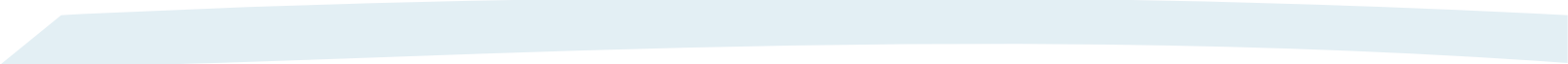


Discussion:

Monitors, Bankers and Bakers



CSE 120

Fall 2007

Wenjing Rao

Hoare Monitors with semaphores

Assume a Hoare-style monitor m with a single condition c (easily generalized for multiple conditions).

Generate:

```
struct m {  
    mutex sem lock=1;  
    counting sem urgent=0, csem=0;  
    int ccount=0, ucount=0  
};
```

Wrap each *entry* method:

```
wait(m.lock);  
code for entry method  
if (m.ucount > 0) signal(m.urgent);  
else signal(m.lock);
```

Hoare Monitors with semaphores II

Replace `c.wait()` with:

```
m.ccount++;  
if (m.ucount > 0) signal(m.urgent);  
else signal(m.lock);  
wait(m.csem);  
m.ccount--;
```

Replace `c.signal()` with:

```
m.ucount++;  
if (m.ccount > 0) {  
    signal(m.csem);  
    wait(m.urgent);  
}  
m.ucount--;
```

Final signals

Most entry methods signal right before leaving the monitor.

- So, why not have signals done as the last statement of an entry method?
- This gets rid of the extra context switch - the monitor lock is passed to the waiting thread and the signalling thread continues to execute.

Does this limit the power of monitors?

Mesa monitor semantics

- `c.notify()` is like `c.signal()` except that the notifying thread does not leave the monitor.
- `c.notifyAll()` unblocks all threads waiting on `c`.

... so, no need for urgent queue

Mesa monitors with semaphores

Assume a Mesa-style monitor m with a single condition c (easily generalized for multiple conditions).

Generate:

```
struct m {  
    mutex sem lock=1;  
    counting sem csem=0;  
    int ccount=0  
};
```

Wrap each *entry* method:

```
wait(m.lock);  
code for entry method  
signal(m.lock);
```

Mesa monitors with semaphores II

Replace `c.wait()` with:

```
m.ccount++;  
signal(m.lock);  
wait(m.csem);  
wait (m.lock);
```

Replace `c.notify()` with:

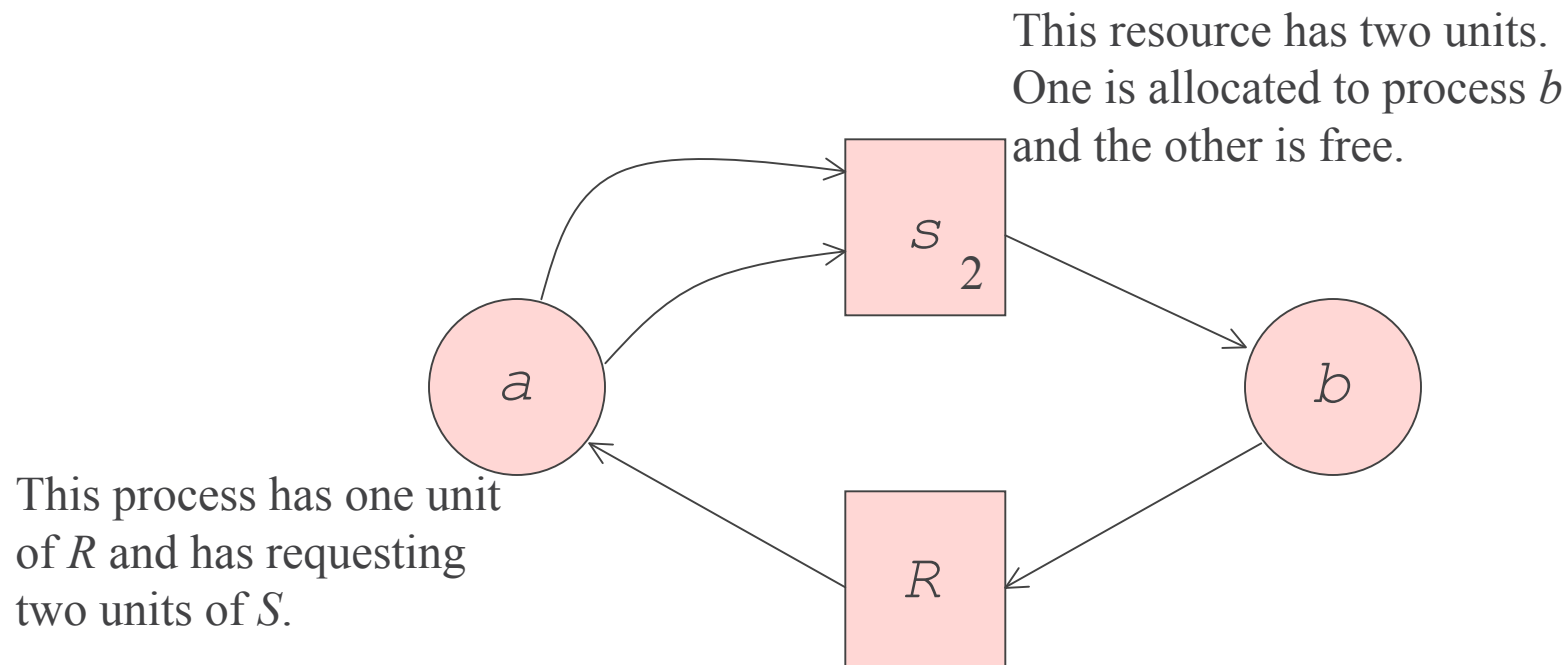
```
if (m.ccount > 0) {  
    signal(m.csem);  
    m.ccount--;  
}
```

Replace `c.notifyAll()` with:

```
while (m.ccount > 0) {  
    signal(m.csem);  
    m.ccount--;  
}
```

Modeling reusable resource allocation

Use bipartite *resource allocation graph*.



