Distributed Computing and Systems

Chalmers university of technology

Wait-free Queue Algorithms for the Real-time Java Specification

Philippas Tsigas

Yi Zhang, SAP Research

Daniel Cederman

Tord Dellsén

RTAS ’06, San Jose,

Apr. 4th – 7th, 2006

Outline

- JAVA Real-time queue Classes
  - RTSJ
  - Non-Blocking Synchronization
- An Algorithmic implementation of these JAVA RT queue classes
  - Algorithm
  - Previous work
  - Evaluation
- Conclusions & Future

Real-time Specification for JAVA

- To make JAVA more suitable for real-time programming
  - Real-time threads `NoHeapRealtimeTreads`
  - Memory Management which bypasses the garbage collection
  - Wait-free synchronization between non-real-time threads and real-time threads:

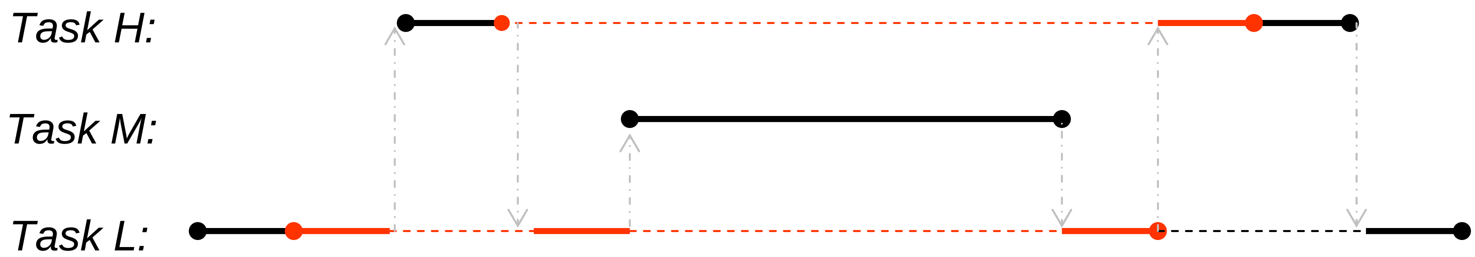
    `WaitFreeReadQueue, WaitFreeWriteQueue`
  - ……

Wait-free Synchronization in RTSJ

- Why?
  - RTSJ supports priority-based real-time systems.
  - Lock-based synchronizations introduce the priority inversion problem.
  - Protocols to solve the priority inversion problem bring dependencies with the garbage collector in JVM.

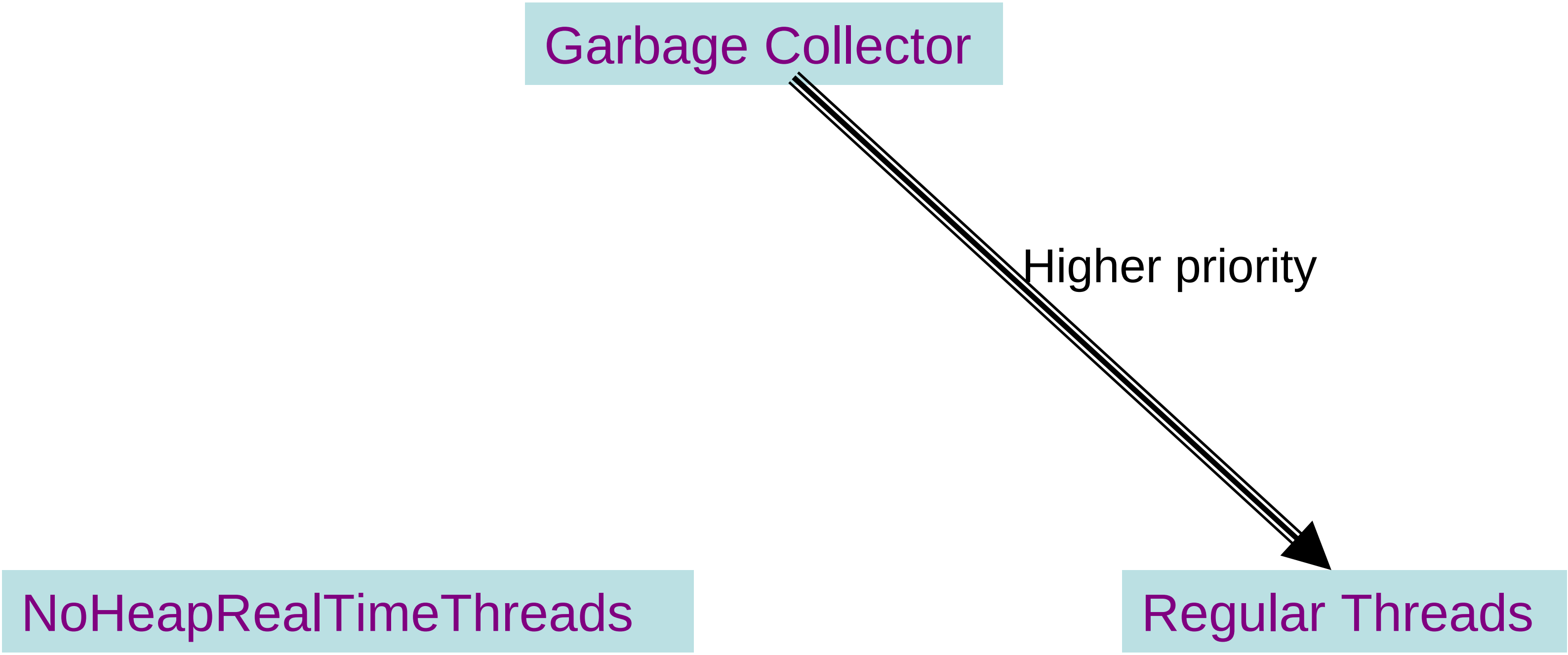
Lock-Based Synchronization -> Priority Inversion

- A high priority task is delayed due to a low priority task holding a shared resource. The low priority task is delayed due to a medium priority task executing.



