Deterministic
Fast User Space Synchronization

Alexander Züpke

alexander.zuepke@hs-rm.de

RheinMain University of Applied Sciences

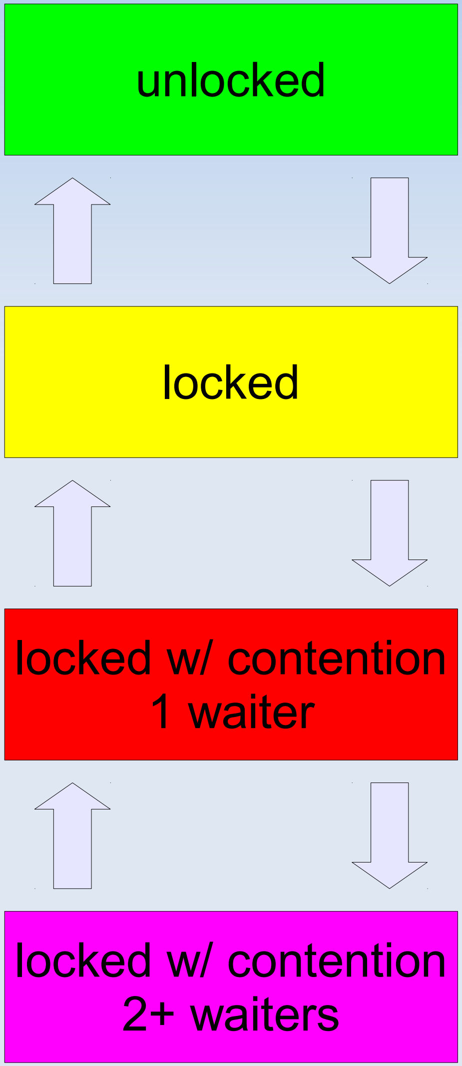
Wiesbaden, Germany

Overview

- Futex Basics
- Challenge: Futexes for Partitioning Systems
- New Approach
  - Mutexes
  - Condition Variables
  - Locking of Wait Queues
  - Robustness
- Future Work
- Summary

Mutex State Transitions

- Unlocked ↔ Locked
  - Fast path: use atomic ops
  - No system call involved!
- Contention: first waiter
  - Atomically indicate pending waiters
  - System call: suspend caller
  - Kernel allocates a wait queue object
- Contention: multiple waiters
  - Append to existing wait queue
  - Wait queue order depends, sorting if necessary



Futexes in Linux

- Futex := 32-bit integer variable in user space
- atomic CAS or LL/SC operations in the fast path
- Glibc provides:
  - Mutexes and Condition Variables
  - Semaphores, Reader-Writer Locks, Barriers, …
- Linux kernel provides system calls to:
  - suspend the caller
  - wake a given number of waiters
- First prototype in Linux kernel version 2.5.7

Futexes in Linux

- Futex API

```
#include <linux/futex.h>

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

- Operations
  - `FUTEX_WAIT` Suspend calling thread on futex `uaddr`
  - `FUTEX_WAKE` Wake `val` threads waiting on futex `uaddr`
  - `FUTEX_REQUEUE` Move threads waiting on `uaddr` to `uaddr2`
  - … more operations available → see `FUTEX(2)` man page

Motivation

- Linux Implementation
  - Requires system calls only on contention
  - Supports an arbitrary number of futexes
  - No kernel resources required until suspension
  - Also supports PI mutexes & condition variables

- Futexes are really nice

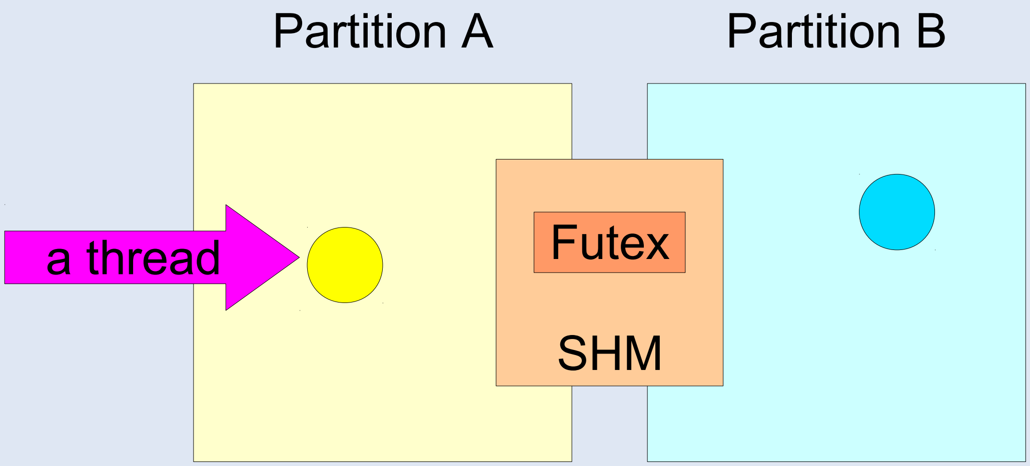
  … for Un*x Kernels

Motivation

- Linux Implementation
  - Requires system calls only on contention
  - Supports an arbitrary number of futexes
  - No kernel resources required until suspension
  - Also supports PI mutexes & condition variables
- But:
  - Can we use futexes in partitioned environments?
  - For highly safety critical systems?
  - Kernels without SLAB allocator?

