

Introduction to Deadlock



CSE 120
Spring 2002

Deadlock

Indicates set of processes blocked waiting for events that can be generated only by the blocked processes.

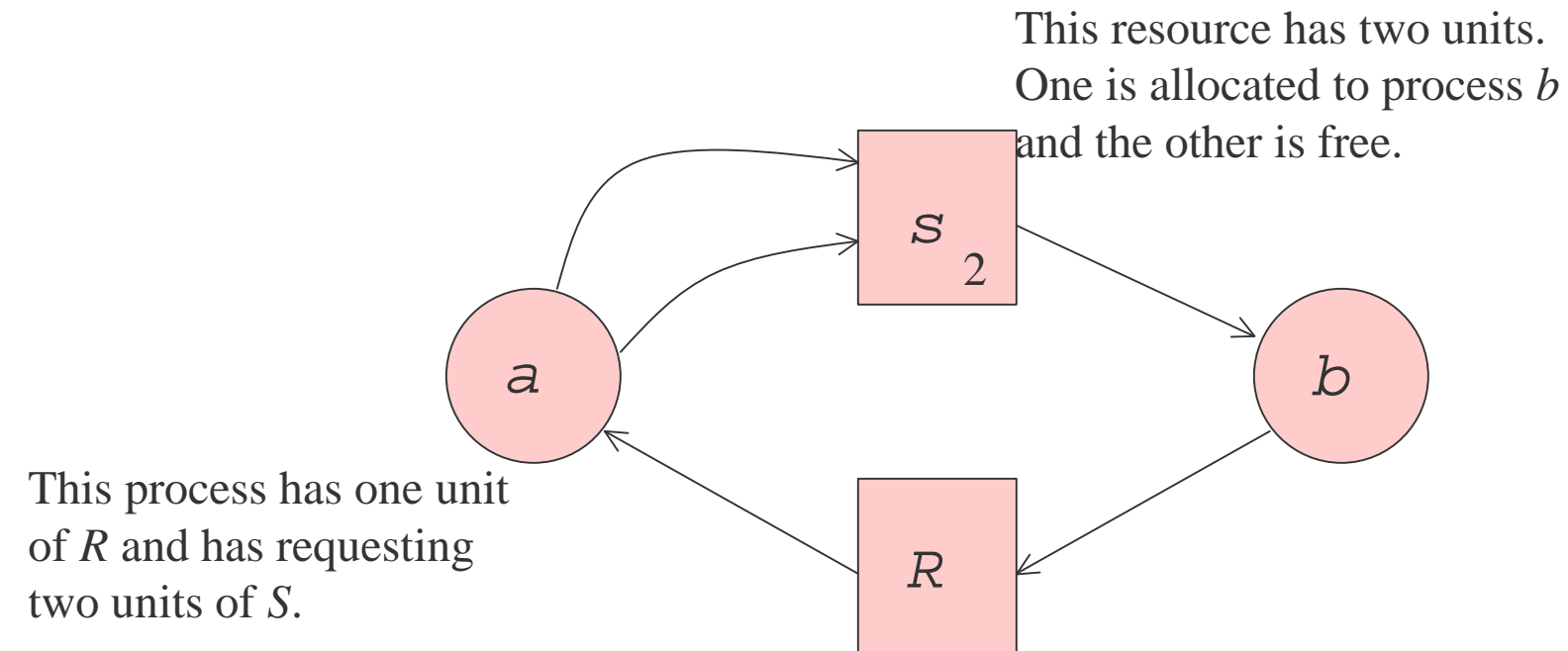
... often characterized as a problem arising from *resource allocation*.

Reusable Resources

- *Reusable* resources can be modeled as a pool of identical and exchangeable resources:
 - $\text{allocate}(R_1, k_1, R_2, k_2, \dots)$ allocate k_1 units of R_1 ,
 k_2 units of R_2 , etc, only
when all are free.
 - $\text{free}(r_1, r_2, \dots)$ free resources r_1, r_2, \dots
Process must own the units
Can then be allocated again.
- Other situations can also lead to deadlock.
 - We don't consider such deadlocks here.

Modeling reusable resource allocation

Use bipartite *resource allocation graph*.