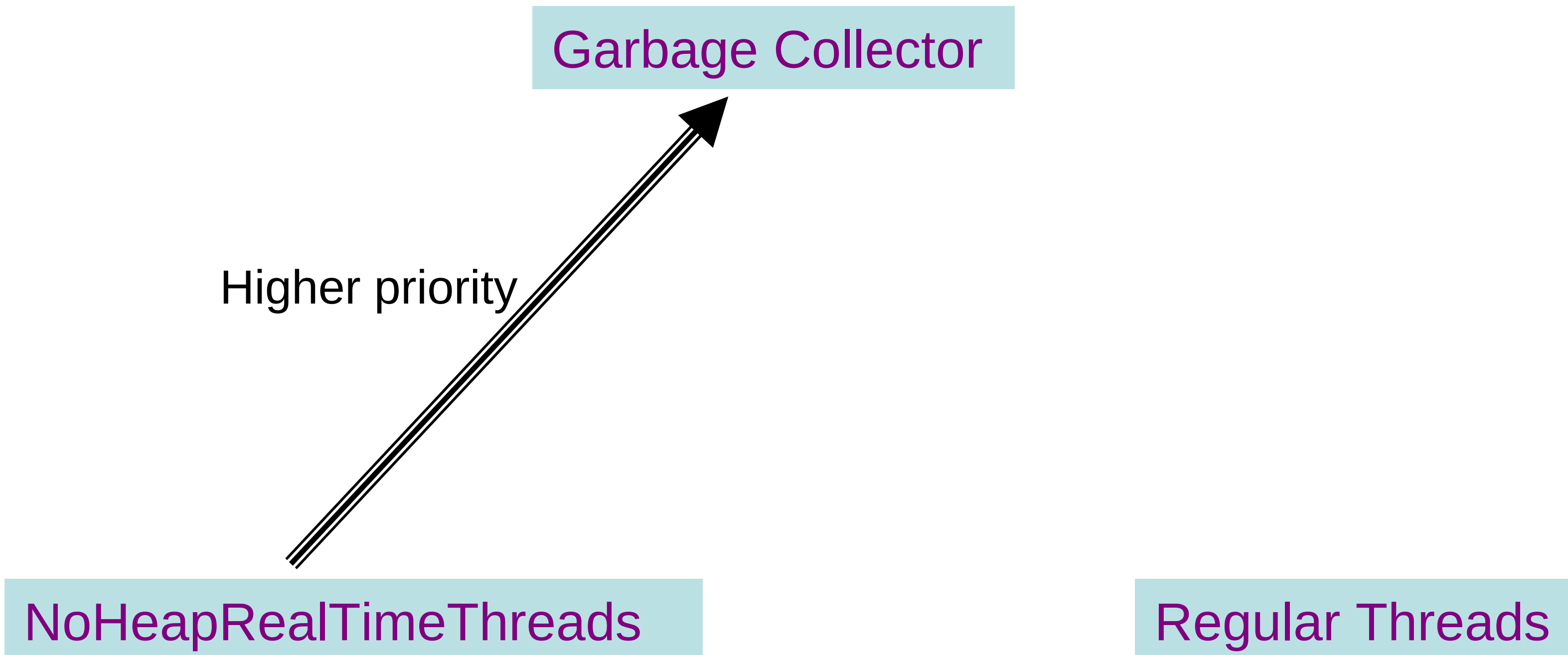
- Solutions: Priority inheritance protocols

Lock-based Synchronizations for RTJ threads



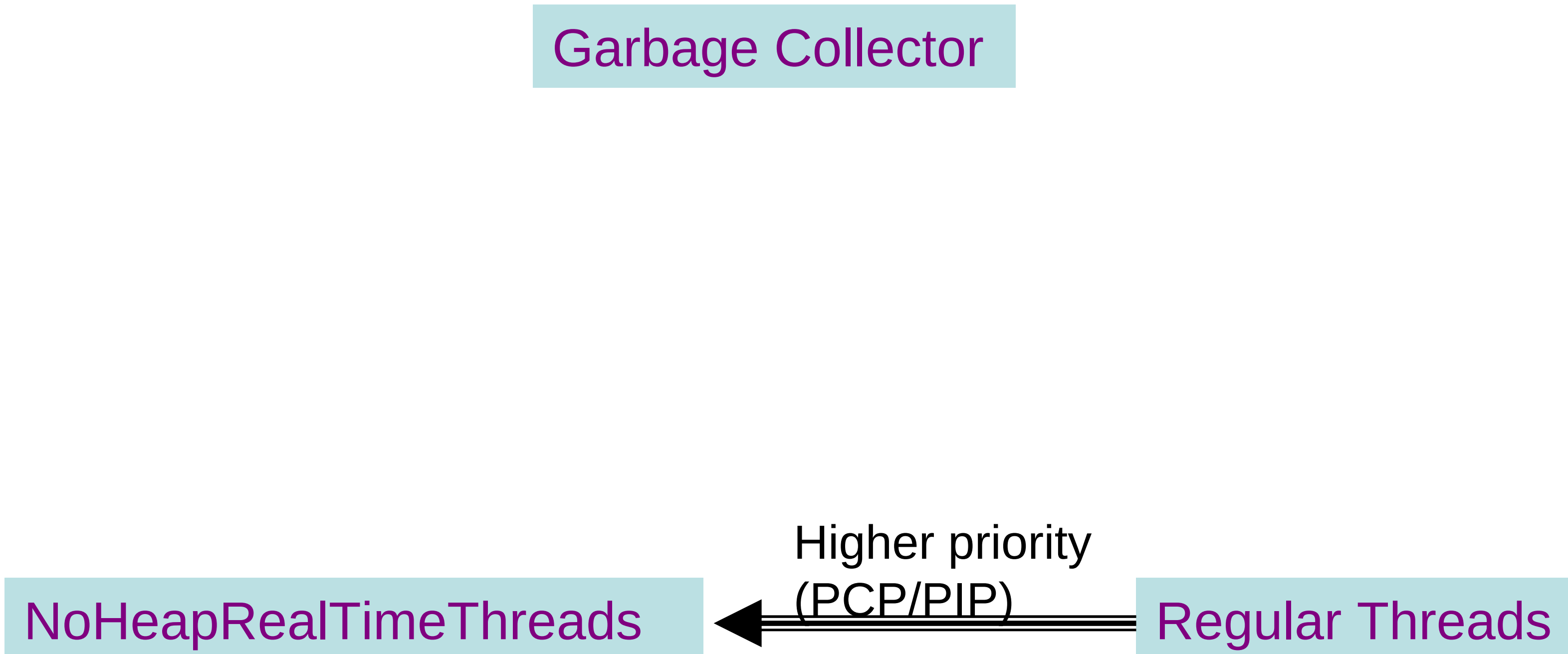
- Regular java threads must wait until the collector reaches a preemption-safe point.

Lock-based synchronizations for RTJ threads



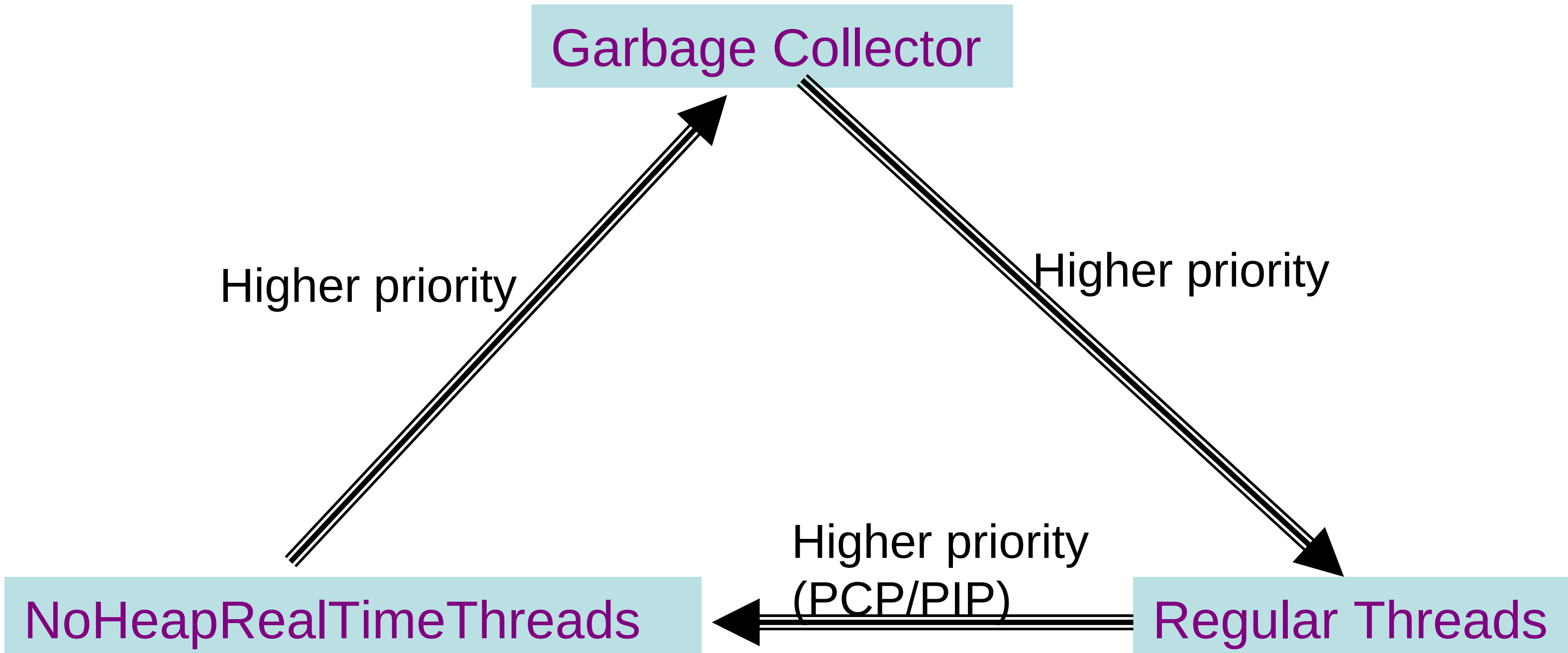
- NoHeapRealtimeThread may interrupt the garbage collector at any time.

