RTEMS Thread Queue Simulation

Andrew Butterfield

January 16, 2023

Contents

1 Introduction

This document provides simulations of the various thread queue designs used in the RTEMS operating system. These range from simple first-in first-out (FIFO) designs, through priority-based approaches, to algorithms for SMP that support schedulability analysis.

We focus on thread queues for RTEMS Tasks waiting to gain access to a shared resource, via a mutex lock.

The basic building blocks are a simple FIFO queue, and a priority queue that assumes all threads have a unique priority.

The most complex form is a round-robin/FIFO queue whose contents are priority queues, one for each scheduling cluster. This is used to implement the Multiprocessor Resource Sharing Protocol(MrsP) thread queue algorithm.

Key papers are: [BW13], [Bra13], [CBHM15], [GZBW17], [ZGBW17], [Gom19], [ZGW+20], [ZCW+21] .

Most relevent RTEMS documents: [RTEa], [RTEb], and [RTEc].

2 RTEMS Thread Queues

```
module Queues
    ( FIFOQ
    , isEmptyFIFOQ,  viewFIFOQ,  enqueueFIFO,  dequeueFIFO
    , Priority, PRIOQ
    , isEmptyPRIOQ,  viewPRIOQ,  enqueuePRIO,  dequeuePRIO
    , Cluster, CLSTRQ
    , isEmptyCLSTRQ, viewCLSTRQ, enqueueCLSTR, dequeueCLSTR
    ) where
```

Here we define the three types of queues used for thread synchronisation and scheduling.

2.1 FIFO Queues

See [RTEa, §3.5].

We model a FIFO queue as a Haskell list, parameterised by content object type, with enqueue and dequeue operations, and an emptiness check.

```
type FIFOQ obj = [obj]

isEmptyFIFOQ :: FIFOQ obj -> Bool
isEmptyFIFOQ = null

viewFIFOQ :: Show obj => FIFOQ obj -> String
viewFIFOQ = show

enqueueFIFO :: obj -> FIFOQ obj -> FIFOQ obj
enqueueFIFO thing fifoq = fifoq ++ [thing]

dequeueFIFO :: MonadFail m => FIFOQ obj -> m (obj,FIFOQ obj)
dequeueFIFO [] = fail "empty FIFO queue"
dequeueFIFO (thing:restq) = return (thing,restq)
```

We have a variant of a FIFO queue called round-robin (RR).

In this, the dequeue operation immediately performs an enqueue operation with the item just dequeued. Typically the queue is initially setup by enqueuing all desired items, and subsequent operations consists solely of dequeing.

```
dequeueRR :: MonadFail m => FIFOQ obj -> m (obj,FIFOQ obj)
dequeueRR [] = fail "empty RR queue"
dequeueRR (thing:restq) = return (thing,restq++[thing])
```

2.2 Priority Queues

See [RTEa, §3.5].

We model a priority queue as a Haskell list, parameterised by content object type, with enqueue and dequeue operations, and an emptiness check.

```
type Priority = Int
type PRIOQ obj = [(Priority,obj)]

isEmptyPRIOQ :: PRIOQ obj -> Bool
isEmptyPRIOQ = null

viewPRIOQ :: Show obj => PRIOQ obj -> String
viewPRIOQ = show

enqueuePRIO :: obj -> Priority -> PRIOQ obj -> PRIOQ obj
enqueuePRIO thing p [] = [(p,thing)]
enqueuePRIO thing p prioq@(first@(q,_):restq)
  | p < q       =  (p,thing) : prioq
  -- p == q, insert as per FIFO, after those of same priority (c-user 3.5)
  | otherwise   =  first      : enqueuePRIO thing p restq

dequeuePRIO :: MonadFail m => PRIOQ obj -> m (obj,Priority,PRIOQ obj)
dequeuePRIO [] = fail "empty PRIO queue"
dequeuePRIO ((p,thing):restq) = return (thing,p,restq)
```

2.3 Clustered Scheduling Queues (SMP)

See [RTEa, §3.5,§5.4].