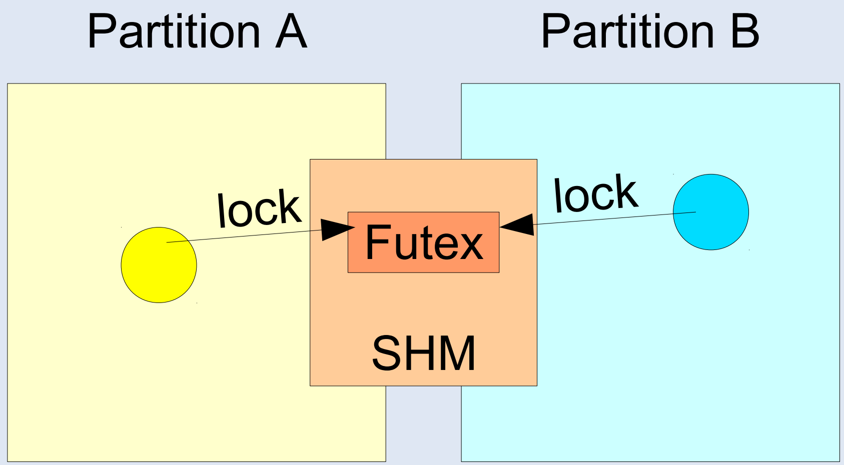
Motivation

- Define "Partitioning"
  - space and time partitioning
  - Isolated (groups of) processes
  - kernel resources are partitioned



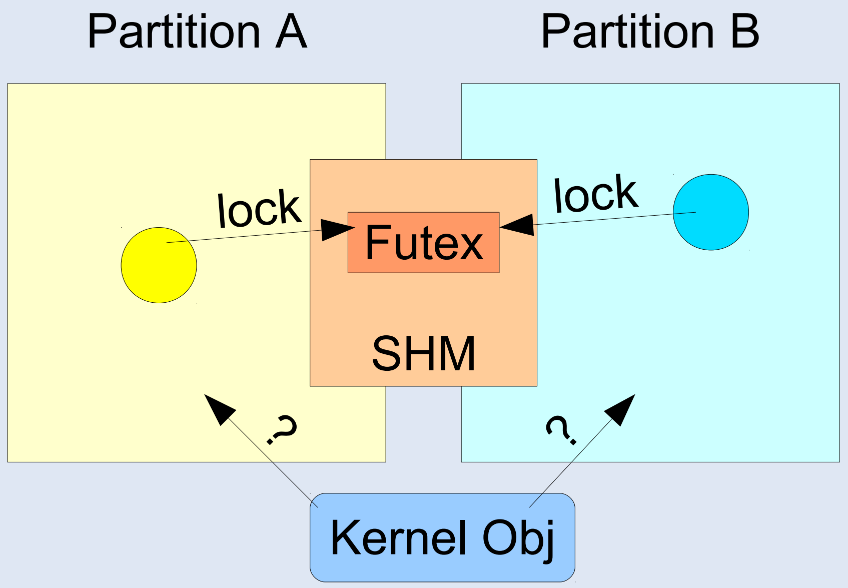
Motivation

- Define ”Partitioning”
  - space and time partitioning
  - Isolated (groups of) processes
  - kernel resources are partitioned



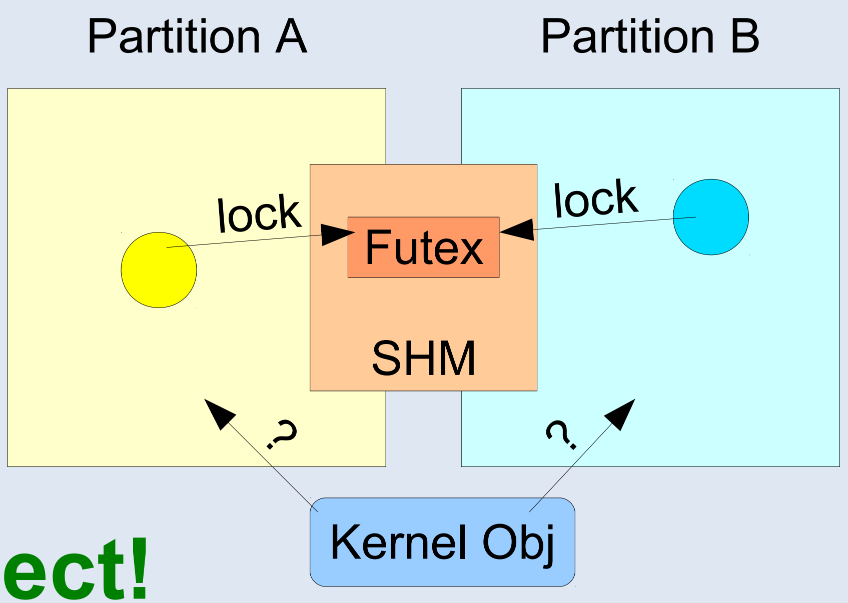
Motivation

- Define ”Partitioning”
  - space and time partitioning
  - Isolated (groups of) processes
  - kernel resources are partitioned
- Problem
  - Q: Wait queue belongs to Partition A or Partition B?
  - Pre-allocated w. queues?
    - Too pessimistic!



Motivation

- Define ”Partitioning”
  - space and time partitioning
  - Isolated (groups of) processes
  - kernel resources are partitioned
- Problem
  - Q: Wait queue belongs to Partition A or Partition B?
  - Pre-allocated w. queues?
    - Too pessimistic!
- Idea: **get rid of kernel object!**



Motivation

- Get rid of the kernel object!

- The Linux Futex implementation uses:
  - array of futex hash entries
    - lock
    - list head in-kernel objects
  - in-kernel object
    - list node in futex hash
    - key (futex address)
    - wait queue
    - lock pointer

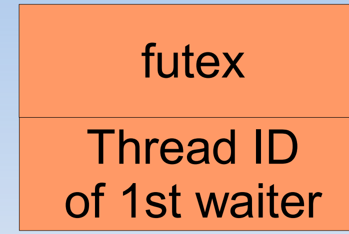
Motivation

- Get rid of the kernel object!