Lock-based synchronizations for RTJ threads



- When NHRTs use locks to synchronize with regular Java threads, PCP/PIP may prompt the priorities of regular java threads.

Lock-based synchronizations for RTJ threads

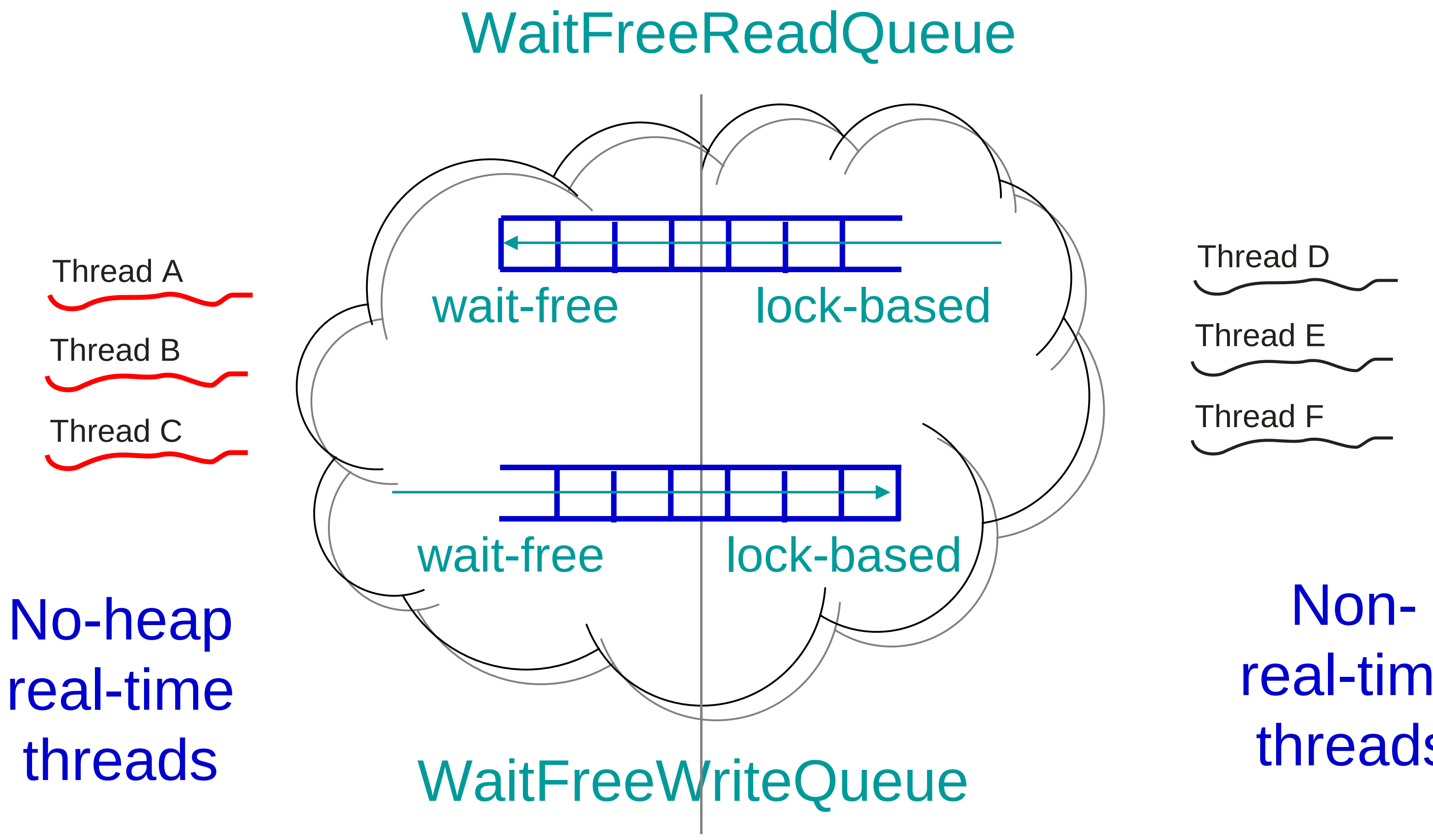


- Which one has the highest priority? Catch-22

Non-Blocking Algorithms

- **Lock-Free**. Guarantees that always one operation is making progress.
  - Combined with scheduling information, schedulability analysis can be done.
- **Wait-Free**. Guarantees that any operation will finish in a finite time.
  - Schedulability analysis can be done directly.

Java Real-time Queue Classes



Outline

- JAVA Real-time queue Classes
  - RTSJ
  - Non-Blocking Synchronization
- An Algorithmic implementation of these JAVA RT queues
  - Algorithm
  - Previous work
  - Evaluation
- Conclusions

Our Algorithm

- Design principles
  - Multiple threads can access the wait-free queue at the same time at both ends (real-time and non-real-time).
  - Using only READ/WRITE primitives to ensure the “Write Once, Run Anywhere” principal.
  - Integrate the uni-direction property of the wait-free queues in our implementation.

Wait-free Methodologies in Priority-based Real-time Systems

- Operations by high priority tasks are atomic operations for low priority tasks.
  - Read/Write has the same power as the "Compare and Swap" primitive. [Ramamurthy, Moir, Anderson 1996]
- Announce and help scheme
  - Tasks make announcements of their intention in an announcement array.
  - Each priority has a correspond location in the announcement array.
  - Each task will help the announcements from the lowest priority to its own priority.

Our WaitFreeWriteQueue Implementation

- Extended a sequential implementation based on Link-List.
- Each task announces its enqueue operation.
- Each task helps all announcements up to its own priority.
  - If the task is not preempted, all pending announcements will be helped.
  - If the task is preempted by a high priority task, all announcements includes its own will be helped by the high priority task.

The “enabled late write” Problem

- The problem happens when a task is preempted just before its write operation.
- When the task is resumed, it may overwrite a wrong value to the location.
- Previous solution to the problem is based on a voting scheme. [Ramamurthy, Moir, Anderson 1996]
- Our solution has two parts: i) directs threads to write on different locations when possible ii) by a careful algorithmic designing we make sure that every shared writing will agree on the content.

