Combining RTSJ with Fork/Join
A Priority-based Model

Cláudio Maia, Luís Nogueira, Luís Miguel Pinho

JTRES 2011, York, United Kingdom
September, 27, 2011

CISTER Research Centre in
Real-Time Computing Systems
FCT Research Unit 608

Agenda

- Motivation
- Challenges & Research Direction
- Framework Proposal & Related work
- Fork/Join Model & Work Stealing
- System Model & Integration Challenges
- Future Work

Motivation

- Embedded systems are starting to incorporate multiple processor architectures
  - Uniprocessor architectures are **not efficient** to implement anymore
  - **Reduction** in the production costs and improved energy efficiency
- **Stringent** operation requirements, such as
  - low memory footprint
  - low power consumption
  - timing constraints

Motivation

- OSes and Java VMs running on uniprocessor systems are **multiprogrammed environments**
  - Applications execute concurrently in order to maximise the utilisation of system resources
- Evolution from uniprocessor systems to multiprocessor systems
  - It is not sufficient to **migrate or adapt** current sequential programming models or tools
  - Penalty: **underutilisation** of system resources
- Natural Evolution
  - Applications need to be **parallelised** so that system throughput is increased, through the efficient management of system resources.

Challenges

- Creation of new **parallel programming models**
  - Efficiently take advantage of parallel platforms and architectures
  - Requires
    - data structures
    - algorithms and
    - code generation tools
- Programming models should be independent on the number of processors
    - Particularly as the number of cores largely increases
    - Nº tasks < nº of processors

Research Direction

- Explore **new programming models** that combine
  - parallel systems
  - embedded real-time systems
- Solve the **limitations** of current embedded real-time OS and VM environments
  - Lack of programming models and tools to handle the parallel execution of applications

Framework Proposal

- Parallel execution of dynamic real-time applications
  - Objective of optimising resource utilisation
- Applications are composed by a set of complex tasks that can be divided into smaller units of execution
- Integrates RTSJ with the Fork/Join model
- Goal is to execute on top of a real-time Java virtual machine
  - Advantages
    - Open-source nature, platform-independence, and application's portability
    - RTSJ
    - Drawback: performance

