CSE 451: Operating Systems
Spring 2017

Module 11
Deadlock

John Zahorjan

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Definition

- A thread is deadlocked when it's waiting for an event that can never occur
  - I'm waiting for you to clear the intersection, so I can proceed
    - but you can't move until he moves, and he can't move until she moves, and she can't move until I move
  - Thread A is in critical section 1, waiting for access to critical section 2; thread B is in critical section 2, waiting for access to critical section 1
  - I'm trying to book a vacation package to Tahiti – air transportation, ground transportation, hotel, side-trips. It's all-or-nothing – one high-level transaction – with the four databases locked in that order. You're trying to do the same thing in the opposite order.

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Four conditions must exist for deadlock to be possible

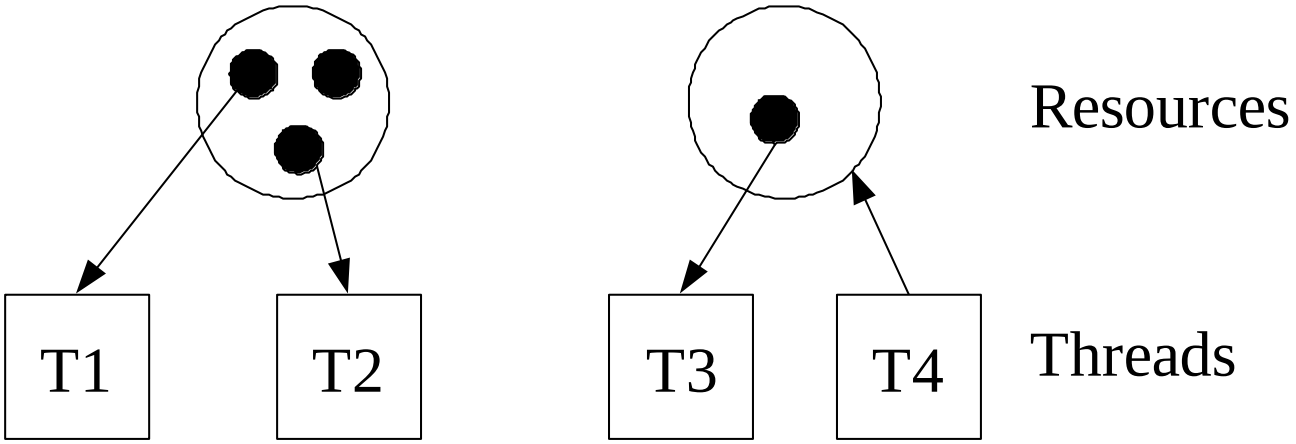
1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

We'll see that deadlocks can be addressed by attacking any of these four conditions.

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Resource Graphs

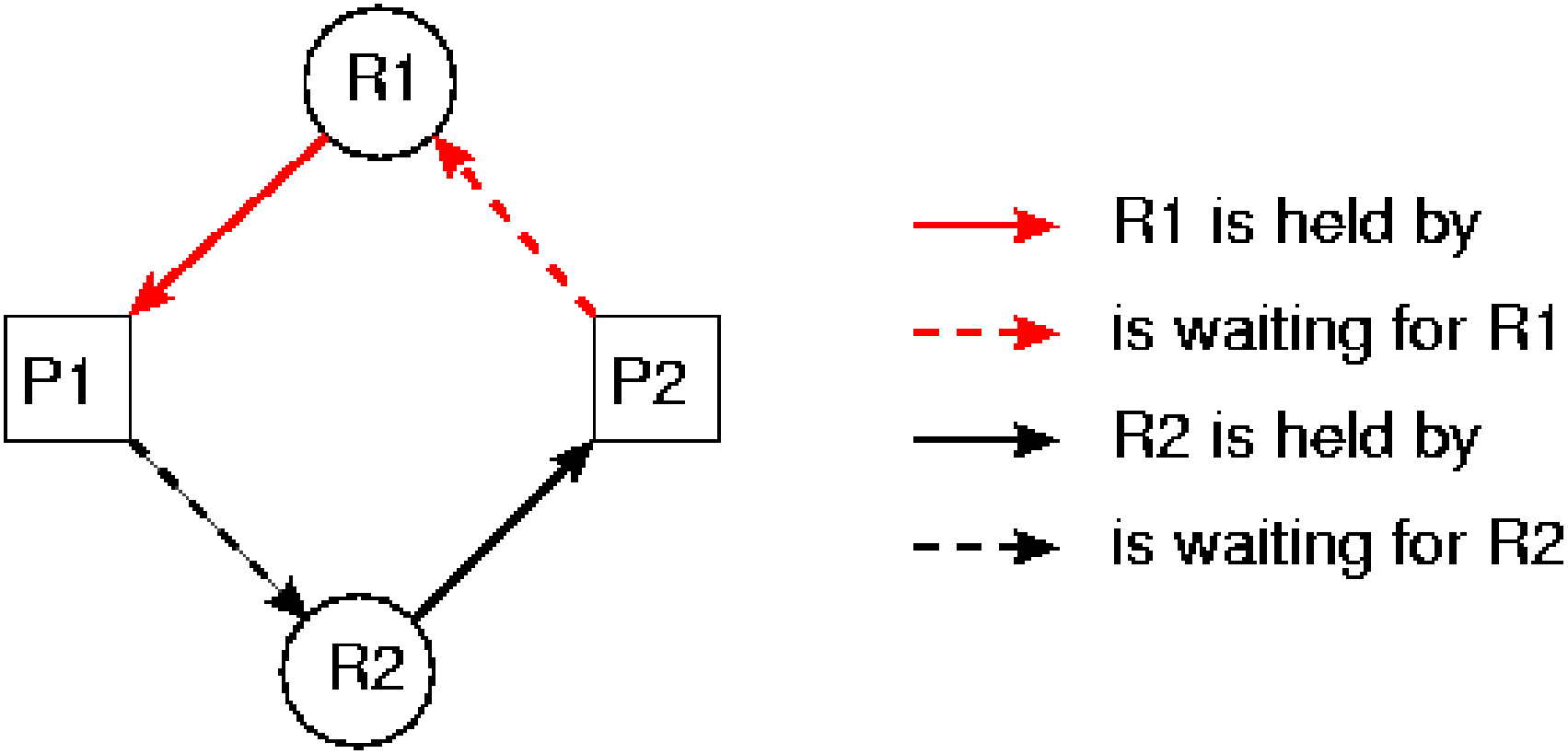
- Resource graphs are a way to visualize the (deadlock-related) state of the threads, and to reason about deadlock