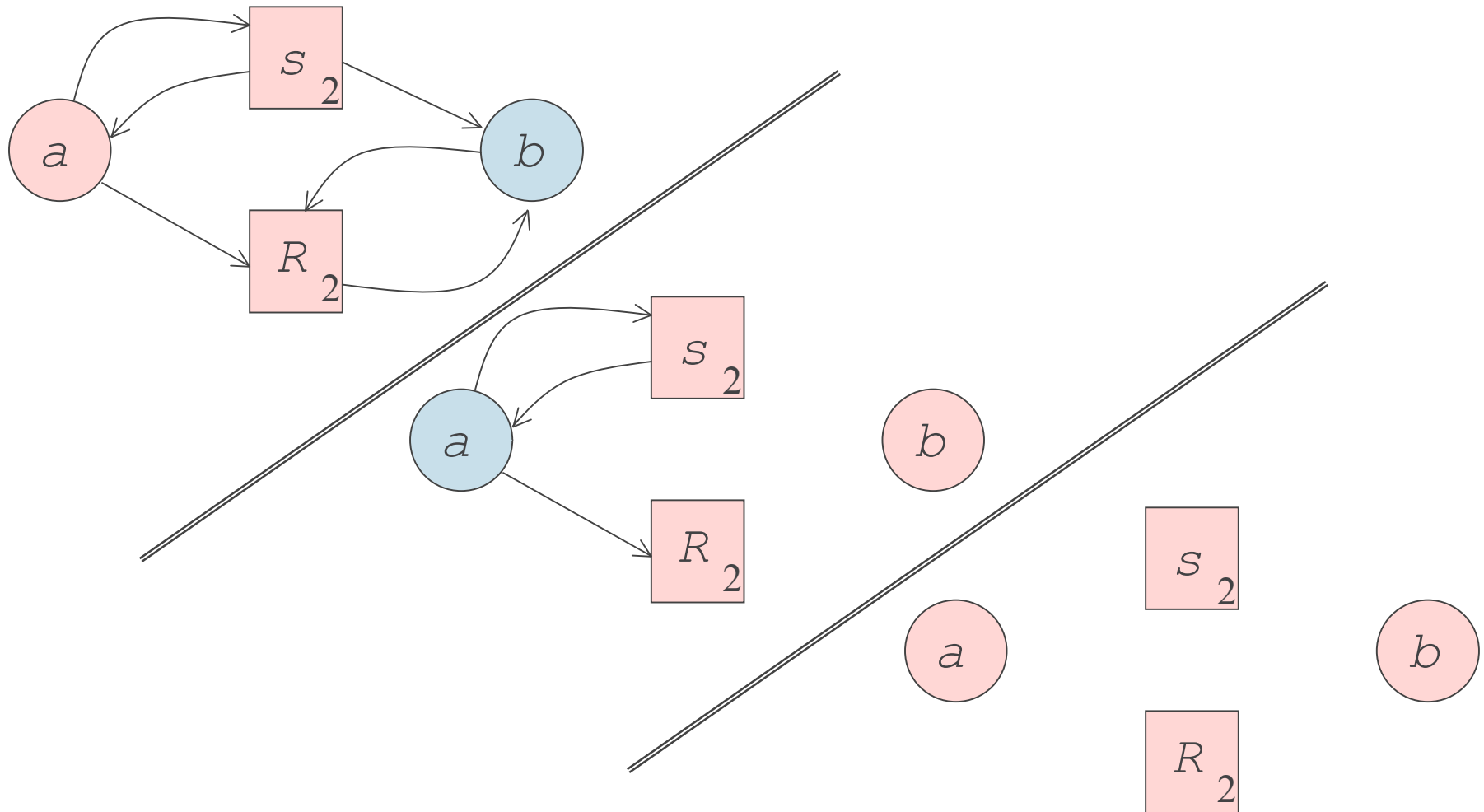
Deadlock theorem

- A resource allocation graph is *reduced* by repeatedly:
 - Select any process for which all outstanding requests can be granted;
 - Erase all edges incident on that process
- ... until no such processes remain.

If the resulting graph contains no edges, then the graph is *completely reduced*.

A system is deadlocked iff the resource allocation graph cannot be completely reduced

Example of reduction



Modeling with RAGs

Suppose there are x thing that can be allocated. Should you model them as x units of one resource, or as x resources where each has one unit?

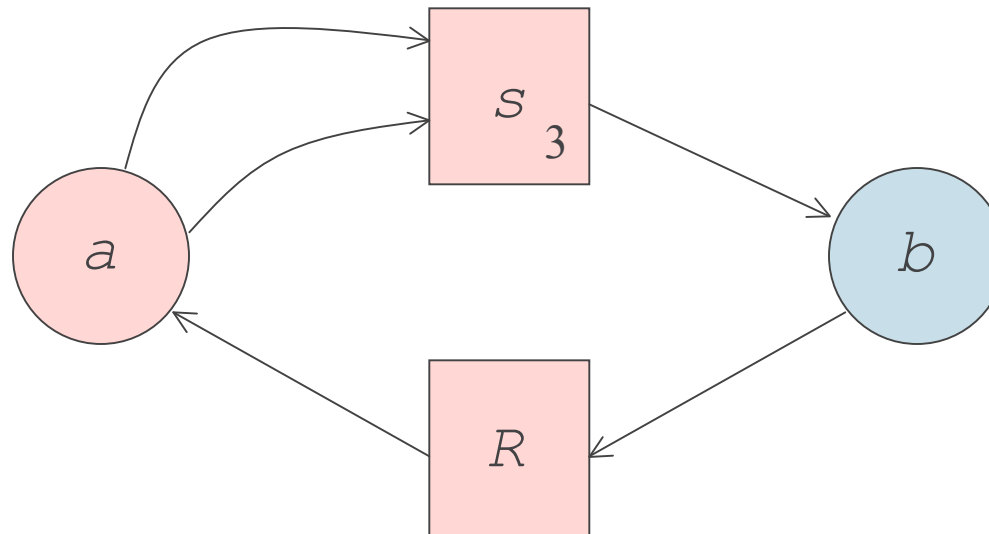
It depends on whether they are equivalent to each other. When a process wishes to allocate i of the x things, does it need specific things, or will any i do?

