Lecture 7

Deadlock

30 September, 2011

semaphore S = T = 1

Process 1	Process 2
:	:
down(S)	down(T)
down(T)	down(S)
up(T)	up(S)
up(S)	up(T)

⋮

down(S)

down(T)

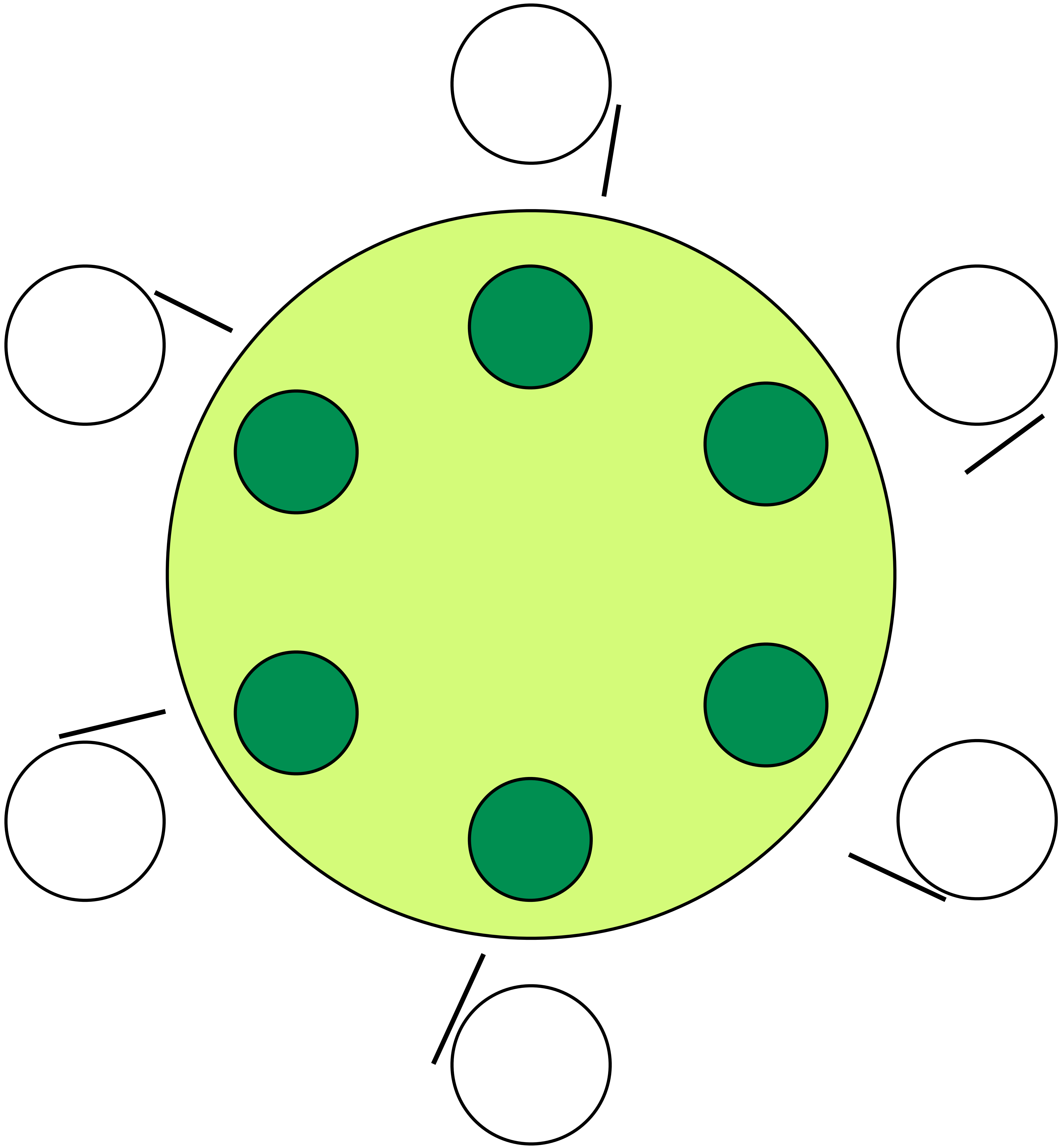
⋮

down(T)

down(S)

```
while (1)
  think
  wait till left chopstick is available
  pick up left chopstick
  wait till right chopstick is available
  pick up right chopstick
  eat
  put down left chopstick
  put down right chopstick
```

```
while (1)
  think
  down(chopstick[i])
  down(chopstick[(i+1)%N]
  eat
  up(chopstick[i])
  up(chopstick[(i+1)%N]
```



Resource

acquire resource
(wait until available)
use resource
release resource

software

vs.

hardware

resource

preemptive

vs.

non-preemptive

resource

single copy

vs.

multiple copies

of a resource

4

conditions
for deadlock

mutual exclusion

each resource must be either
assigned to exactly one process
or is available

```
while (1)
  think
  down(chopstick[i])
  down(chopstick[(i+1)%N]
  eat
  up(chopstick[i])
  up(chopstick[(i+1)%N]
```

hold and wait

processes holding resources granted earlier can request for new resource

```
while (1)
  think
  down(chopstick[i])
  down(chopstick[(i+1)%N]
  eat
  up(chopstick[i])
  up(chopstick[(i+1)%N]
```