For cluster scheduling, each scheduler has its own priority queue, and these queues are themselves placed in a global round-robin queue.

```
type Cluster = Int
type CLSTRQ obj =  FIFOQ (Cluster,PRIOQ obj)

isEmptyCLSTRQ :: CLSTRQ obj -> Bool
isEmptyCLSTRQ = all isEmptyPRIOQ . map snd

viewCLSTRQ :: Show obj => CLSTRQ obj -> String
viewCLSTRQ = show


enqueueCLSTR :: obj -> Priority -> Cluster -> CLSTRQ obj -> CLSTRQ obj
enqueueCLSTR thing p c [] = [(c,[(p,thing)])]
enqueueCLSTR thing p c (first@(c',prioq):rest)
  | c == c'     =  (c',enqueuePRIO thing p prioq):rest
  | otherwise   =  first : enqueueCLSTR thing p c rest

dequeueCLSTR :: MonadFail m => CLSTRQ obj -> m (obj,Priority,Cluster,CLSTRQ obj)
dequeueCLSTR [] = fail "empty CLSTR queue"
dequeueCLSTR ((c,prioq):restq)
  = do (thing,p,prioq') <- dequeuePRIO prioq
       if isEmptyPRIOQ prioq' -- delete empty queues (???)
         then return (thing,p,c,restq)
         else return (thing,p,c,restq ++ [(c,prioq')])
```

3 RTEMS Simulation Runner

```
module Runner
    ( requestInput
    , interactive
    , batch
    ) where
import System.IO
import Queues
```

Here we provide a simple mechanism/language to run simulations
The basic idea is to declare some objects, and then invoke actions upon them.
The language is line based, each line starting with a keyword.

3.1 Objects

We segregate objects by their type.

```
data Object
```

Uninterpreted arbitrary test objects

```
  = Uint
```

FIFO FIFO queue objects

```
  | FIFO Object
```

PRIO Priority queue objects

```
  | PRIO Object
```

CLSTR Cluster queue objects

```
  | ClusterQ Object
```

The queue objects are themselves parameterised with a content object.

3.2 Simulation State

We define the state to be a collection of named objects:

```
type NamedObject obj = (String,obj)
data SimState
  =  State {
       arbobjs   :: [String] -- basically tokens naming themselves
     , fifoQs     :: [NamedObject (FIFOQ String)]
     , prioQs     :: [NamedObject (PRIOQ String)]
     , clusterQs :: [NamedObject (CLSTRQ String)]
     }
  deriving Show

initstate = State [] [] [] []
```

3.3 Running Simulations

3.3.1 Interactive Simulation

```
requestInput prompt = do
  putStr prompt
  hFlush stdout
  getLine

interactive = request initstate

request s = do
  cmd <- requestInput "Cmd> "
  s' <- doCommand s cmd
  putStrLn ("State:\n"++show s')
  request s'
```

3.3.2 Batch Simulation

```
batch sfn = do
  cmds <- fmap lines $ readFile sfn
  putStrLn ("Running '"++sfn)
  putStrLn ("Initial State:\n"++show initstate)
  s' <- perform initstate cmds
  putStrLn ("Final State:\n"++show s')

perform :: SimState -> [String] -> IO SimState
perform s [] = do { putStrLn "Completed" ; return s }
perform s (cmd:cmds) = do
  s' <- doCommand s cmd
  putStrLn ("State:\n"++show s')
  perform s' cmds
```

3.4 Simulation Commands

```
doCommand :: SimState -> String -> IO SimState
doCommand s cmd = do
  putStrLn ("\n> "++cmd)
  case words cmd of
    []  ->  return s
    ("new":what:args) -> makeNewObject s what args
    _ -> do putStrLn ("Unrecognised command '"++cmd++"'")
            return s
```