- 1 or more identical units of a resource are available
- A thread may hold resources (arrows to threads)
- A thread may request resources (arrows from threads)

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Deadlock



- A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above)

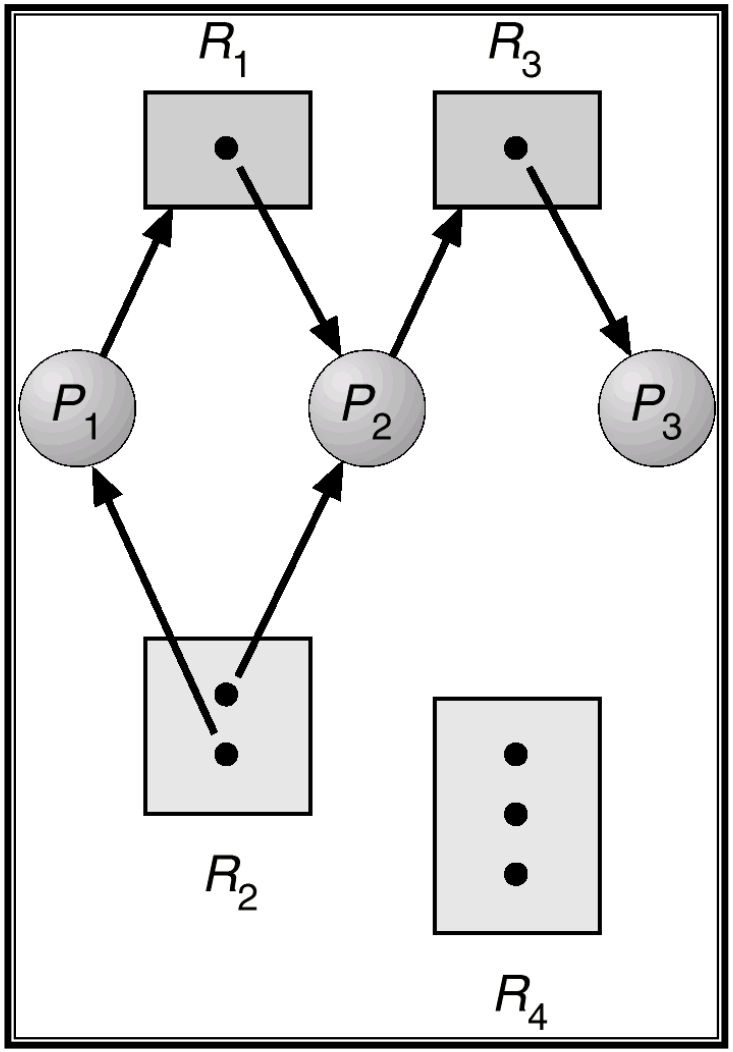
© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Graph reduction

- A graph can be *reduced* by a thread if all of that thread's requests can be granted
  - in this case, the thread eventually will terminate – all resources are freed – all arcs (allocations) to/from it in the graph are deleted
- Miscellaneous theorems (Holt, Havender):
  - There are no deadlocked threads iff the graph is completely reducible
  - The order of reductions is irrelevant

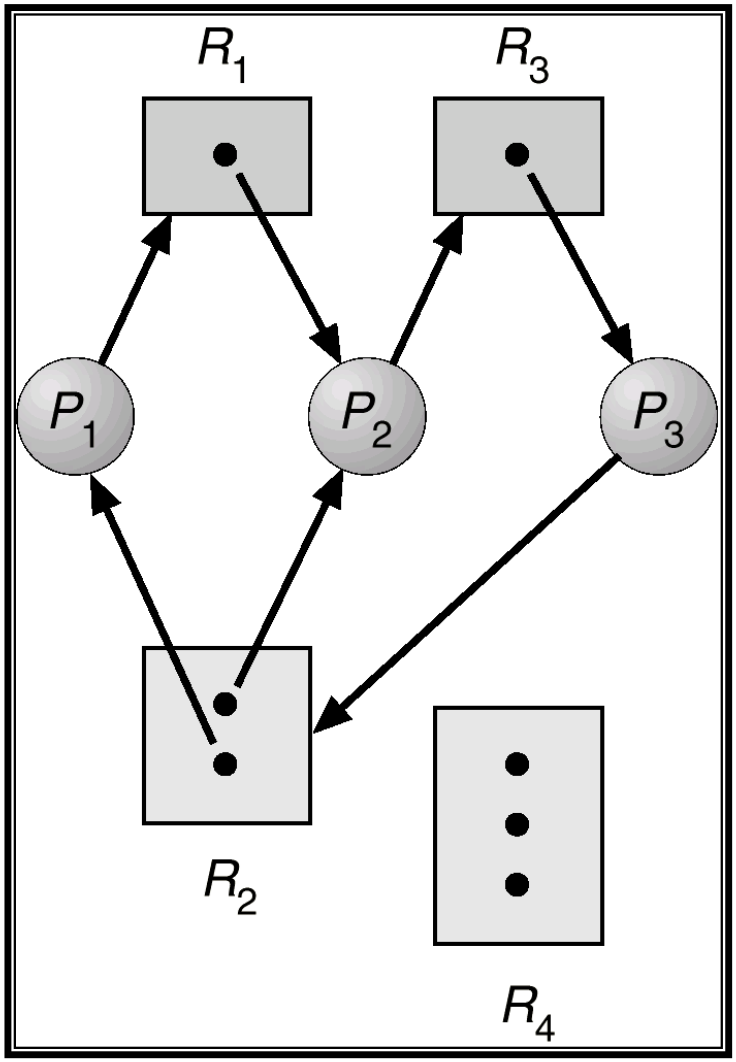
© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Resource allocation graph with no cycle