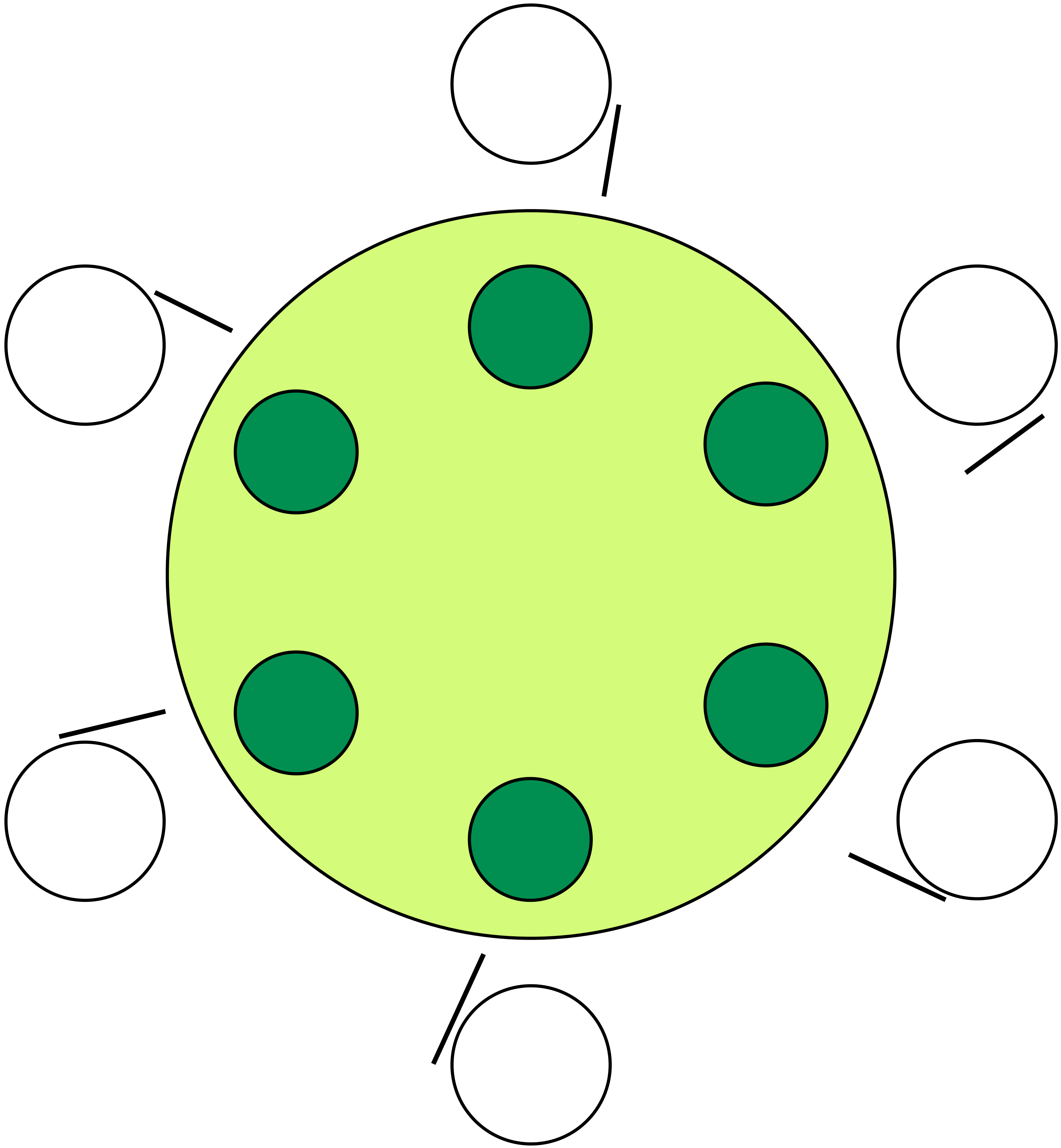
no preemption

resources granted cannot be
forcefully taken away

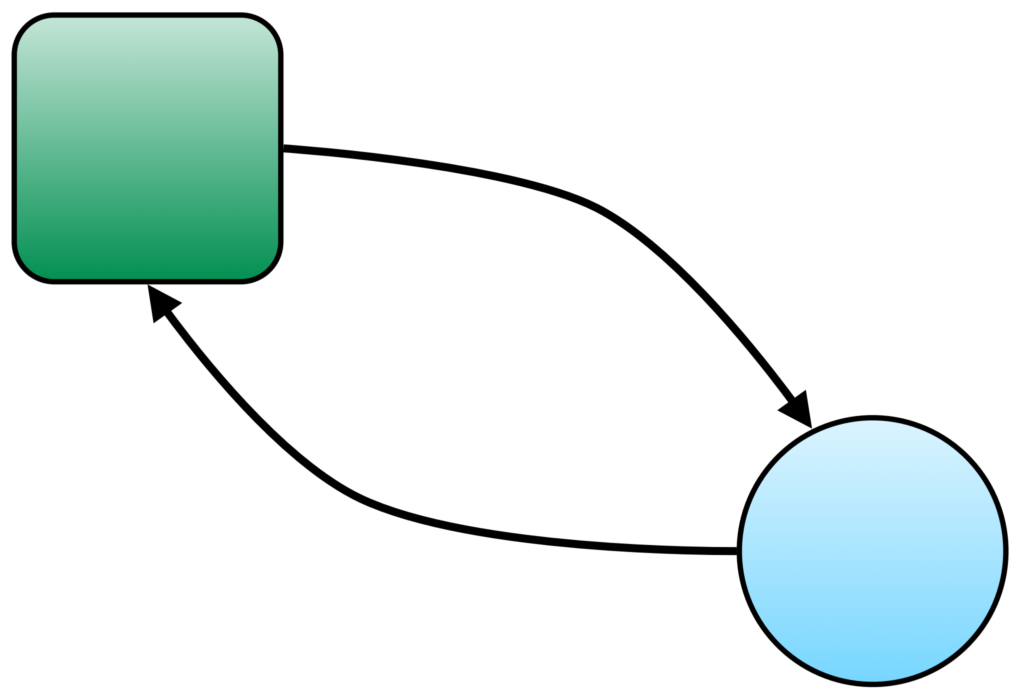
```
while (1)
  think
  down(chopstick[i])
  down(chopstick[(i+1)%N]
  eat
  up(chopstick[i])
  up(chopstick[(i+1)%N]
```