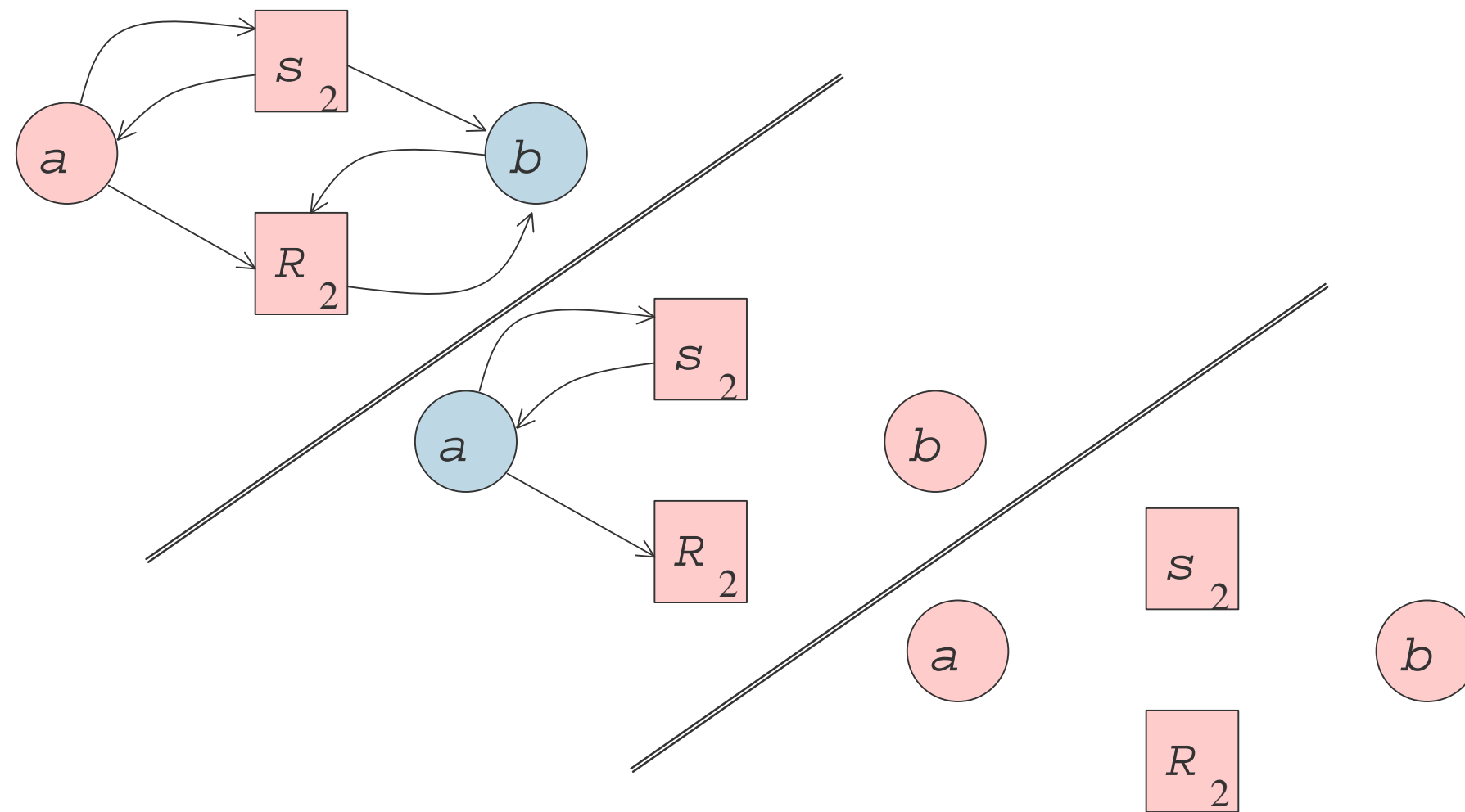
Deadlock theorem

- 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 things 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?

Dealing with Deadlock

Detection

- When detected, decide which process to roll back or abnormally terminate.

Avoidance

- Before allocating a resource, ensure that the resulting state cannot lead to deadlock.

• Prevention

- Ensure that deadlock is not possible.