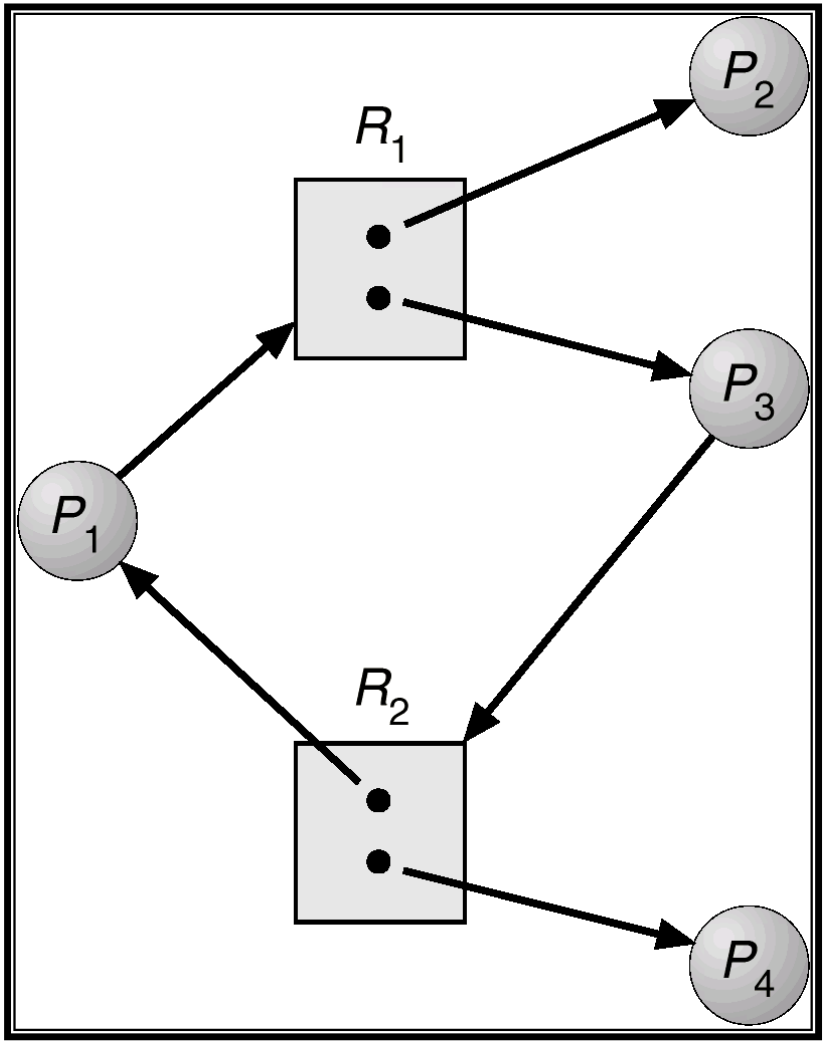


What would cause a deadlock?

Resource allocation graph with a deadlock



Resource allocation graph with a cycle but no deadlock



Handling Deadlock

- Eliminate one of the four required conditions
  - Mutual Exclusion
    - Clearly we're not going to eliminate this one!
  - Hold and Wait
  - No Preemption
  - Circular Wait

- Broadly classified as:
  - Prevention, or
  - Avoidance, or
  - Detection (and recovery)

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Prevention

Applications must conform to behaviors guaranteed not to deadlock

- Eliminating hold and wait
  - each thread obtains all resources at the beginning
  - blocks until all are available
    - drawback?
- Eliminating circular wait
  - resources are ordered (numbered)
  - each thread obtains resources in sequence order (which could require acquiring some before they are actually needed)
    - why does this work?
    - pros and cons?

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Avoidance

Less severe restrictions on program behavior

- Eliminating circular wait
  - each thread states its maximum claim for every resource type
  - system runs the Banker's Algorithm at each allocation request
    - Banker $\Rightarrow$ incredibly conservative
    - if I were to allocate you that resource, and then everyone were to request their maximum claim for every resource, could I find a way to allocate remaining resources so that everyone finished?
  - More on this in a moment…

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Detect and recover

- Every once in a while, check to see if there's a deadlock
  - how?
- If so, eliminate it
  - how?

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

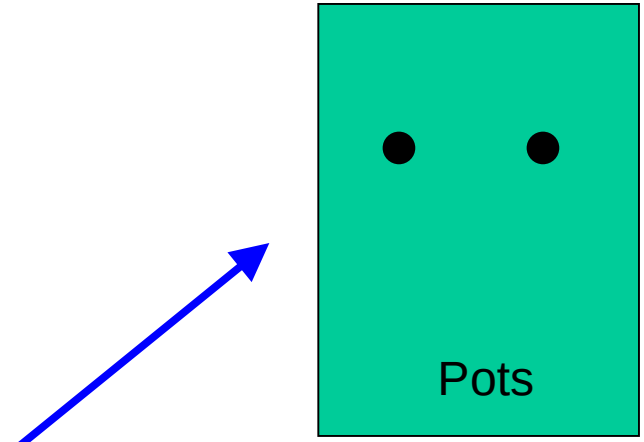
Avoidance: Banker's Algorithm example

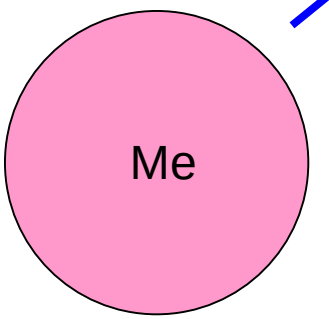
- Background
  - The set of controlled resources is known to the system
  - The number of units of each resource is known to the system
  - Each application must declare its maximum possible requirement of each resource type

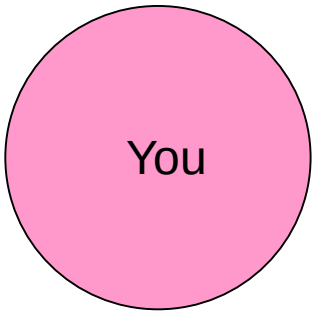
- Then, the system can do the following:
  - When a request is made
    - pretend you granted it
    - pretend all other legal requests were made
    - can the graph be reduced?
      - if so, allocate the requested resource
      - if not, block the thread until some thread releases resources, and then try pretending again

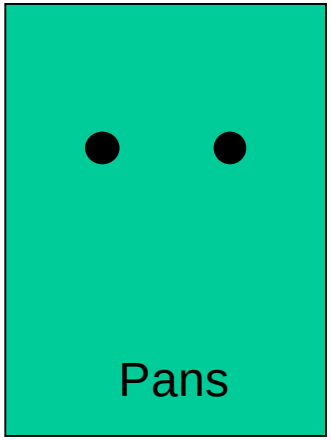
© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

