librtpi

Conditional Variables for Real-Time Applications

Grațian Crișan

gratian.crisan@ni.com,
gratian@gmail.com

About me

- I work for NI (formerly known as National Instruments)
  - makes hardware & software for test, measurement, and automation
- Real-Time OS group for the past decade
  - PREEMPT_RT based Linux kernels
  - ARM and Intel x86_64 architectures
  - distribution based on OpenEmbedded/Yocto
- Maintainer for the Linux kernel shipping on our RT hardware
- Often debug nasty RT issues (too often related to locking primitives)

Agenda

Real-Time concepts

Conditional variables and monitors

Problems with conditional variable in libpthread

Librtpi (re)implementation of condvars

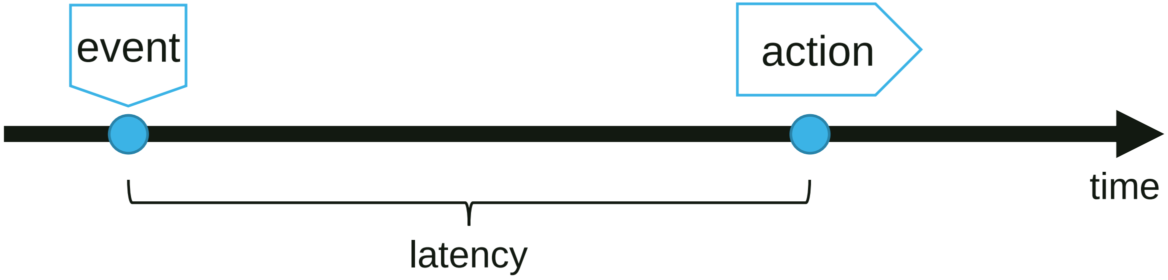
Future ideas and questions

Real-Time

Real World

Any system that interacts with the real, physical world must synchronize with it

Deterministic response to stimulus



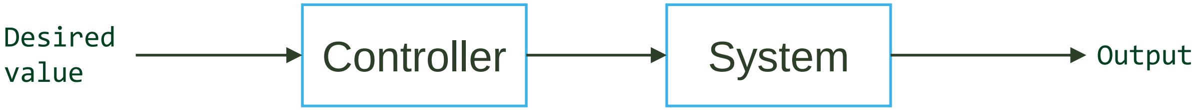
Events can be:

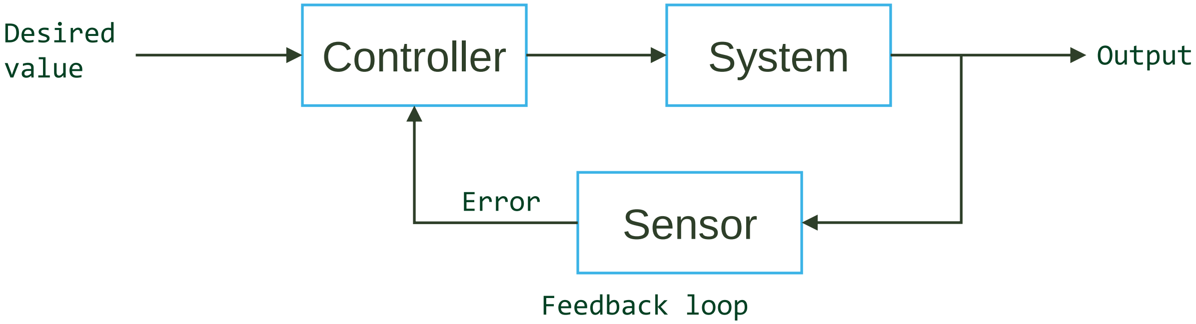
- Asynchronous
- Synchronous (clock driven)

We want the latency to be:

- Predictable
- Bounded

Traditional Real-Time applications

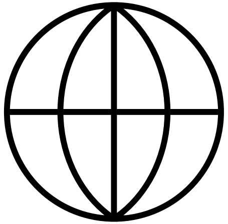




Real-Time applications today

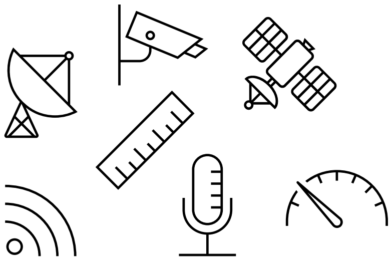
Sensory processing
- sensor fusion
- complex filters
- image recognition
- classification
- estimation

World Model



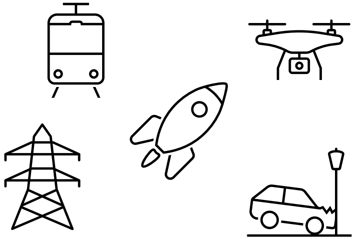
Behavior generation
- planners
- executors
- AI in the loop
- HIL in the loop

