Operating Systems

Deadlock

Lecture 7

Michael O'Boyle

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

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently.

Transaction 1 transfers \$25 from account A to account B, and

Transaction 2 transfers \$50 from account B to account A

Four conditions must exist for deadlock to be possible

1. Mutual Exclusion

2. Hold and Wait

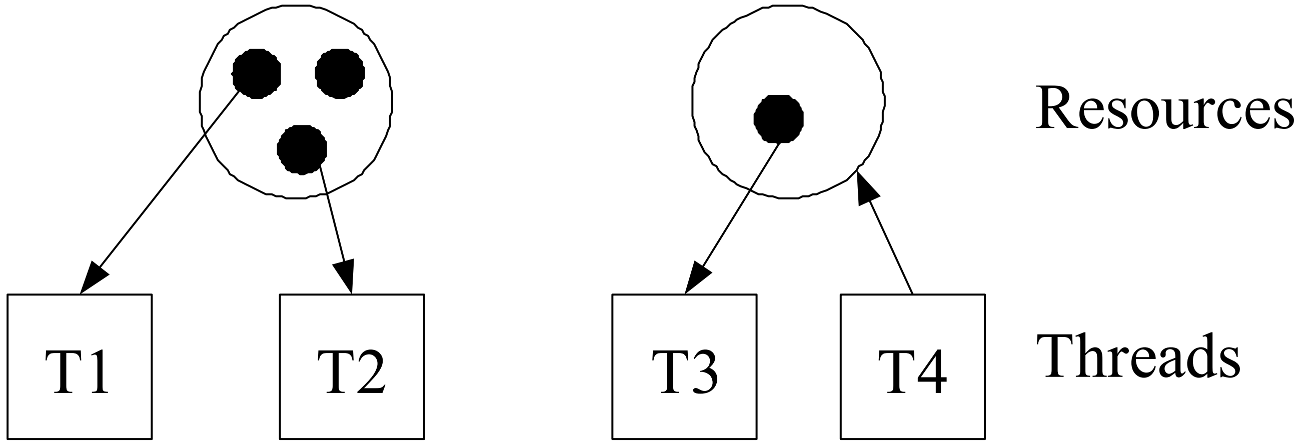
3. No Preemption

4. Circular Wait

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

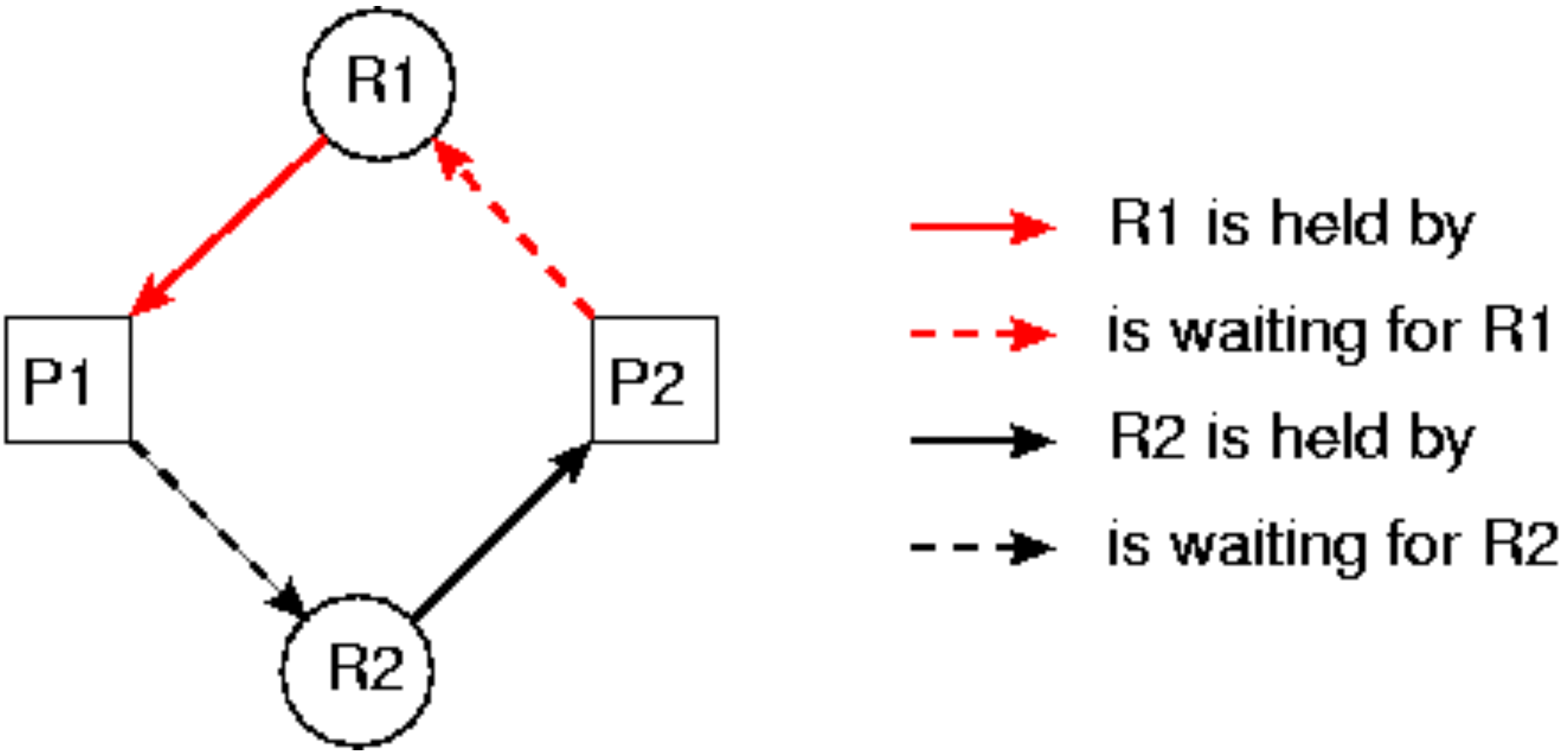
Resource Graphs

- 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)