Detection (I)

- Maintain resource allocation graph. If an allocation request blocks, then reduce the graph.
 - If cannot fully reduce, there is a deadlock.
- Can detection be done more efficiently?

Detection (II)

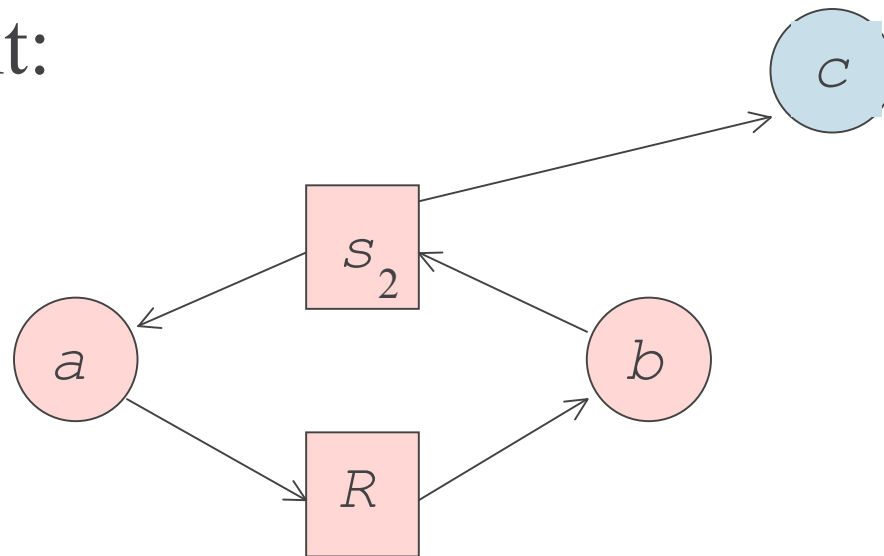
- A cycle in the RAG is necessary but not sufficient:



Detection (III)

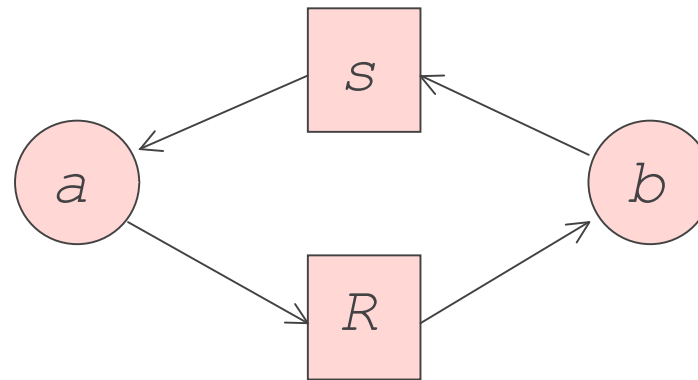
A RAG is *expedient* if there are no grantable requests.

- A cycle in an expedient RAG is not sufficient:



Detection (IV)

If all resources are *single unit*, then a cycle is necessary and sufficient.



