

# *Distributed Synchronization with Shared Semaphore Sets*

Cristian Țăpuș

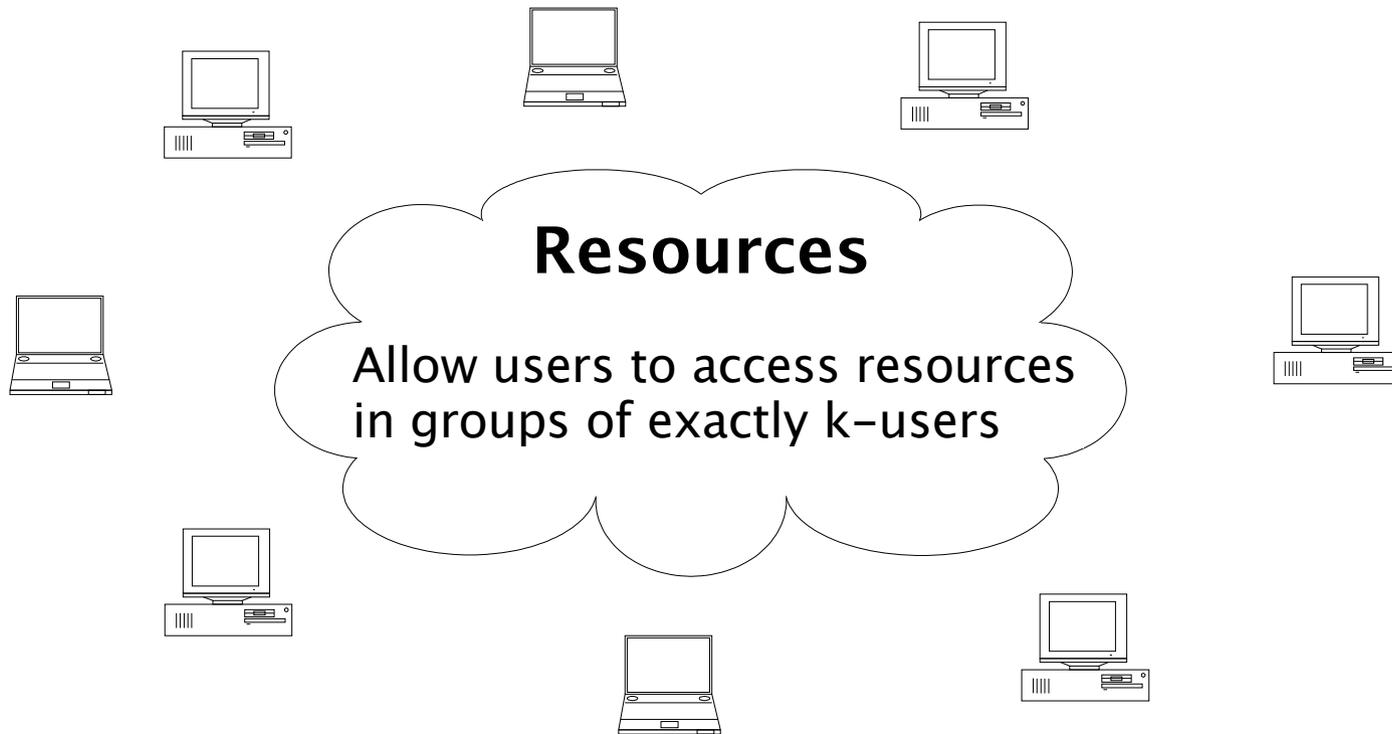
Jason Hickey



Mojave

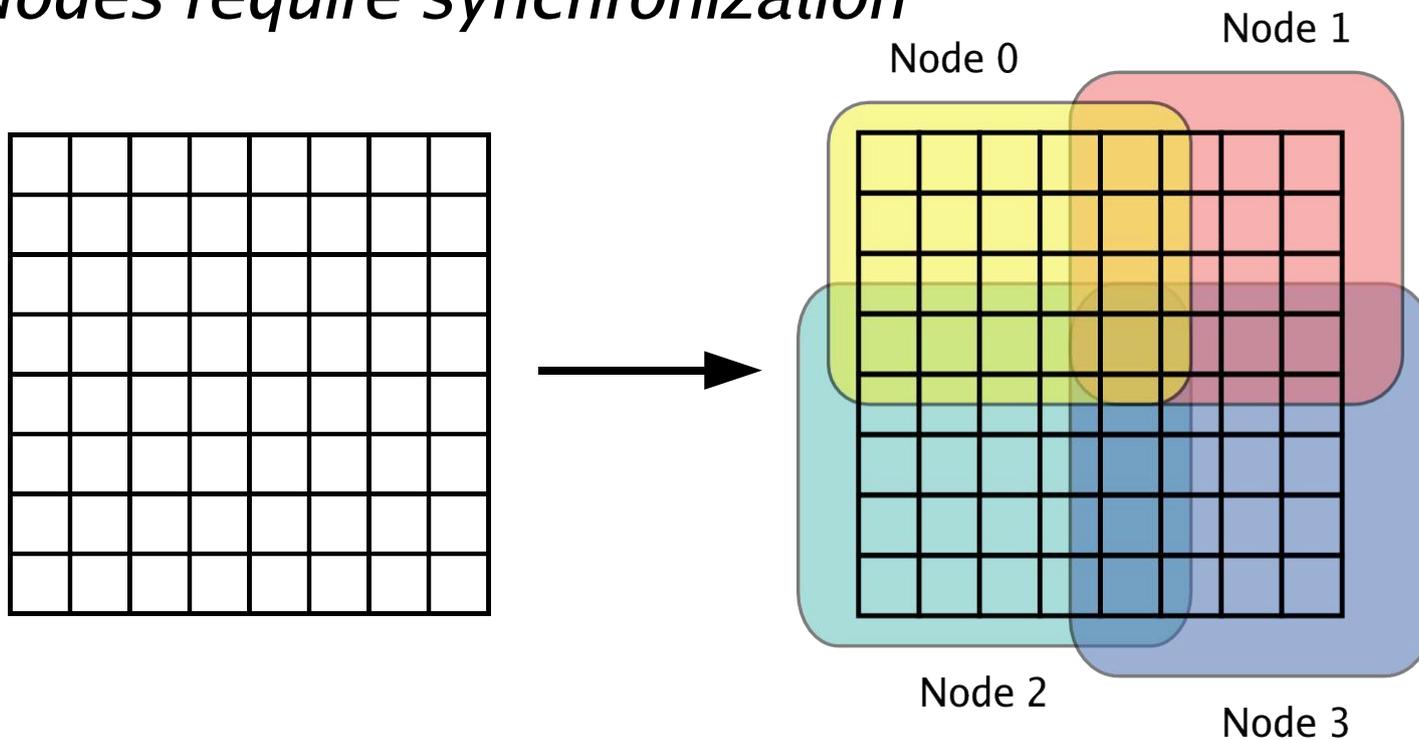
# What is Distributed Synchronization

- Resource sharing problem
  - *Restrict access to resources for economic and/or safety reasons*



# Why Distributed Synchronization?

- Scientific computing
  - *Divide computation space for efficiency*
  - *Nodes require synchronization*