Deadlock

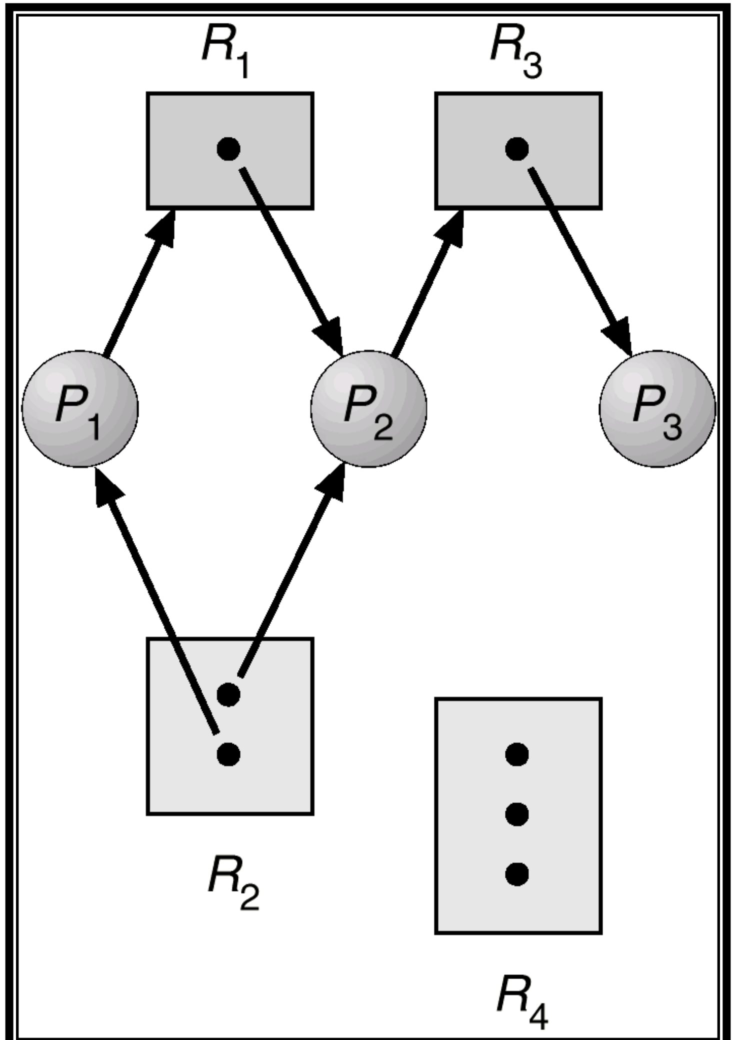


- A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above)

Graph reduction

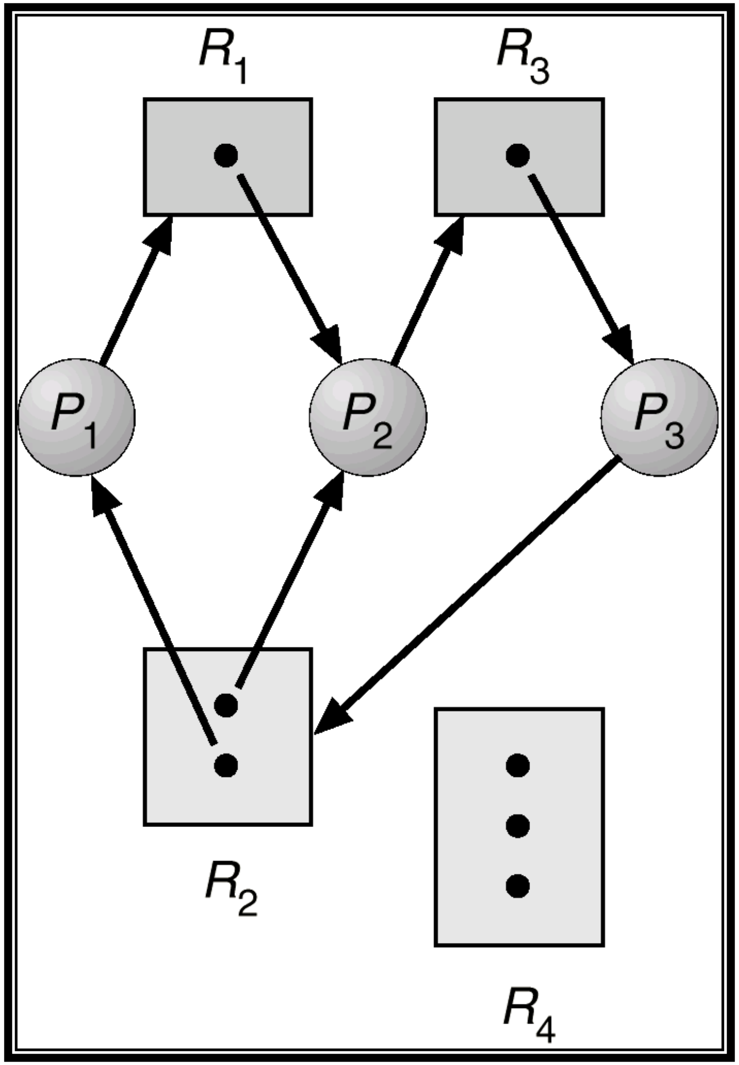
- A graph can be *reduced* by a thread if all of that thread's requests can be granted
  - in this case, the thread eventually will terminate – all resources are freed – all arcs (allocations) to/from it in the graph are deleted
- Miscellaneous theorems (Holt, Havender):
  - There are no deadlocked threads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