1. I request a pot





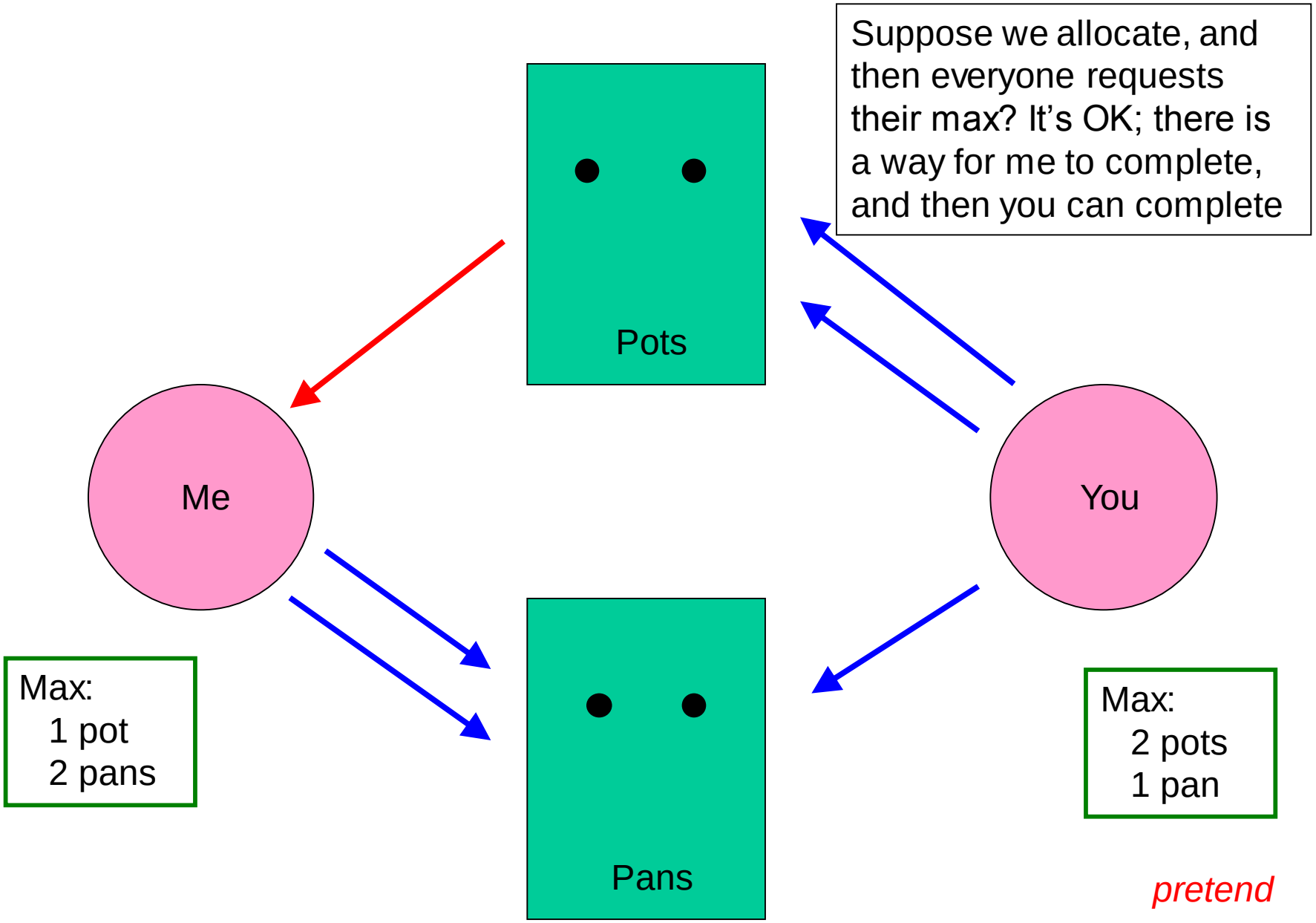




Max:
1 pot
2 pans

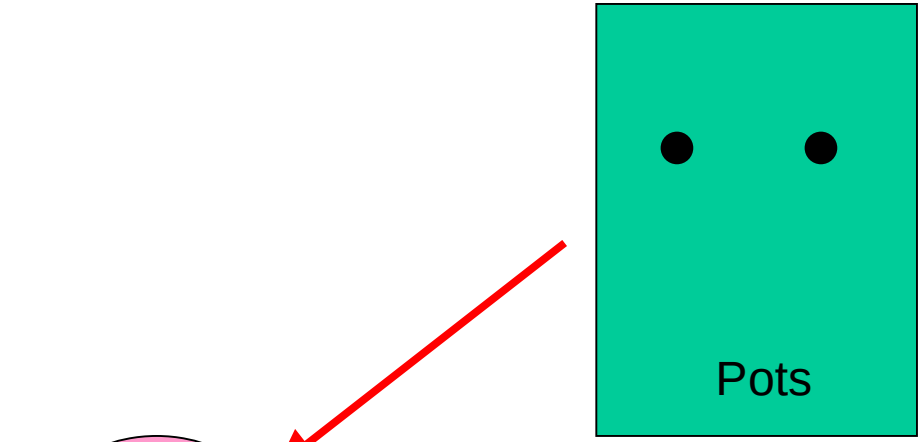
Max:
2 pots
1 pan

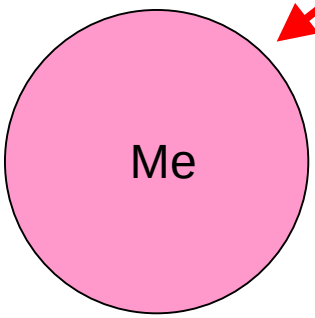
© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

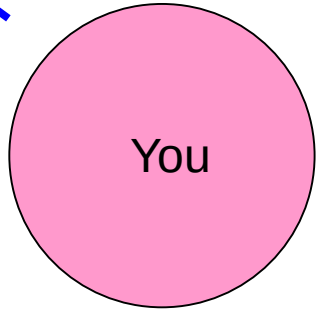


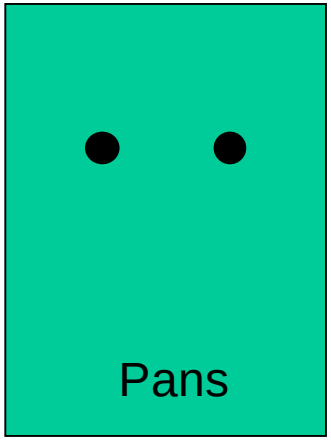
© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