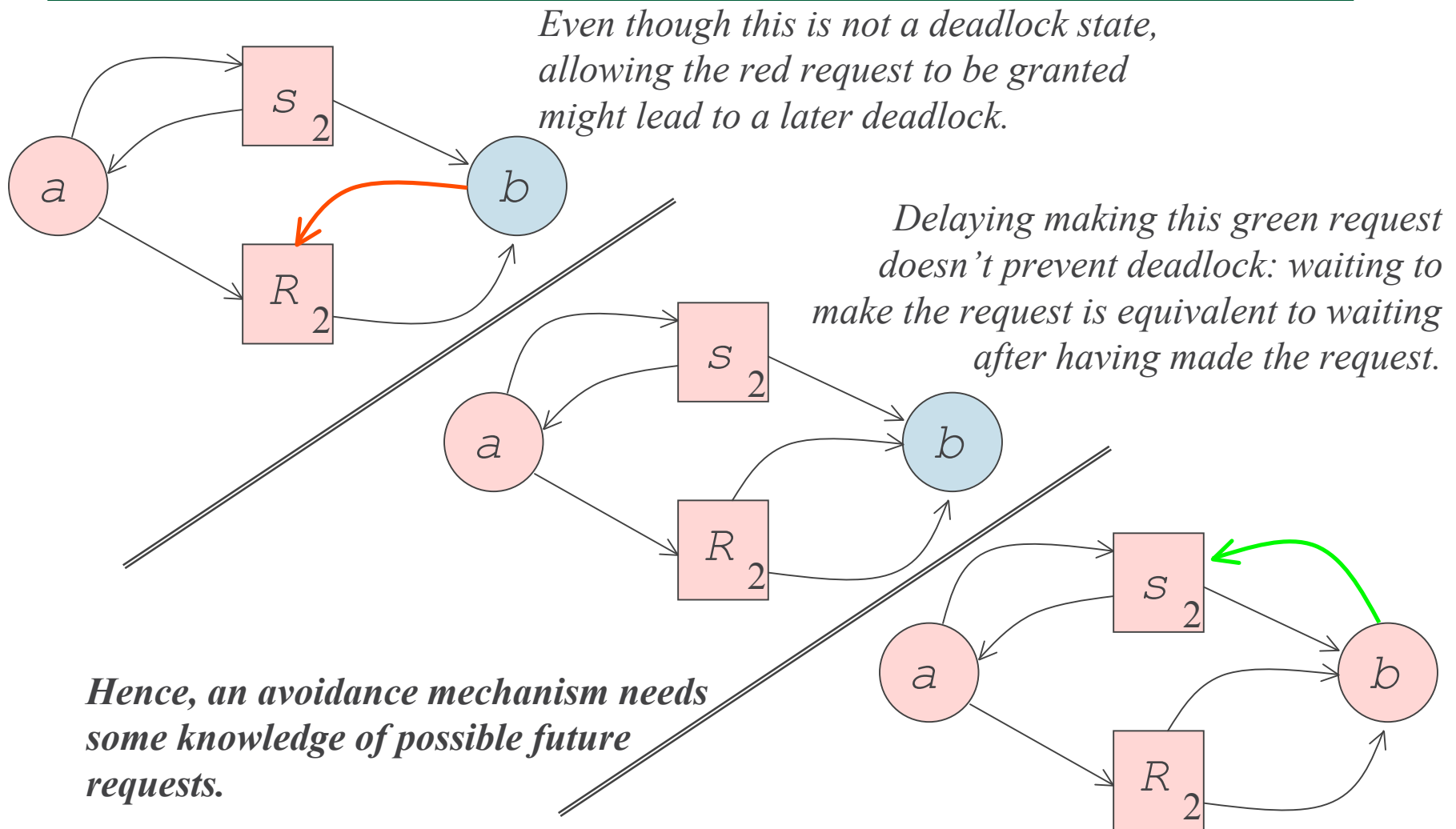
If single unit requests on multiple unit resources, then a *knot* is necessary and sufficient.

Detection (VI)

In summary,

- Reduction of RAG always detects deadlock.
- If single unit resources, cycle is necessary and sufficient.
- If single unit requests, knot is necessary and sufficient.
- What you do once a deadlock is detected is usually messy.

Avoidance (I)



Avoidance (II)

- For each process p and resource R , p initially declares the *maximum number units of R* it will ever require at any time.