Enabled-Late-Write

Thread A's Variables:
i = undefined

Shared Variables:
Counter = 0

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

Thread A's Variables:
i = 0

Shared Variables:
Counter = 0

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

Thread A's Variables:
i = 1

Shared Variables:
Counter = 0

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

Thread A's Variables:
i = 1

Shared Variables:
Counter = 0

Thread B starts helping
Thread A

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

Thread A's Variables:
i = 1

Shared Variables:
Counter = 0

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

Thread A's Variables:
i = 1

Shared Variables:
Counter = 1

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

Thread A's Variables:
i = 1

Shared Variables:
Counter = 1

Thread B increments the counter for itself

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

Thread A's Variables:
i = 1

Shared Variables:
Counter = 2

Thread B increments the counter for itself

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

Enabled-Late-Write

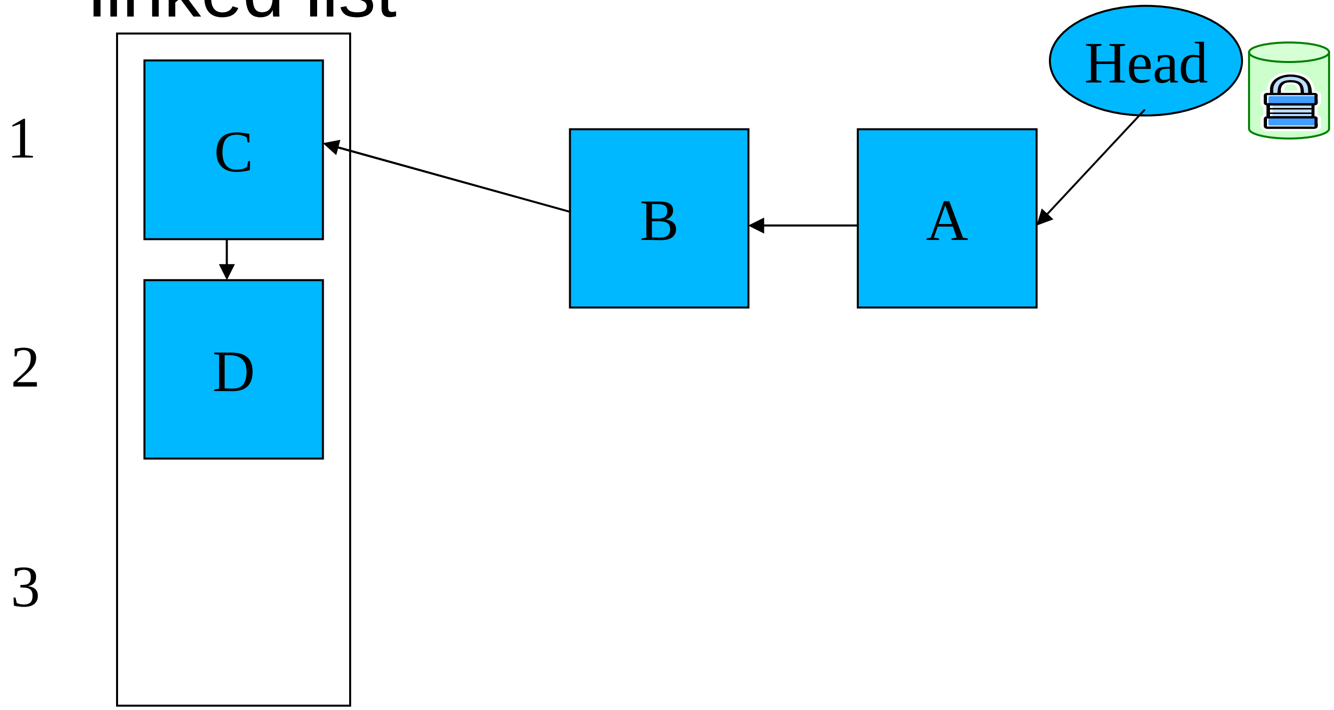
Thread A's Variables:
i = 1

Shared Variables:
Counter = 1

```
void incCounter()
{
  int i = counter;
  i = i + 1;
  counter = i;
}
```

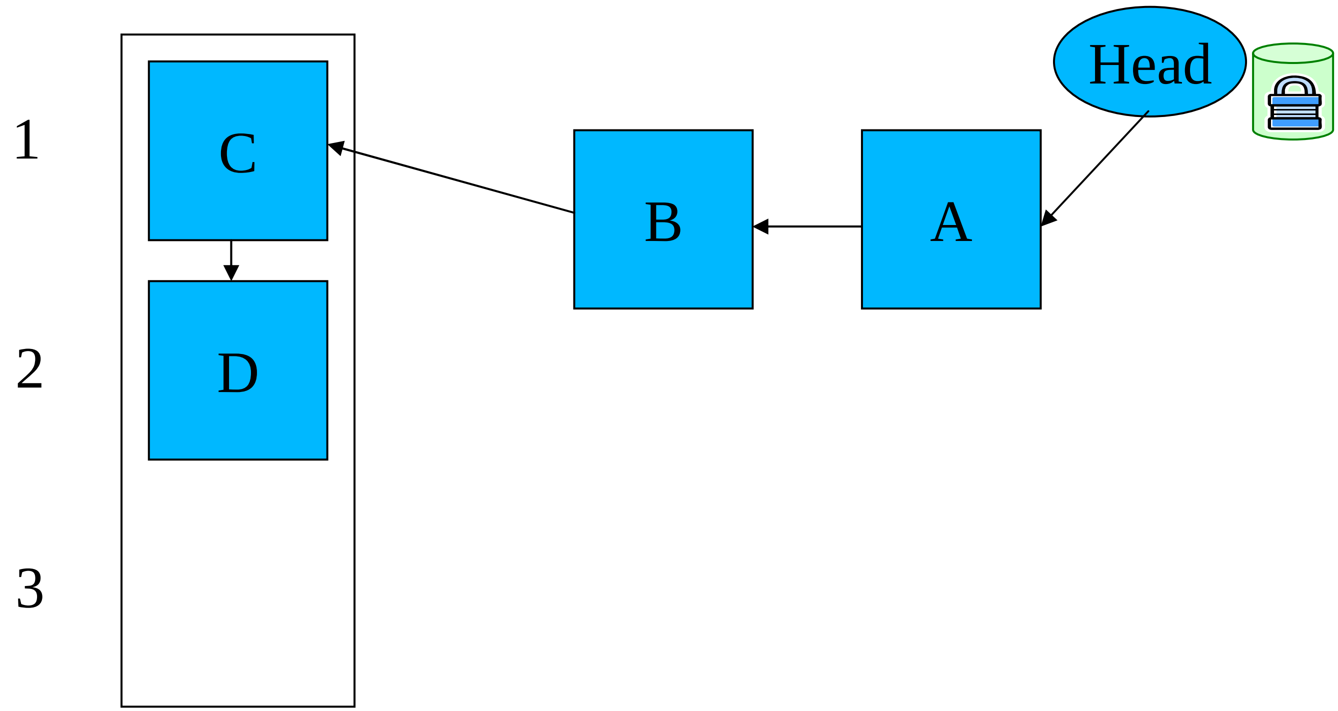
Wait-Free Write Queue

- Instead of only one tail it has a tail for each priority, but only one is the *actual* tail of the linked list