Complex sensors



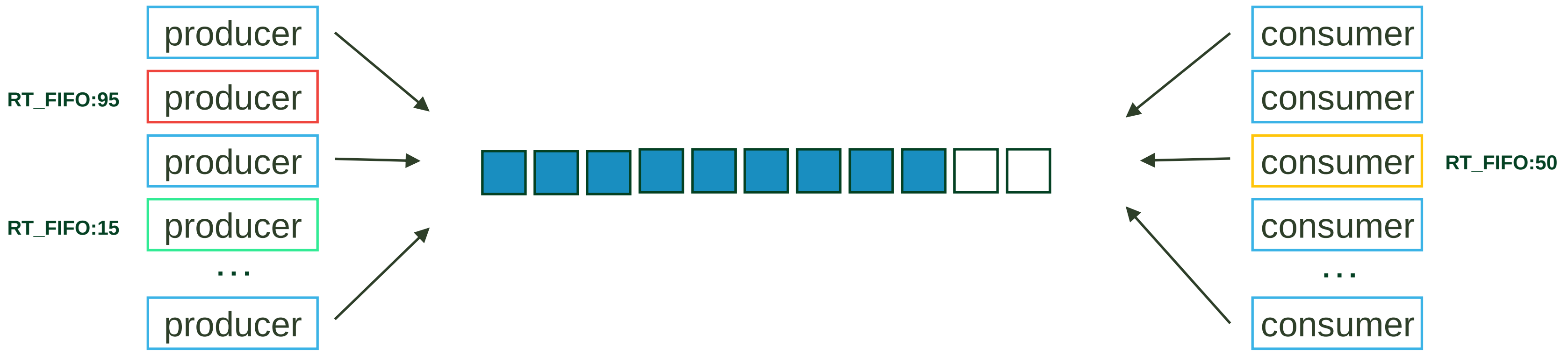
Real World

Complex actuators



Data in Real-Time applications

Solving the bounded multiple producer/consumer problem with RT constraints



How can a thread wait for a condition to be true?

- Spinning until condition becomes true
  - very inefficient (wastes CPU cycles)
  - can live-lock a CPU when used with RT threads
- Explicit queue
  - threads can put themselves on when some state of execution is not as desired (by waiting on the condition)
  - some other thread, when it changes said state, can wake one or more waiting threads (by signaling the condition)

Conditional Variable

A synchronization primitive that provides a queue for threads waiting for a resource.

Operations:

- **wait** - add calling thread to the queue and put it to sleep (potentially with a timeout)
- **signal** - remove a thread from the queue and wake it up
- **broadcast** - remove and wake-up all threads on the queue

Monitor

A synchronization construct that allows threads to have both mutual exclusion and the ability to wait for a certain condition.

Composed of:

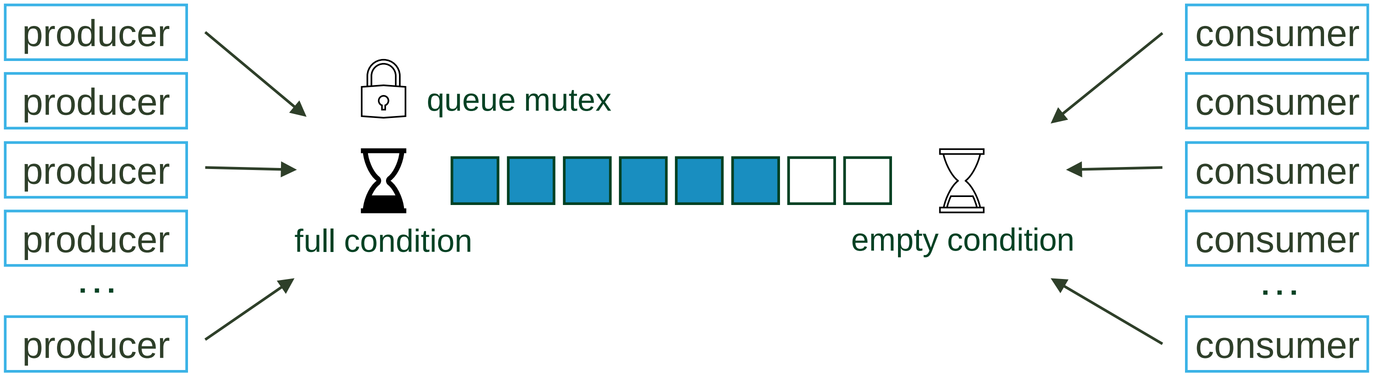
- a lock object - provides the mutual exclusion (mutex)
- one or more condition variables - provides the queues to wait on after atomically releasing the mutex

Higher-level languages (e.g. C#, D) support monitors natively.