- The Linux Futex implementation uses:
  - array of futex hash entries
    - lock
    - list head in-kernel objects
  - in-kernel object
    - list node in futex hash
    - key (futex address)
    - wait queue
    - lock pointer

put into TCB

Requirements



- Identify correct wait queue
  - use thread ID of the first waiter
  - **put thread ID into user space, next to futex**
- Wait queue implementation in linear space
  - a priority sorted wait queue would be nice
- Locking of the wait queue
  - assume a single kernel lock for now

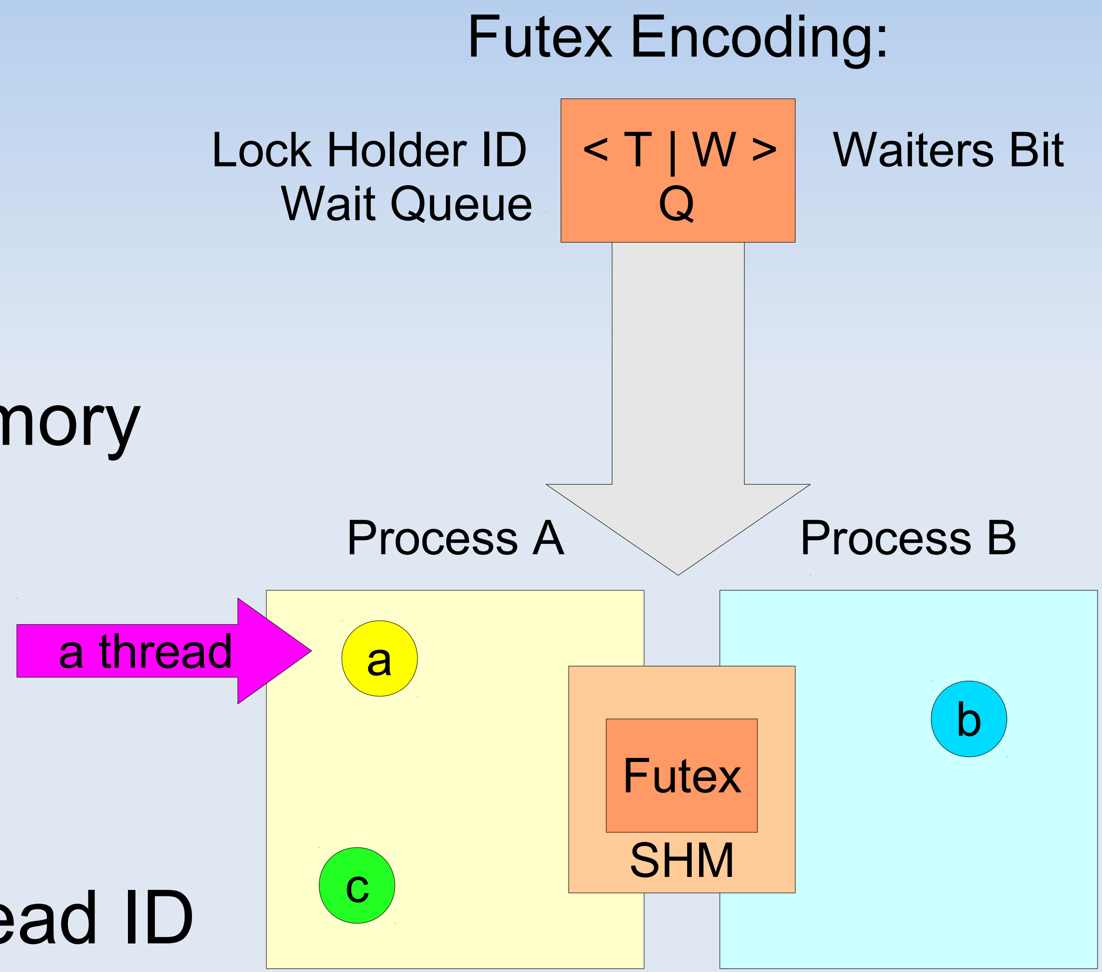
    → more on that later

Requirements

- Algorithms need bounded WCET
  - depends on # of waiters
  - # of waiters probably not known in advance
    → tricky across partition boundaries
- Wait Queues
  - doubly-linked lists are O(1) ... except searching
  - sorted wait queues with O(log n) are acceptable
    if the upper bound of O(log n) is known
  - O(n) is only acceptable if n is bounded
  - Pick FIFO-ordered doubly-linked list for now