3.4.1 Creating New Objects

```
makeNewObject :: SimState -> String -> [String] -> IO SimState
makeNewObject s what args
  | what == "arb"   =  makeNewArbitraryObjects s args
  | otherwise   =  do putStrLn ("Unknown object type '"++what++"'")
                      return s
```

```
makeNewArbitraryObjects :: SimState -> [String] -> IO SimState
makeNewArbitraryObjects s args
  = return s{ arbobjs = args ++ arbobjs s }
```

4 Program Mainline

```haskell
module Main where

import System.Environment

import Queues
import Runner

main :: IO ()
main = do
  putStrLn "\n\tThread Q Simulator\n"
  args <- getArgs
  case args of
    [] -> interactive
    (["-i"]) -> interactive
    (["-b"]) -> do
      simFileName <- requestInput "Enter simulation filename: "
      batch simFileName
    (["-b",simFileName]) -> batch simFileName
    ([simFileName]) -> batch simFileName
    _ -> usage
  putStrLn "\n\tFinished!\n"

usage = putStrLn $ unlines
  [ "usage: tqsim [-i|-b] [fname]"
  , " -i         :  run iteractively (default if no args)"
  , " -b         :  request sim file name"
  , " -b fname   :  use sim file 'fname'"
  , " fname mmm  :  use sim file 'fname'"
  ]
```

5 Test Program

```haskell
main :: IO ()
main = putStrLn "Thread Q Sim Test suite not yet implemented"
```

References

[Bra13] Björn B. Brandenburg. A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, pages 292–302, 2013.

[BW13] A. Burns and A. J. Wellings. A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, 2013.

[CBHM15] Sebastiano Catellani, Luca Bonato, Sebastian Huber, and Enrico Mezzetti. Challenges in the Implementation of MrsP. In *Reliable Software Technologies - Ada-Europe 2015*, pages 179–195, 2015.

[Gom19] Ricardo Gomes. Analysis of MrsP Protocol in RTEMS Operating System. Master's thesis, CISTER, Departmento de Engenharia Informática, Instituto Superior de Engenharia do Porto (ISEP), Portugal, 2019.

[GZBW17] Jorge Garrido, Shuai Zhao, Alan Burns, and Andy J. Wellings. Supporting nested resources in mrsp. In Johann Blieberger and Markus Bader, editors, *Reliable Software Technologies - Ada-Europe 2017 - 22nd Ada-Europe International Conference on Reliable Software Technologies, Vienna, Austria, June 12-16, 2017, Proceedings*, volume 10300 of *Lecture Notes in Computer Science*, pages 73–86. Springer, 2017.

[RTEa] The RTEMS Project contributors. RTEMS Classic API Guide. `https://docs.rtems.org/branches/master/c-user.pdf`.

[RTEb] The RTEMS Project contributors. RTEMS Software Engineering. `https://docs.rtems.org/branches/master/eng.pdf`.

[RTEc] The RTEMS Project contributors. RTEMS User Manual. `https://docs.rtems.org/branches/master/user.pdf`.

[ZCW+21] Shuai Zhao, Wanli Chang, Ran Wei, Weichen Liu, Nan Guan, Alan Burns, and Andy J. Wellings. Priority assignment on partitioned multiprocessor systems with shared resources. *IEEE Trans. Computers*, 70(7):1006–1018, 2021.

[ZGBW17] Shuai Zhao, Jorge Garrido, Alan Burns, and Andy J. Wellings. New schedulability analysis for mrsp. In *23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2017, Hsinchu, Taiwan, August 16-18, 2017*, pages 1–10. IEEE Computer Society, 2017.

[ZGW+20] Shuai Zhao, Jorge Garrido, Ran Wei, Alan Burns, Andy J. Wellings, and Juan Antonio de la Puente. A complete run-time overhead-aware schedulability analysis for mrsp under nested resources. *J. Syst. Softw.*, 159, 2020.