2. You request a pot





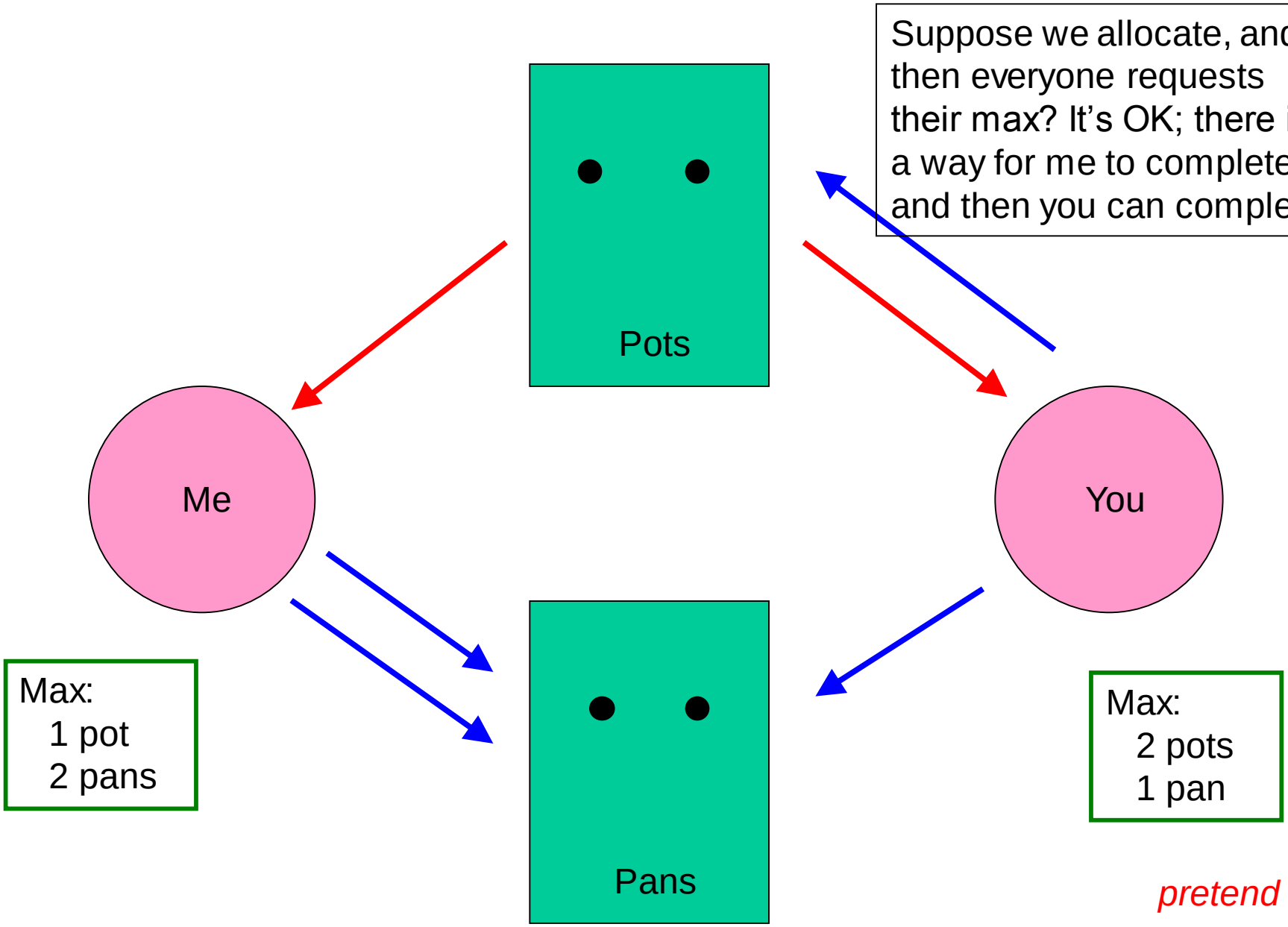




Max:
1 pot
2 pans

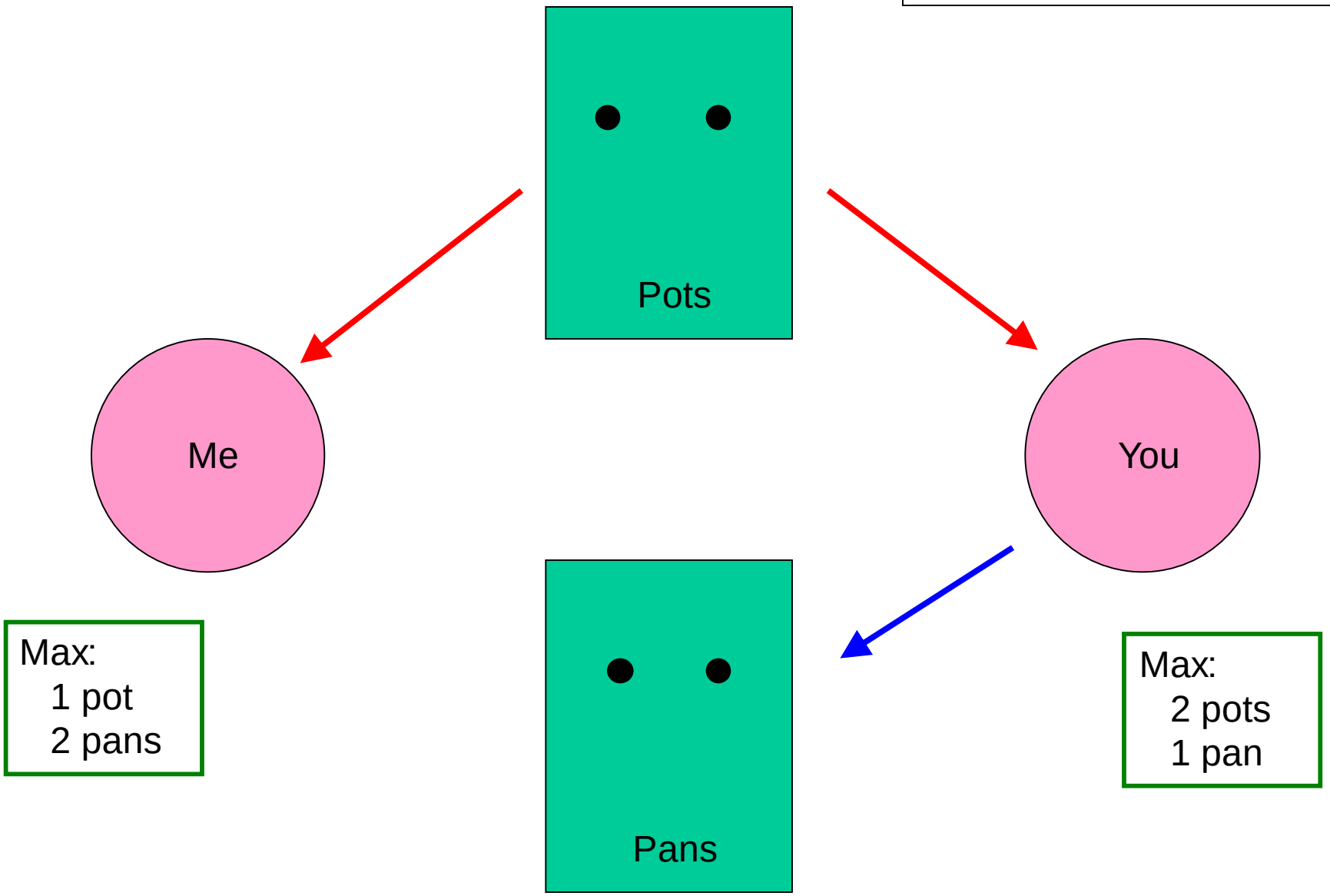
Max:
2 pots
1 pan

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

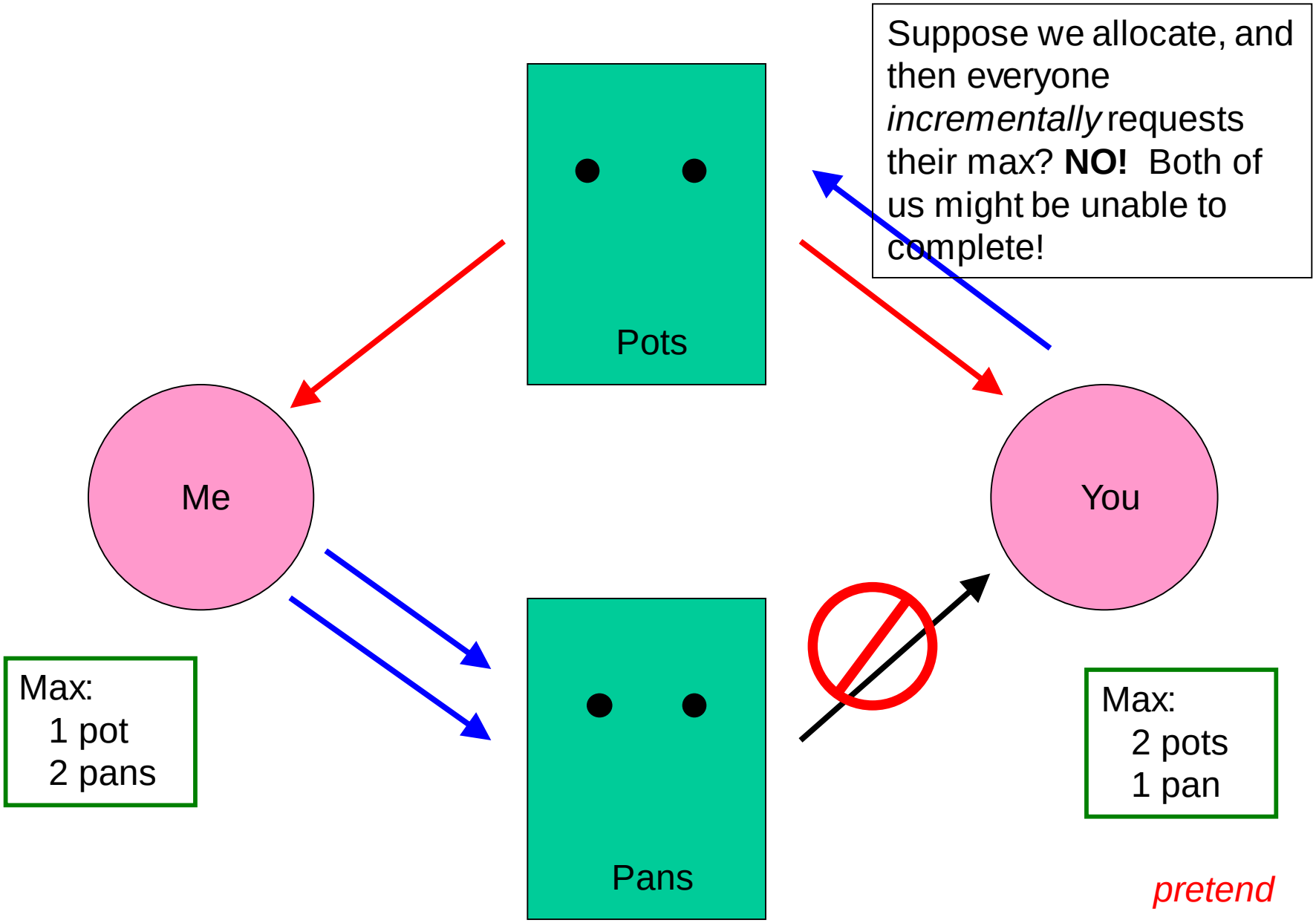


© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

3a. You request a pan

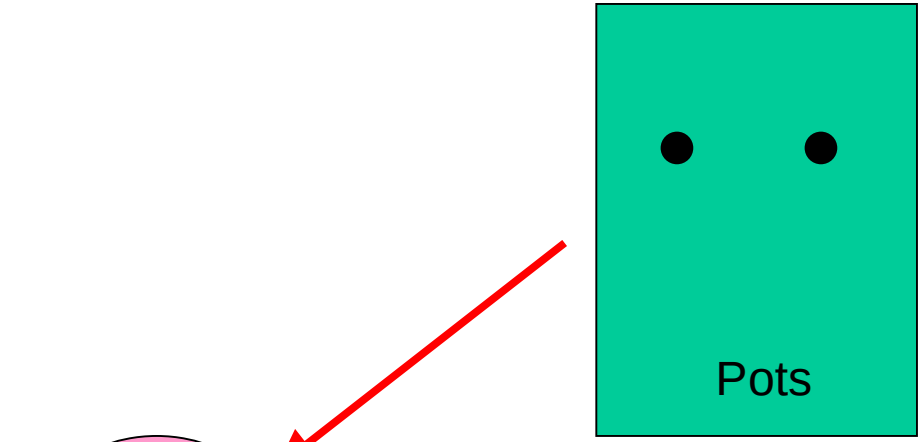


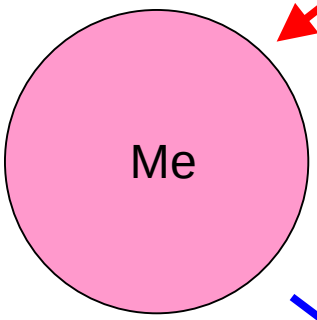
© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