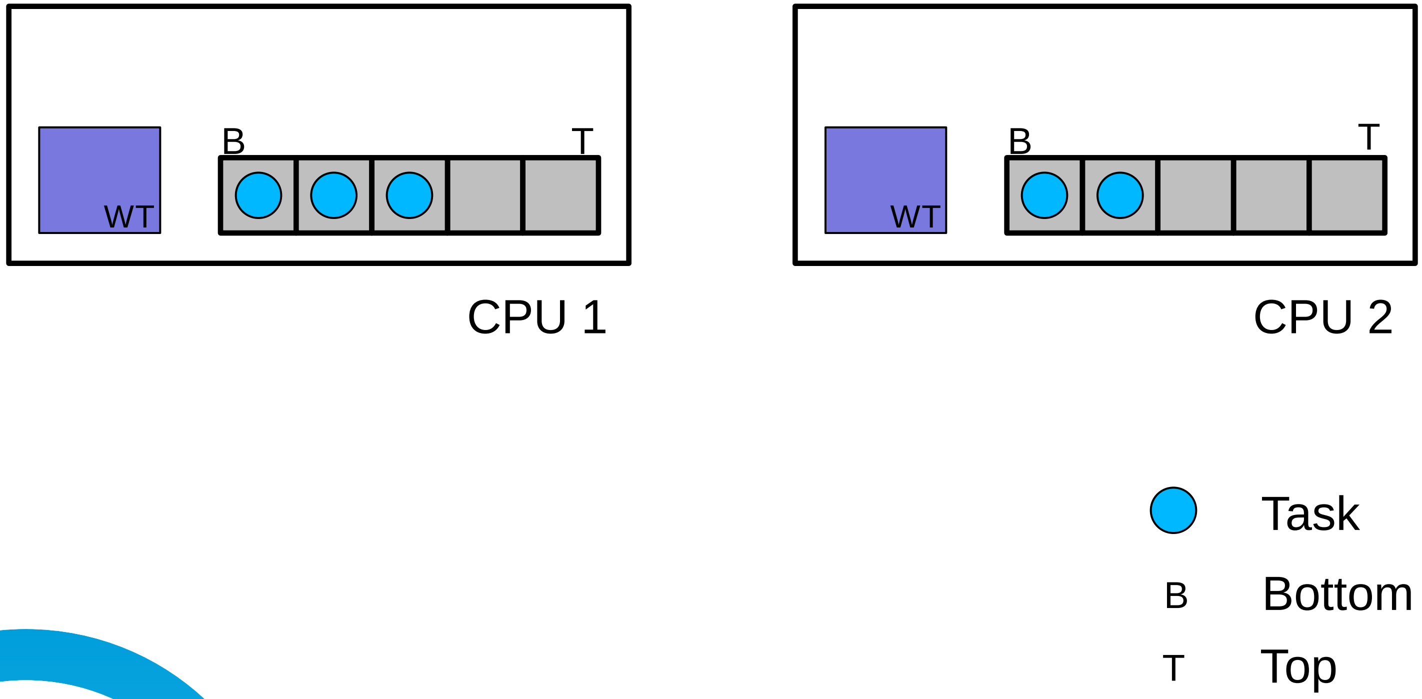
Related Work

- RTSJ
    - Limitations concerning multiprocessor support
        - Mapping of schedulable objects to processors
        - A fixed priority scheduler with a single run queue per priority level (global, partitioned and mixed require adaptation)
        - ...
    - Garbage collection on multiprocessors
        - Has to be further studied
- Parallel Systems
    - Cilk, Java Fork/Join, OpenMP
        - Encourage programmers to divide their applications into parallel blocks which are assigned to processors

Fork/Join Model Concepts

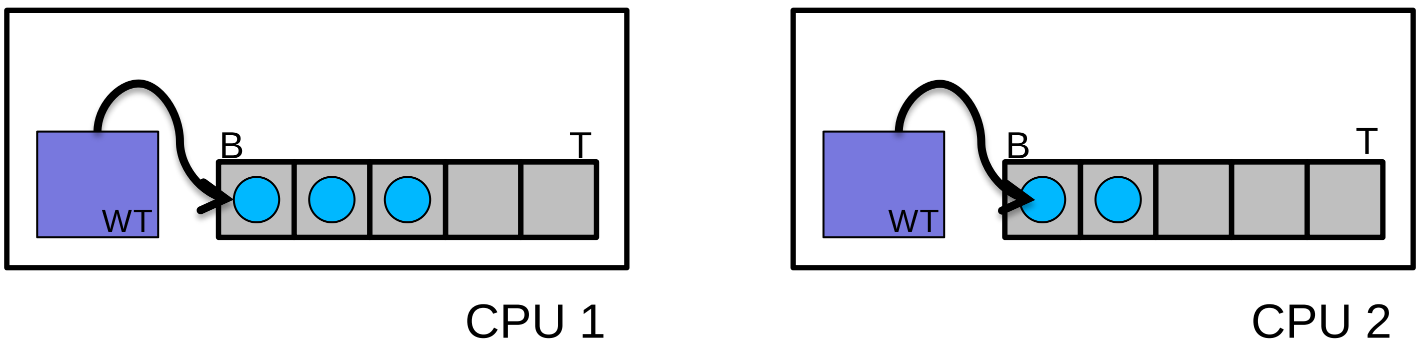
- Principle of divide and conquer
  - Fork tasks into subtasks in a recursive manner
  - Join to wait until subtasks complete (blocking point)
  - Examples: Fibonacci, Image processing
- Implementations rely on work-stealing
  - **Worker Thread (WT)** per processor with its own scheduling double-ended queue (**deque**)
  - Deques support LIFO and FIFO operations
  - **LIFO**
    - WT processing their own deques
  - **FIFO**
    - WT steals work from other worker threads
  - Subtasks generated by tasks are pushed into that WT deque
  - WT become idle when there’s no work to do

Work-Stealing (Visual Representation)



Work-Stealing (Visual Representation)