Resource allocation graph with no cycle

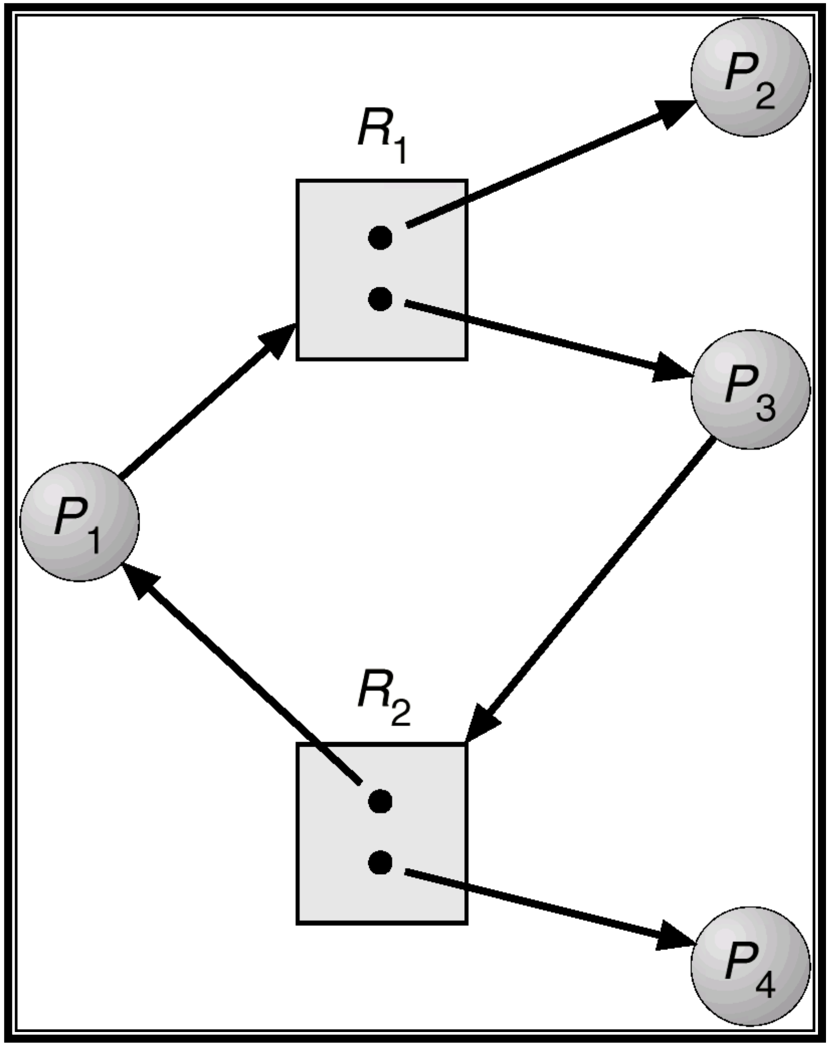


What would cause a deadlock?

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
  - Hold and Wait
  - No Preemption
  - Circular Wait

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

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No (resource) Preemption** –
  - If a process holding some resources requests another unavailable resource all resources currently held are released
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait**

  – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Avoidance

Less severe restrictions on program behavior

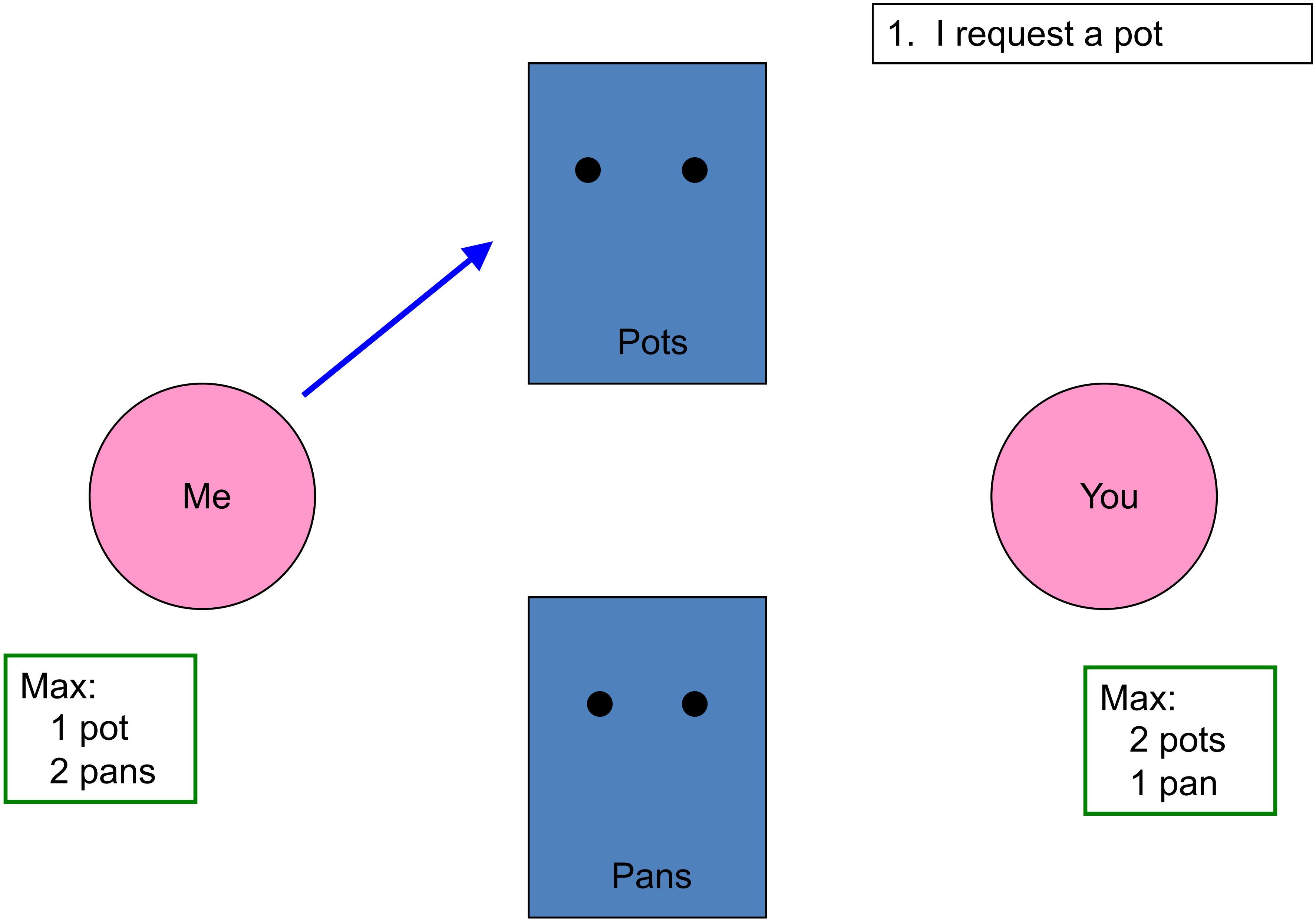
- Eliminating circular wait
  - each thread states its maximum claim for every resource type
  - system runs the Banker's Algorithm at each allocation request
    - Banker $\Rightarrow$ highly conservative
- More on this shortly