circular waiting

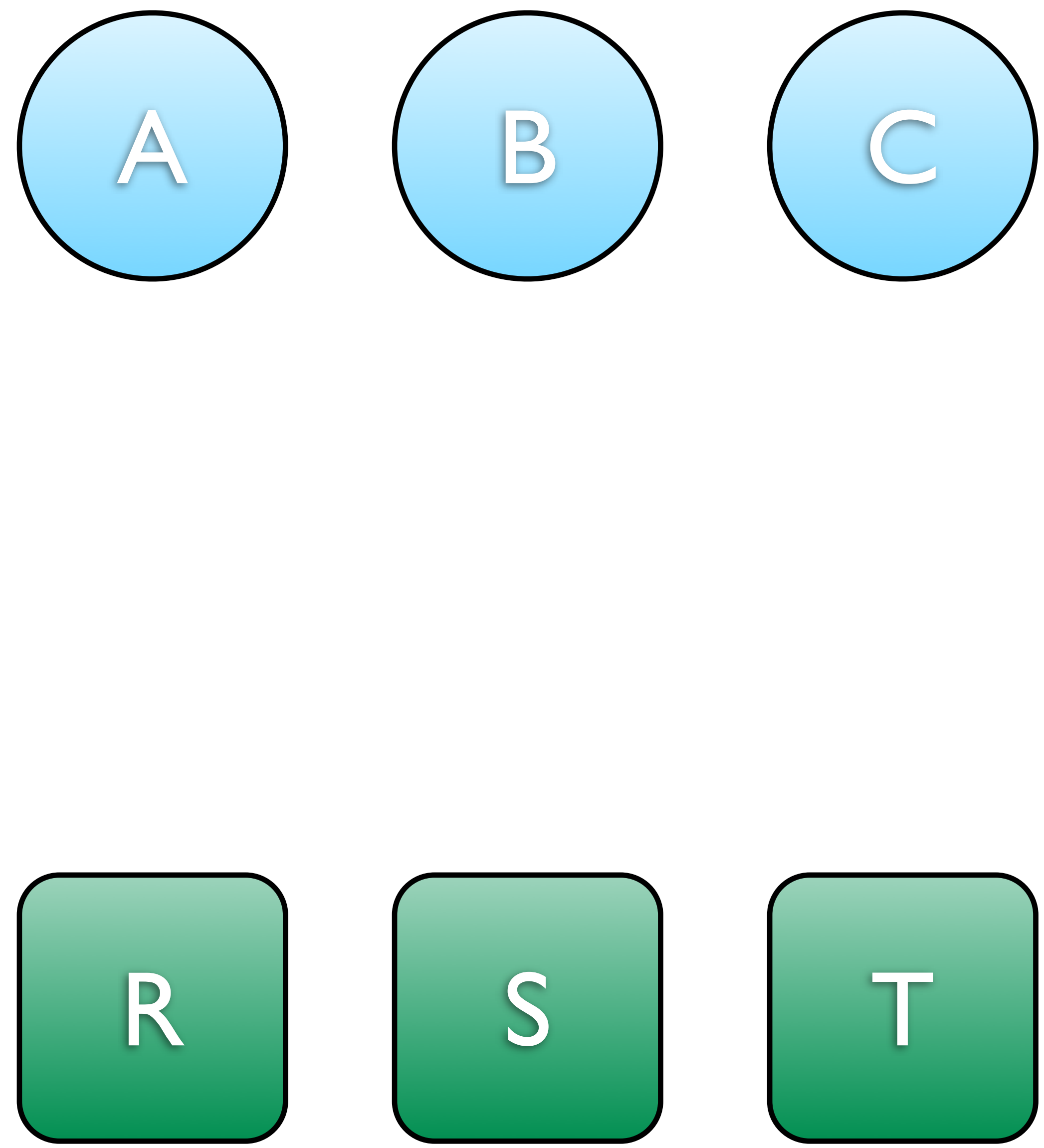
a circular chain of processes, each waiting for a resource held by the next member of the chain



Deadlock Modeling



A requests **R**
B requests **S**
C requests **T**
A requests **S**
B requests **T**
C requests **R**



Deadlock Detection

is the system deadlocked, and if so, which process are involved?

Deadlock Detection
(if each resource type has one copy)

1. periodically build a resource graph

2. run depth first search on the graph to detect cycle

Deadlock Detection

(if each resource
type has multiple copies)