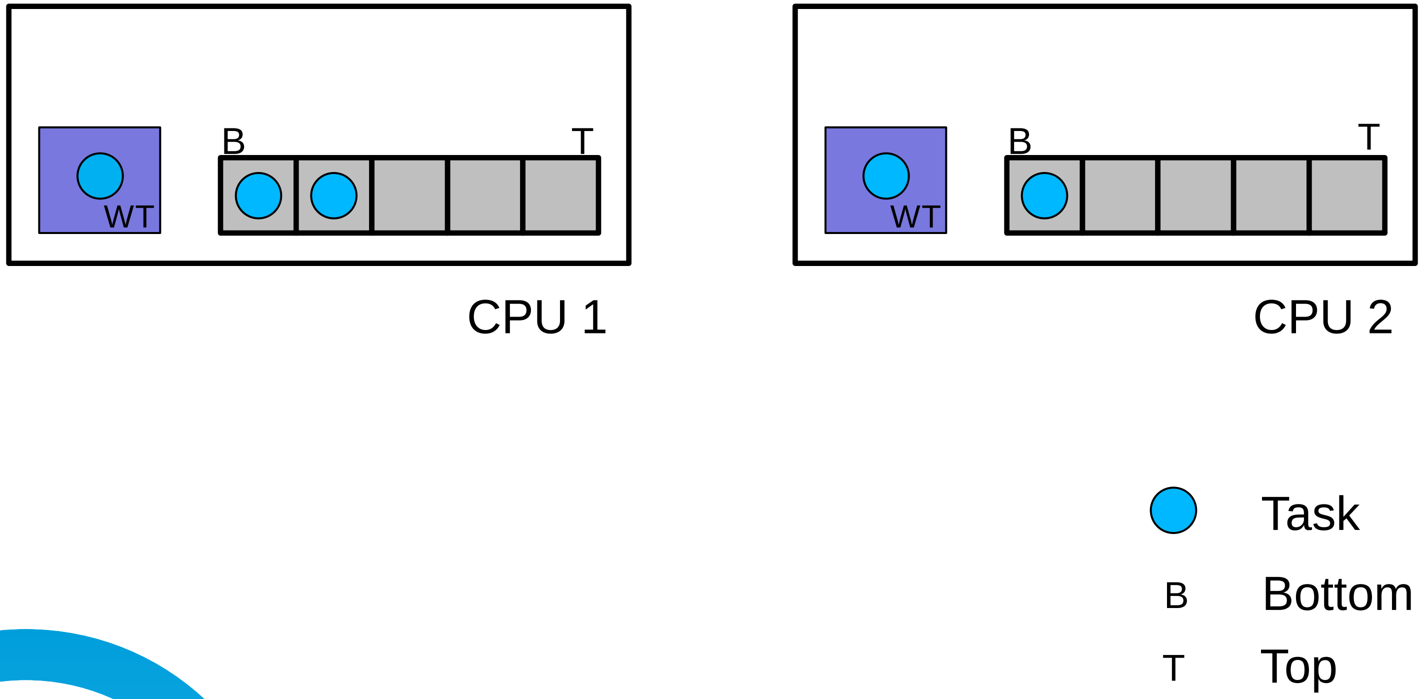
- Work Threads process work from the bottom of the queue



Task
B Bottom
T Top

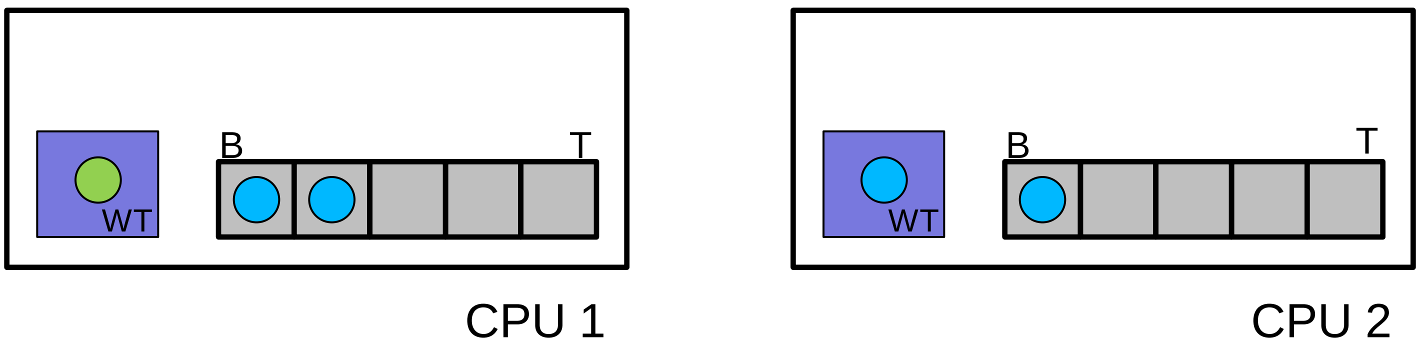
Work-Stealing (Visual Representation)