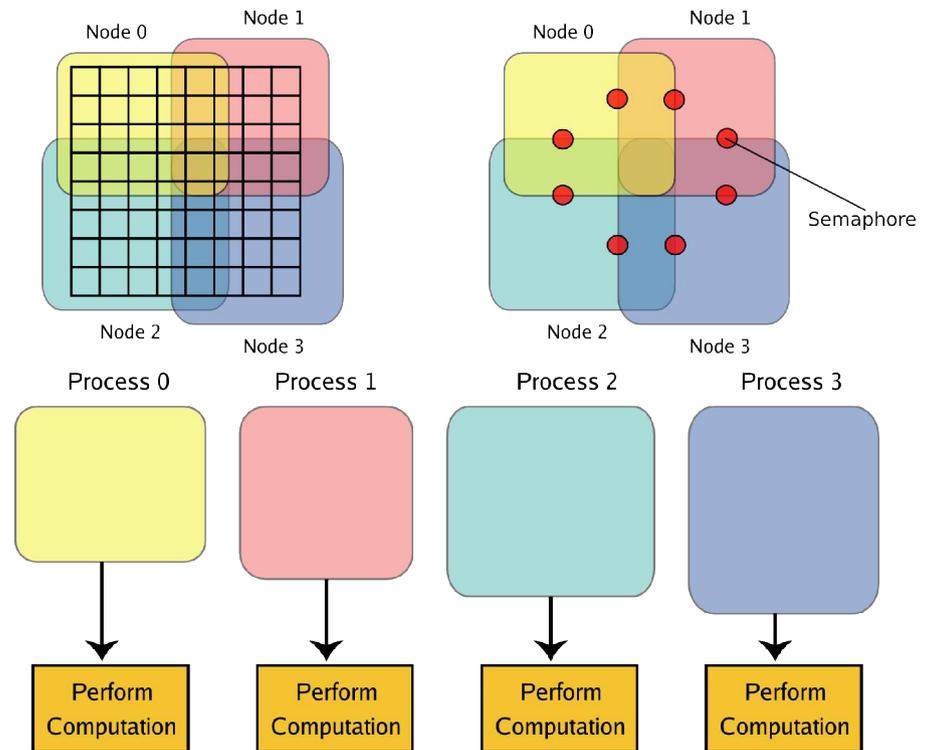
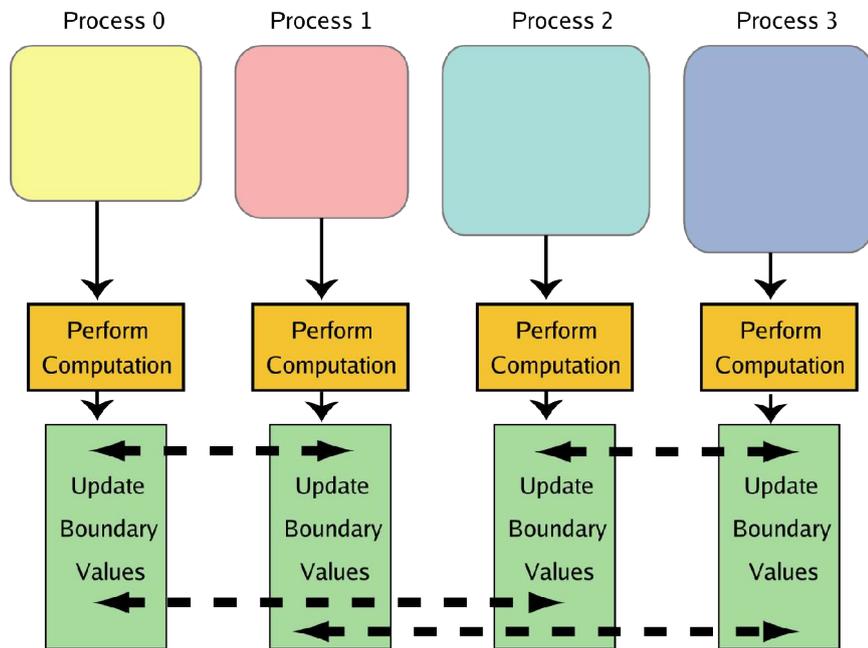


# Why Semaphores?

- Alternatives (i.e. MPI)
  - *Complex*
  - *Require more programming knowledge*
  - *Error-prone*
- Semaphores are a classic method for enforcing synchronization
  - *Introduced in 1960s by Dijkstra*
  - *Simple programming paradigm*
  - *Proberen and Verhogen operations*
- Need synchronization without shared memory



# MPI vs. Shared Mem/Semaphores Implementation



# Why Semaphore Sets?



- Atomically acquire/release multiple resource
- Less prone to deadlock
- Simpler, more elegant code
- Could increase performance due to locality



# Outline

- System V IPC API
- KDIPC System Overview
- Implementation details
- Providing Consistency
- Conclusion and Future Work



# System V IPC API

- System V IPC API is a useful, popular model
  - *Simple semantics for shared objects*
  - *Designed for tightly coupled systems*
  - *Relies on hardware for maintaining consistency*
  - *Linux kernel implementation for single and multi-processor machines*
- Shared semaphores
  - *Semget: register a shared semaphore set*
  - *Semop: perform an atomic operation on one or more semaphores in the shared semaphore set*
  - *Semctl: “control” operations on the shared semaphore set*



# *KDIPC System Overview*

- When generalizing to distributed IPC, semantics must be preserved
  - *Distributed environments can't rely on hardware support for maintaining consistency*
  - *Software support for reliable data distribution*
  - *Only minimal changes to application's source code*
- Programs expect *sequential consistency*
- Total order communication protocol
  - *Obvious choice*
  - *Usually expensive*