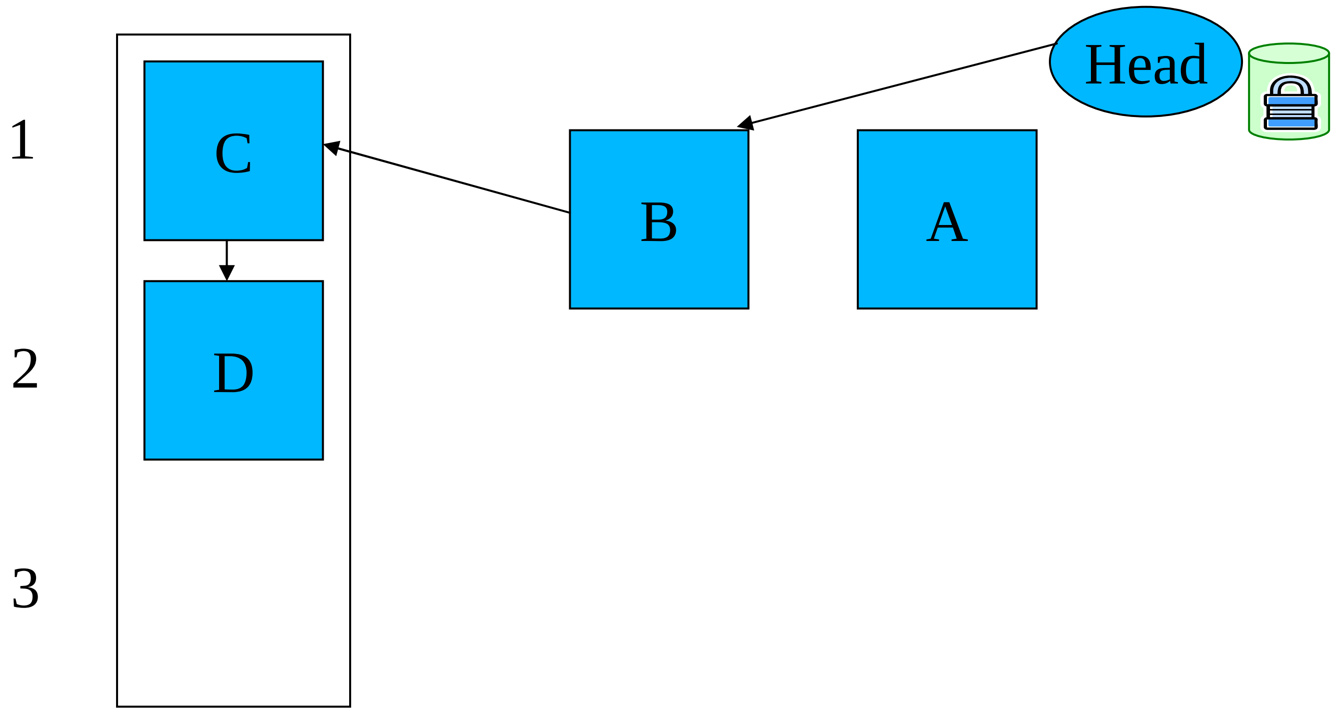
Reading from the queue

- Reading is simple since it's blocking, just make the head variable point to the next node in the list



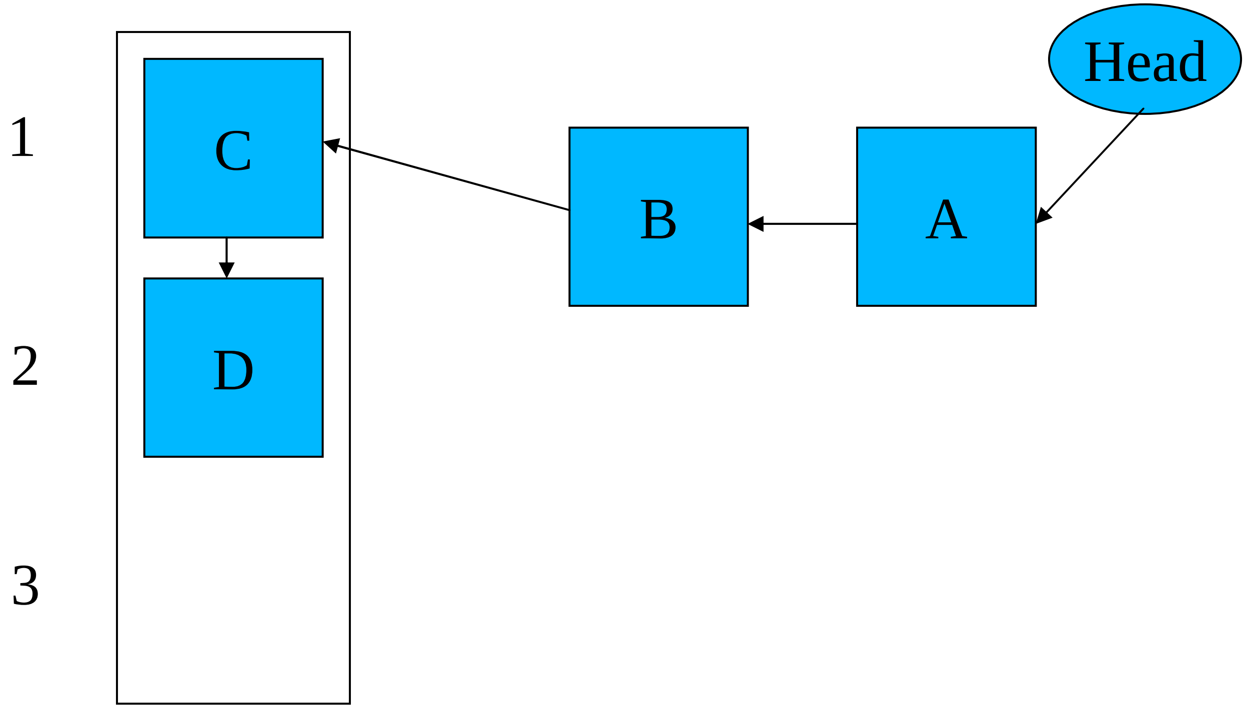
Reading from the queue

- Reading is simple since it's blocking, just make the head variable point to the next node in the list



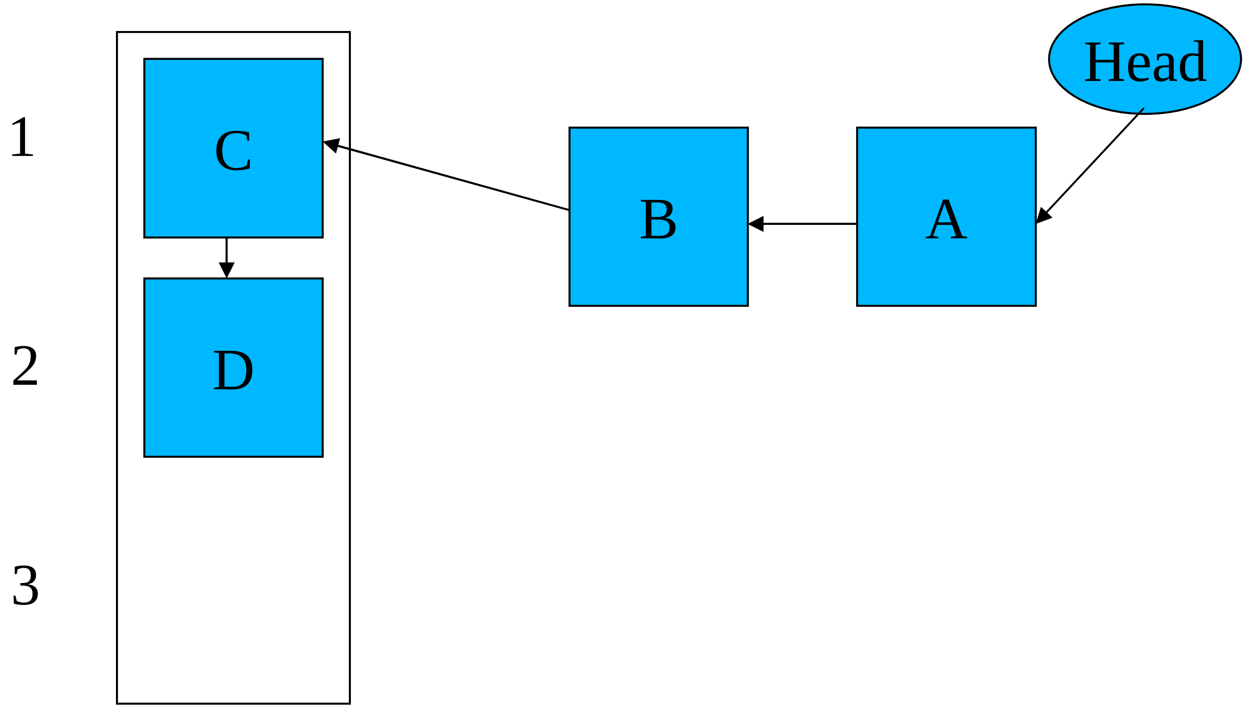
Writing to the queue

- First we need to find the actual tail.