Mutex Protocol

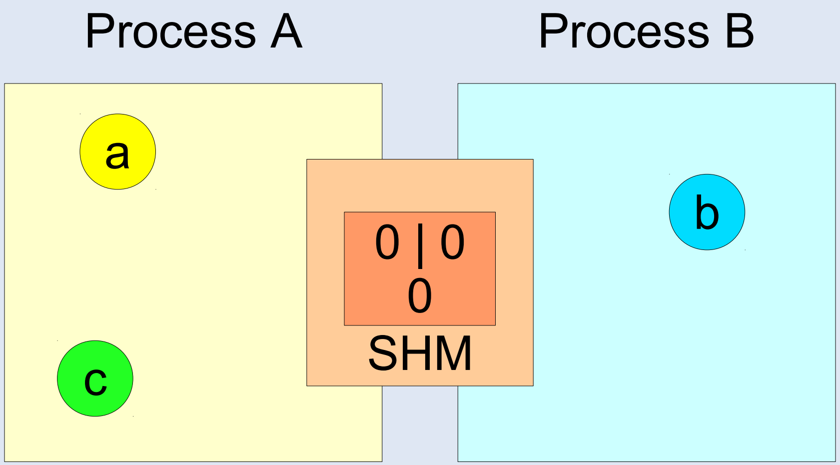
- Example
  - 2 processes
  - 3 threads
  - futex in shared memory
  - mutex protocol



- Symbols
  - T: lock holder's thread ID
  - W: bit indicating non-empty wait queue
  - Q: thread ID of first waiting thread

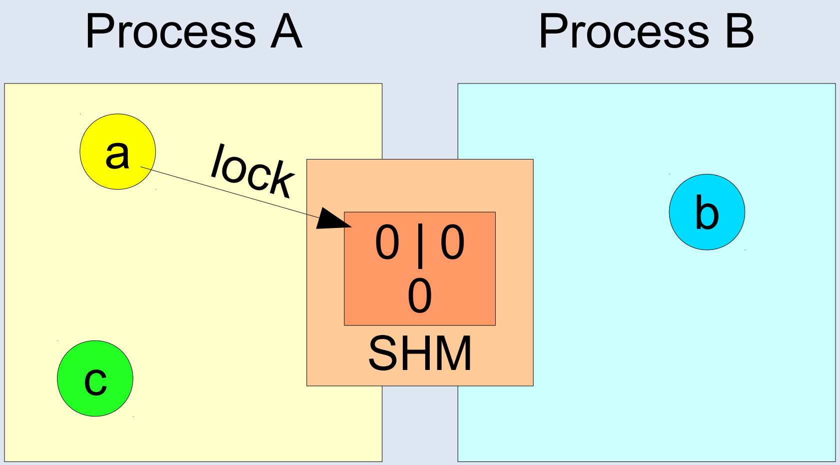
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



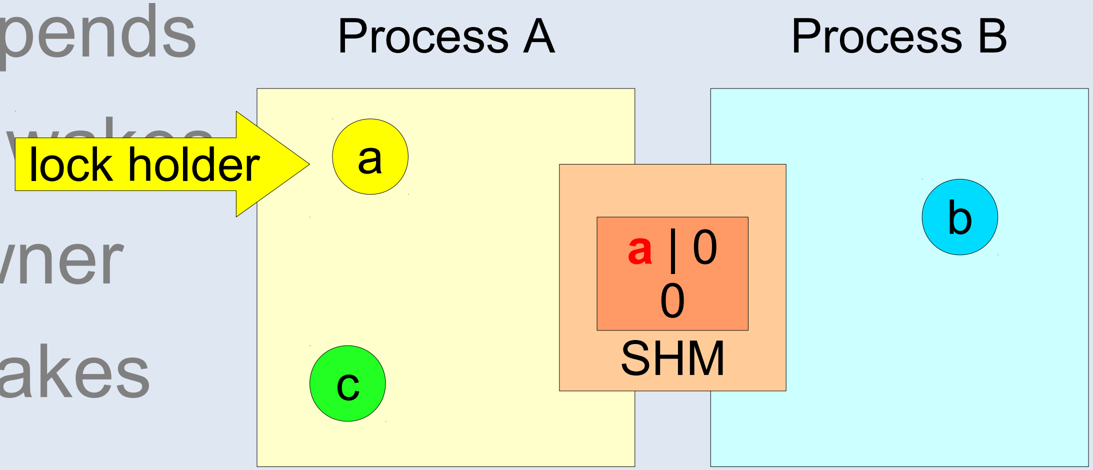
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



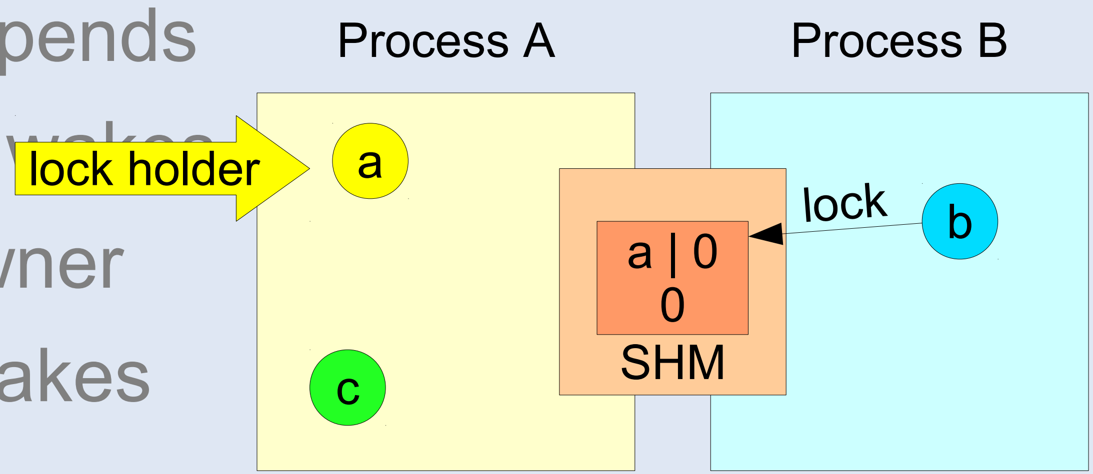
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



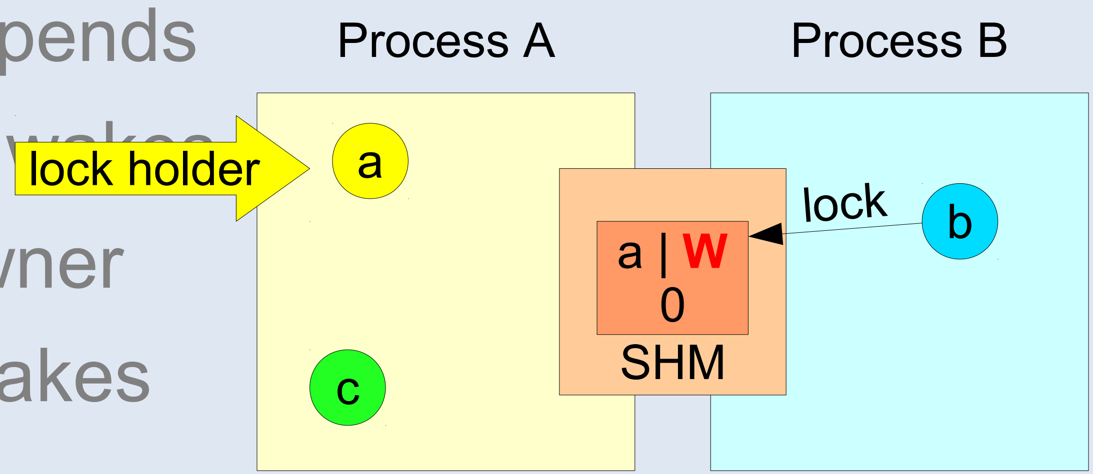
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



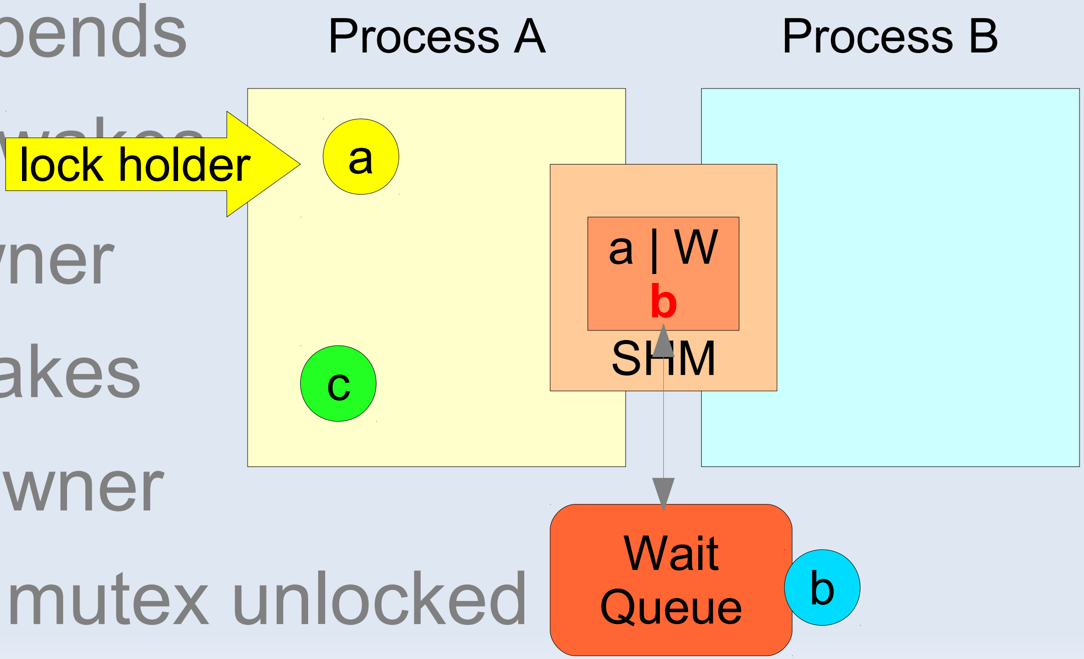
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



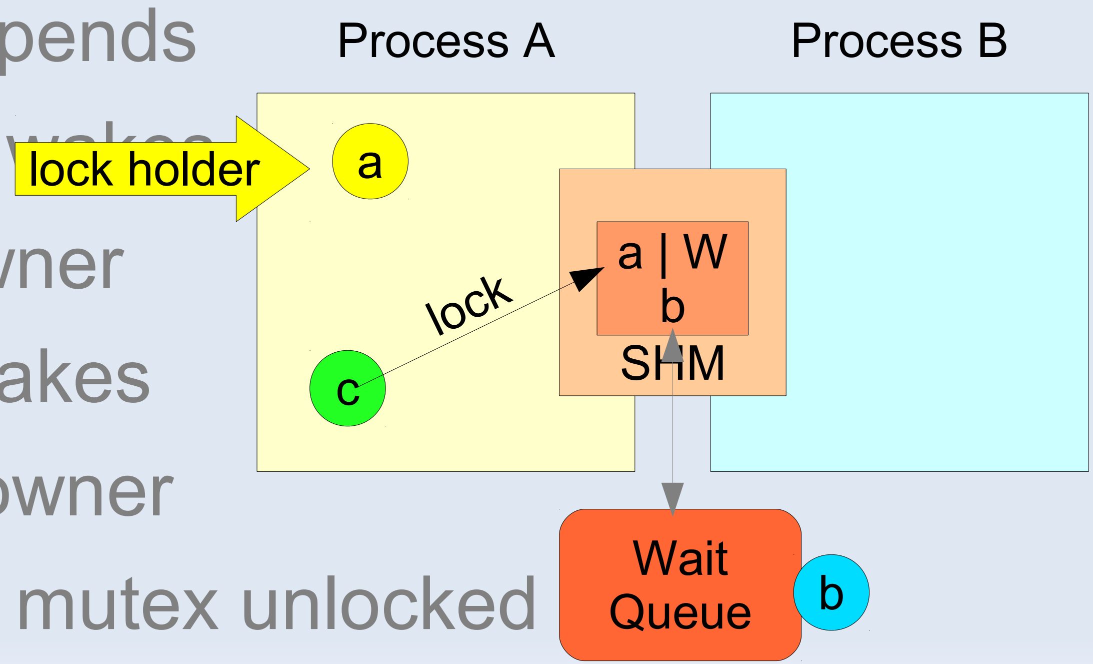
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



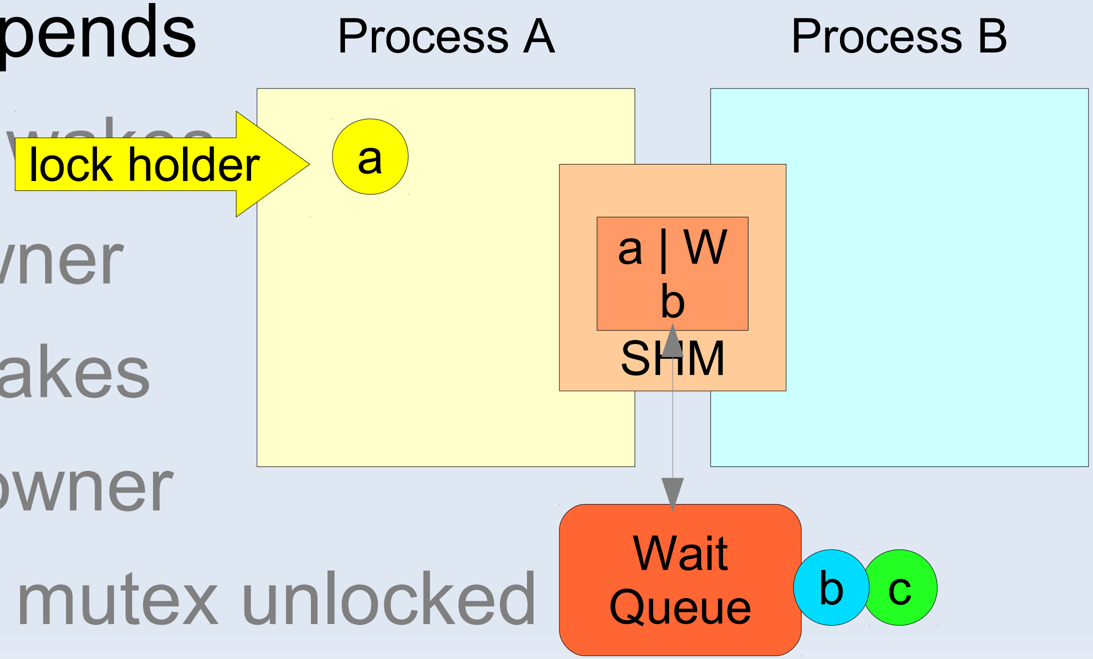
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



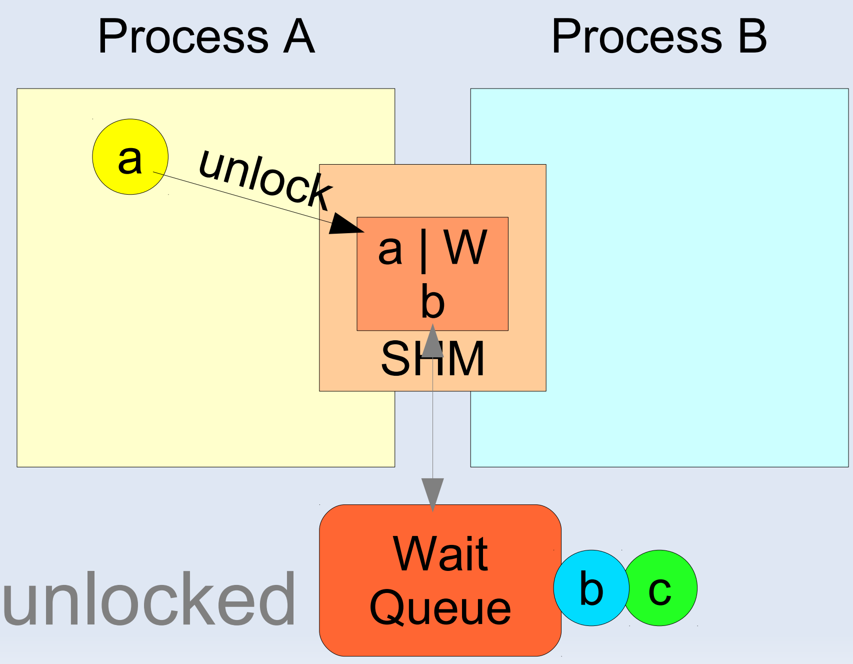