resources in existence

R	S	T	U
4	2	3	1

resources available

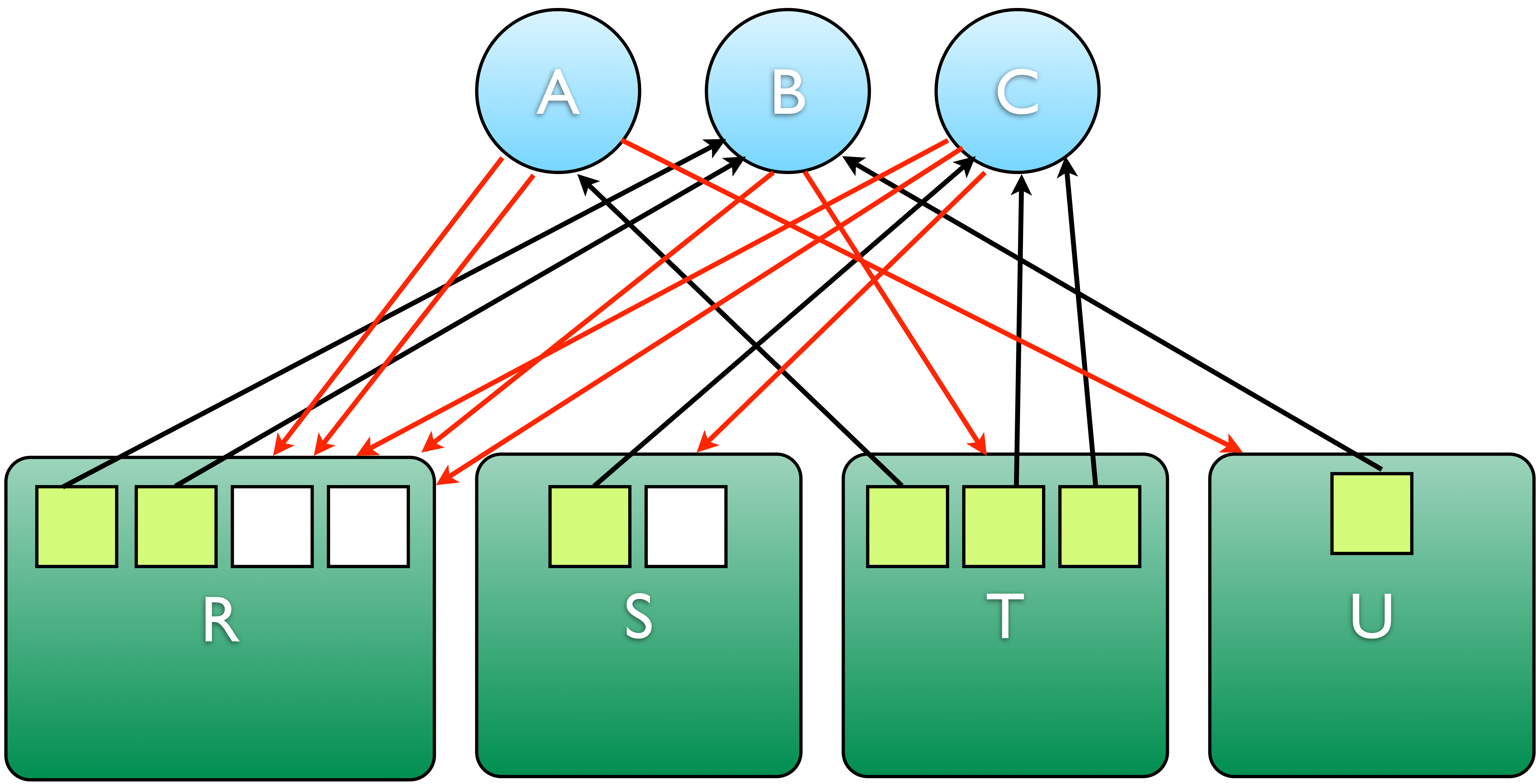
R	S	T	U
2	1	0	0

allocation matrix

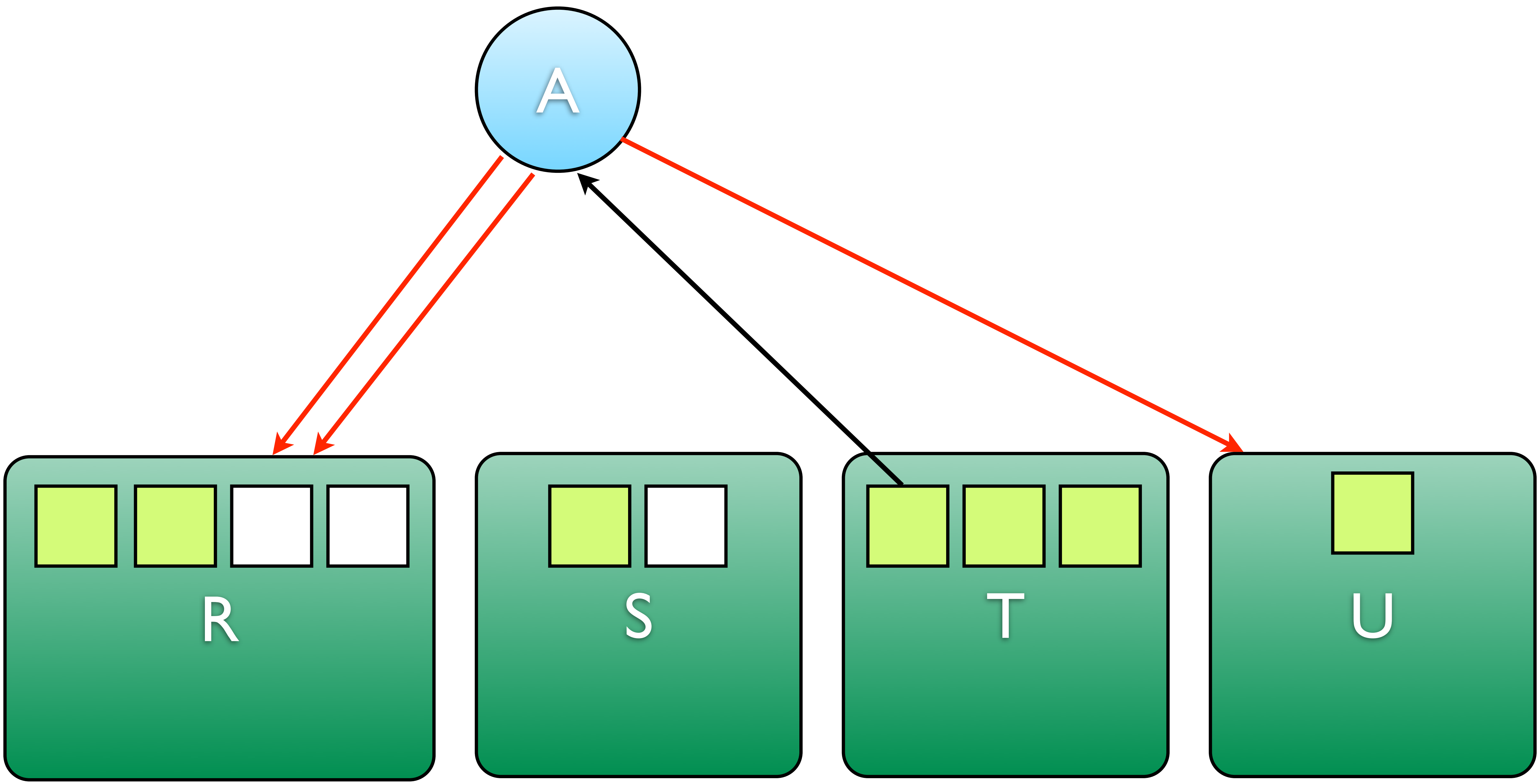
	R	S	T	U
A	0	0	1	0
B	2	0	0	1
C	0	1	2	0

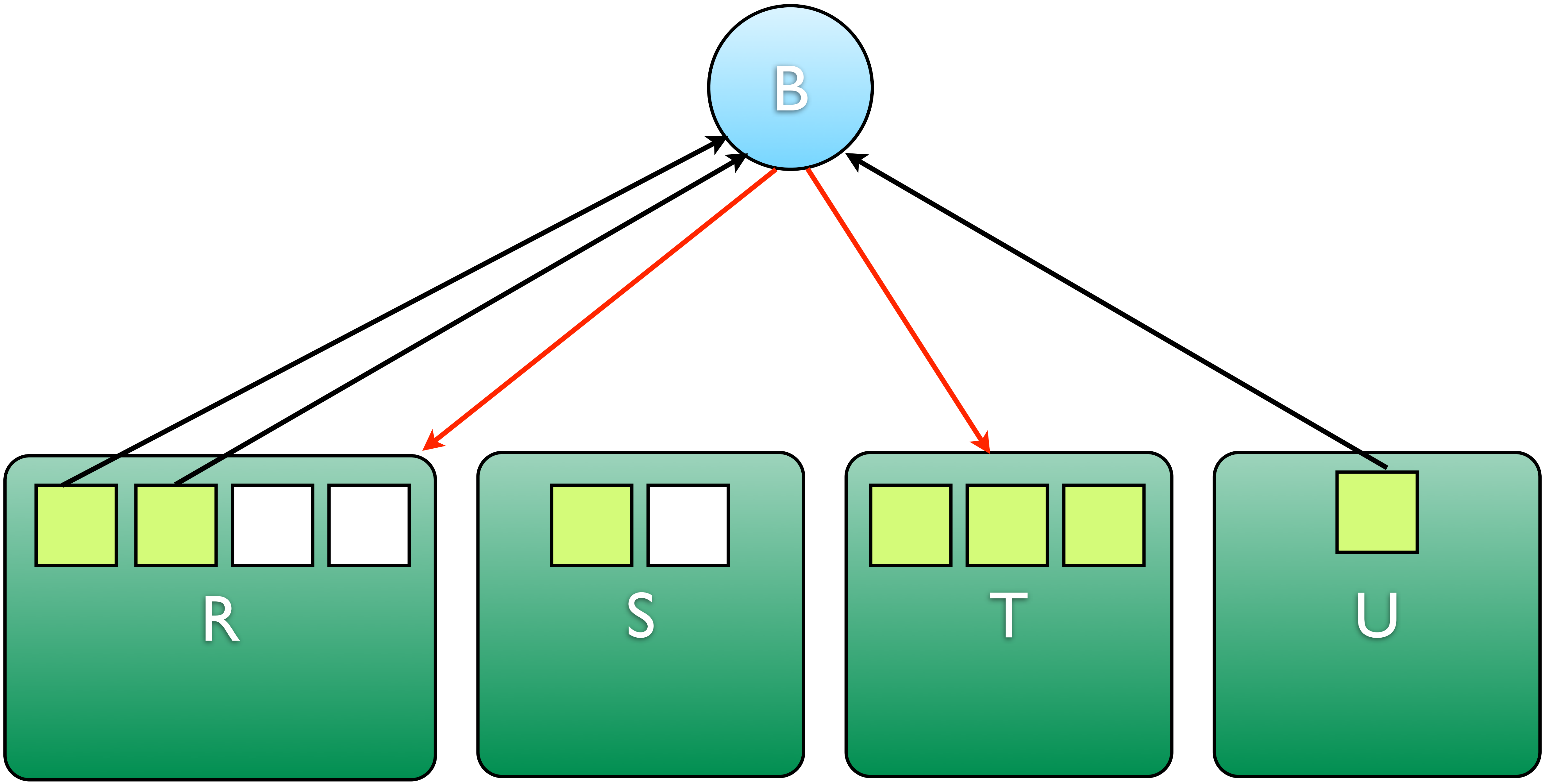
request matrix

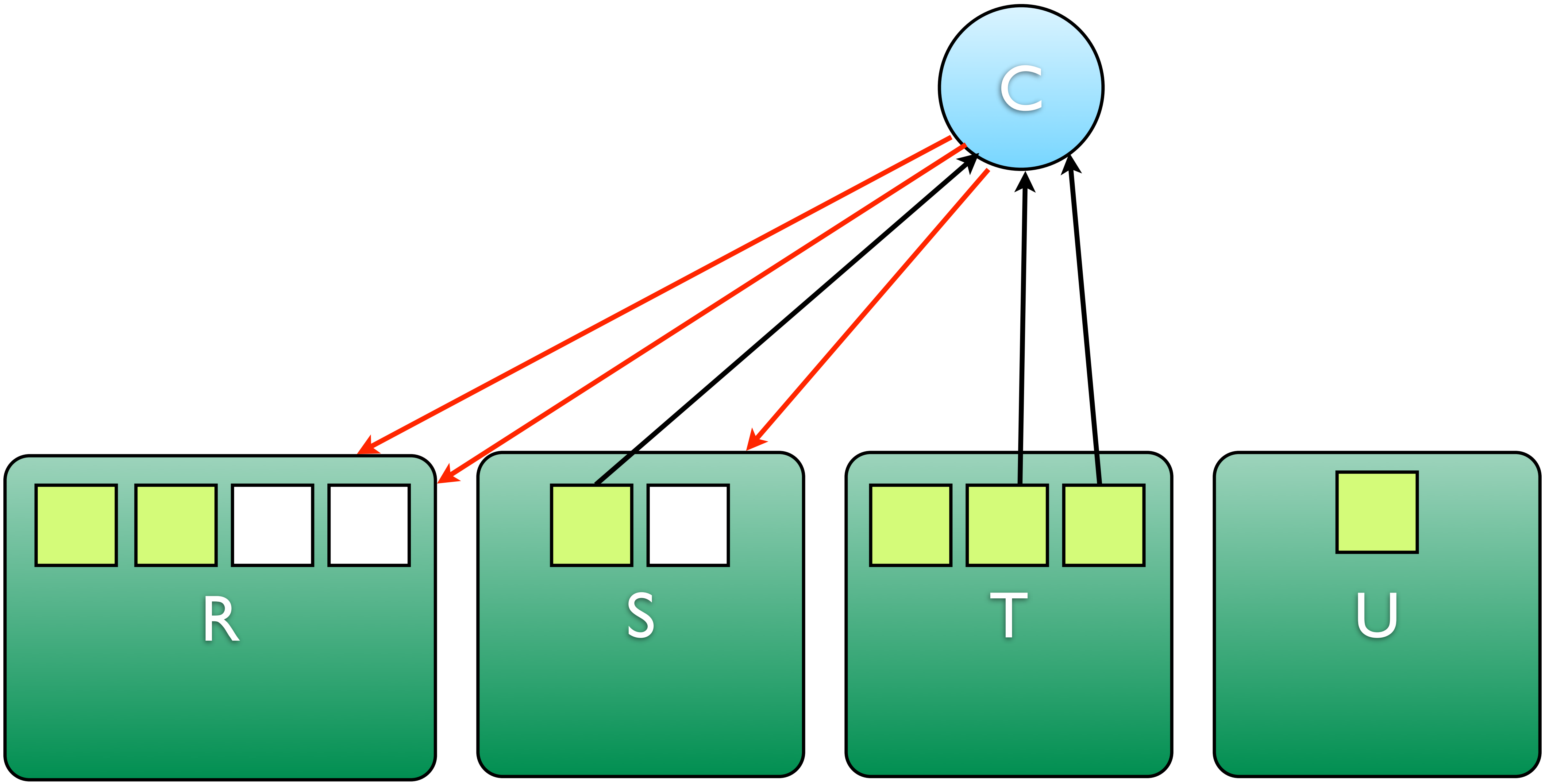
	R	S	T	U
A	2	0	0	1
B	1	0	1	0
C	2	1	0	0

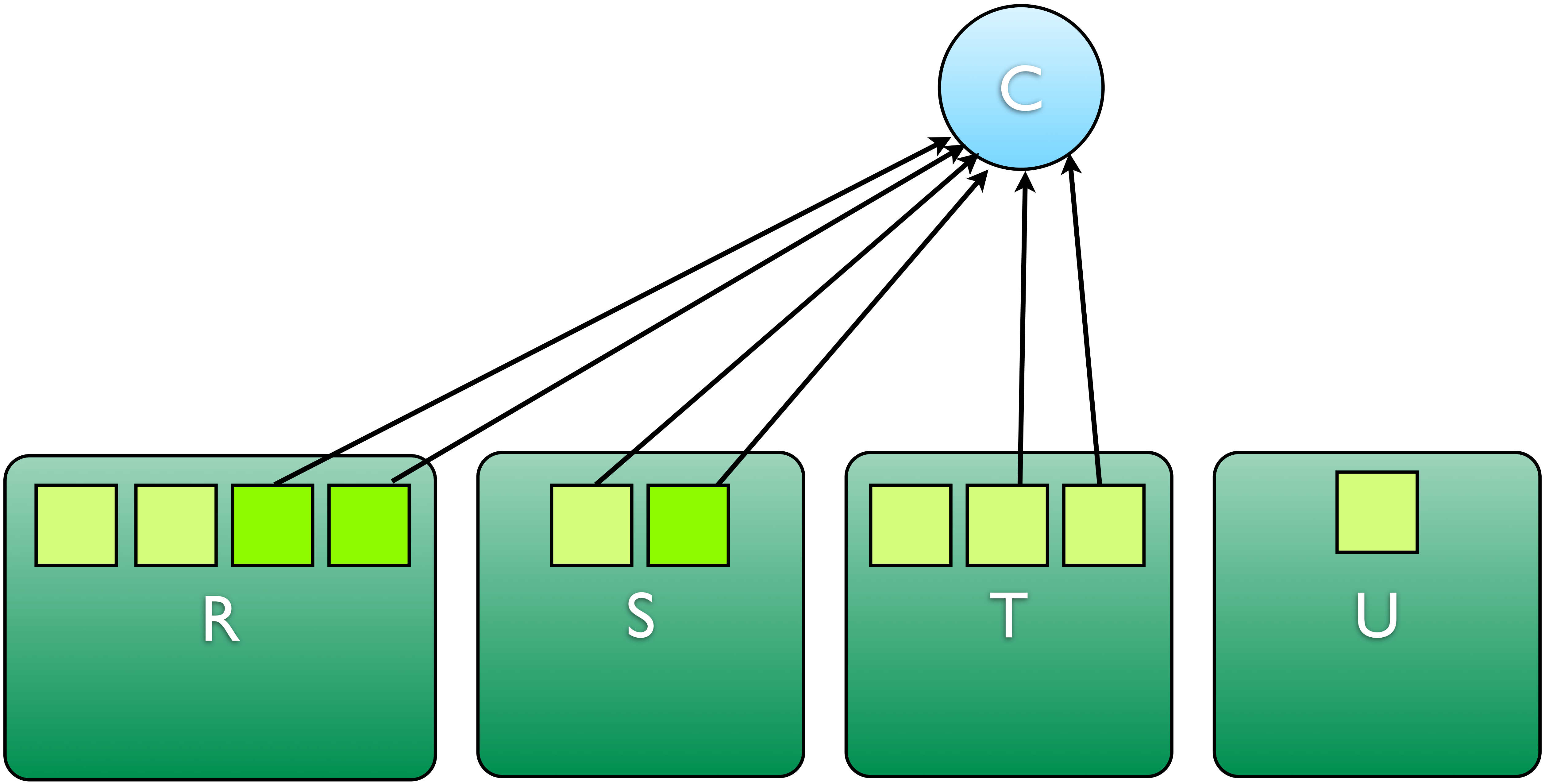