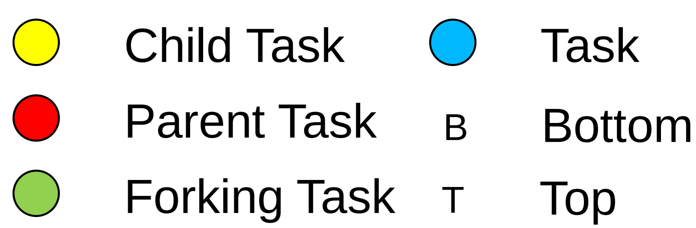
- Work Threads process work from the bottom of the queue



Work-Stealing (Visual Representation)

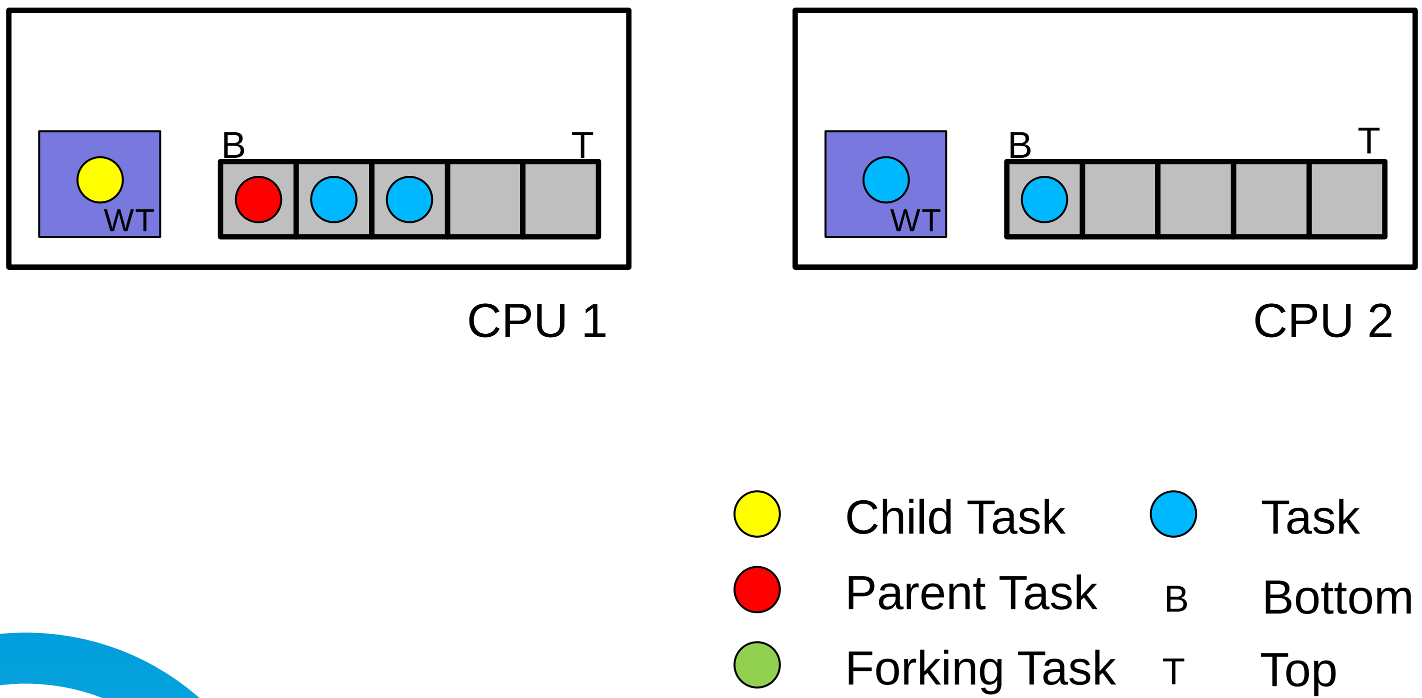
- If a task spawns a new child, then the parent is pushed to the bottom of the deque and the processor executes the child task