In C/C++ they must be constructed from a mutex and conditional variables.

Monitor design rule

Multiple condition variables can be associated with the same mutex, but not vice versa.



Hoare-style monitors (most theory)

- Signaler passes lock to waiter
- Waiter runs immediately
- Condition is guaranteed to hold while waiter runs
- Waiter gives lock back to signaler when it exits the critical section or if it waits again

Mesa-style monitors (most real OSes)

- Signaler keeps lock
- Waiter simply put on ready queue
- Might have to wait for the lock again
- Must recheck condition

Making a resource available (Mesa-style)

```
lock(mutex)
...
/* make resource available */
...
signal(cond)
/* or broadcast(cond) */

unlock(mutex)
```

Waiting for a resource (Mesa-style)

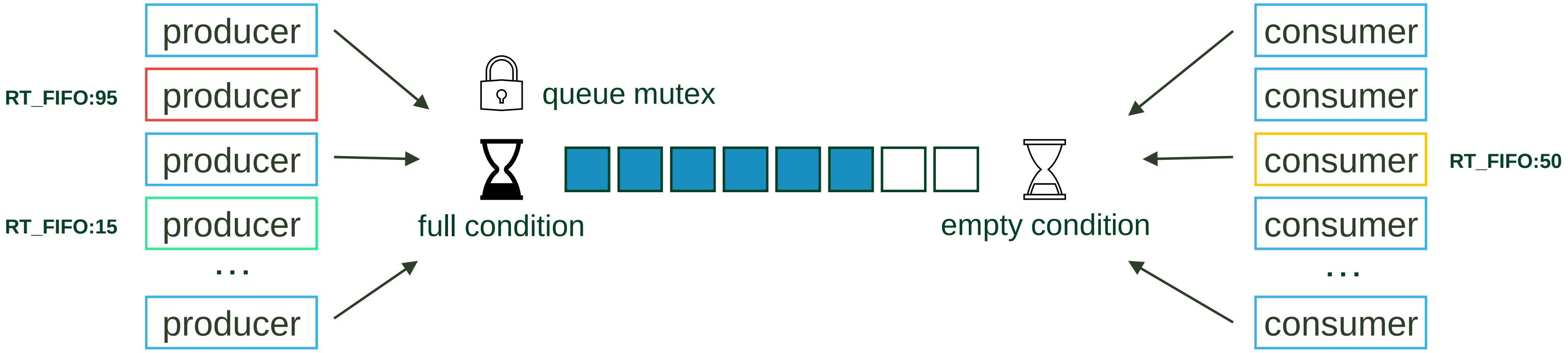
```
lock(mutex)
while (no_resource)
    wait(cond, mutex)
...
/* after wait we own the mutex
and can use the resource */
...
unlock(mutex)
```