A
B
C
R
S
T
U

is there a process whose requests can be satisfied?


A
R
S
T
U


B
R
S
T
U


C
R
S
T
U


C
R
S
T
U

resources in existence

R	S	T	U
4	2	3	1

resources available

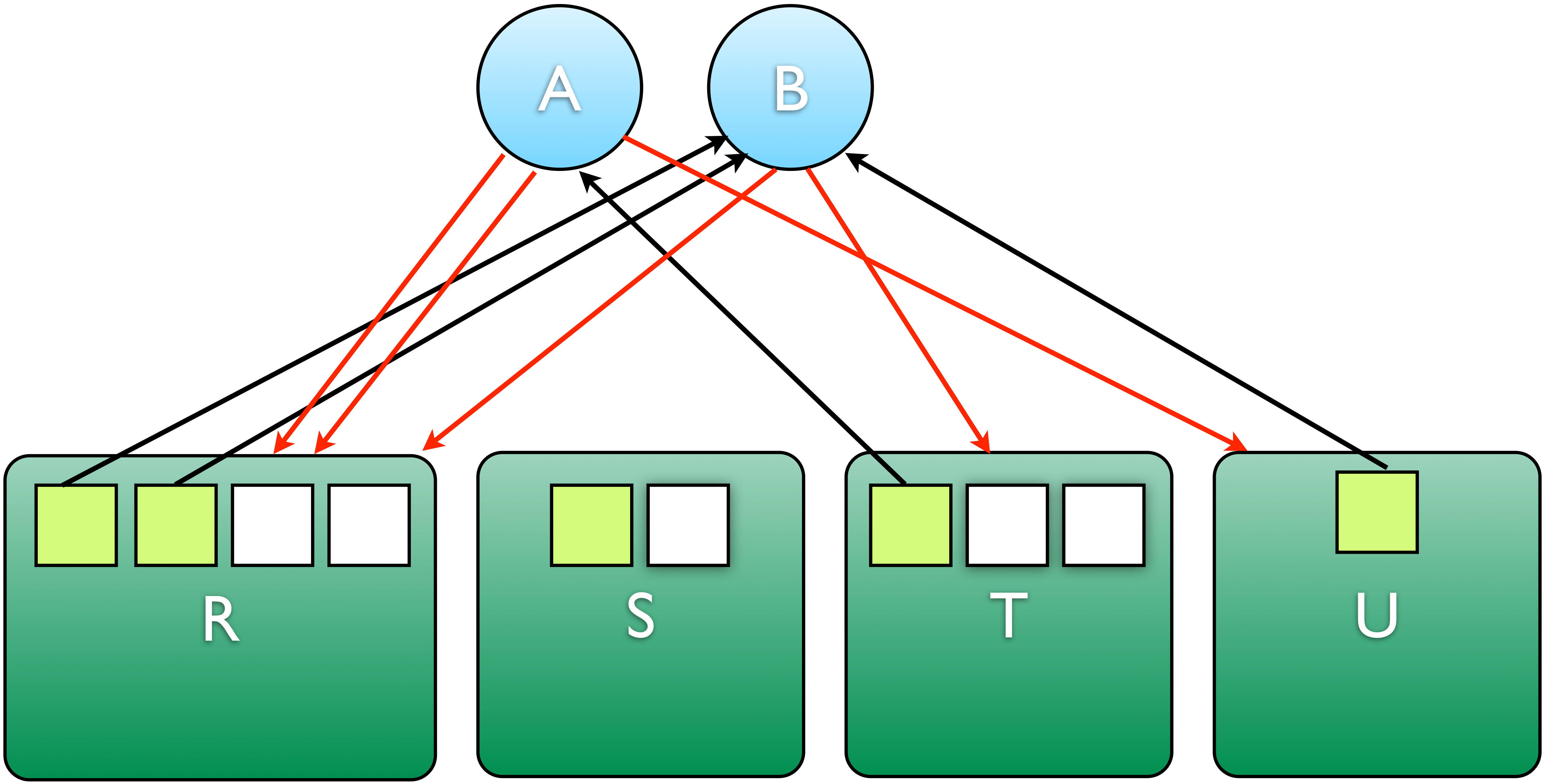
R	S	T	U
2	1	0	0

allocation matrix

	R	S	T	U
A	0	0	1	0
B	2	0	0	1
C	0	1	2	0

request matrix

	R	S	T	U
A	2	0	0	1
B	1	0	1	0
C	2	1	0	0


A
B
R
S
T
U

resources in existence

R	S	T	U
4	2	3	1

resources available

R	S	T	U
2	2	2	0

allocation matrix

	R	S	T	U
A	0	0	1	0
B	2	0	0	1
C	0	0	0	0

request matrix

	R	S	T	U
A	2	0	0	1
B	1	0	1	0
C	0	0	0	0

resources in existence

R	S	T	U
4	2	3	1

resources available

R	S	T	U
4	2	2	1

allocation matrix

	R	S	T	U
A	0	0	1	0
B	0	0	0	0
C	0	0	0	0

request matrix

	R	S	T	U
A	2	0	0	1
B	0	0	0	0
C	0	0	0	0

suppose we have
deadlock,
now what?