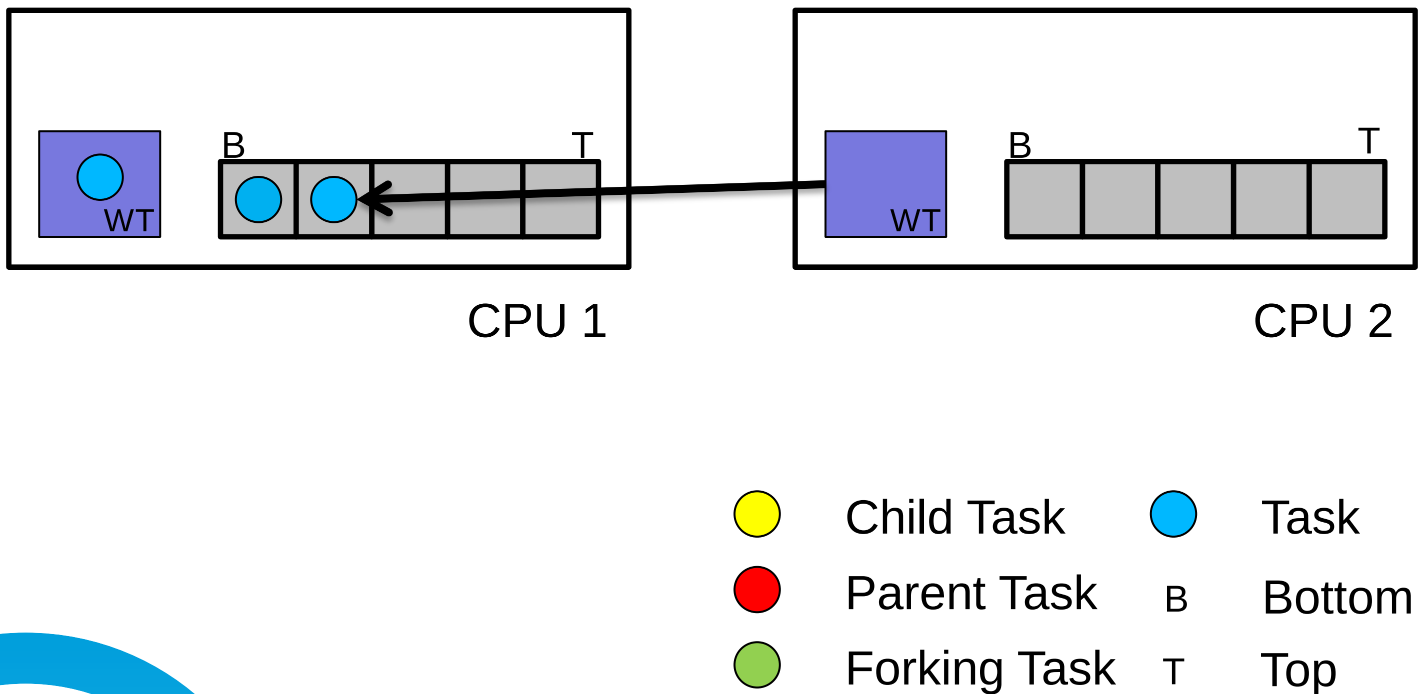
Work-Stealing (Visual Representation)

- If a task spawns a new child, then the parent is pushed to the bottom of the deque and the processor executes the child task