while loop necessary due to allowed spurious wake-ups

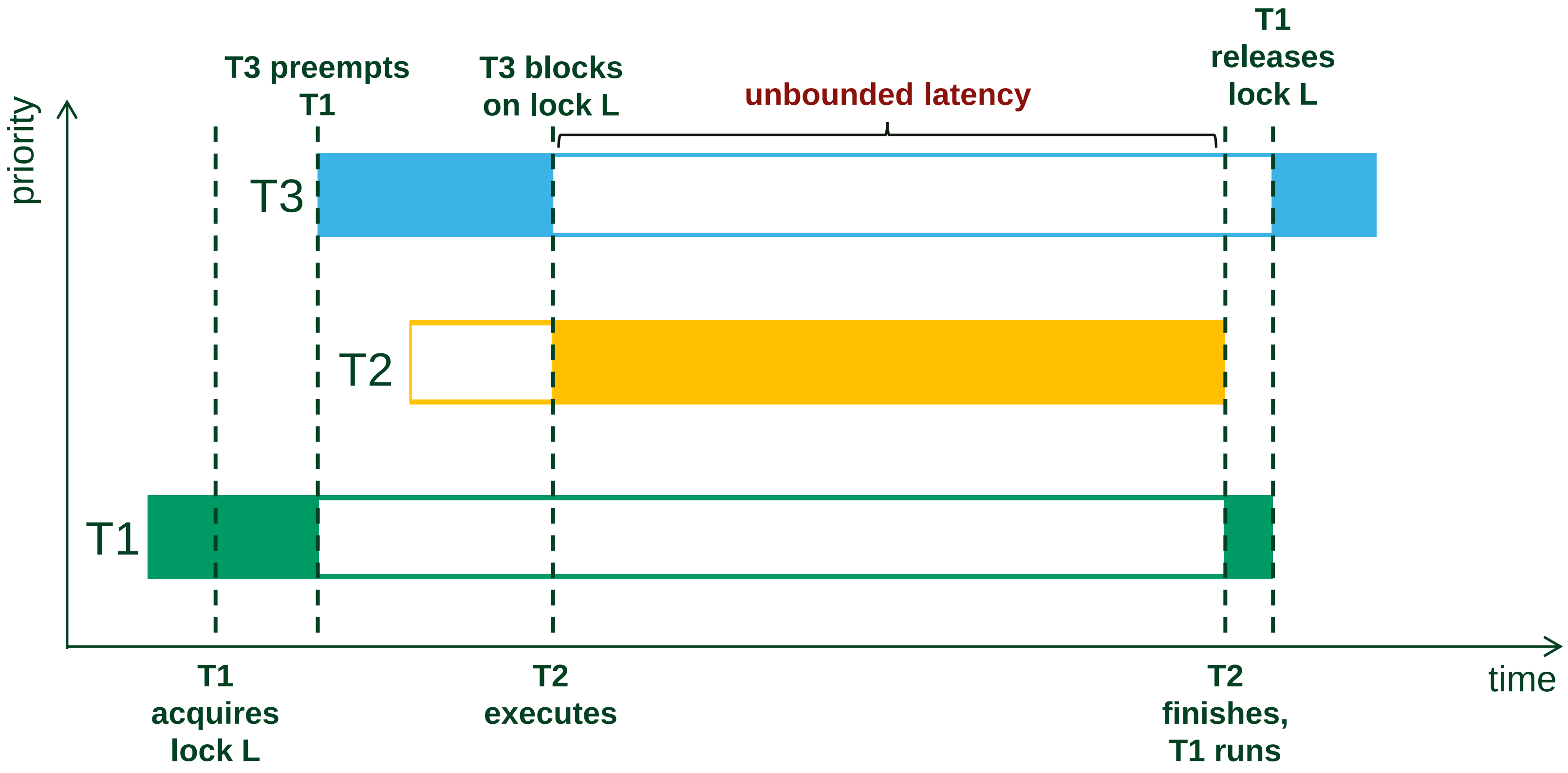
atomically releases the mutex and waits on cond