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



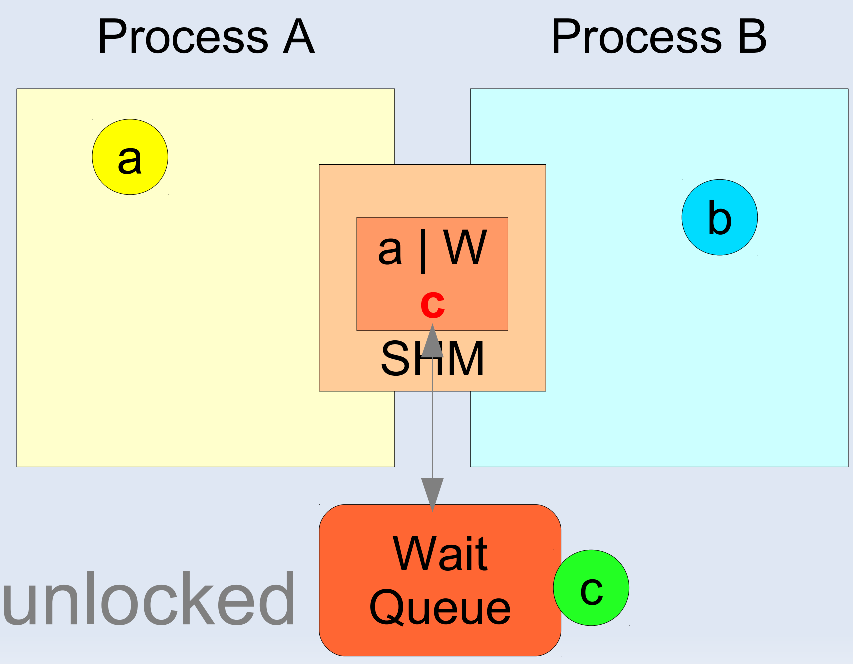
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



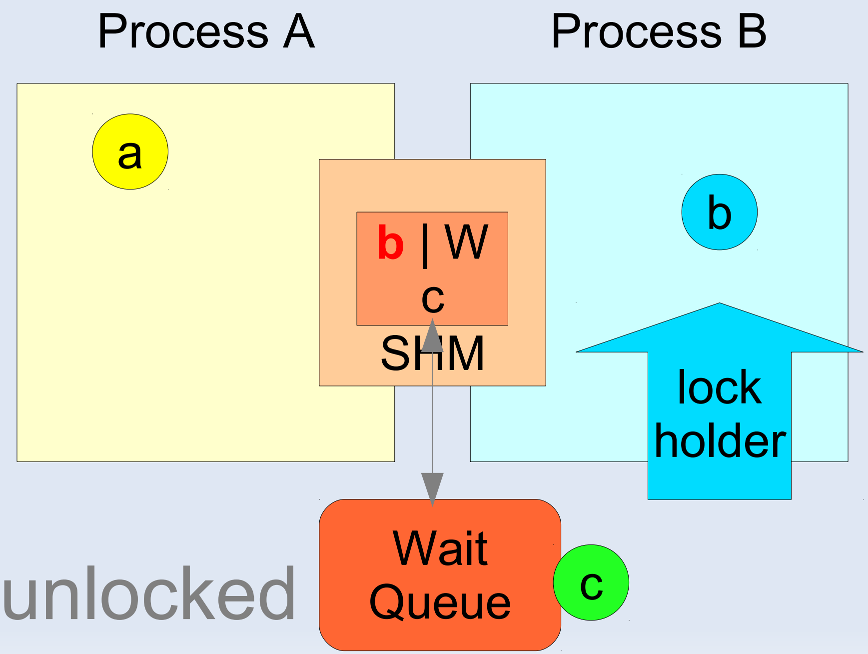
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



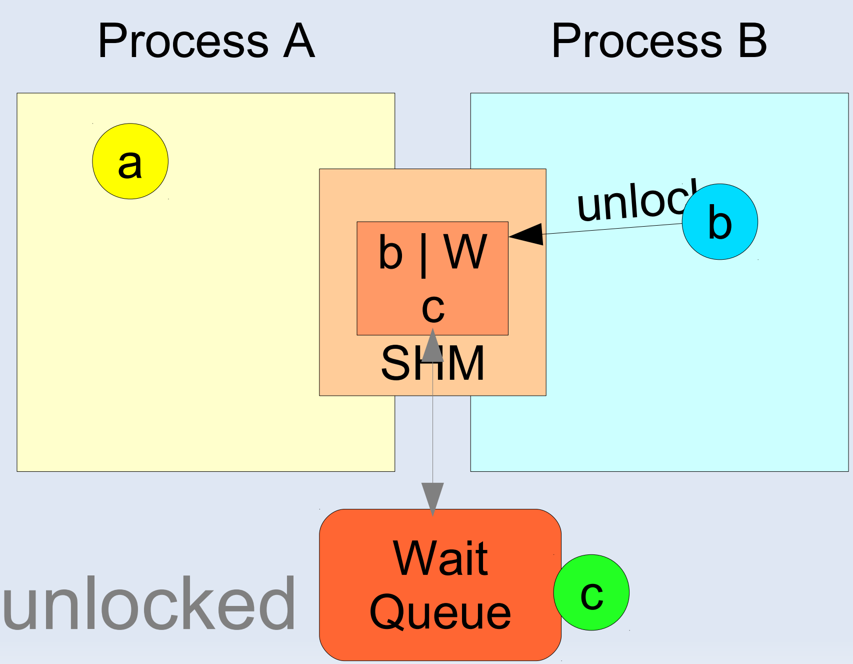
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



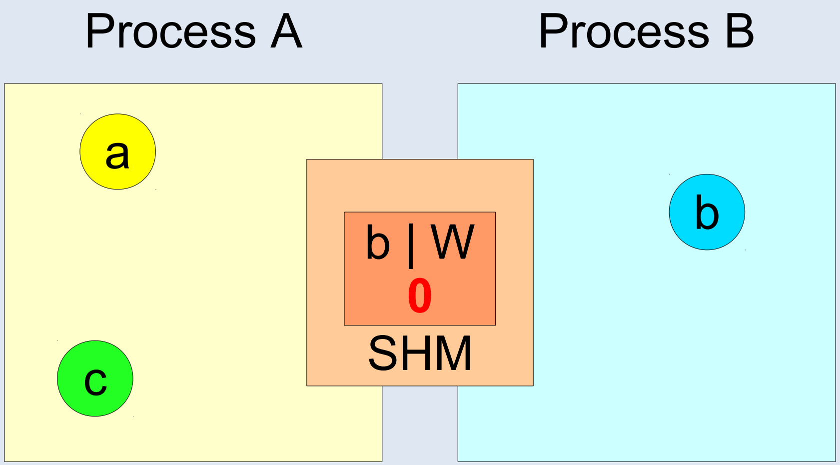
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



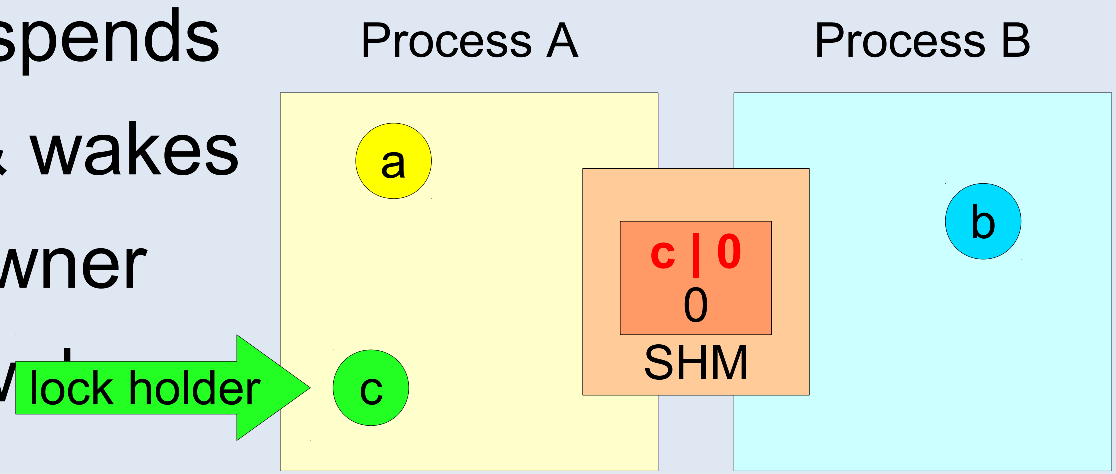
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



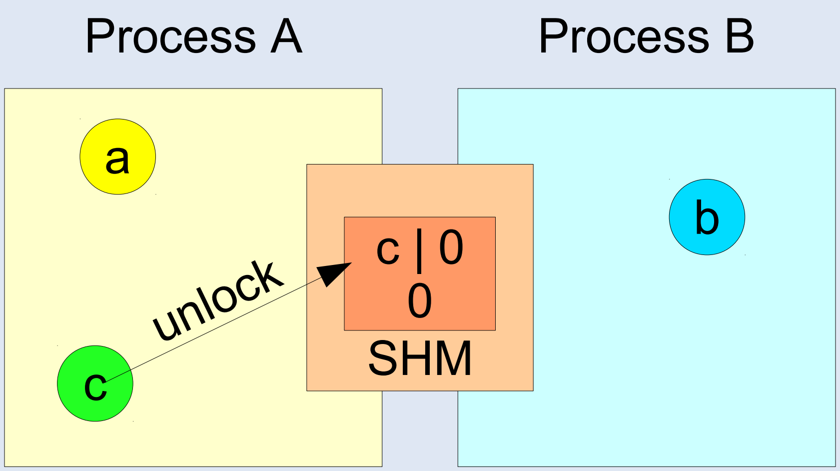
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & w
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



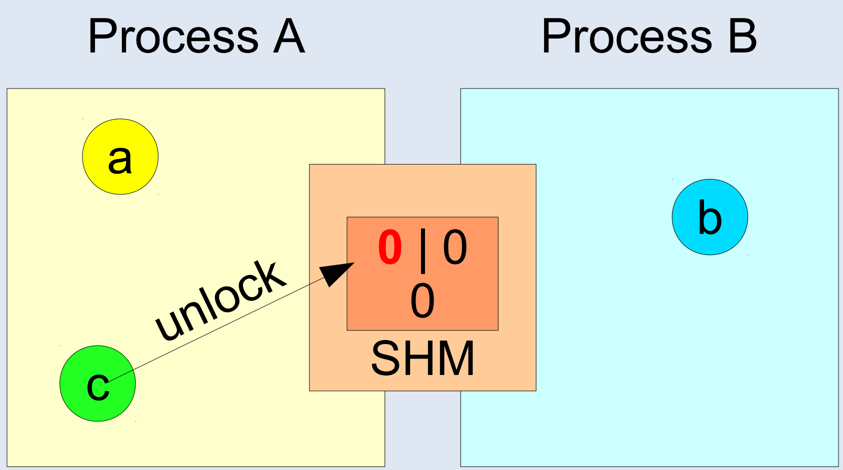
Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



Mutex Protocol

- Sequence
  - 0. initial state: mutex unlocked
  - 1. yellow tries to lock & suceeds
  - 2. blue tries & sets W & suspends
  - 3. green tries & suspends
  - 4. yellow unlocks & wakes
  - 5. blue becomes owner
  - 6. blue unlocks & wakes