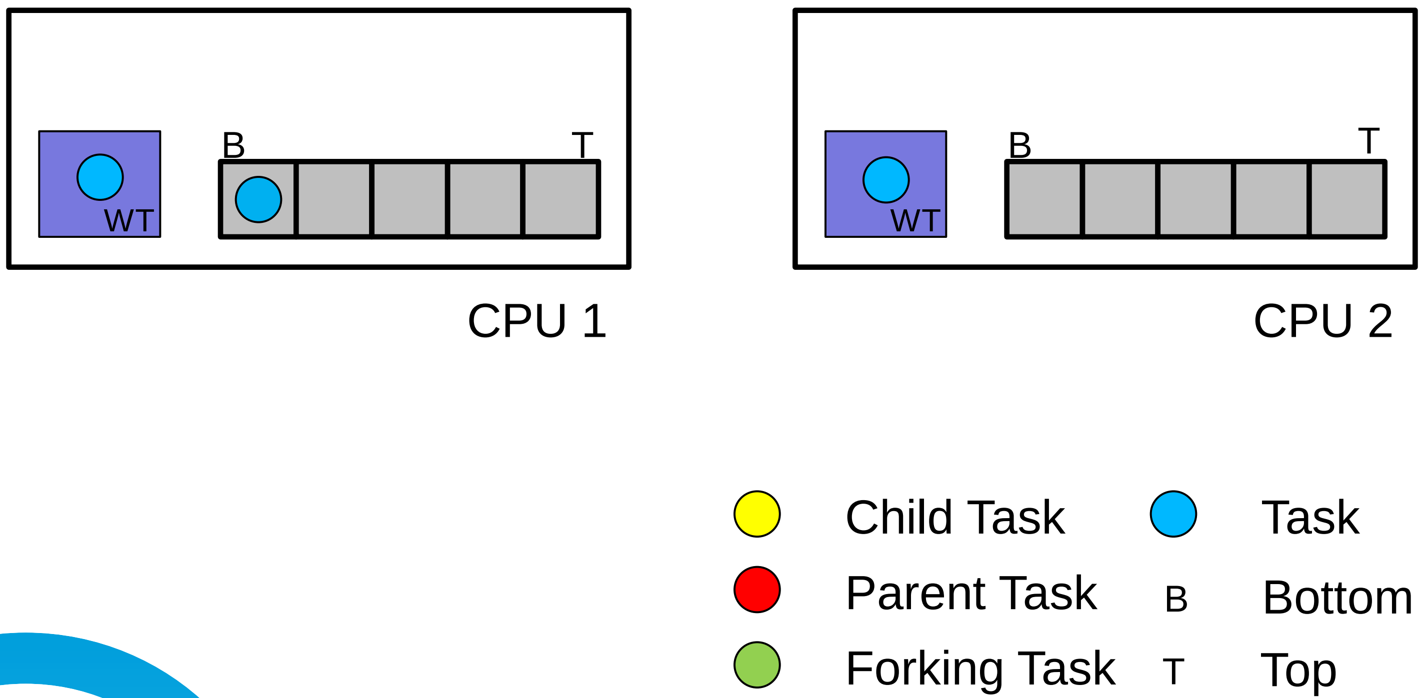
Work-Stealing (Visual Representation)

- If a deque is empty, the Worker Thread steals work (the topmost task) from other processor's deque



Work-Stealing (Visual Representation)

- If a deque is empty, the Worker Thread steals work (the topmost task) from other processor's deque

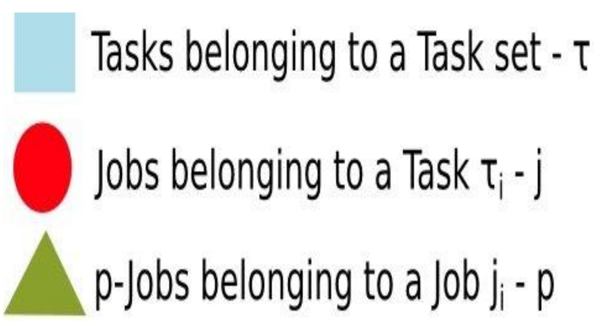


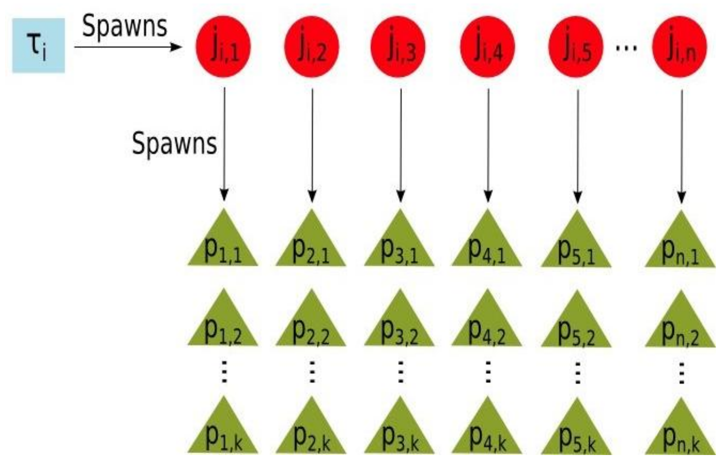
Work Stealing Advantages

- Reducing task contention
  - LIFO
    - WT Processing own tasks
  - FIFO
    - WT stealing from the opposite side of the deque
- Initial tasks generate more work, which affect
    - Amount of stealing operations
    - Task decompositions