Monitor Real-Time design constraint

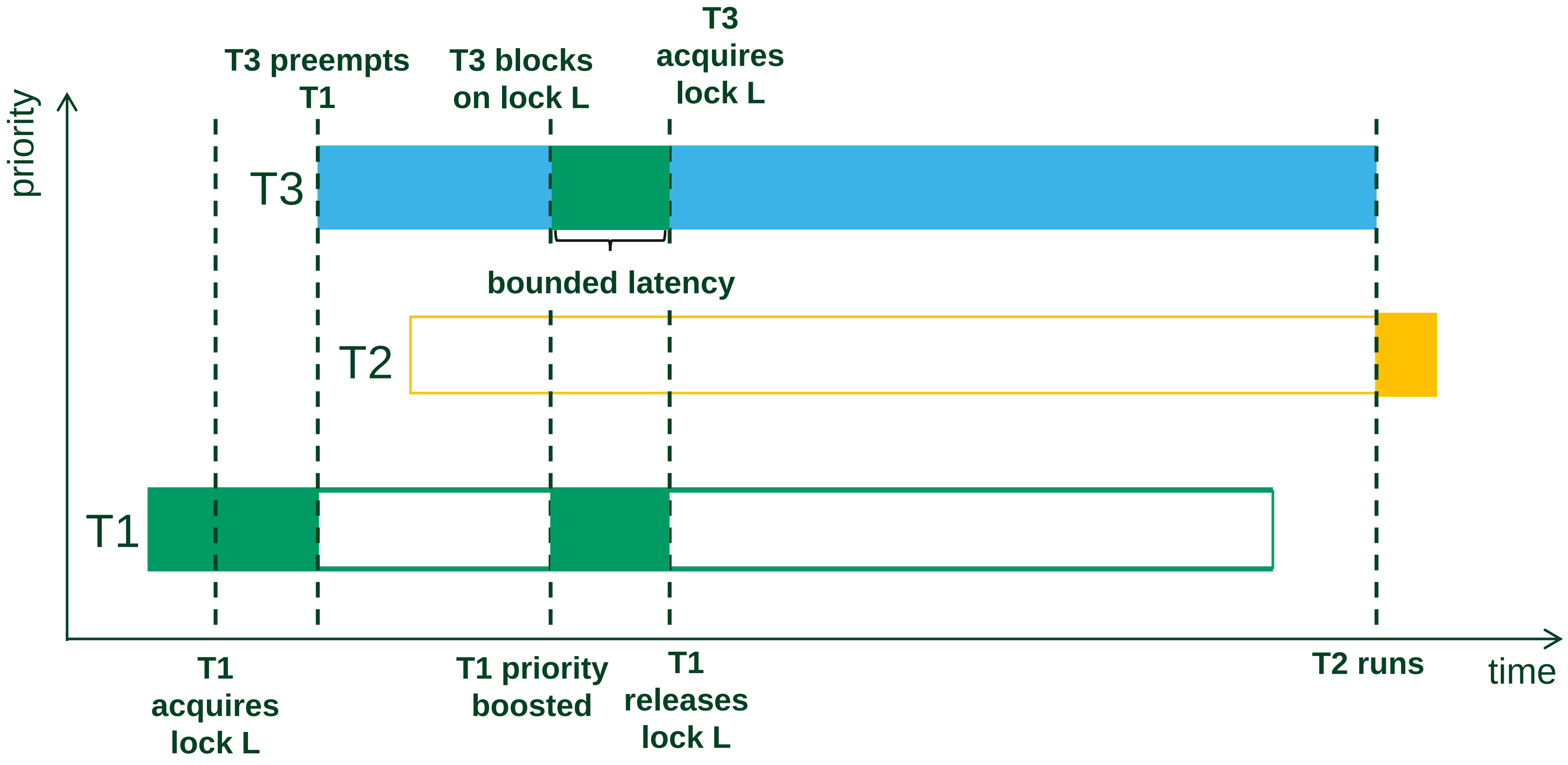
Threads are woken in priority order



Priority inversion



Priority inheritance



Bug 11588 - pthread condvars are not priority inheritance aware

Status:	NEW	**Reported:**	2010-05-11 18:45 UTC by Darren Hart
		Modified:	2019-11-18 03:52 UTC (History)
Alias:	None	**CC List:**	17 users (show)
Product:	glibc	**See Also:**	
Component:	nptl (show other bugs)	**Host:**	
Version:	2.12	**Target:**	
		Build:	
Importance:	P2 enhancement	**Last reconfirmed:**	
Target Milestone:	---		
Assignee:	Not yet assigned to anyone	**Flags:**	fweimer: security-
URL:			
Keywords:			
Depends on:			
Blocks:			

https://sourceware.org/bugzilla/show_bug.cgi?id=11588 gratian@quark:~

```
       FUTEX_CMP_REQUEUE_PI (since Linux 2.6.31)
              This  operation is a PI-aware variant of FUTEX_CMP_REQUEUE.  It requeues waiters
              that are blocked via FUTEX_WAIT_REQUEUE_PI on uaddr from a non-PI  source  futex
              (uaddr) to a PI target futex (uaddr2).

              As with FUTEX_CMP_REQUEUE, this operation wakes up a maximum of val waiters that
              are waiting on the futex at uaddr.  However, for  FUTEX_CMP_REQUEUE_PI,  val  is
              required  to be 1 (since the main point is to avoid a thundering herd).  The re-
              maining waiters are removed from the wait queue of the source futex at uaddr and
              added to the wait queue of the target futex at uaddr2.

              The val2 and val3 arguments serve the same purposes as for FUTEX_CMP_REQUEUE.

       FUTEX_WAIT_REQUEUE_PI (since Linux 2.6.31)
              Wait on a non-PI futex at uaddr and potentially be requeued (via a FUTEX_CMP_RE-
              QUEUE_PI operation in another task) onto a PI futex at uaddr2.  The wait  opera-
              tion on uaddr is the same as for FUTEX_WAIT.

              The  waiter  can  be removed from the wait on uaddr without requeueing on uaddr2
              via a FUTEX_WAKE operation in another task.  In this  case,  the  FUTEX_WAIT_RE-
              QUEUE_PI operation fails with the error EAGAIN.

              If timeout is not NULL, the structure it points to specifies an absolute timeout
              for the wait operation.  If timeout is NULL, the  operation  can  block  indefi-
              nitely.
 Manual page futex(2) line 489 (press h for help or q to quit)
```

Bug 13165 - pthread_cond_wait() can consume a signal that was sent before it started waiting

Status: RESOLVED FIXED

Alias: None

Product: glibc
Component: nptl (show other bugs)
Version: 2.14

Importance: P2 normal
Target Milestone: 2.25
Assignee: Torvald Riegel

URL:
Keywords:

Depends on:
Blocks:

Reported: 2011-09-07 19:14 UTC by Mihail Mihaylov
Modified: 2017-01-01 21:32 UTC (History)
CC List: 8 users (show)

See Also:
Host:
Target:
Build:
Last reconfirmed:

Flags: fweimer: security-

https://sourceware.org/bugzilla/show_bug.cgi?id=13165

POSIX Austin Group defect #609

ID	Category	Severity	Type	Date Submitted	Last Update
0000609	[1003.1(2004)/Issue 6] System Interfaces	Editorial	Clarification Requested	2012-09-20 14:18	2016-05-17 22:13

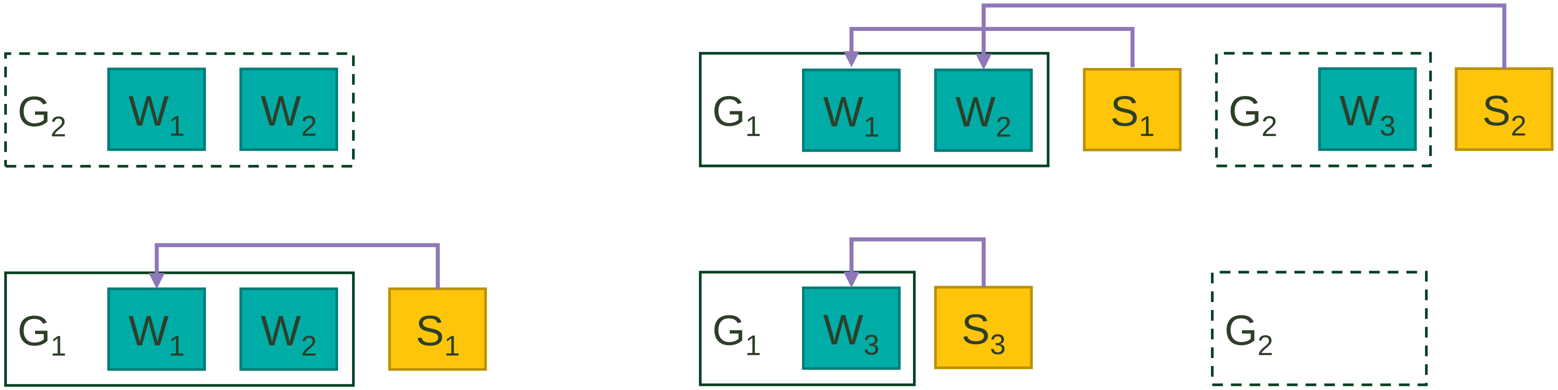
Reporter	mmihaylov	**View Status**	public		
Assigned To	ajosey				
Priority	normal	**Resolution**	Open		
Status	Under Review				

Name	Mihail Mihaylov
Organization	
User Reference	
Section	pthread_cond_broadcast, pthread_cond_signal
Page Number	1043
Line Number	33043 - 33046
Interp Status	---
Final Accepted Text	

Summary	0000609: It is not clear what threads are considered blocked with respect to a call to pthread_cond_signal() or pthread_cond_broadcast()

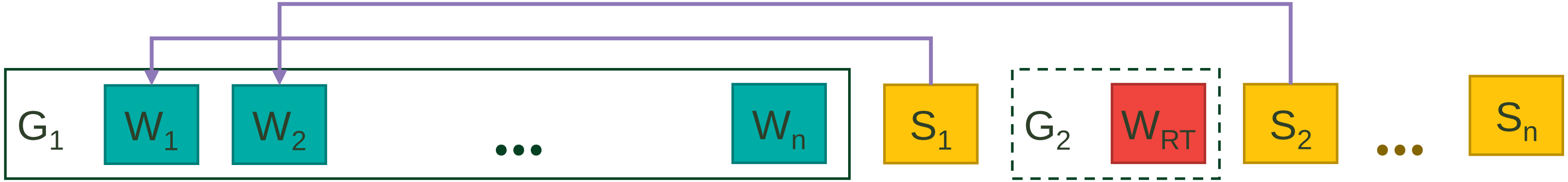
Current design of glibc conditional variables

- New waiters start in non-eligible group G2
- Group G1 contains only eligible waiters
- A signal will wake some thread in G1
- When all waiters in G1 are signaled, G2 becomes the new G1



Problems with current design

- New RT priority waiters start in non-eligible group G2
  - will have to wait until G1 is completely signaled
- Signaling is done with a FUTEX_WAKE operation
  - woken threads must contend for the associated mutex (thundering herd)



Bug 11588 – no known solution for glibc > 2.24

Torvald Riegel 2017-01-11 11:50:41 UTC **Comment 56**

```
The new condition variable implementation is now committed upstream.  It should be
the base for any improvement suggestions from now on.

How to support PI for condvars has also been discussed at the Linux Real-Time
Summit 2016: https://wiki.linuxfoundation.org/realtime/events/rt-summit2016
/schedule
So far, there is no known solution for how to achieve PI support given the current
constraints we have (eg, available futex operations, POSIX requirements, ...).
```