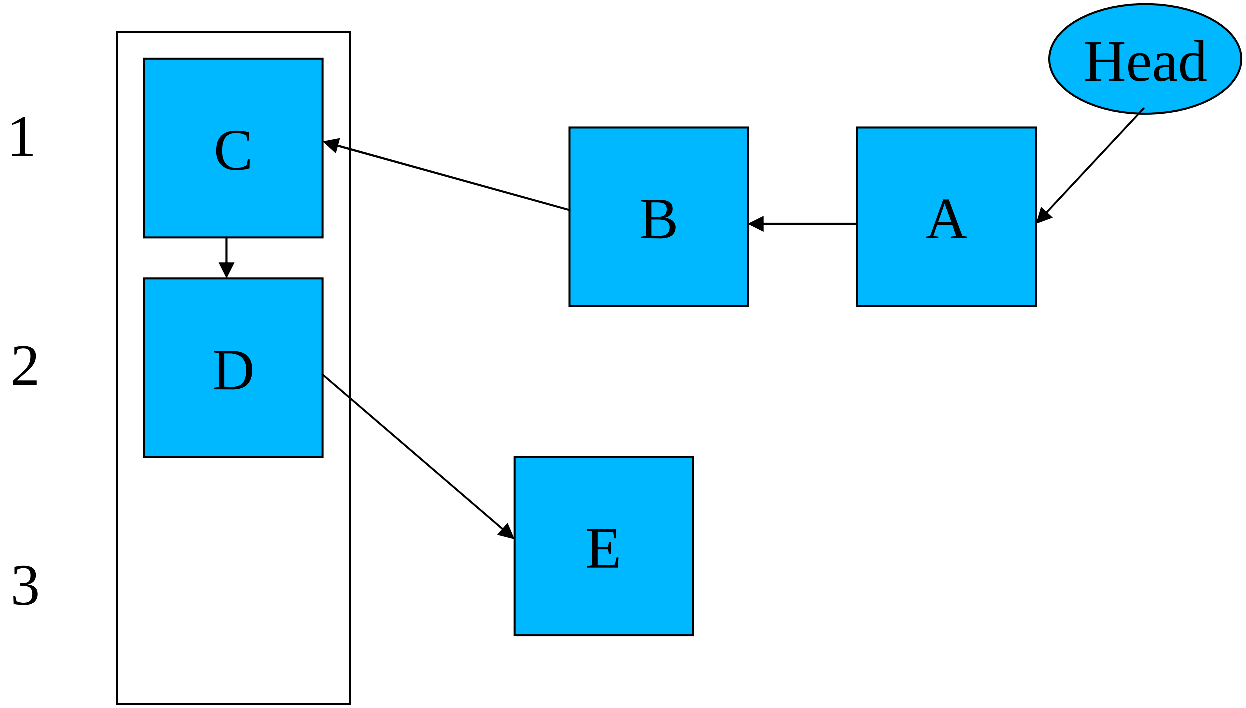
Writing to the queue

- We go through the tail array looking for the node that doesn't point to another node