  - 7. green becomes owner
  - 8. green unlocks → mutex unlocked



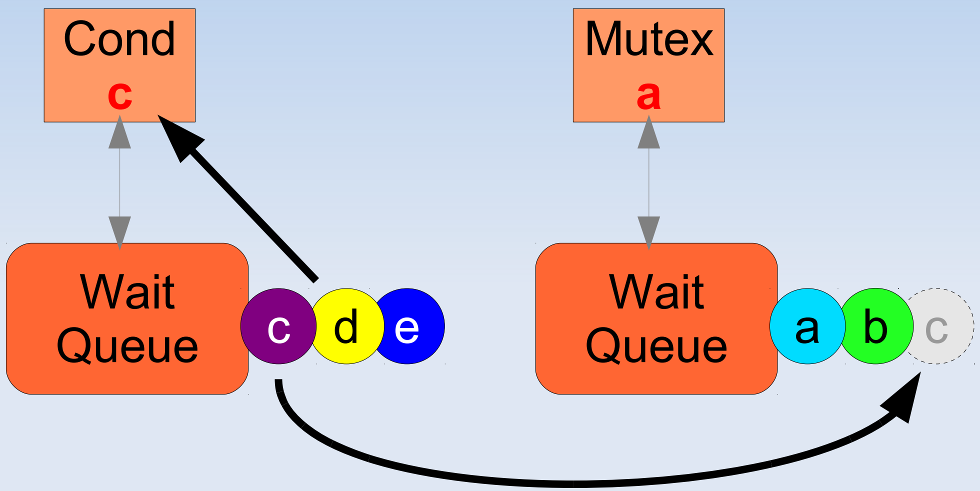
Condition Variables

- Condition Variables have a supporting Mutex
- CVs also use futexes
  - Futex value: atomic counter
  - Wait queue: maintain waiting threads

- `cond_wait()`
  - Releases mutex (caller of `cond_wait()` holds mutex)
  - Calls kernel to suspend on futex
  - On return: caller is owner of the mutex again

Condition Variables

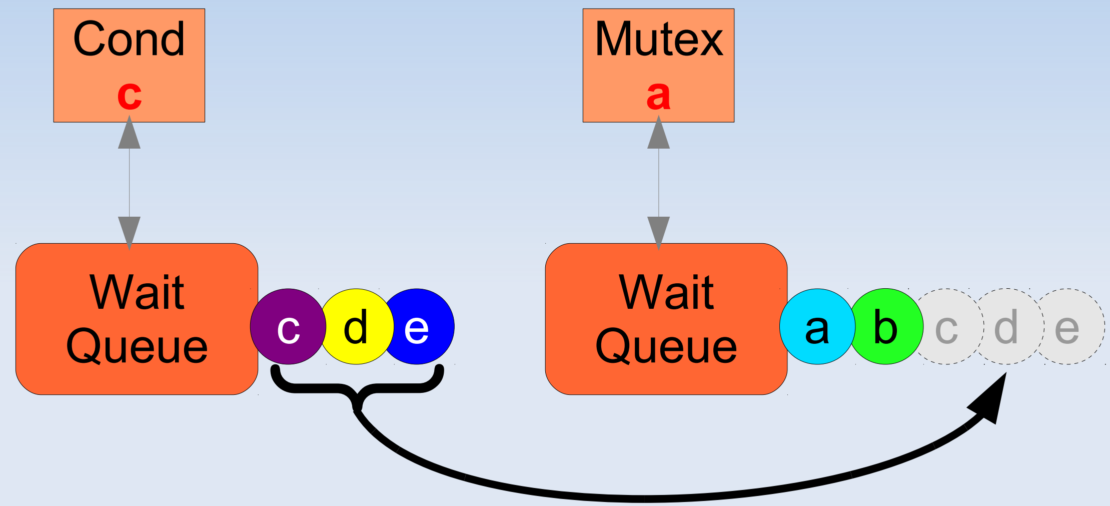
- `cond_signal()`



  - Atomically increment futex value
  - Call kernel to move first waiter
    from Condition Variable wait queue
    to Mutex wait queue
  - can be done in O(1) using doubly-linked lists

Condition Variables

- `cond_broadcast()`



  - Atomically increment futex value
  - Call kernel to move all waiters
    from Condition Variable wait queue
    to Mutex wait queue
  - can be done in O(1) using doubly-linked lists

Wait Queue Locking

- Wait queues are maintained by the kernel