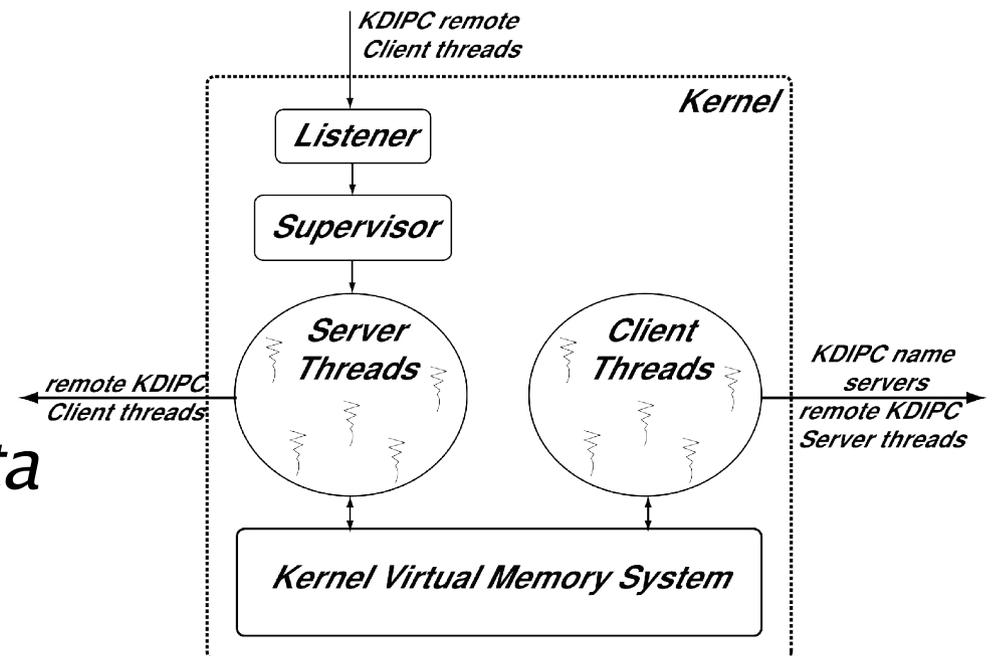
# *KDIPC System Overview*

- Kernel level implementation (2.4.1 kernel)
  - *Pros*
    - *Speed*
    - *Integration with existing System V IPC API*
    - *Possibly becoming part of the Kernel distribution*
  - *Challenges*
    - *Hard to debug*
    - *No documentation for the Linux kernel*
    - *Continuously changing environment*



# Implementation Details

- Listener for incoming requests
- Supervisor manages Server and Client Threads
- Server Thread
  - “serves” requests from remote Clients
- Client Thread
  - “retrieves” required data from remote locations