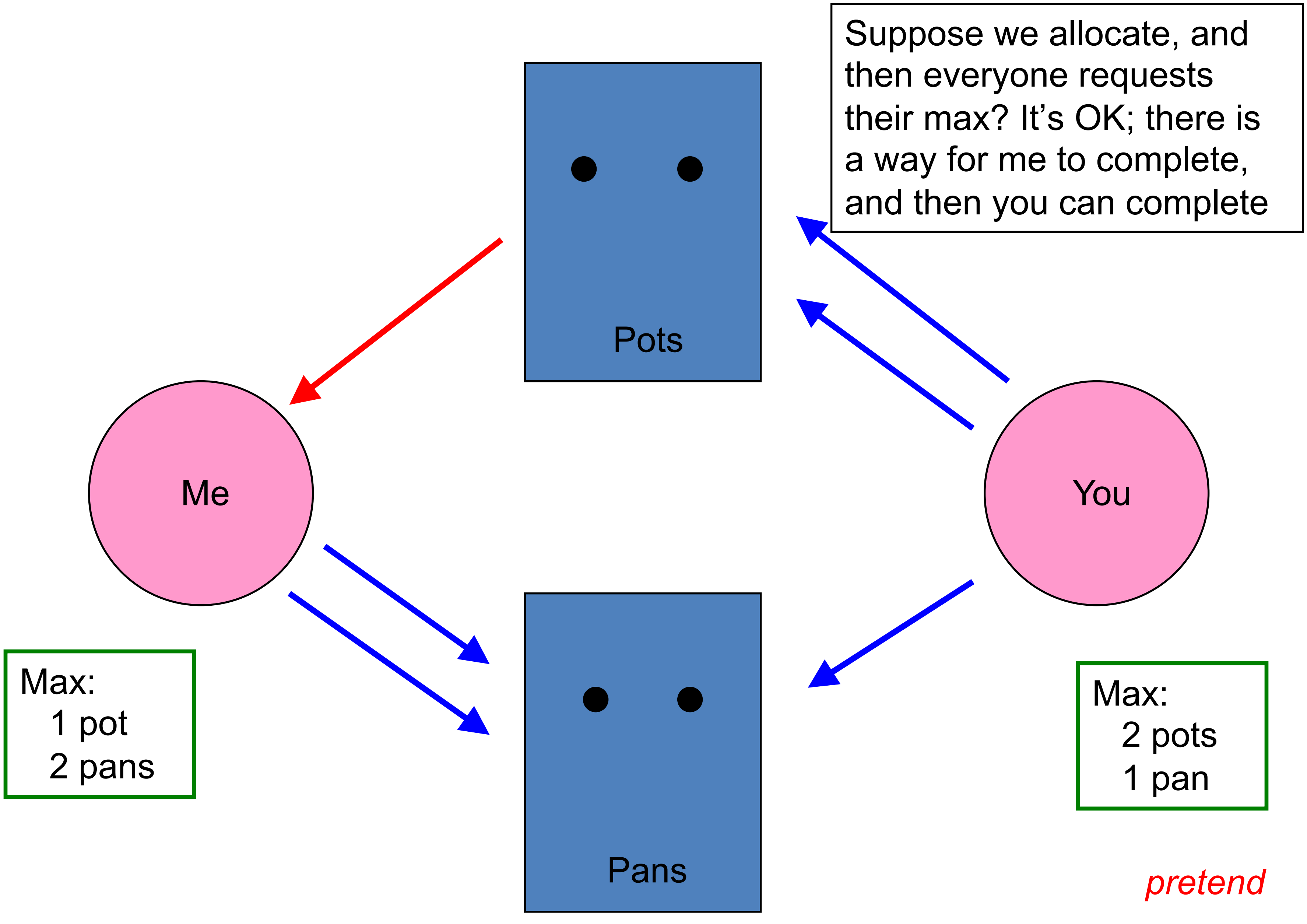
Detect and recover

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

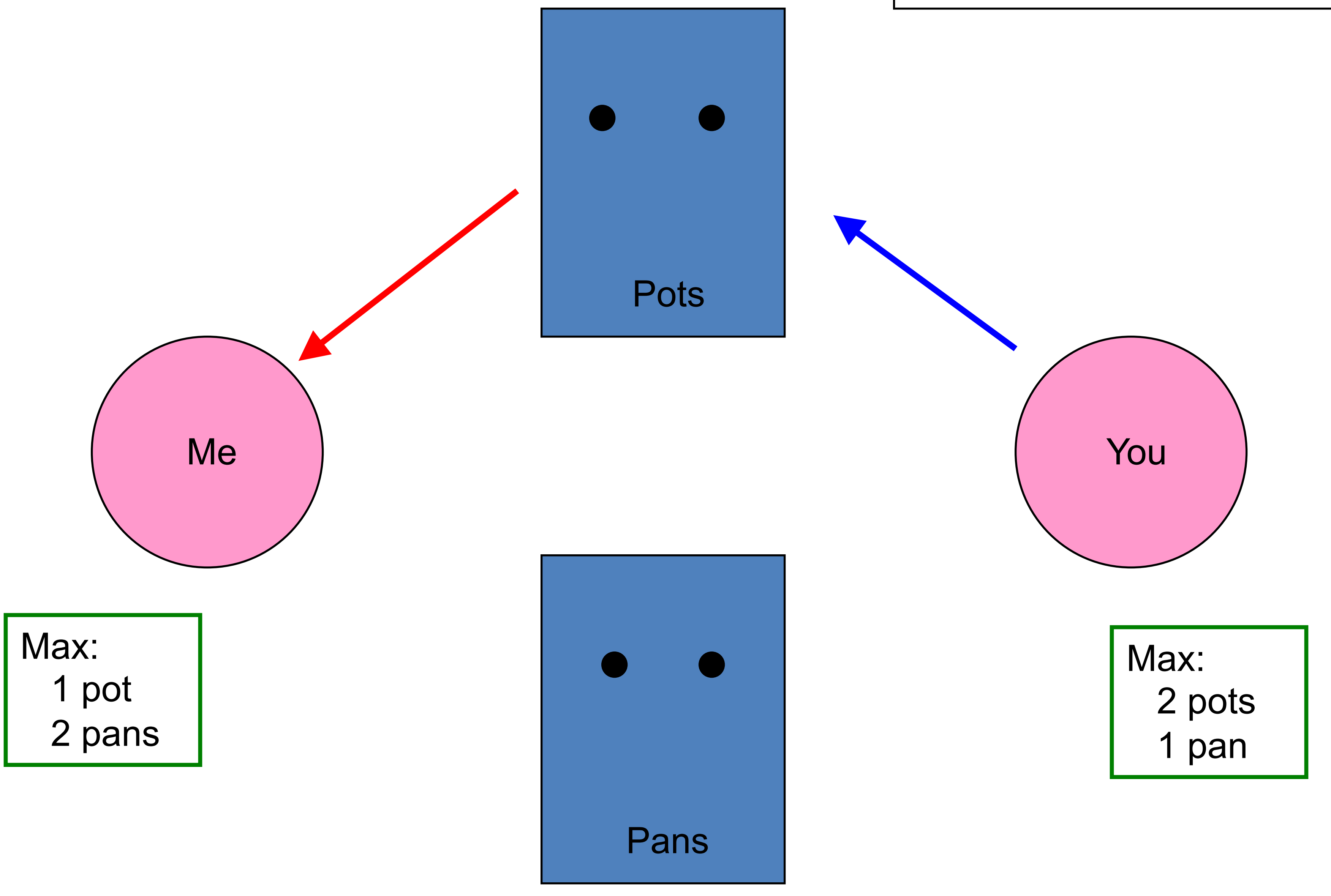
Avoidance: Banker's Algorithm example

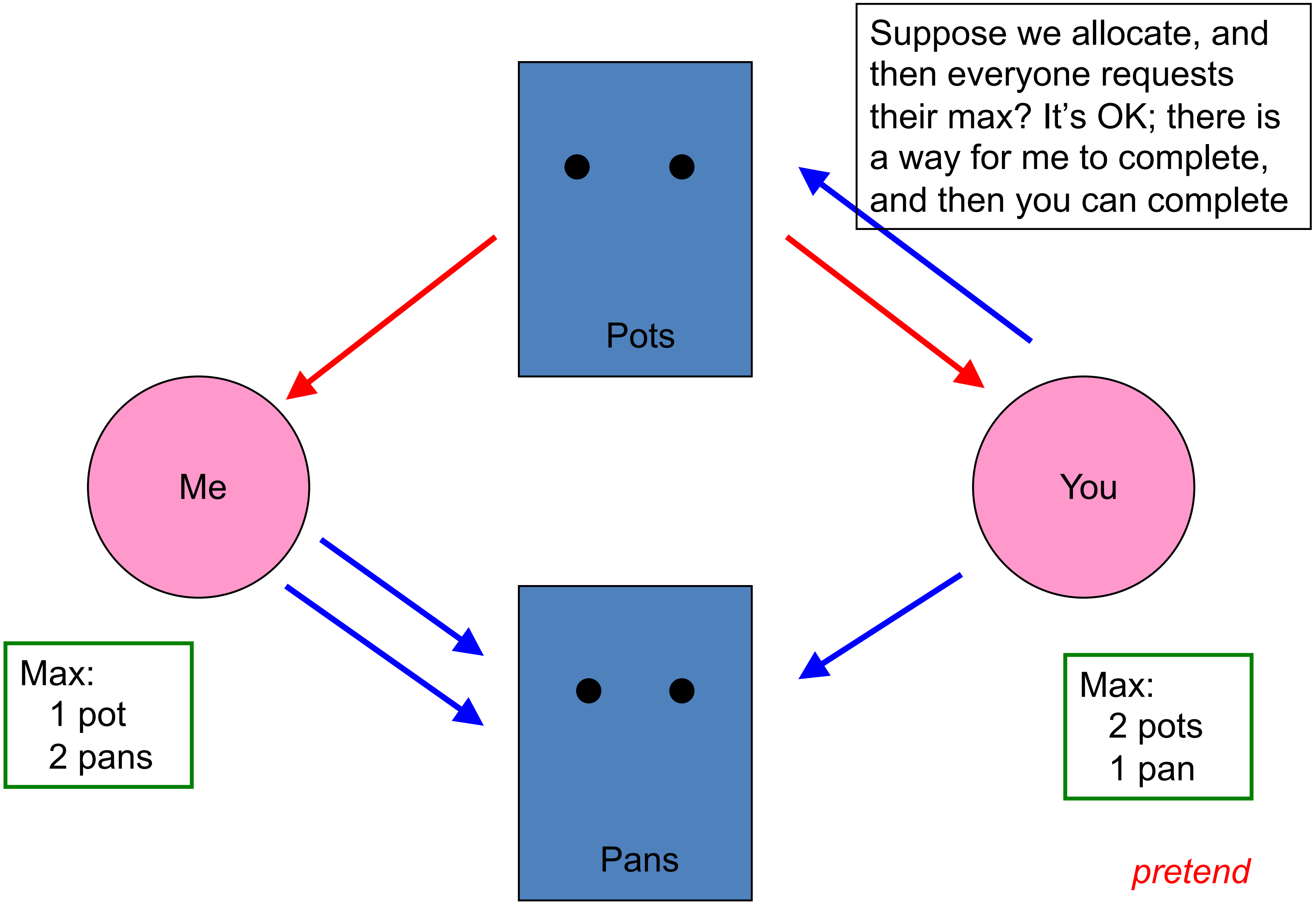
- Background
  - The set of controlled resources is known to the system
  - The number of units of each resource is known to the system
  - Each application must declare its maximum possible requirement of each resource type
- Then, the system can do the following:
  - When a request is made
    - pretend 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