  → need proper locking in the kernel
- Futex scope specific approaches:
  - single address space
    - possibly use an existing per-adspace lock
  - single partition
    - use an existing per-partition lock
  - across partitions
    - use a system wide lock or a global kernel lock
  - can use existing locks or introduce new ones

Wait Queue Locking

- Hashed address approach
  - futex address → hash() → select lock in an array
    - Single address space → virtual address
    - Multiple address spaces → physical address
  - The lock array needs to be pre-allocated
- Both approaches should be combined
  - scope approach ensures proper timing
  - hashing for scalability
- Also check partition privileges!

Robustness

- What happens if the user manipulates the thread ID of the first waiter in user space?
  - ID set to zero or an invalid value

    → no waiters found
    - But kernel can still remove waiters safely from the wait queue
  - ID of a thread waiting on another futex
    - Sanity checks apply → no waiter woken up
  - …
- Errors same as a thread never unlocking a mutex

  → Futex users have to trust each other

Future Work

- Sorted wait queues
  - for priority inheritance (PI)
    or other priority inversion protocols
  - e.g. sorted by priority, deadline, …
- Other scheduling algorithms
  - using dynamic priorities like EDF
  - for mixed criticality systems

Summary

- Implemented features:
  - Pthread mutexes in different flavours
    - Error Checking
    - Recursive
  - Pthread condition variables
  - Pthread rwlocks, barriers, pthread_once()
  - POSIX semaphores
  - waiting with relative and absolute timeouts
  - both ”private” and ”shared” futexes

Summary

- Mutexes and Condition Variables
  with FIFO ordering
- No in-kernel memory allocator required!
- Linear kernel memory usage
- Using linked lists, all operations are in O(1) time

- when adding ”wake arbitrary # of waiters”
  and not just migrating queues,
  we get the same flexibility as in Linux

  → unfortunately this needs O(n) time

Thank You!

Any Questions?