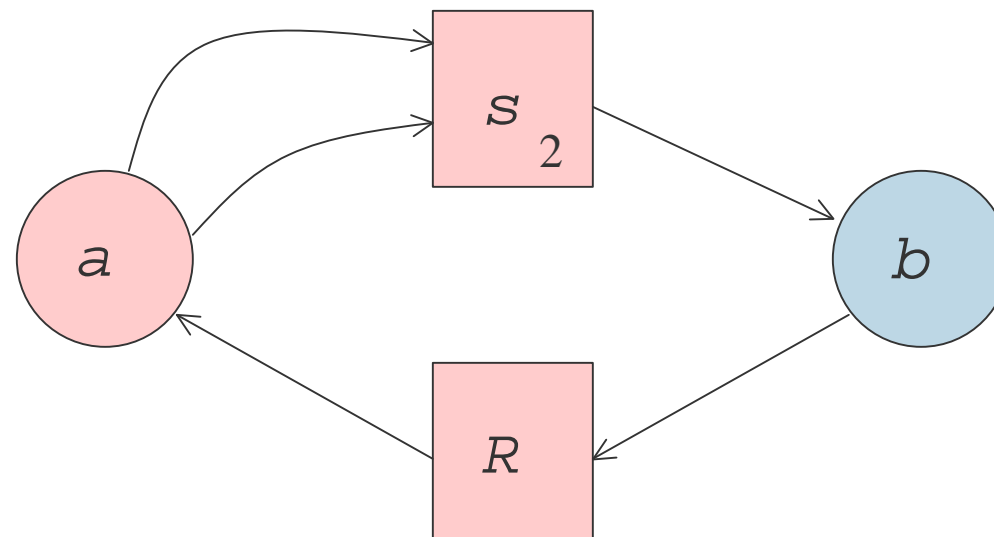
Detection (I)

- 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)

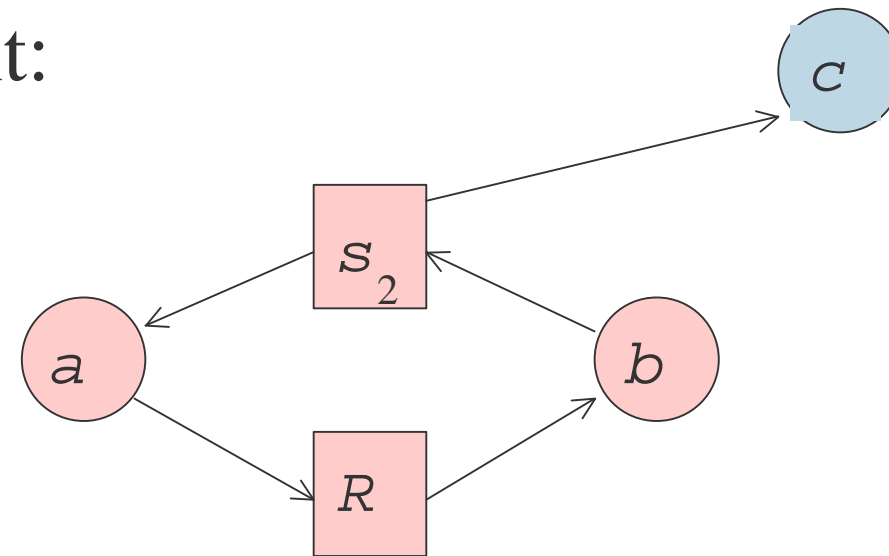
- A cycle in the RAG is necessary but not sufficient:



Detection (III)

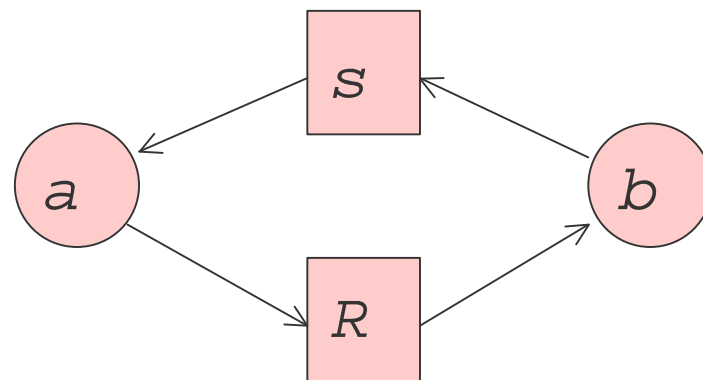
A RAG is *expedient* if there are no grantable requests.