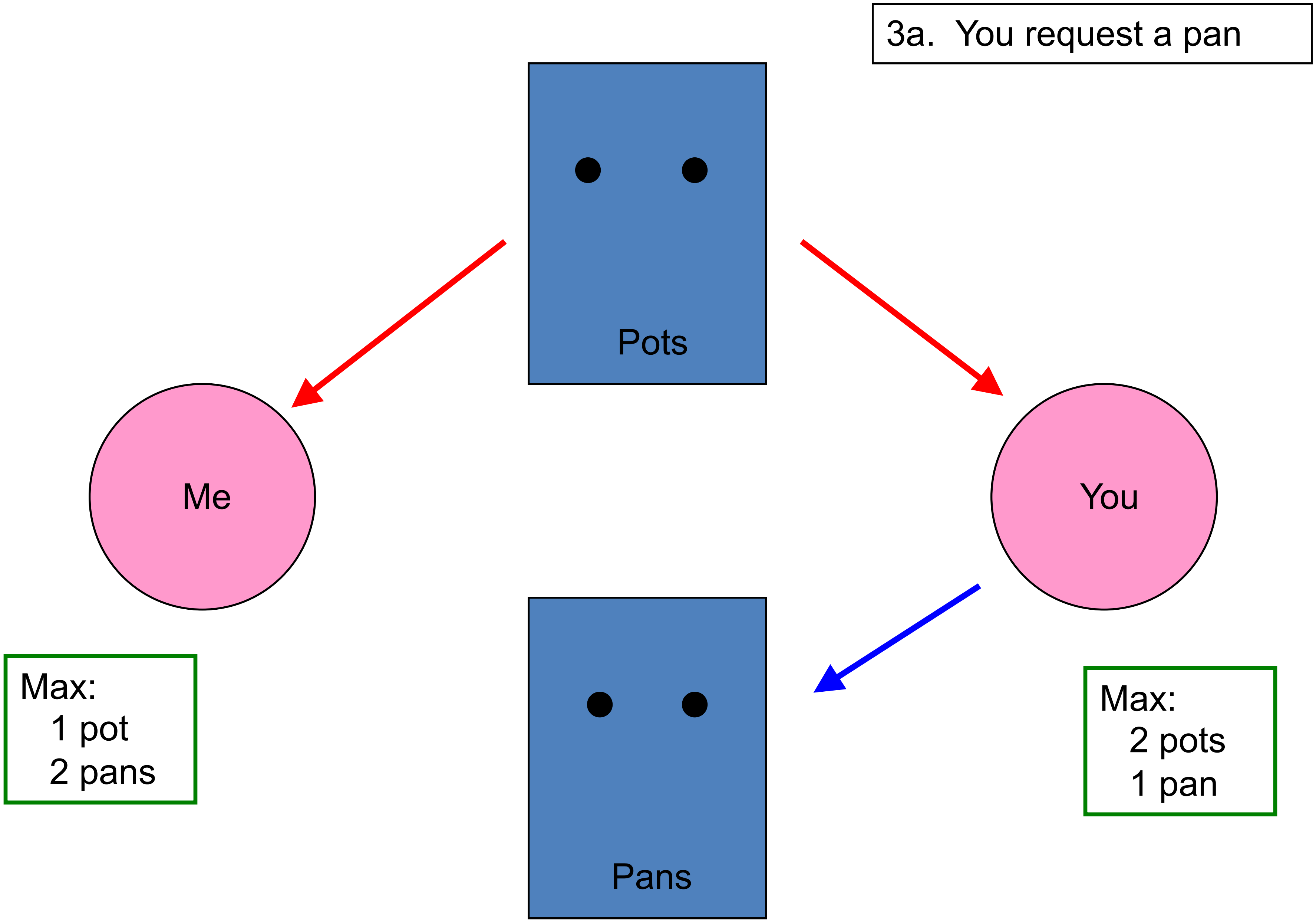

1. I request a pot
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan


Suppose we allocate, and then everyone requests their max? It's OK; there is a way for me to complete, and then you can complete
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan
pretend