Priority inheritance support in libpthread

With priority inheritance support:

- pthread_mutex_* — FUTEX_LOCK_PI/UNLOCK_PI (enabled via mutex attributes)

Without priority inheritance support:

- pthread_barrier_*
- pthread_rwlock_*
- sem_*
- pthread_spin_*

pthread_barrier_*, pthread_rwlock_* — FUTEX_WAIT/WAKE

pthread_rwlock_*, sem_* — FUTEX_WAIT_BITSET/WAKE

pthread_spin_* — user-space spinning

The librtpi project inception and history

- I presented on the problem at RT Summit 2017 (video)
- Sebastian Andrezej Siewior set-up a meeting with: Darren Hart, Peter Zijlstra, Julia Cartwright, and me
- Given the glibc constraints we decided to try a standalone implementation
- Darren put together the initial spec and github project
- Sebastian and Julia worked on fleshing it out at Summit on a Summit 2018
- Darren and Julia presented a status update at Linux Plumbers 2018 (video)
- I worked with Darren to fix corner cases and bugs, add tests, and tweak the API (ELC 2019)

Librtpi design goals

- Priority inheritance by default
- Waiters will be woken in priority order
- Signaler must hold the lock
- Avoid “thundering herd” effect
- Default to CLOCK_MONOTONIC for timed waits
- Opaque data types to allow for future expansion
- API as close as possible to the POSIX pthread specification

Librtpi license, build, and test

- LGPL 2.1
  - makes it possible to link/reuse glibc code
  - broadly usable in industry
- Autotools build system
- Travis CI (github)

pi_mutex

```
int pi_mutex_init(pi_mutex_t *mutex,
                  uint32_t flags);
int pi_mutex_destroy(pi_mutex_t *mutex);

int pi_mutex_lock(pi_mutex_t *mutex);
int pi_mutex_trylock(pi_mutex_t *mutex);
int pi_mutex_unlock(pi_mutex_t *mutex);
```

```
#define DEFINE_PI_MUTEX(mutex, flags)
#define RTPI_MUTEX_PSHARED    0x1

pi_mutex_t *pi_mutex_alloc(void);
void pi_mutex_free(pi_mutex_t *mutex);
```

Porting POSIX code to pi_mutex

```
int pi_mutex_init(pi_mutex_t *mutex,
                  uint32_t flags);
int pi_mutex_destroy(pi_mutex_t *mutex);

int pi_mutex_lock(pi_mutex_t *mutex);
int pi_mutex_trylock(pi_mutex_t *mutex);
int pi_mutex_unlock(pi_mutex_t *mutex);
```

```
#define DEFINE_PI_MUTEX(mutex, flags)
#define RTPI_MUTEX_PSHARED    0x1

pi_mutex_t *pi_mutex_alloc(void);
void pi_mutex_free(pi_mutex_t *mutex);
```

pi_mutex_lock() implementation

```
if (!__sync_bool_compare_and_swap(&mutex->futex, 0, pid))
  syscall(SYS_futex, …);
```

```
int futex(int *uaddr,
          int futex_op,
          int val,
          const struct timespec *timeout,
          int *uaddr2,
          int val3);
```

PI futex address (&mutex->futex)

FUTEX_LOCK_PI [| FUTEX_PRIVATE_FLAG]

0: deadlock detection, unused

pi_mutex_unlock() implementation

```
if (!__sync_bool_compare_and_swap(&mutex->futex, pid, 0))
  syscall(SYS_futex, …);
```

```
int futex(int *uaddr,
          int futex_op,
          int val,
          const struct timespec *timeout,
          int *uaddr2,
          int val3);
```

PI futex address (&mutex->futex)

FUTEX_UNLOCK_PI [| FUTEX_PRIVATE_FLAG]

0: deadlock detection, unused

pi_cond

```
int pi_cond_init(pi_cond_t *cond,
                 uint32_t flags);
int pi_cond_destroy(pi_cond_t *cond);

int pi_cond_wait(pi_cond_t *cond,
                 pi_mutex_t *mutex);
int pi_cond_timedwait(pi_cond_t *cond,
                      pi_mutex_t *mutex,
                      const struct timespec *abstime);
int pi_cond_signal(pi_cond_t *cond,
                   pi_mutex_t *mutex);
int pi_cond_broadcast(pi_cond_t *cond,
                      pi_mutex_t *mutex);
```

```
#define DEFINE_PI_COND(condvar, flags)
#define RTPI_COND_PSHARED \
        RTPI_MUTEX_PSHARED

pi_cond_t *pi_cond_alloc(void);
void pi_cond_free(pi_cond_t *cond);
```