- A cycle in an expedient RAG is not sufficient:



Detection (IV)

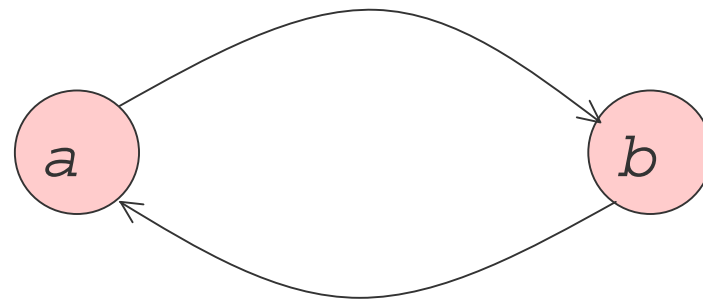
If all resources are single unit, then a cycle is necessary and sufficient.



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

Detection (V)

For single unit resources, we can use the simpler *waits for* graph:



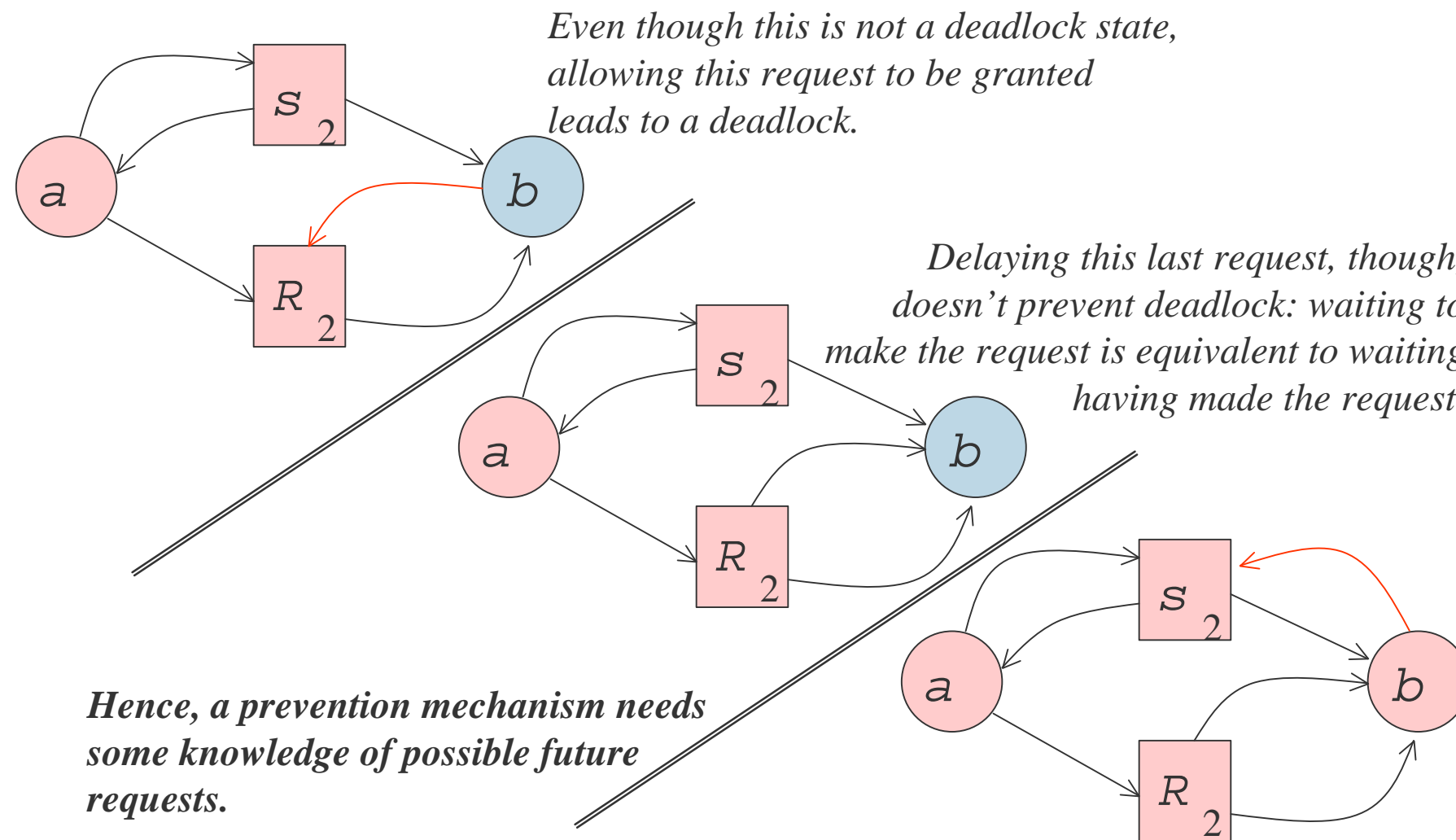
... a cycle is equivalent to deadlock.