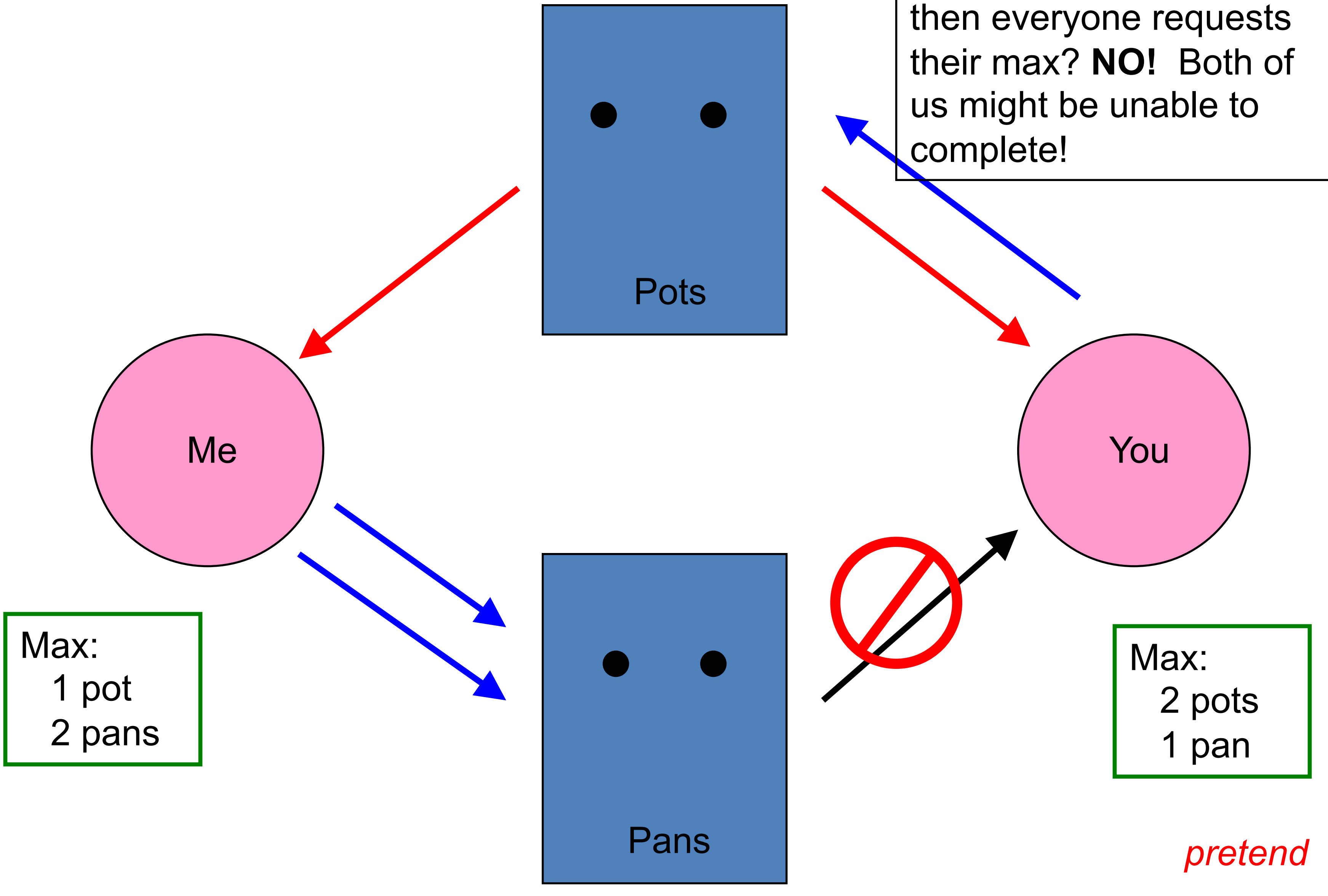
2. You request a pot



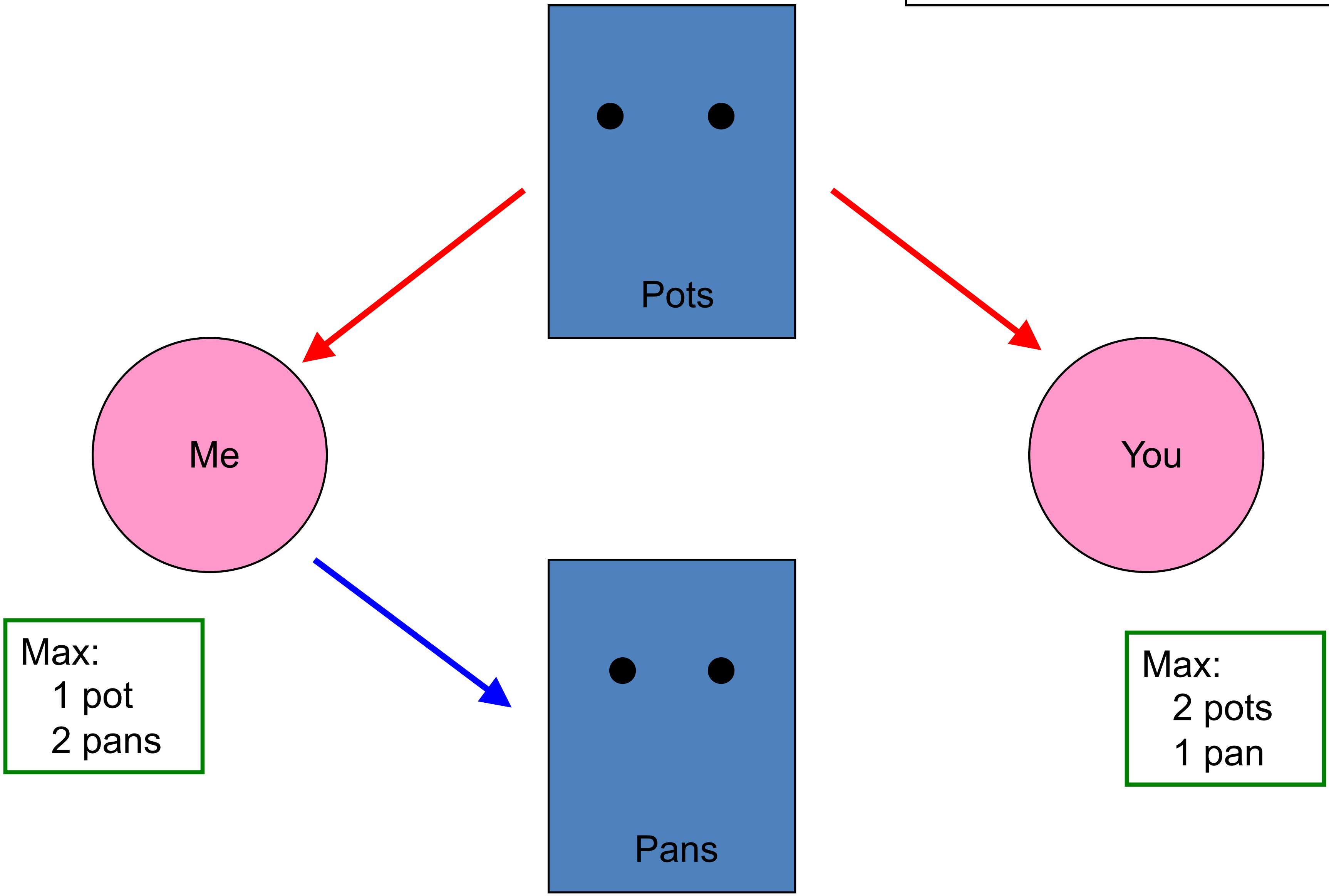

Suppose we allocate, and then everyone requests their max? It's OK; there is a way for me to complete, and then you can complete
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan
pretend

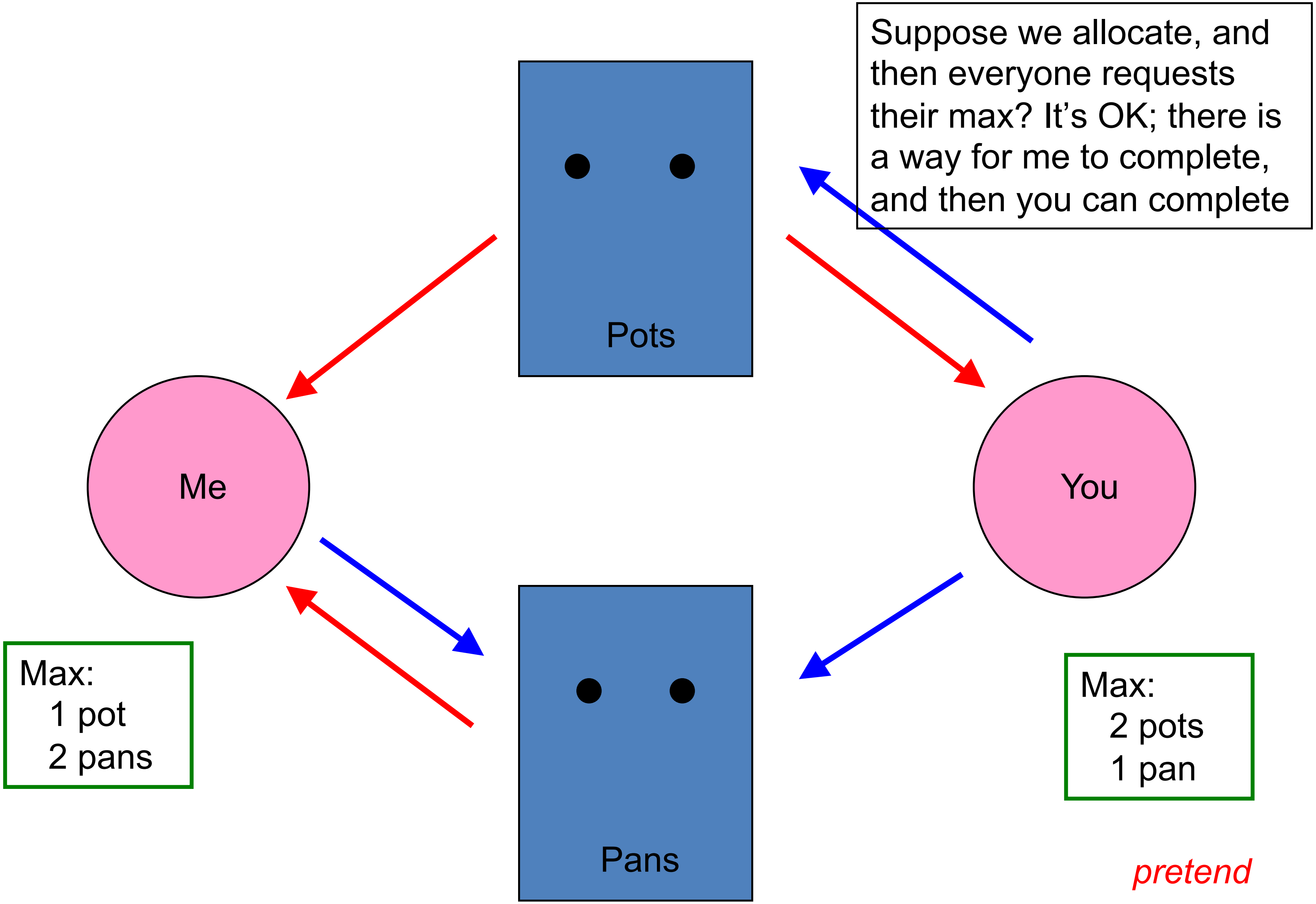

3a. You request a pan
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan

Suppose we allocate, and then everyone requests their max? **NO!** Both of us might be unable to complete!