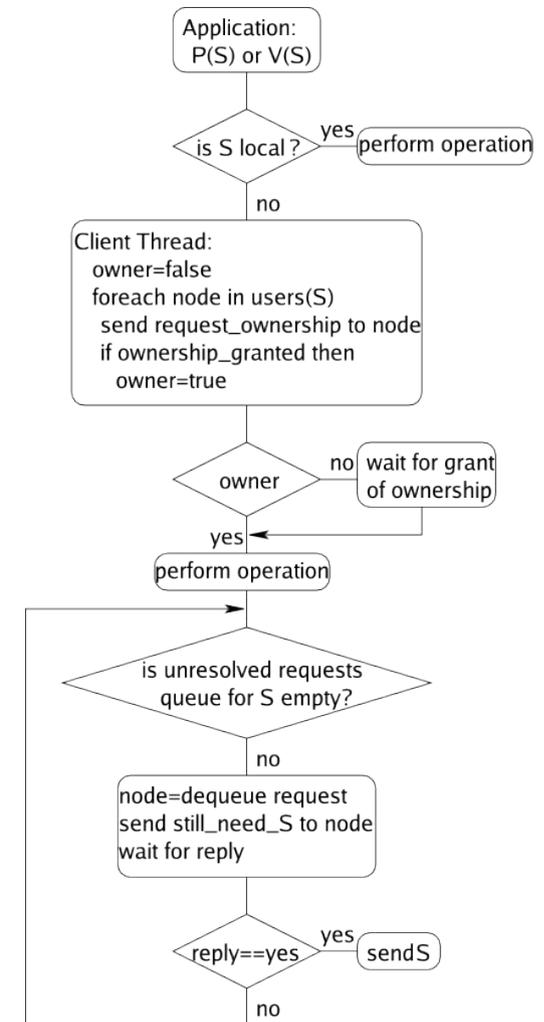
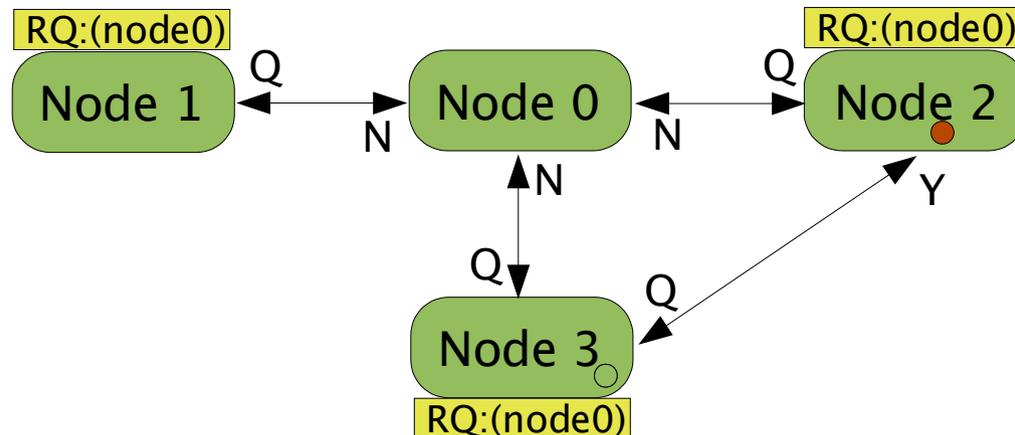
# Semantics of the access operation

- Accesses are blocked until semaphore set state can satisfy the operation
  - Semaphore:  $(1,0,5,3,2,4)$
  - Operation:  $(-1,0,0,-1,-1,0) \Rightarrow (0,0,5,2,2,4)$  **SUCCESS**
  - Operation:  $(-1,0,0,-1,-3,0) \Rightarrow (0,0,5,2,-1,4)$  **BLOCKED**
- Each semaphore set has a “sleeping” processes queue associated with it: (pid,op)
- When operations in the queue can be performed, the semaphore set is updated accordingly and the process is woken up



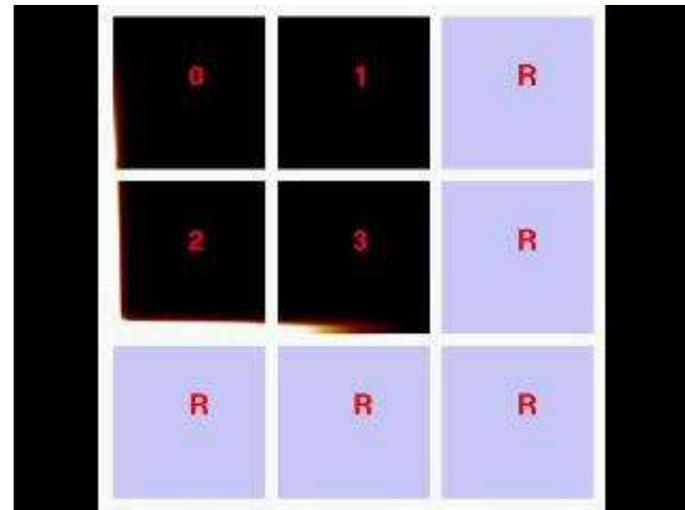
# Updates and consistency

- Keep one active copy of a semaphore set (S)
- If S is not local request if from remote location



# Experimental Setup

- Heat propagation application
- Linux cluster (dual 750Mhz PIII) using KDIPC
- Implementation used
  - *Distributed shared memory*
  - *Distributed semaphores*
- Performance
  - *~5% faster than MPI implementation*
- Simpler Code



# Conclusion and Future Work

- Kernel level inter-process communication library
  - *Provides synchronization mechanisms*
  - *Based on semaphore sets (System V IPC API)*
  - *Is independent of shared memory constructs*
- Upgrade code to latest kernel version
  - *Make it more robust and more modular*
- Investigate other protocols for consistency
  - *Efficient group communication*
- Use replication for increased parallelism
- Collaboration with Kerrighed team