Detection (VI)

In summary,

- Reduction of RAG always works.
- If single unit resources, cycle is necessary and sufficient.
- If single unit requests, knot is necessary and sufficient.
- What you do when 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

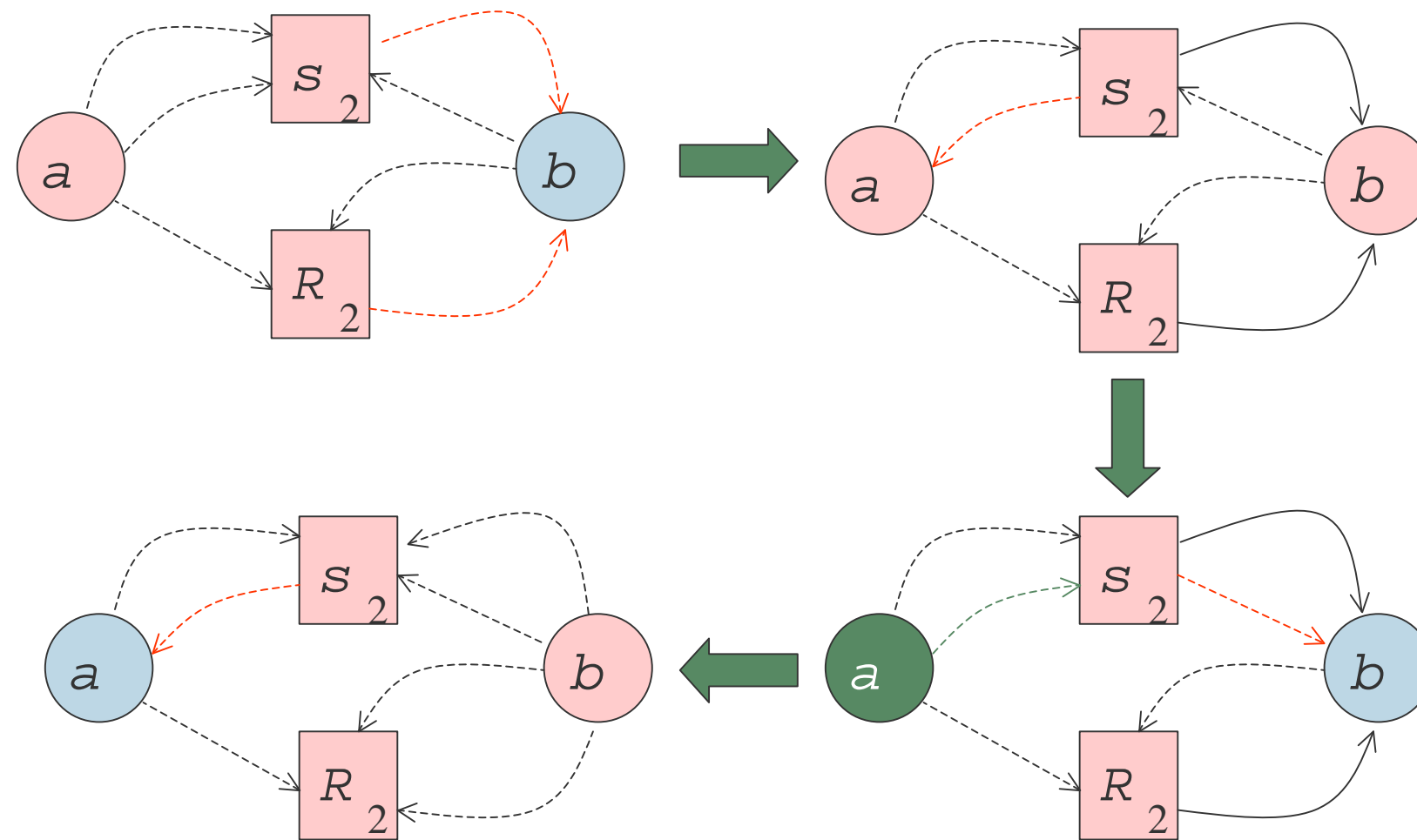
- The resource allocation graph initially consists of dashed request edges for these maximum claims.