Maximum Claims

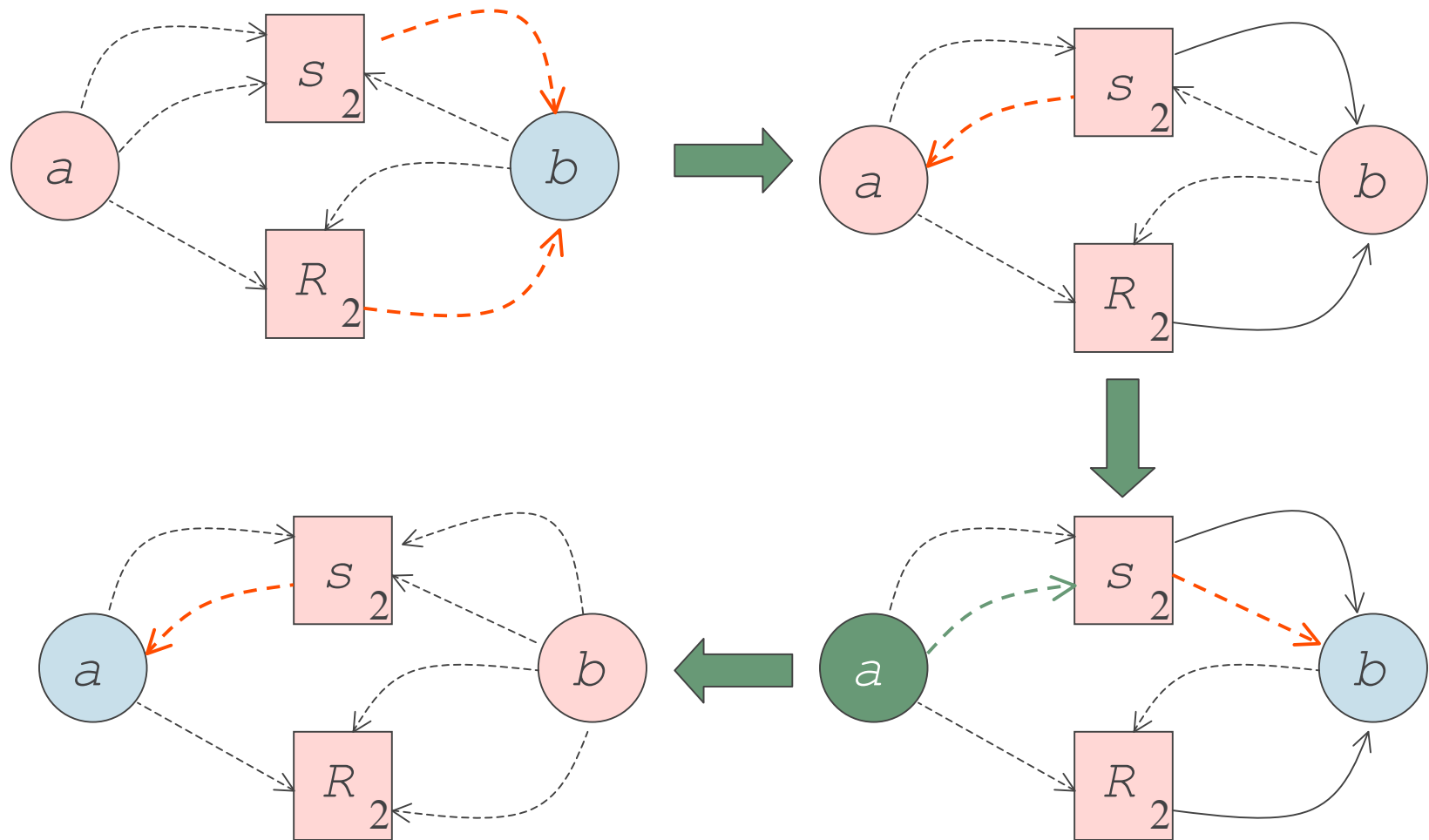
- The resource allocation graph initially consists of dashed request edges for these maximum claims.

Maximum Claims Graph

Avoidance (III)

- When a process p attempts to allocate resources, the equivalent number of dashed edges are reversed.
 1. If there are not enough dashed edges, then p is attempting to exceed its maximum claims and is terminated.
 2. If the resulting graph can be fully reduced, then the allocation is allowed and the reversed dashed arrows are made solid.
 3. Otherwise, they are reversed back and the allocation is delayed.
 4. Re-attempt allocation after any resource is release.