System Model

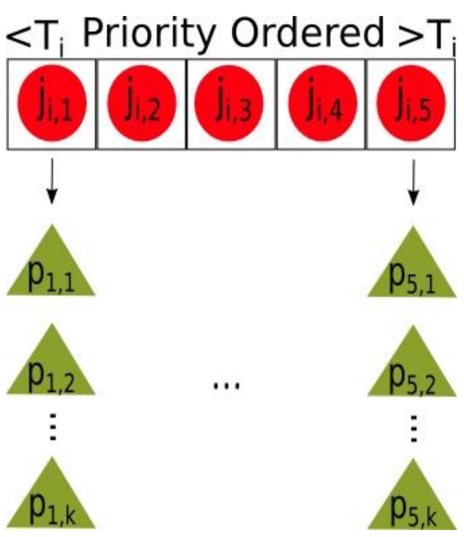
- Sporadic and independent tasks on *m* identical processors
- Tasks release jobs at sporadic time intervals and the execution requirements are only known at runtime
- Jobs may spawn a set of parallel jobs (FJ tasks)
- p-Jobs – work units that can be executed in different processors at the same time instant

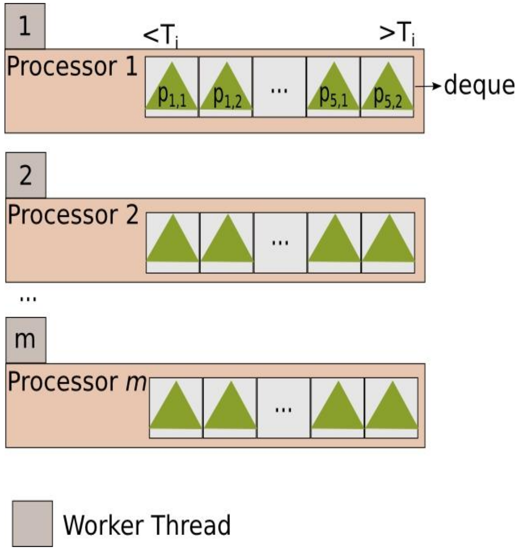




System Model

- Jobs are scheduled according to its priority and placed in a global submission queue
- p-Jobs inherit the timing properties of the job that spawn it
- Each processor has its own worker thread and deque where p-Jobs will be pushed/popped according to a WS policy





WS Priority-Inversion

- Two cores execute two threads
  - T_m in Core 1 (medium priority)
  - T_h in Core 2 (high priority)
- T_m generates p-Jobs (placed in Core 1 deque)
- Meanwhile, T_{h2} (high priority) is ready and preempts T_m in Core 1
- T_{h2} p-Jobs are placed in Core 1's deque, pushing older p-Jobs (T_m) to the end of the queue
- If Core 2 has no work to do, it may steal older p-Jobs from Core 1's deque (generated by T_m) causing priority inversion
- However, if work stealing wouldn't be applied, Core 2 would remain idle

Integration Challenges (RTSJ/FJ)

- Task Scheduling
  - Respect the properties of both
    - Real-time tasks and work-stealing
  - Therefore, we should carefully take into account
    - Timing properties of real-time tasks through feasibility analysis
    - Impacts of task migration
    - Predictability of the system
- Memory Management
  - Garbage collection (it is always a concern ☺)
  - Memory regions per WT
    - Using portals to share p-Jobs maybe a solution (due to the imposed scope assignment rules)
- Native multiprocessor support in the JVM

Future Work

- The definition and specification of a real-time scheduling algorithm based on work-stealing
    - Considering the preliminary system model just presented
- Implementation of this scheduling scheme using RTSJ and FJ
- Specification of memory-related concepts
    - Scopes/ GC

Thank You!

Questions?