Porting POSIX code to pi_cond

```
int pi_cond_init(pi_cond_t *cond,
                 uint32_t flags);
int pi_cond_destroy(pi_cond_t *cond);

int pi_cond_wait(pi_cond_t *cond,
                 pi_mutex_t *mutex);
int pi_cond_timedwait(pi_cond_t *cond,
                      pi_mutex_t *mutex,
                      const struct timespec *abstime);
int pi_cond_signal(pi_cond_t *cond,
                   pi_mutex_t *mutex);
int pi_cond_broadcast(pi_cond_t *cond,
                      pi_mutex_t *mutex);
```

```
#define DEFINE_PI_COND(condvar, flags)
#define RTPI_COND_PSHARED \
        RTPI_MUTEX_PSHARED

pi_cond_t *pi_cond_alloc(void);
void pi_cond_free(pi_cond_t *cond);
```

pi_cond_signal() / broadcast() implementation

```
/* called with the mutex locked (per API) */
cond->cond++;
cond->wake_id = cond->cond;

ret = syscall(SYS_futex, …, FUTEX_CMP_REQUEUE_PI,…);
if (ret >= 0)
     return 0;

/* retry on EAGAIN */
return errno;
```

Futex syscall used for signaling

```
int futex(int *uaddr,
          int futex_op,
          int val,
          uint32_t val2,
          int *uaddr2,
          int val3);
```

non-PI futex waiters are queued on

FUTEX_CMP_REQUEUE_PI [| FUTEX_PRIVATE_FLAG]

number of threads to wake (required to be 1)

number of threads to requeue
(0:signal, INT_MAX: broadcast)

target PI futex to requeue threads on

pi_cond_wait() / timedwait() implementation

```
cond->cond++;
wake_id = cond->wake_id;
pi_mutex_unlock(mutex);

ret = syscall(SYS_futex, …, FUTEX_WAIT_REQUEUE_PI,…);
if (!ret)
     return 0;      /* normal wakeup and we own the lock */

pi_mutex_lock(mutex);
/* retry on EAGAIN unless we’ve raced with a signaler */
return errno;
```

Futex syscall used for waiting

```
int futex(int *uaddr,
          int futex_op,
          int val,
          const struct timespec *timeout,
          int *uaddr2,
          int val3);
```

non-PI futex thread waits on

FUTEX_WAIT_REQUEUE_PI [| FUTEX_PRIVATE_FLAG]

futex word value (race detection)

absolute timeout (NULL: wait forever)

PI futex thread gets requeued on
(a.k.a. user mutex/monitor mutex)

Current status

- Glibc tests and API change merged at: https://github.com/dvhart/librtpi
- Still owe Darren some pull requests: https://github.com/gratian/librtpi/commits/latest
  - locking fixes, pi_mutex fix for process shared case
  - simplified sequence counters / race detection
  - get rid of internal private mutex
  - CLOCK_REALTIME support
  - cancellation support (?)
  - general clean-ups, documentation, error checks etc. (~25 commits ahead)

Current status (cont'd)

- librtpi.so ~ 34KB (x86_64)
- All tests pass*
- Used in production at NI
- Want do an "official" release when remaining issues merged

```
PASS: test_api
PASS: tst-cond1
[sudo] password for gratian:
PASS: tst-condpi2.sh
======================================
Testsuite summary for librtpi 0.0.1
======================================
# TOTAL: 3
# PASS:  3
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
======================================
```

```
PASS: tst-cond1
PASS: tst-cond2
PASS: tst-cond3
PASS: tst-cond4
PASS: tst-cond5
PASS: tst-cond6
PASS: tst-cond7
PASS: tst-cond8
PASS: tst-cond9
PASS: tst-cond10
PASS: tst-cond11
PASS: tst-cond12
PASS: tst-cond13
PASS: tst-cond16
PASS: tst-cond18
PASS: tst-cond19
PASS: tst-cond20
PASS: tst-cond21
PASS: tst-cond22
PASS: tst-cond24
PASS: tst-cond25
PASS: tst-cond-except
========================================
Testsuite summary for librtpi 0.0.1
========================================
# TOTAL: 22
# PASS:  22
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
========================================
```

Future

- Users, testers, and contributors
  - https://github.com/dvhart/librtpi
  - https://github.com/gratian/librtpi/tree/latest

- Extend it into a user space toolbox for Real-Time design
  - other locking primitives relevant for RT
  - RT safe queues for arbitrary data types
  - circular buffers, priority queues, IPC mechanisms
  - other building blocks useful for RT applications

- Your ideas and questions