Avoidance (IV)



Bakery Algorithm

- Peterson's algorithm implements mutual exclusion for two threads.
- A *tournament tree* can be used to implement mutual exclusion for multiple threads.
- The *Bakery algorithm* is a more direct way to implement mutual exclusion for multiple threads: $O(n)$ rather than $O(n \log n)$ comparisons.
 - Called this because it is similar to taking a number for service in a deli or a bakery.

The Bakery Algorithm, I

```
int turn[n] = 0, 0, ..., 0;
int number = 1, next = 1;

// code for process i
int j;
while (1) {
    < turn[i] = number; number = number + 1; >
    < await (turn[i] == next); >

    critical section

    < next = next + 1; >

    noncritical code

}
```

The Bakery Algorithm, II

```
int turn[n] = 0, 0, ..., 0;

// code for process i
int j;
while (1) {
    < turn[i] = max(turn[0], ..., turn[n-1]) + 1; >
    for (j = 0; j < n; j++)
        if (j != i) < await (turn[j] == 0 || turn[i] < turn[j]); >

    critical section

    < turn[i] = 0; >

    noncritical code

}
```

The Bakery Algorithm, III

```
int turn0 = 0, turn1 = 0;
cobegin
  while (1) {
    turn0 = turn1 + 1;
    while (turn1 != 0 && turn0 > turn1) ;
    critical section
    turn0 = 0;
    noncritical code
  }
  ||
  while (1) {
    turn1 = turn0 + 1;
    while (turn0 != 0 && turn1 > turn0) ;
    critical section
    turn1 = 0;
    noncritical code
  }
coend
```

... but this isn't live.

The Bakery Algorithm, IV

```
int turn0 = 0, turn1 = 0;
cobegin
  while (1) {
    turn0 = turn1 + 1;
    while (turn1 != 0 && turn0 > turn1) ;
    critical section
    turn0 = 0;
    noncritical code
  }
  ||
  while (1) {
    turn1 = turn0 + 1;
    while (turn0 != 0 && turn1 ≥ turn0) ;
    critical section
    turn1 = 0;
    noncritical code
  }
coend
```

... but this isn't safe

The Bakery Algorithm, VI

```
int turn0 = 0, turn1 = 0;
int choosing0 = 0, choosing1 = 0;
cobegin
  while (1) {
    choosing0 = 1;
    turn0 = turn1 + 1;
    choosing0 = 0;
    while (choosing1) ;
    while (turn1 != 0 && turn0 > turn1) ;
    critical section
    turn0 = 0;
    noncritical code
  }
||
  while (1) {
    choosing1 = 1;
    turn1 = turn0 + 1;
    choosing1 = 0;
    while (choosing0) ;
    while (turn0 != 0 && turn1 ≥ turn0) ;
    critical section
    turn1 = 0;
    noncritical code
  }
coend
```

The Bakery Algorithm, VII

... can make symmetric by defining $[a, b] > [c, d]$ to be $(a > b) \vee ((a = b) \wedge (c > d))$

```
int turn0 = 0, turn1 = 0;
int choosing0 = 0, choosing1 = 0;
cobegin
  while (1) {
    choosing0 = 1;
    turn0 = turn1 + 1;
    choosing0 = 0;
    while (choosing1) ;
    while (turn1 != 0 && [turn0, 0] > [turn1, 1]) ;
    critical section
    turn0 = 0;
    noncritical code
  }
|| ...
coend
```

The Bakery Algorithm, VIII

```
int turn[n] = 0, 0, ..., 0;
int choosing[n] = 0, 0, ..., 0;
// code for process i
int j;
while (1) {
    choosing[i] = 1;
    turn[i] = max(turn[0], ..., turn[n-1]) + 1;
    choosing[i] = 0;
    for (j = 0; j < n; j++) if (j != i) do {
        while (choosing[j]) ;
        while (turn[j] != 0 && [turn[i], i] > [turn[j], j]) ;
    }

    critical section

    turn[i] = 0;

    noncritical code

}
```