3b. I request a pan

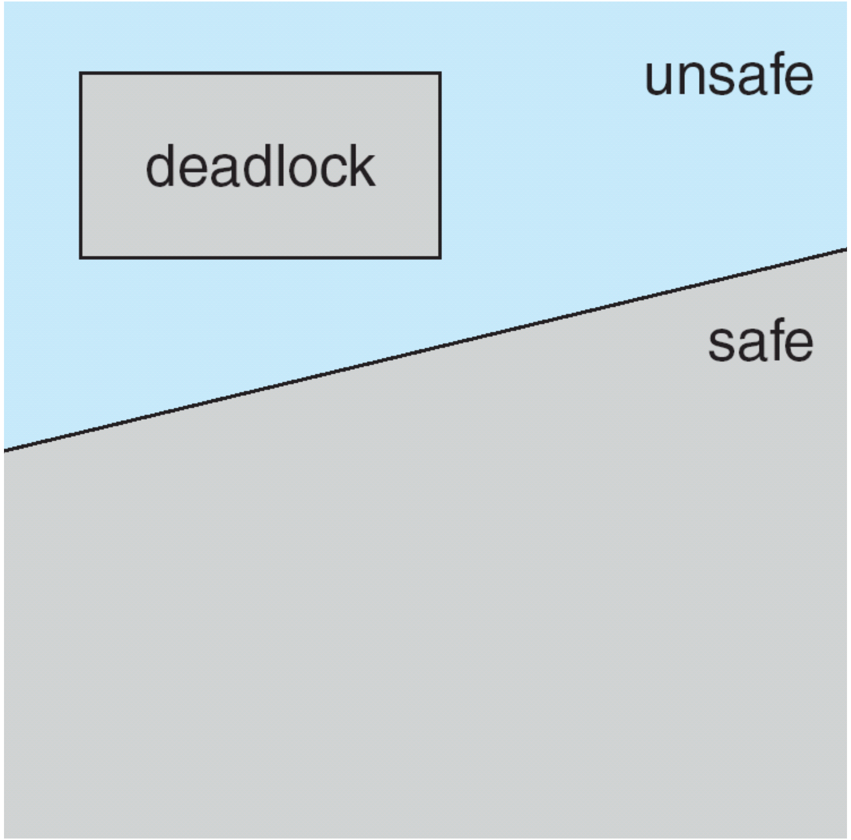



Suppose we allocate, and then everyone requests their max? It's OK; there is a way for me to complete, and then you can complete
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan
pretend

Safe State

- When requesting an available resource decide if allocation leaves the system in a safe state
- In **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems
  - such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe, Unsafe, Deadlock State



Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If Available [j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) $\boldsymbol{Need_i \le Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$

   Finish*[*i*] = *true

   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i\,[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \leq Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), *B* (5instances), and *C* (7 instances)
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

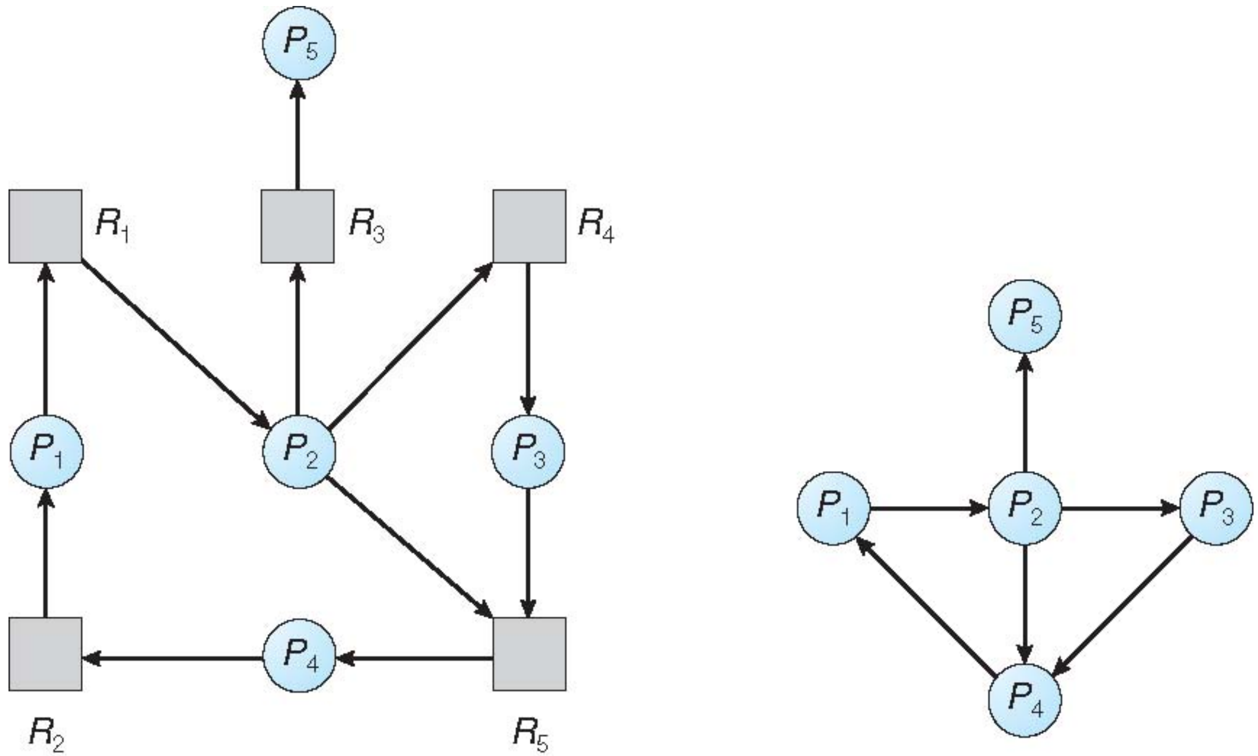
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.
  - If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph
  - requires an order of n^2 operations,
  - where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a) (b)

Resource-Allocation Graph Corresponding wait-for graph

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily,
  - there may be many cycles in the resource graph
  - we would not be able to tell which deadlocked processes "caused" the deadlock.

Recovery from Deadlock:

- Process Termination
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated
  - In which order should we choose to abort?
- Resource Preemption
  - **Selecting a victim** – minimize cost
  - **Rollback** – return to some safe state, restart process for that state
  - **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

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