1. preempt
2. rollback
3. terminate

Deadlock Avoidance

if we know the resources required by a process,
can we avoid deadlock by careful allocation?

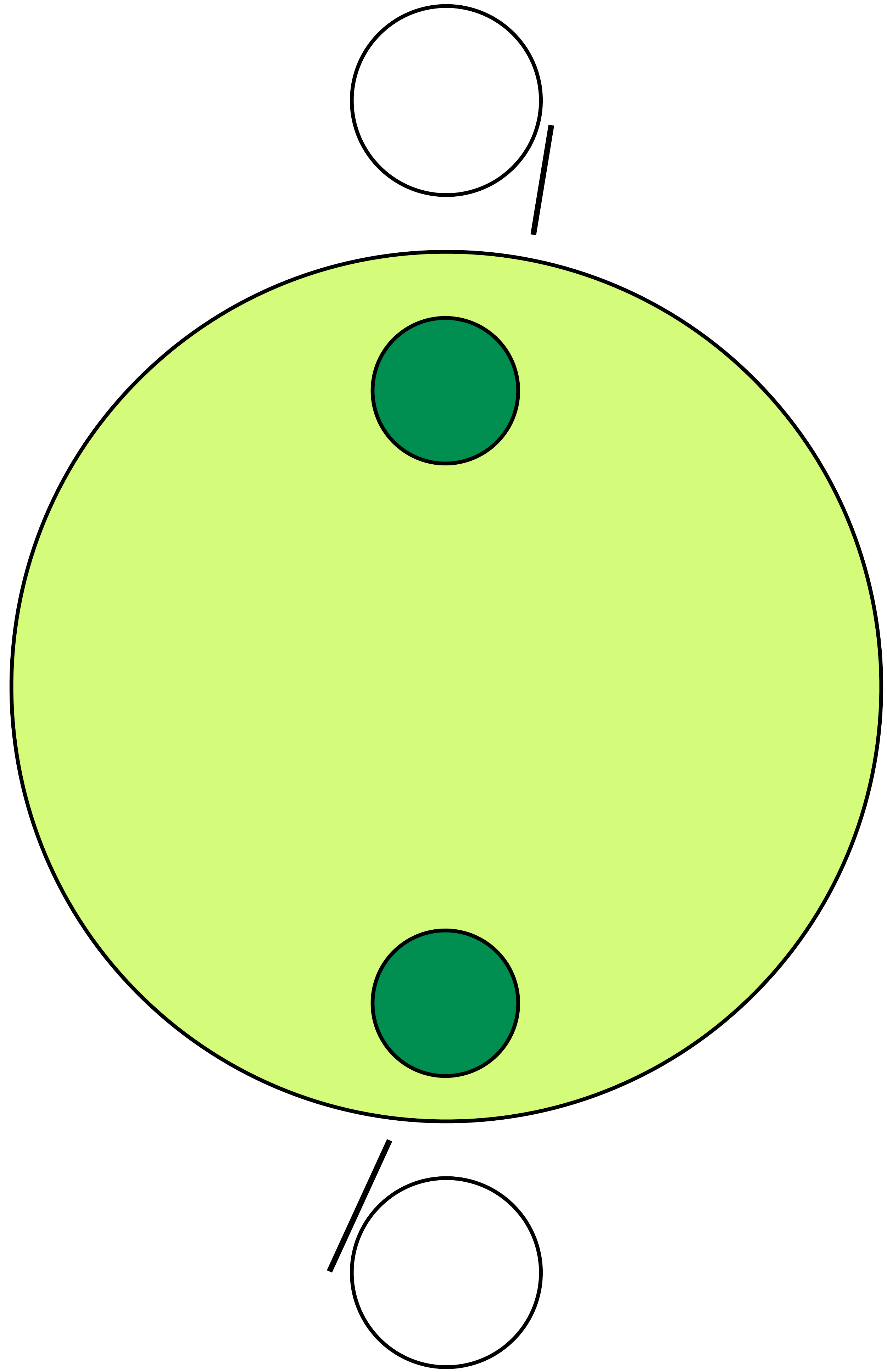
Deadlock Prevention

can we set some rules that prevent deadlock?