Maximum Claims Graph

Avoidance (III)

- When a process attempts to allocate resources, the equivalent number of dashed edges are reversed.
 - If there are not sufficient dashed edges, then the process is attempting to exceed its maximum claims, and is therefore terminated.
 - If the resulting graph can be fully reduced, then the allocation is allowed and the reversed dashed arrows are made solid.
 - Otherwise, they are reversed back and the allocation is delayed.
 - Delayed allocations are re-attempted after any resource release.

Avoidance (IV)



Avoidance (V)

- This algorithm (written in a more conventional manner) is called the *Banker's Algorithm (Dijkstra)*.
- Applicable only if one can give maximum claims.

Prevention (I)

The four conditions needed for deadlock are:

1. Mutual exclusion of resource units.
2. Processes can hold resources when they request other resources.
3. Resources can not be preemptively taken away from a process.
4. Circular wait.

... *prevention* is done by breaking one of these conditions.

Prevention (II)

We can break (2): *hold and wait*.

A process allocates all resources it needs at once.

... *static two phase locking* uses this.

Prevention (III)

Break (4): *cyclic waiting*. Done usually with *hierarchical locking*:

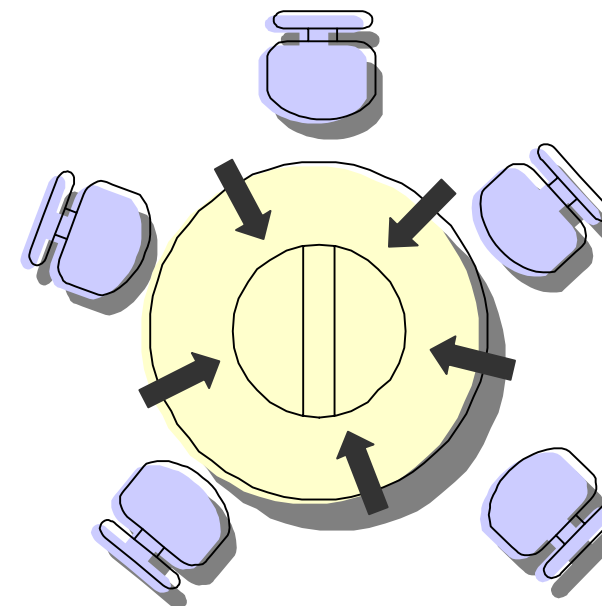
- Assign to each resource r an integer level $L(r)$.
- Let $R(p)$ be the resources allocated to p . It can allocate r only if

$$\forall r' \in R(p): L(r') < L(r).$$

Dining Philosophers (I)

- Five philosophers $\{p_0, p_1, p_2, p_3, p_4\}$
 - *Thinking or eating*
- Five forks $\{f_0, f_1, f_2, f_3, f_4\}$
 - *Free or in use.*
- p_i needs f_i and $f_{i\oplus 1}$ to eat.
- A deadlocking protocol:

```
Philosopher  $i$ :  
while (1) {  
    pick up  $f_i$ ;  
    pick up  $f_{i\oplus 1}$ ;  
    eat;  
    put down  $f_i$ ;  
    put down  $f_{i\oplus 1}$ ;  
}
```



Dining Philosophers (II)

Use hierarchical locking with $L(f_i) = i$.