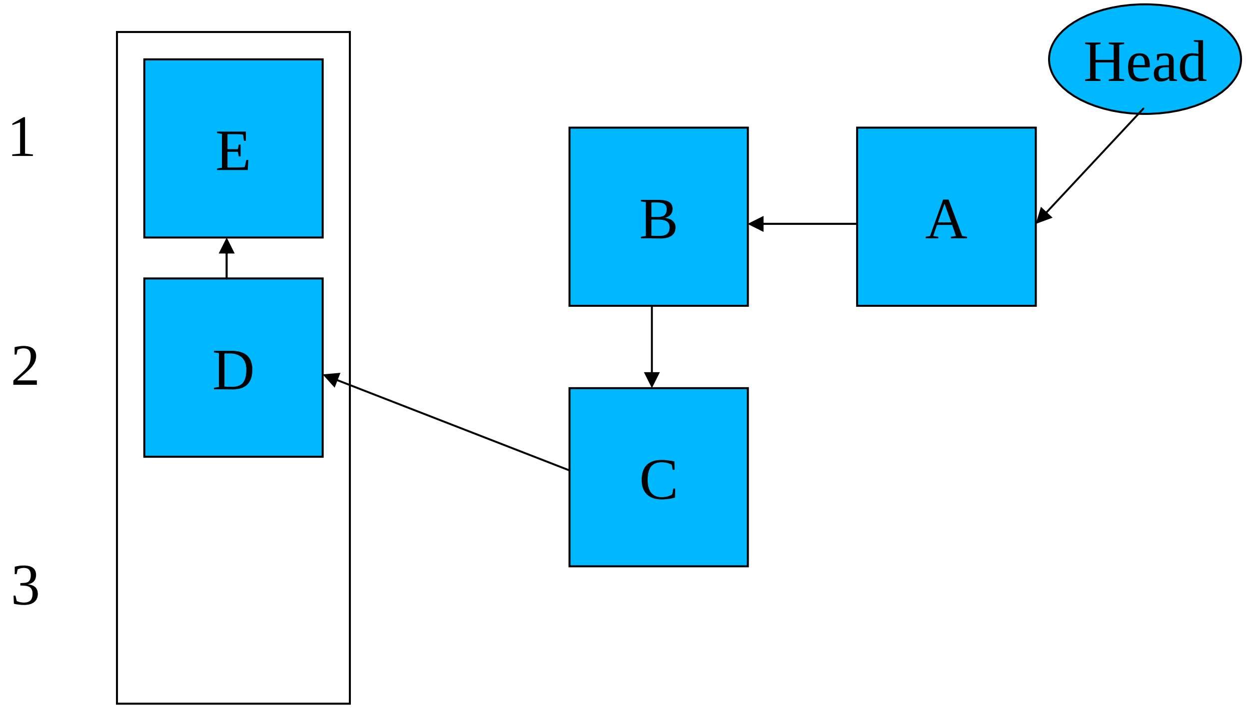
Writing to the queue

- We make it point to the new node



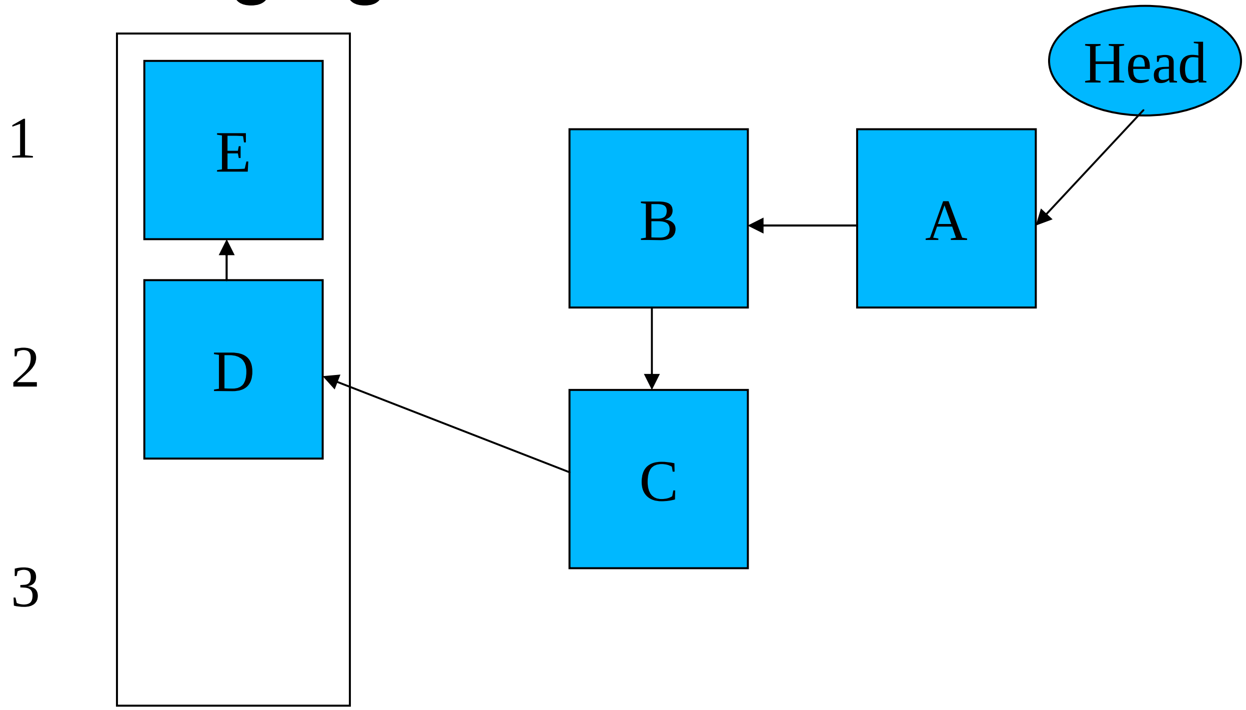
Writing to the queue

- Then we store it in the tail array at the threads priority, in this case 1



Writing to the queue

- By doing this we avoid the enabled-late-write that could have occured when changing the tail



Previous Work

- By combining [Ramamurthy, Moir, Anderson 1996] and [Anderson, Ramamurthy, Jain 97] a fully wait-free linked list for priority based systems can be derived.

Experimental results

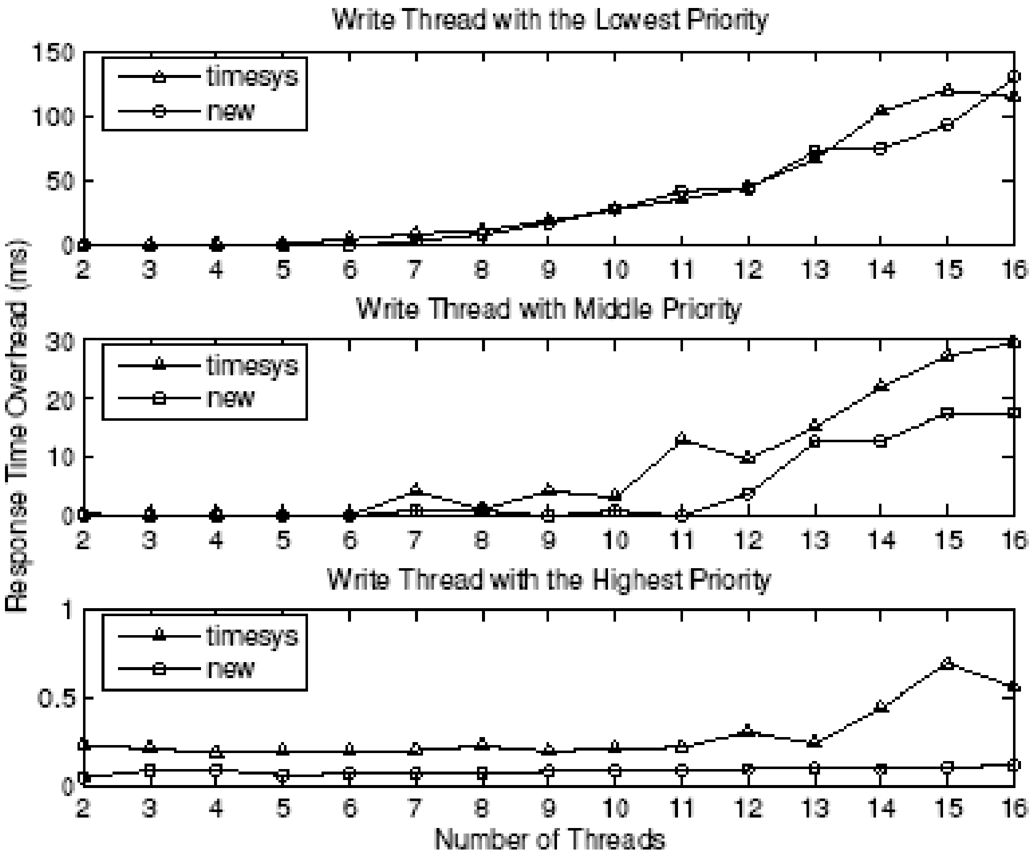
- We compared our implementation with the TimeSys reference implementation.
- We used JVM version 1.0.0 build 547 on top of TimeSys Linux 4.1 Build 155
- We implemented our Queues on top of this reference implementation
- 1100 MHz Intel Celeron processor and 512MB memory (with 100 MHz clock frequency). Its L2 cache was 128kB.

Experimental Results

- We consider the worst case execution times of the implementations (at least 50 test).
- We generated senarios that produce 90% utilization (rate monotonic).

- We implemented PCEP for the HardRealTime to synchronise when accessing the queues

Response Time Overhead



$$T_{ro} = t_f - t_s - C_i$$

Outline

- JAVA Real-time queue Classes
  - RTSJ
  - Non-Blocking Synchronization
- An Algorithmic implementation of these JAVA RT queue classes
  - Algorithm
  - Previous work
  - Evaluation
- Conclusions & Future

Our Contribution

- An algorithmic implementation of the wait-free queues in RTSJ.
- An new solution for the “enabled late write” problem.
- O(N+M) space complexity
- O(N) time complexity