mutual exclusion

each resource must be either
assigned to exactly one process
or is available

allow sharing of resources



hold and wait

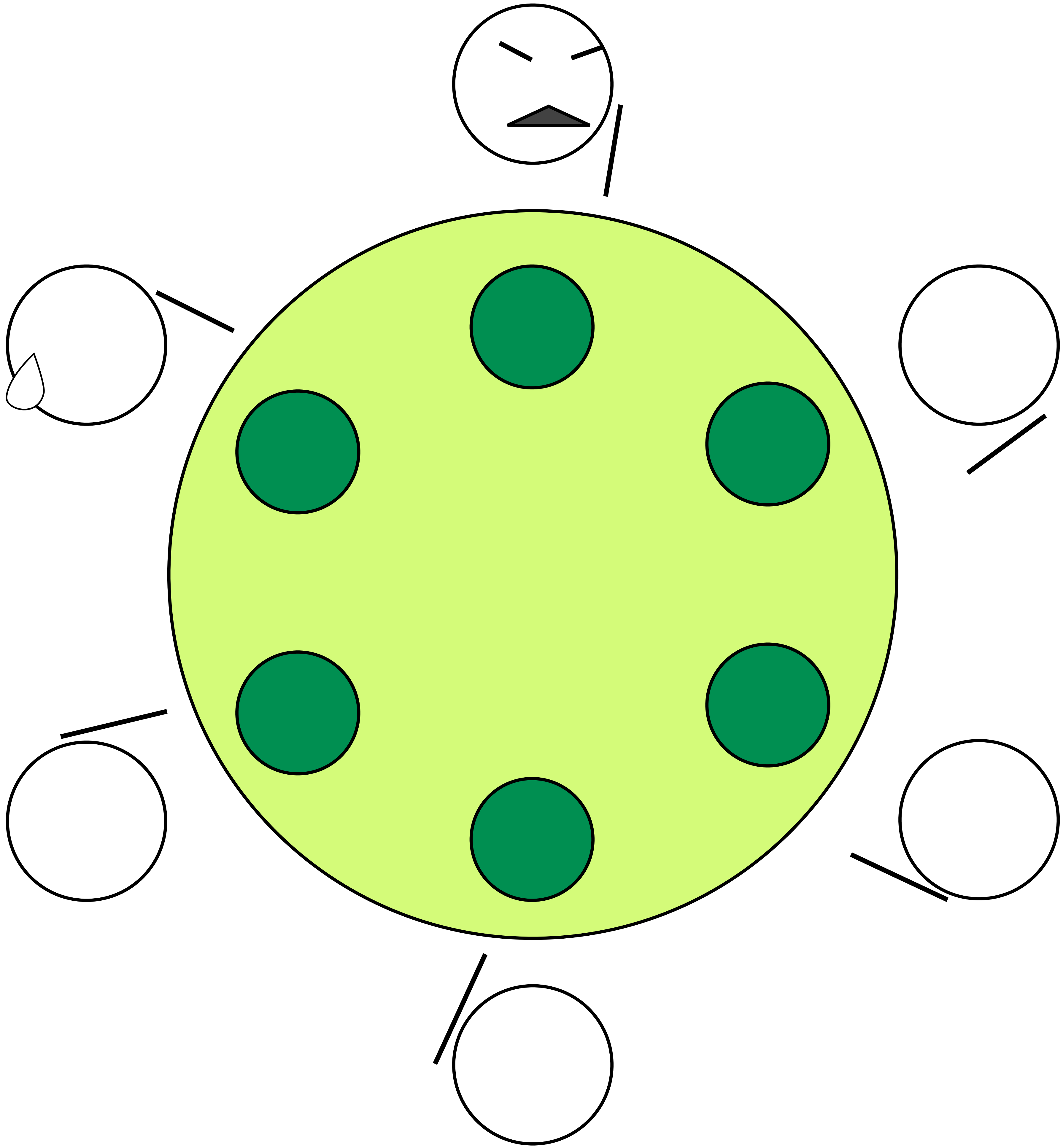
processes holding resources granted earlier can request for new resource allocate only if all resources are available

```
while (1)
    think
    down(mutex)
    state[ i ] = HUNGRY
    test( i )
    up(mutex)
    down(semaphore[ i ])
    eat
    down(mutex)
    state[ i ] = THINK
    test( L )
    test( R )
    up(mutex)
```

no preemption

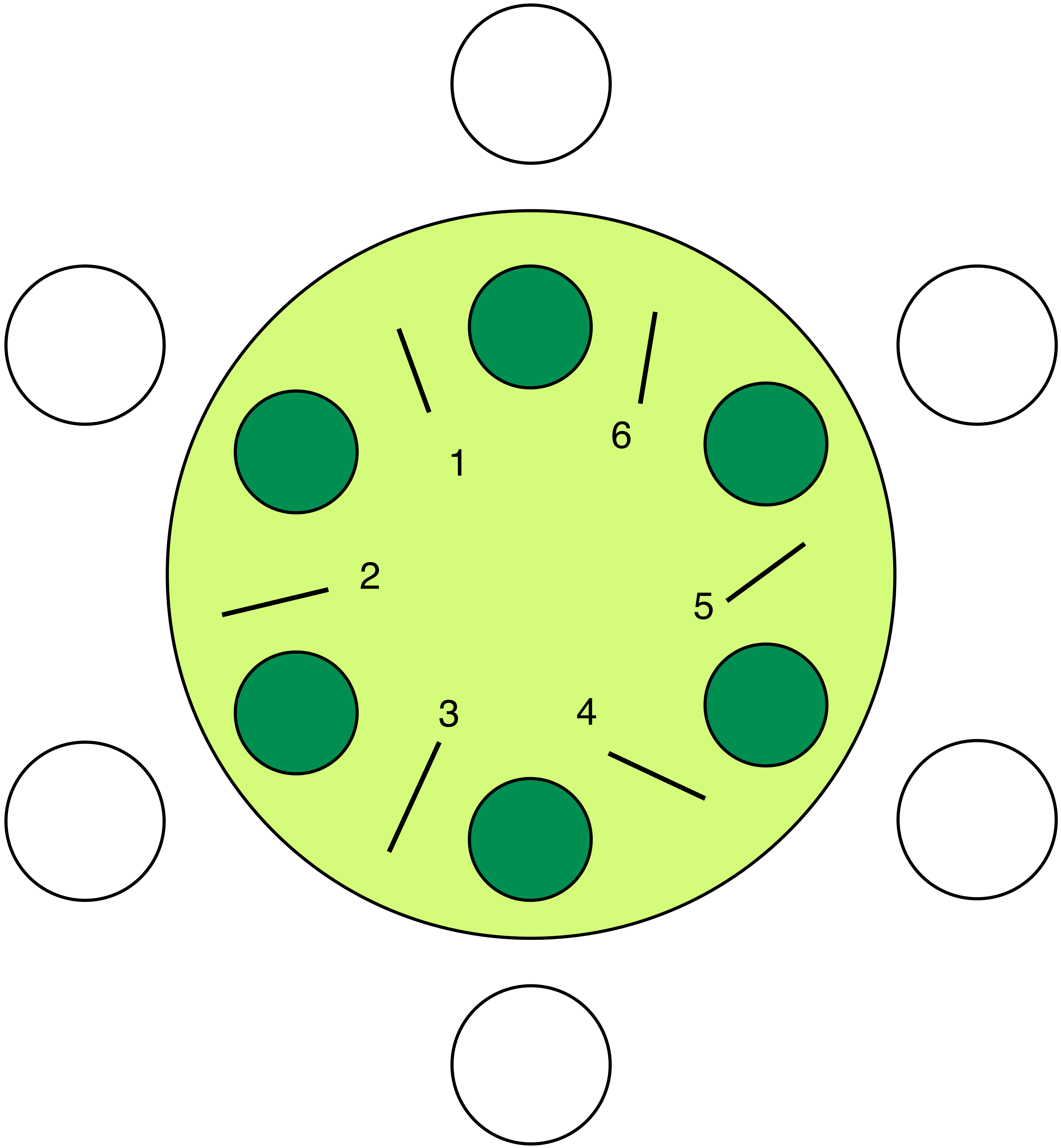
resources granted cannot be
forcefully taken away

allow resources to be preempted



circular waiting

a circular chain of processes, each waiting for a resource held by the next member of the chain order resource numerically and acquire in order



1
6
2
5
3
4

Livelock

Starvation

Process A (low):

down(mutex)

:

work

:

:

up(mutex)

Process B (high):

down(mutex)

:

important tasks

:

:

up(mutex)

Priority Inversion