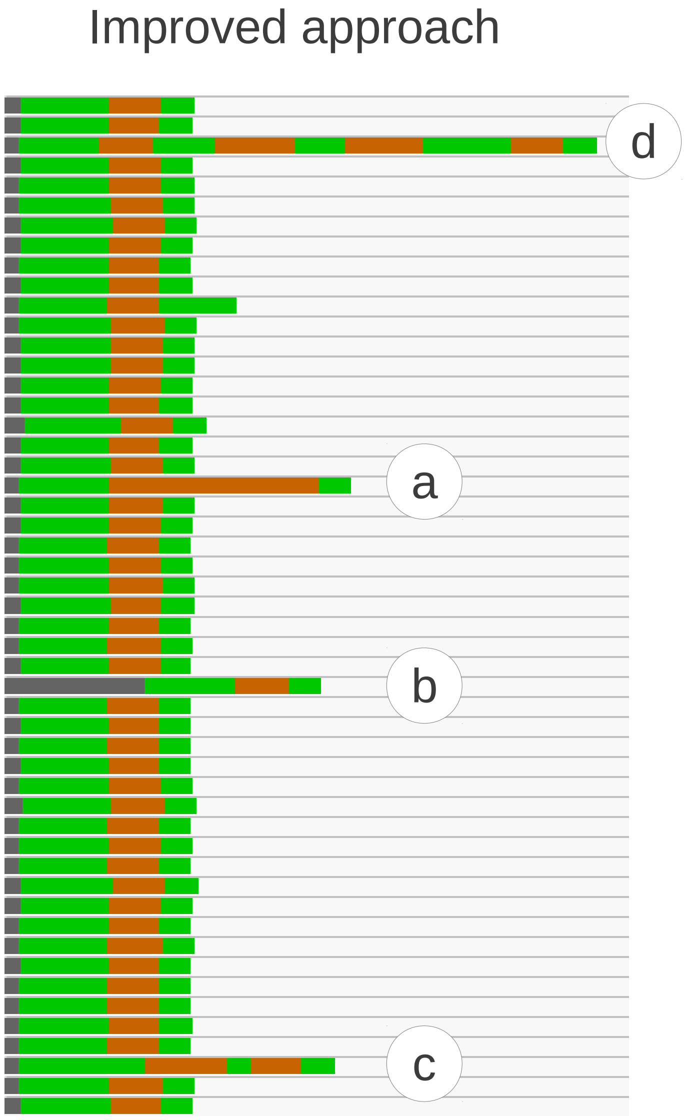
- Blocking should be categorized
  - “Planned” blocking
    - Input/output operation
    - Timer expiration
  - “Unplanned” blocking
    - Mutex contention
- Use the “planned” blockings to help split a process in repeating periods

New approach example





New approach example (explained)



- Basic case
  - One mutex contention
- Other cases :
  - Preempted by a higher priority process:
    - While mutex is contested (a)
    - Before the start of execution (b)
    - While mutex is not contested (c)
  - The higher priority process blocks on another mutex (d)

Conclusion

- Process separated to form individual tasks
  - Using kernel events with no additional instrumentation
  - Allows for better analysis tools for real-time processes
    - Statistics gathering
    - Specific views
- Future work
  - Support for user-defined filters
  - Robust integration with TMF