Future Work

- Wait-Free Memory Management schemas
- Experiments with real-world data

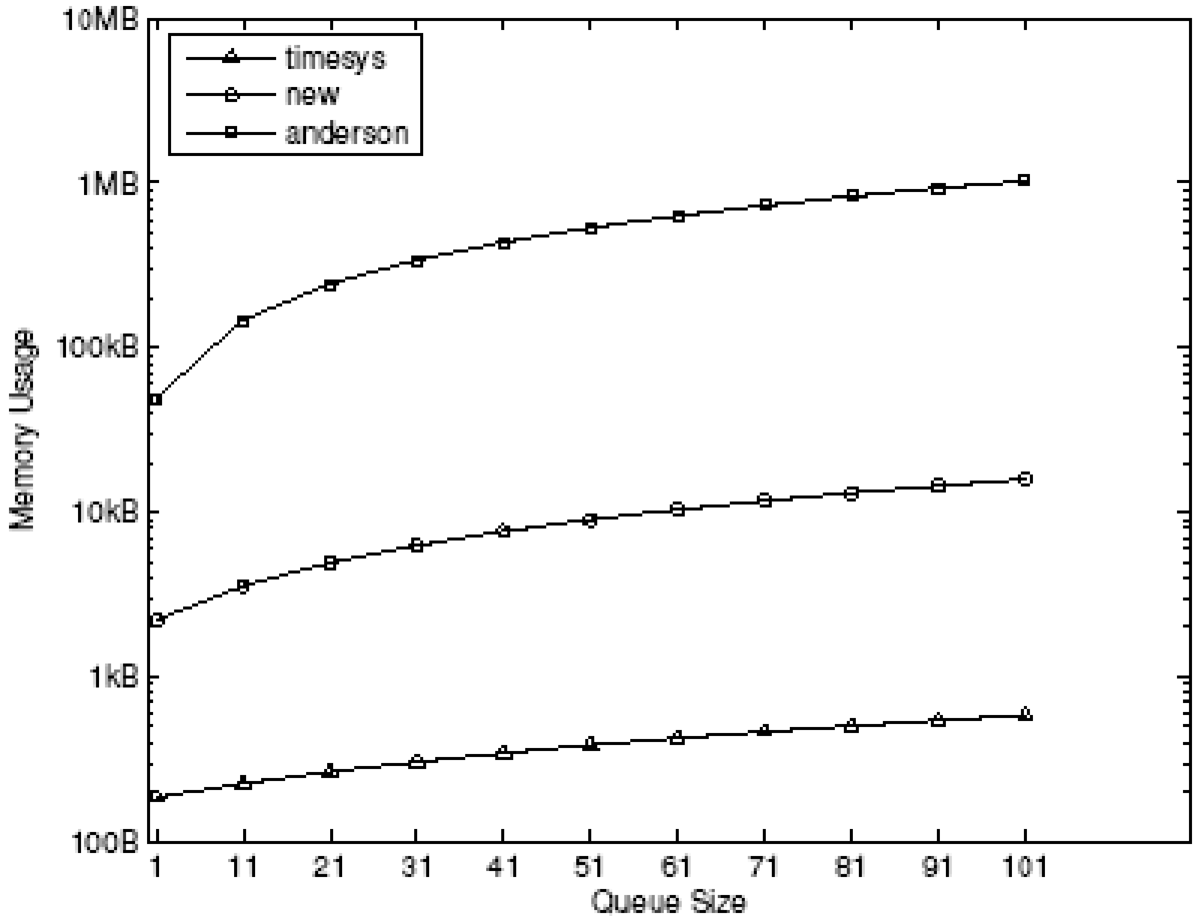
Memory Management

Type	Memory Requirements	Response Time	Comment
Multiple Arrays	O(Q*P)	O(Q)	Simple and very memory intensive.
Single Array	O(Q)	O(Q)	Low memory requirements but linear access.
Stack	O(Q)	O(P*c)	Slow due to emulated CAS.
Queue	O(Q)	O(P)	Fast but complicated.

Table 8-1. Comparison between Memory Managers

This table sums up the benefits and downsides of different memory managers. Q is the maximum size of the queue and P is the number of priorities in the system. The c is a large constant due to emulated CAS.

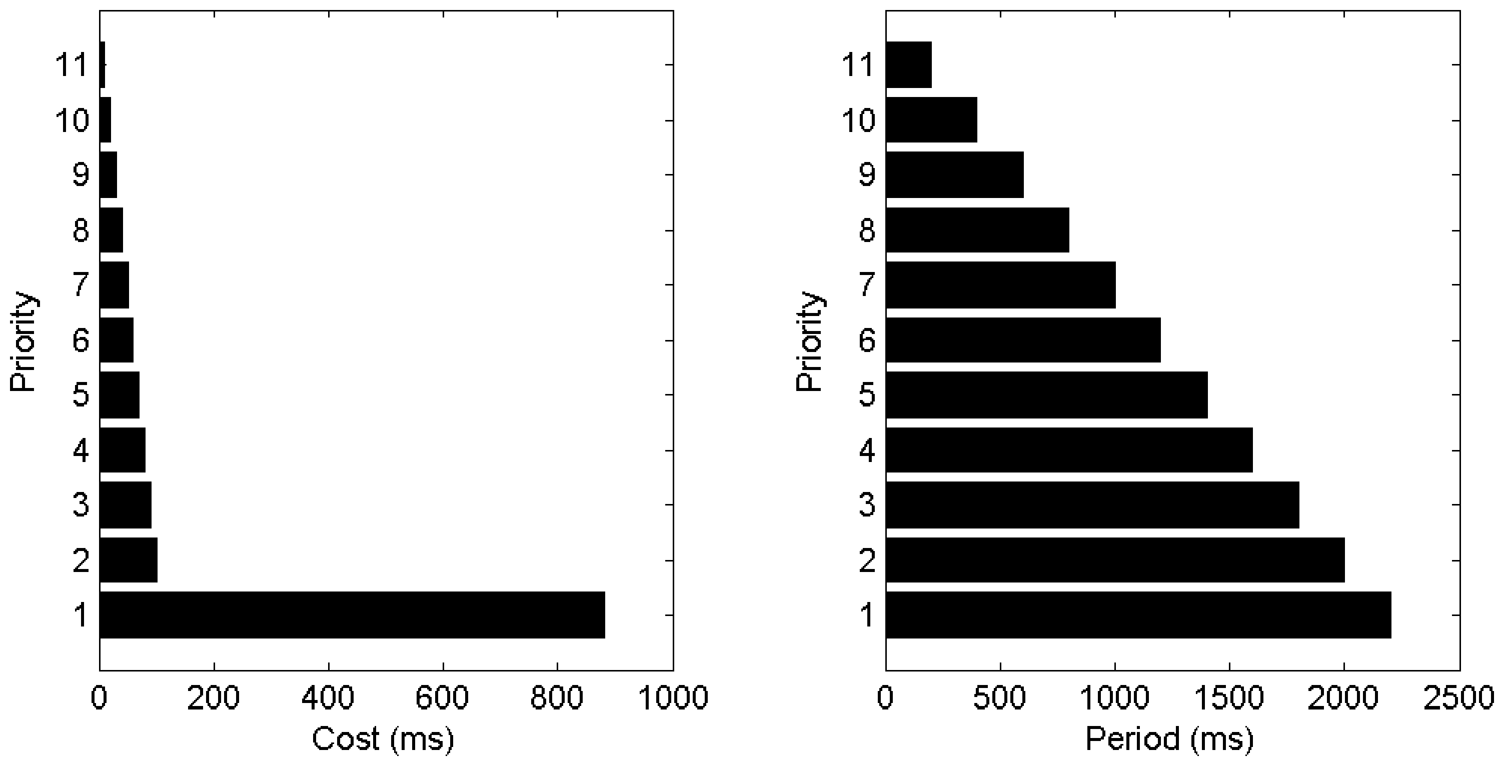
Memory Consumption



Task Set Generation

$$T_i = (n-i) * 20 * C, \qquad i = 0, \ldots, (n-1)$$
$$C_i = \begin{cases} (n-i) * C, & i = 1, \ldots, (n-1) \\ (19-n) * n * C, & i = 0 \end{cases}$$

Test Setup



11
10
9
8
7
6
5
4
3
2
1
Priority
0
200
400
600
800
1000
Cost (ms)
11
10
9
8
7
6
5
4
3
2
1
Priority
0
500
1000
1500
2000
2500
Period (ms)

Name	Queue Type	Implementations	Operations	L.P. Operations	Cost
1. Write Queue	Write Queue	ptyz, timesys, anderson	1 Write	* Reads	5 ms
2. Write Queue - no read	Write Queue	ptyz, timesys, anderson	1 Write	1 Write	1 ms
3. Write Queue - no anderson	Write Queue	ptyz, timesys	20 Writes	* Reads	1 ms
4. Read Queue	Read Queue	ptyz, timesys, anderson	1 Read	* Writes	5 ms
5. Read Queue - no write	Read Queue	ptyz, timesys, anderson	1 Read	1 Read	1 ms
6. Read Queue - no anderson	Read Queue	ptyz, timesys	20 Reads	* Writes	1 ms