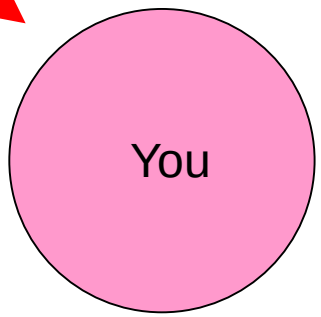


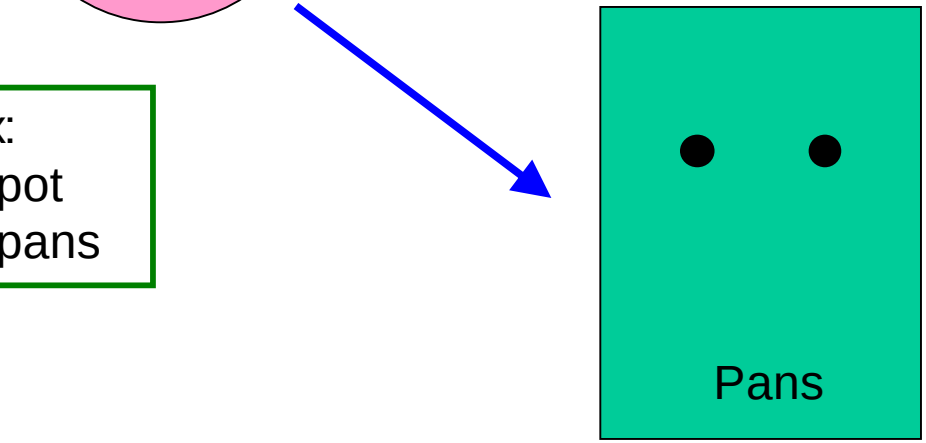
© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

3b. I request a pan





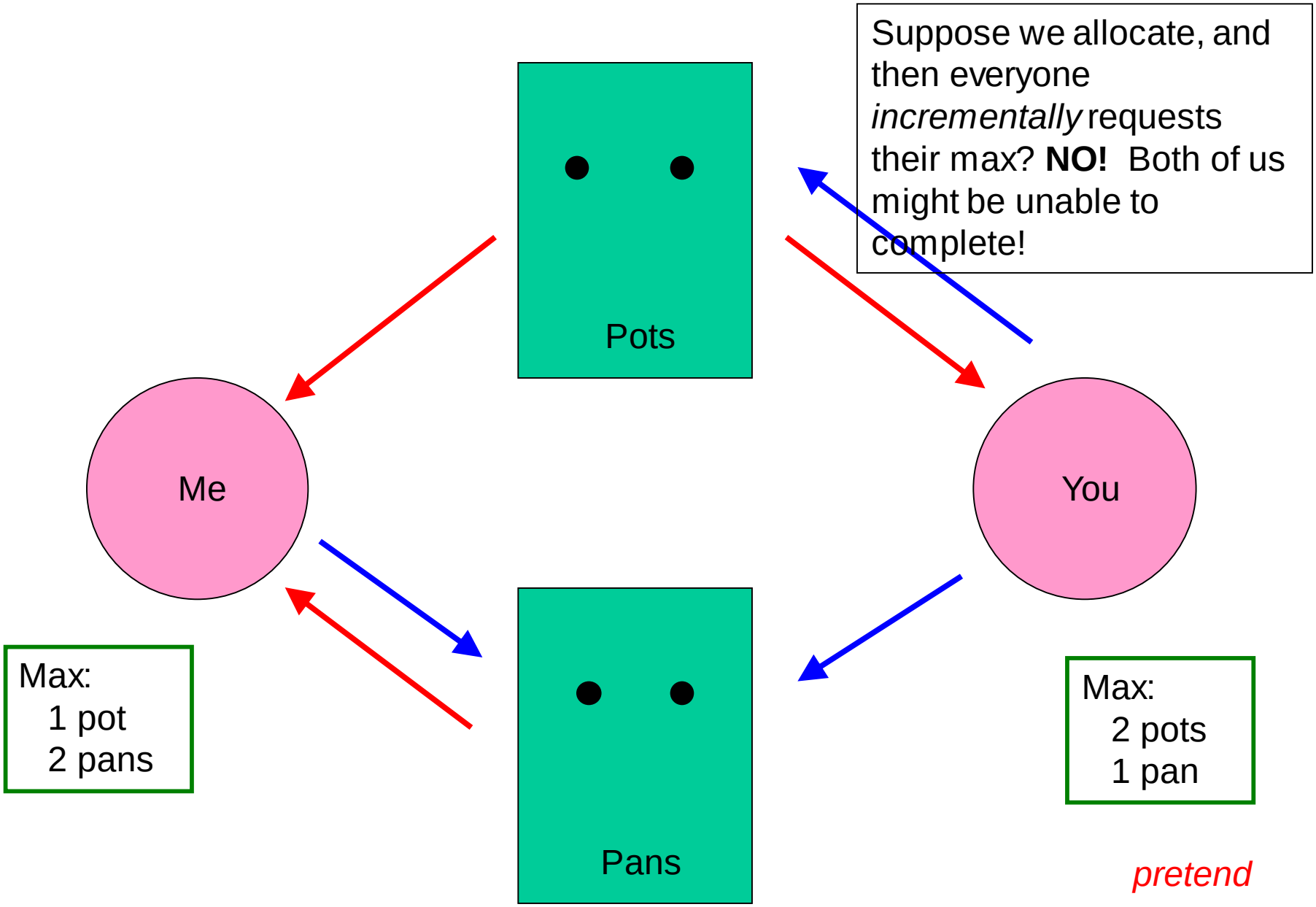




Max:
1 pot
2 pans

Max:
2 pots
1 pan

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski



© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Current practice

- Microsoft SQL Server
  - "The SQL Server Database Engine automatically detects deadlock cycles within SQL Server. The Database Engine chooses one of the sessions as a deadlock victim and the current transaction is terminated with an error to break the deadlock."
- Oracle
  - As Microsoft SQL Server, plus "Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order... For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table."

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

- Windows internals (Linux no different)
  - "Unless they did a huge change in Vista (and from what I've heard they haven't modified this area), the NT kernel architecture is a deadlock minefield. With the multi-threaded re-entrant kernel there is plenty of deadlock potential."
  - "Lock ordering is great in theory, and NT was originally designed with mutex levels, but they had to be abandoned. Inside the NT kernel there is a lot of interaction between memory management, the cache manager, and the file systems, and plenty of situations where memory management (maybe under the guise of its modified page writer) acquires its lock and then calls the cache manager. This happens while the file system calls the cache manager to fill the cache which in turn goes through the memory manager to fault in its page. And the list goes on."

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Non-blocking algorithms

- An algorithm is non-blocking if a slow thread cannot prevent another faster thread from making progress
  - Using locks is not non-blocking because a thread may acquire the lock and then run really really slowly
    - (Why?)