Philosopher i :

$fst = \min(i, i \oplus 1)$;

$snd = \max(i, i \oplus 1)$;

while (1) {

 pick up f_{fst} ;

 pick up f_{snd} ;

 eat;

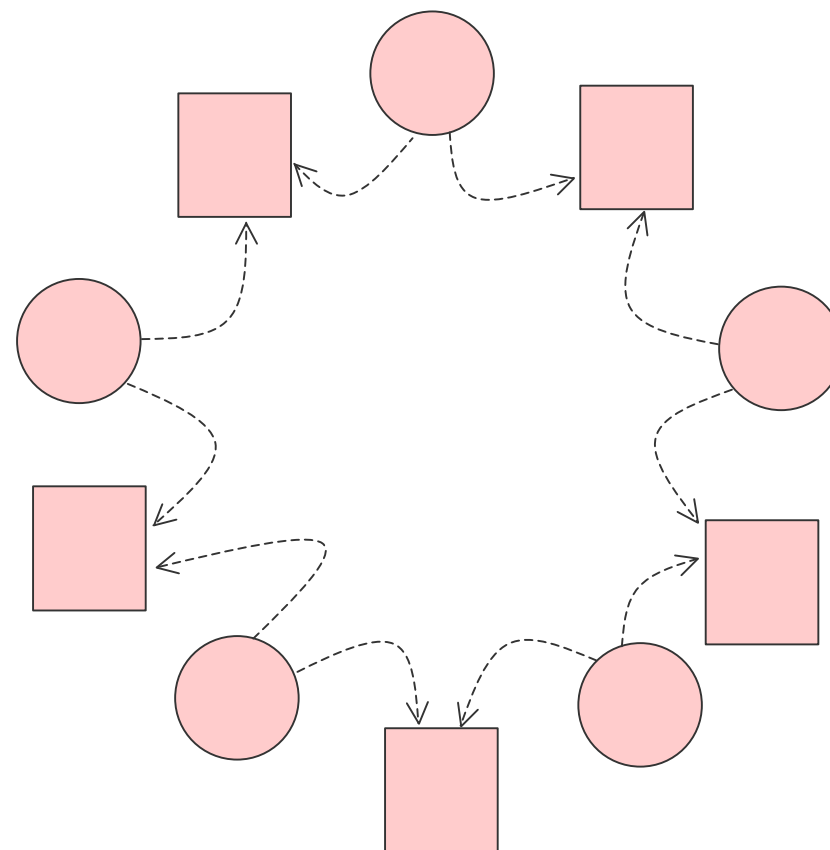
 put down f_{fst} ;

 put down f_{snd} ;

}

Dining Philosophers (III)

Banker's algorithm: maximum claims graph



- Since these are single unit resources, the RAG can be fully reduced iff it does not contain a cycle.

- For there to be a cycle each philosopher must both have a fork and be waiting for a fork.

- Hence, the RAG can be fully reduced when there is at least one thinking philosopher.

So, avoid deadlock by never allowing more than four dining philosophers at a time.

Dining Philosophers (IV)

```
monitor DP {
    const int n = 5;
    int forkFree[n], eating = 0;           // 0 <= eating <= 4
    condition eat[n]; // eating < 4 ∧ fork[i] ∧ fork[i⊕1]
public:
    void entry Eat(int i);
    void entry Think(int i);
    DP(void);
};
```

Dining Philosophers (V)

```
DP::DP(void) {
    int i;
    for (i = 0; i < n; i++) forkFree[0] = 1;
}

void DP::Eat(int i) {
    if (eating == 4 || ! fork[i] || ! fork[(i+1)%n]) eat[i].wait( );
    eating++;
    fork[i] = fork[(i+1)%n] = 0;
}

void entry Think(int i) {
    eating--;
    fork[i] = fork[(i+1)%n] = 1;
    if (fork[(i-1)%n]) eat[(i-1)%n].signal( );
    if (eating < 4 && fork[(i+1)%n] && fork[(i+2)%n])
        eat[(i+1)%n].signal( );
}
```