- Non-blocking algorithms are often built on an atomic hardware instruction, Compare And Swap (CAS)

```
bool CAS(ptr, old, new) {
    if ( *ptr == old ) { *ptr = new;  return true; }
    return false;
}
```

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Non-blocking atomic integer

- ```
  int atomic_int_add(atomic_int *p, int val) {
      int oldval;
      do {
          oldval = *p;
      } while ( ! CAS(p, oldval, oldval+val) );
  ```

- What happens if multiple threads execute this concurrently?
  - Does every thread make progress?
  - Does at least one thread make progress in bounded number of steps?

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski
  ```

Why non-blocking

- What if a thread is pre-empted while holding a lock?
- If there are no locks, can there be deadlock?
- Suppose a low priority thread holds a lock needed by a high priority thread
  - (Alternative solution: priority inheritance)

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Why not non-blocking? (Non-blocking FIFO implementation)

```
structure pointer_t      {ptr: pointer to node_t, count: unsigned integer}
structure node_t         {value: data type, next: pointer_t}
structure queue_t        {Head: pointer_t, Tail: pointer_t}

initialize(Q: pointer to queue_t)
        node = new_node()                  # Allocate a free node
        node->next.ptr = NULL              # Make it the only node in the linked list
        Q->Head = Q->Tail = node           # Both Head and Tail point to it
```

Pointers are stored with a generation number in one 8-byte quantity (32-bit pointer + 32-bit generation number)

From Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms *by Michael & Scott.*

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Non-blocking FIFO: enqueue()

```
enqueue(Q: pointer to queue_t, value: data type)
E1:     node = new_node()                                  # Allocate a new node from the free list
E2:     node->value = value                                # Copy enqueued value into node
E3:     node->next.ptr = NULL                              # Set next pointer of node to NULL
E4:     loop                                               # Keep trying until Enqueue is done
E5:         tail = Q->Tail                                 # Read Tail.ptr and Tail.count together
E6:         next = tail.ptr->next                          # Read next ptr and count fields together
E7:         if tail == Q->Tail                             # Are tail and next consistent?
E8:             if next.ptr == NULL                        # Was Tail pointing to the last node?
E9:                 if CAS(&tail.ptr->next, next, <node, next.count+1>)  # Try to link node at the end of the linked list
E10:                    break                              # Enqueue is done. Exit loop
E11:                endif
E12:            else                                       # Tail was not pointing to the last node
E13:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)  # Try to swing Tail to the next node
E14:            endif
E15:        endif
E16:    endloop
E17:    CAS(&Q->Tail, tail, <node, tail.count+1>)          # Enqueue is done. Try to swing Tail to the inserted node
```

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Non-blocking FIFO: dequeue

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:     loop                                                    # Keep trying until Dequeue is done
D2:         head = Q->Head                                      # Read Head
D3:         tail = Q->Tail                                      # Read Tail
D4:         next = head->next                                   # Read Head.ptr->next
D5:         if head == Q->Head                                  # Are head, tail, and next consistent?
D6:             if head.ptr == tail.ptr                         # Is queue empty or Tail falling behind?
D7:                 if next.ptr == NULL                         # Is queue empty?
D8:                     return FALSE                            # Queue is empty, couldn't dequeue
D9:                 endif
D10:                CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)   # Tail is falling behind. Try to advance it
D11:            else                                            # No need to deal with Tail
                    # Read value before CAS, otherwise another dequeue might free the next node
D12:                *pvalue = next.ptr->value
D13:                if CAS(&Q->Head, head, <next.ptr, head.count+1>)   # Try to swing Head to the next node
D14:                    break                                   # Dequeue is done. Exit loop
D15:                endif
D16:            endif
D17:        endif
D18:    endloop
D19:    free(head.ptr)                                          # It is safe now to free the old dummy node
D20:    return TRUE                                             # Queue was not empty, dequeue succeeded
```

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Performance Results

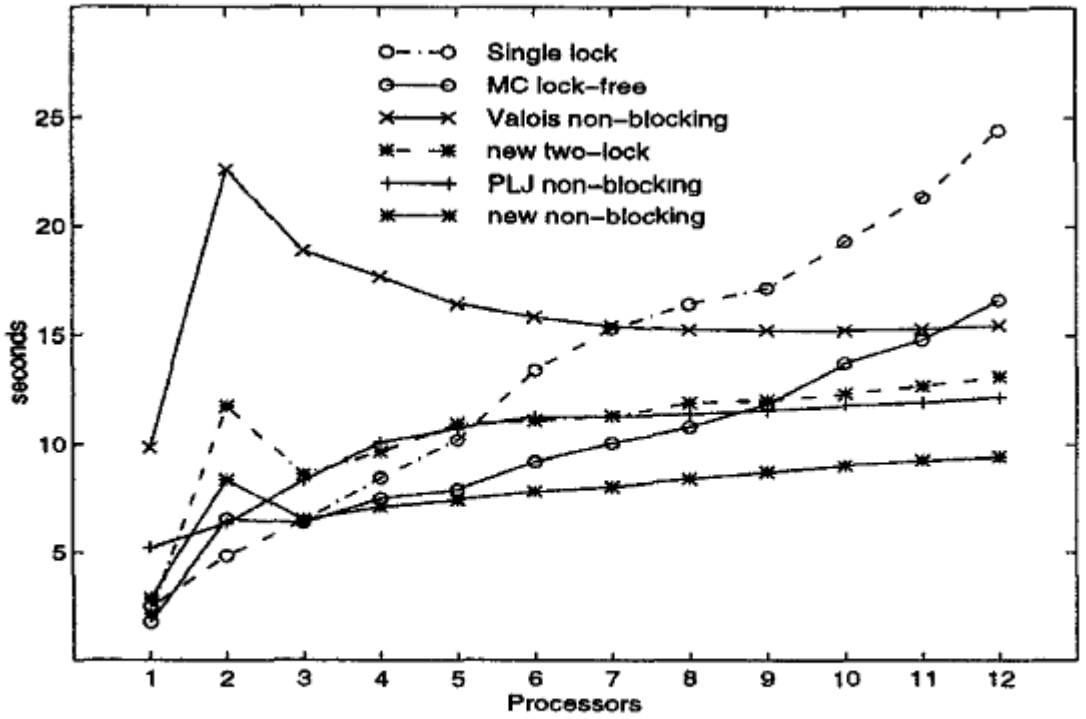


Figure 3: Net execution time for one million enqueue/dequeue pairs on a dedicated multiprocessor.

12 processor Silicon Graphics Challenge

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski

Summary

- Deadlock is bad!
- We can deal with it either statically (prevention) or dynamically (avoidance and/or detection)
- In practice, you'll encounter lock ordering, periodic deadlock detection/correction, and minefields

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski