Lecture 5

Interprocess Communication

9 September, 2011

Interprocess Communication

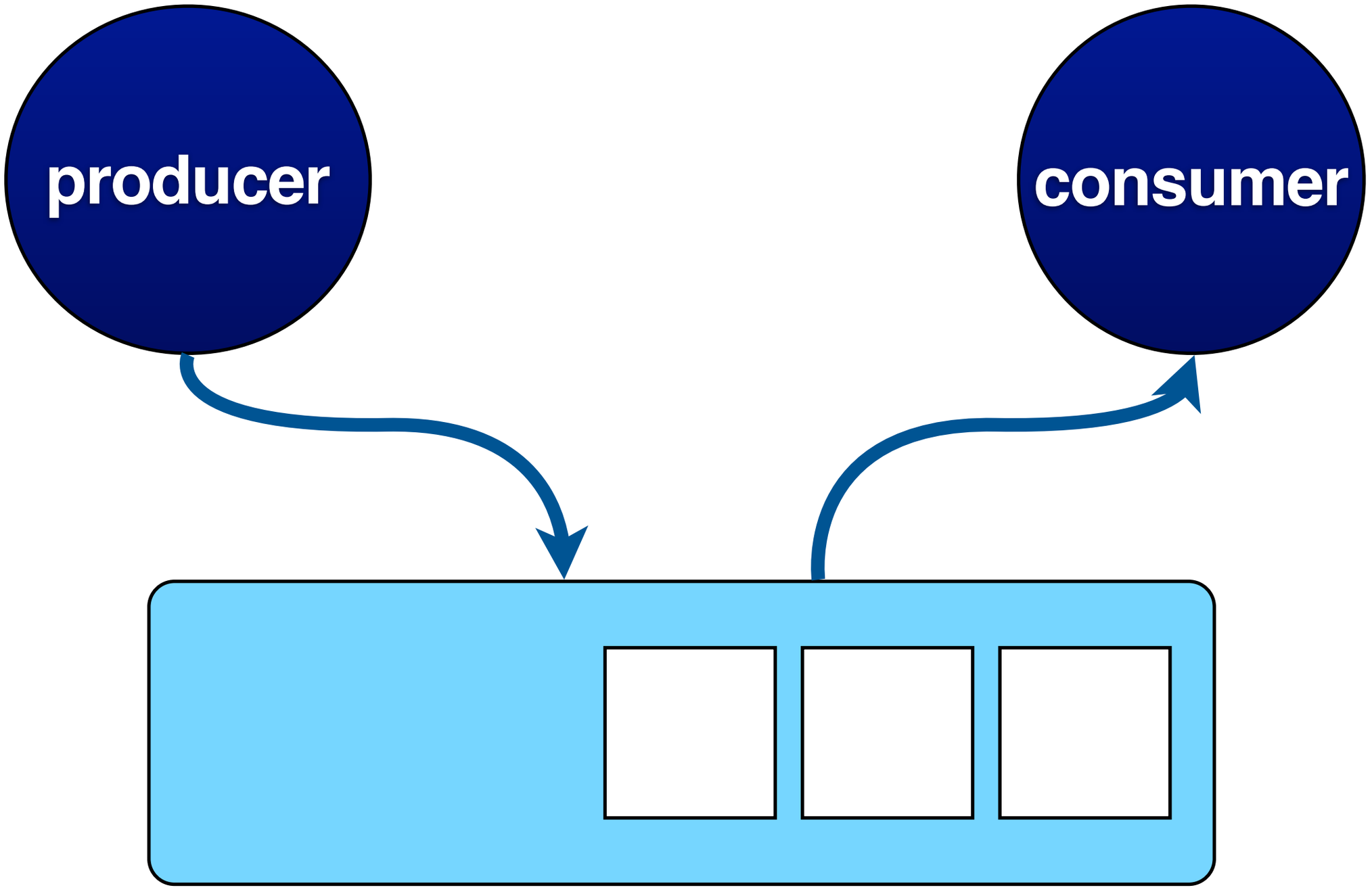
1. mutual exclusion

2. synchronization

the

producer-consumer

problem


producer
consumer

```
while (1)
    if (buffer is full)
        sleep
    if (buffer is empty)
        produce
        wake up consumer
else
        produce
```

```
while (1)
    if (buffer is empty)
        sleep
    if (buffer is full)
        consume
        wake up producer
else
    consume
```

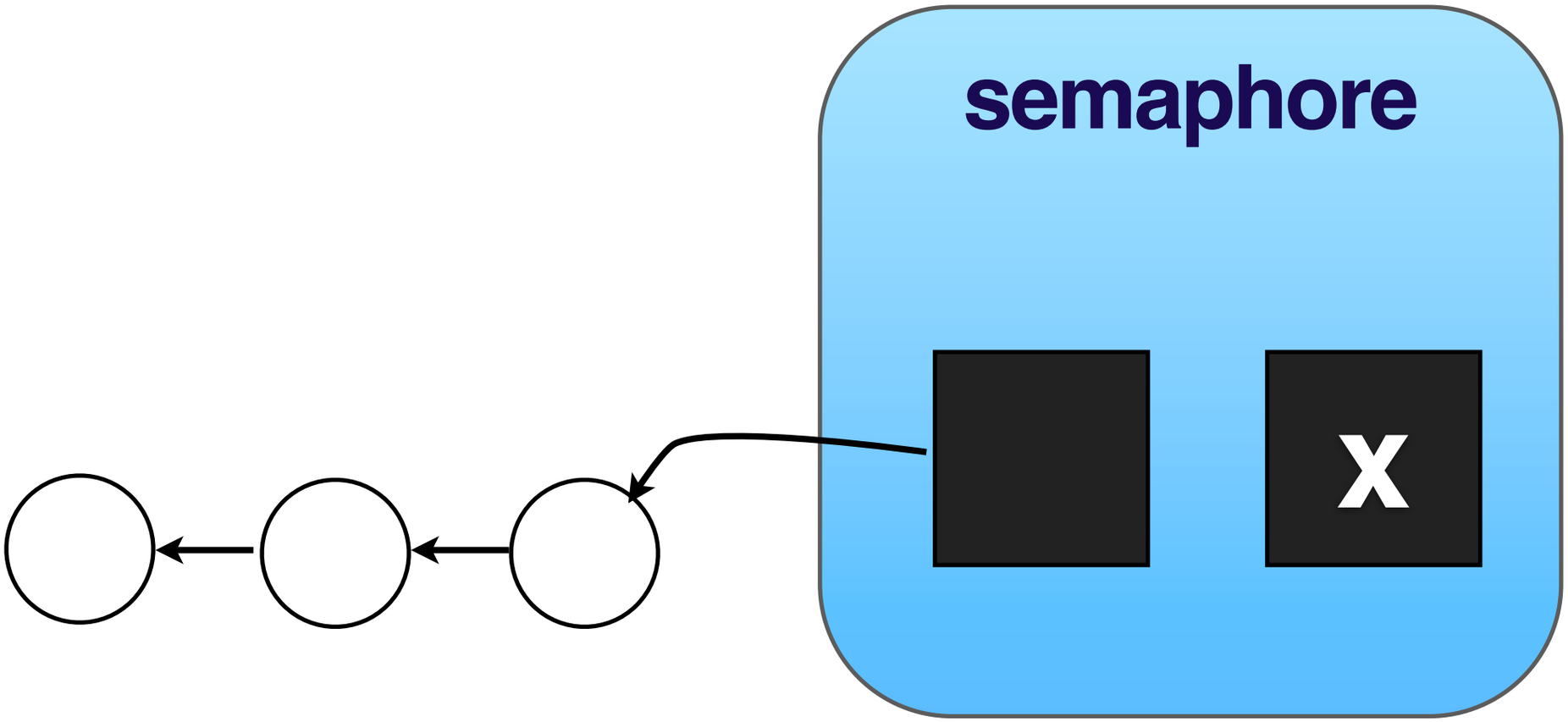
```
while (1)
    if (buffer is full)
        sleep
    if (buffer is empty)
        produce
        wake up consumer
```

```
while (1)
    if (buffer is empty)
        sleep
    if (buffer is full)
        consume
        wake up producer
        .
```

problem:
producer's wake up
call is ignored if
consumer is awake

how to remember “sleep” / “wake” message?

the **semaphore** abstraction



x is an integer

down( )

value = value - 1

if value < 0

sleep (put in wait list)

up( )

value = value + 1

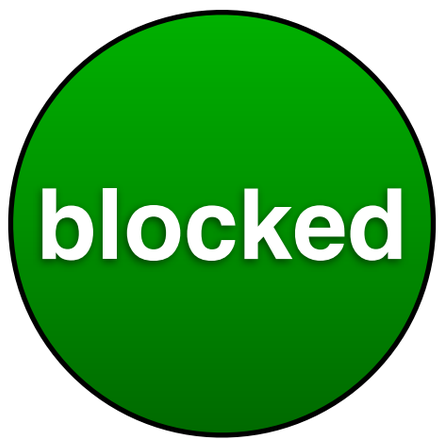
if value <= 0

wake someone


ready


running


new


blocked


exit

up( ) and **down**( ) are **atomic**

can use enter( ) and leave( ) from last lecture to ensure mutual exclusion

operations on semaphore

init(S, i) or S = i

up(S)

down(S)

semaphore in C

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
sem_wait(&s); //down
sem_post(&s); //up
sem_destroy(&s);
```

semaphore S = 0

Process 1

.
.
.
.

down(S)

Process 2

.
.
.
.

up(S)

semaphore S = 1

Process 1

:

down(S)

:

up(S)

:

Process 2

:

down(S)

:

up(S)

:

semaphore free_slots = N
semaphore used_slots = 0

while (1)
down(free_slots)
produce
up(used_slots)

while (1)
down(used_slots)
consume
up(free_slots)

```
semaphore free_slots = N
semaphore used_slots = 0
semaphore mutex = 1

while (1)                          while (1)
down(free_slots)                   down(used_slots)
down(mutex)                        down(mutex)
produce                            consume
up(mutex)                          up(mutex)
up(used_slots)                     up(free_slots)
```

pitfalls of

semaphore

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)

deadlock

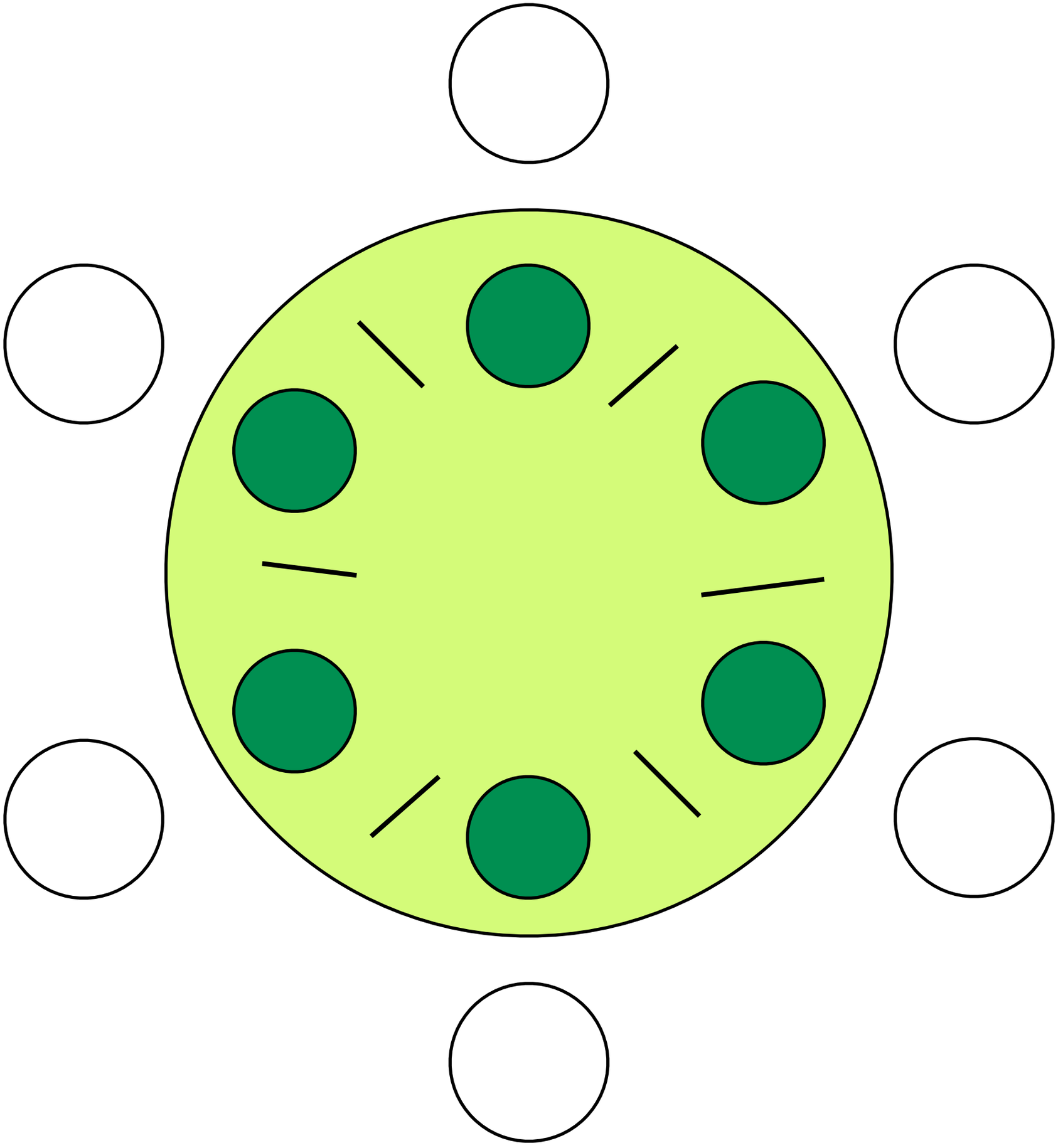
```
semaphore free_slots = N
semaphore used_slots = 0
semaphore mutex = 1

while (1)                          while (1)
down(mutex)                        down(mutex)
down(free_slots)                   down(used_slots)
produce                            consume
up(mutex)                          up(mutex)
up(used_slots)                     up(free_slots)
```

```
while (1)
    down(mutex)
    down(free_slots)
    produce
    up(mutex)
    up(used_slots)
```

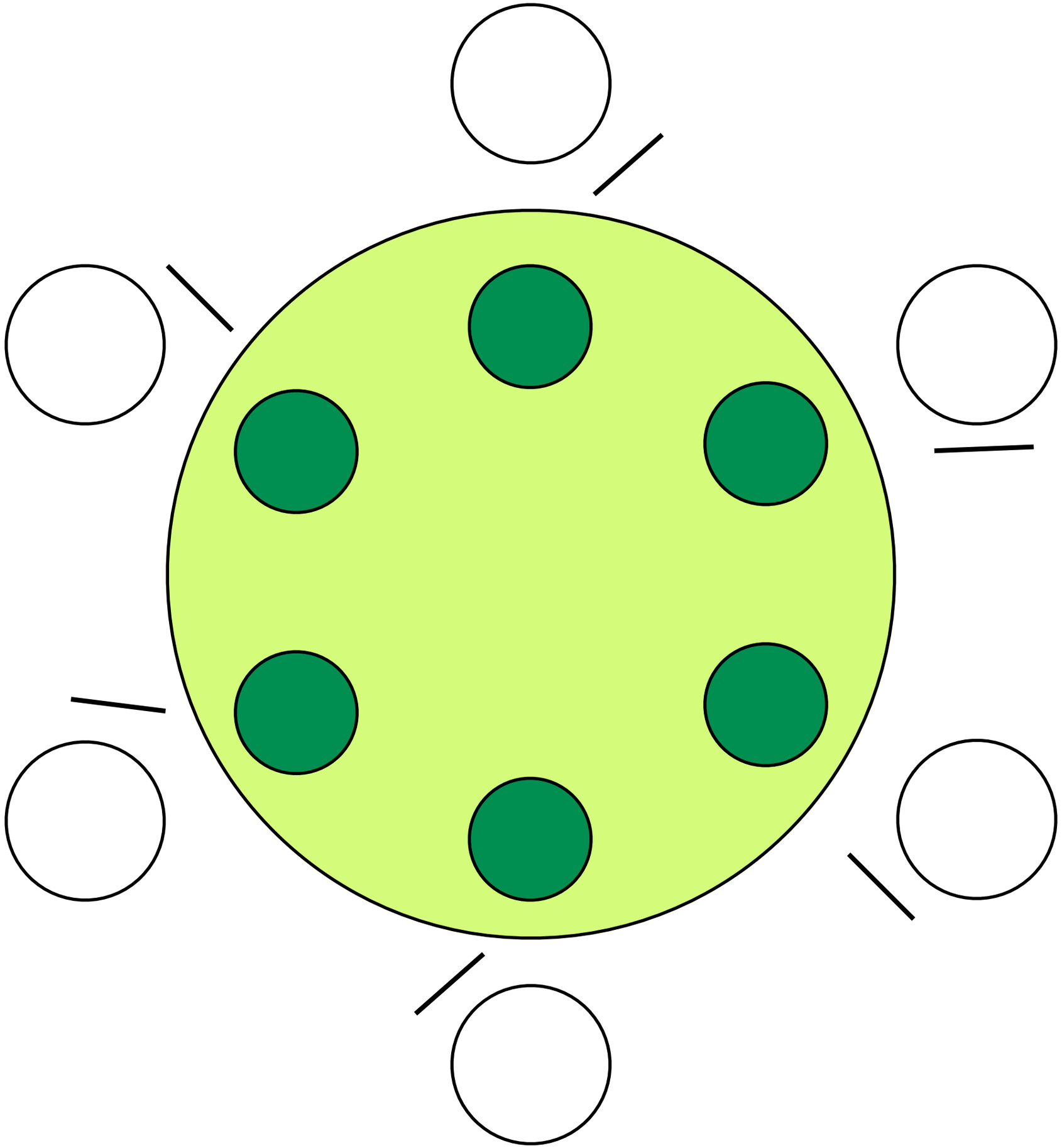
```
while (1)
    down(mutex)
    down(used_slots)
    consume
    up(mutex)
    up(free_slots)
```

the
dining philosophers
problem



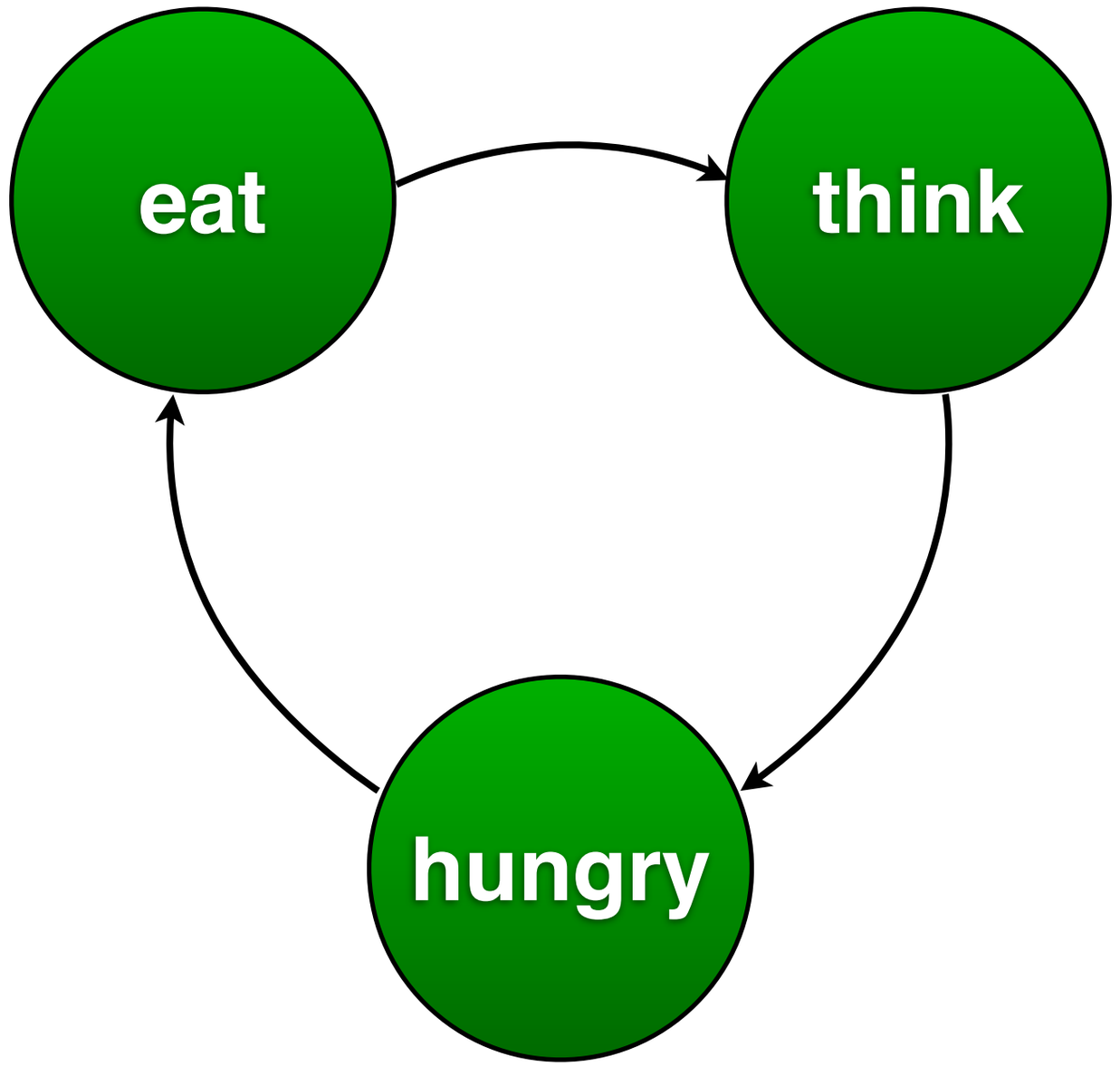
```
while (1)
    think
    pick left chopstick
    pick right chopstick
    eat
    put down left chopstick
    put down right chopstick
```

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



starvation

```
while (1)
    think
    enter( )
        pick left chopstick
        pick right chopstick
        eat
        put down left chopstick
        put down right chopstick
leave( )
```



(may block)

```
while (1)
    think
    if a neighbor is eating
        wait for chopsticks
    eat
    if a neighbor is waiting and is
      ready to eat
        wake up neighbor
```

```
while (1)
    think
    state[ i ] = HUNGRY
    if a neighbor is eating
        wait for chopsticks
    state[ i ] = EAT
    eat
    state[ i ] = THINK
    if a neighbor is waiting
        wake up neighbor
```

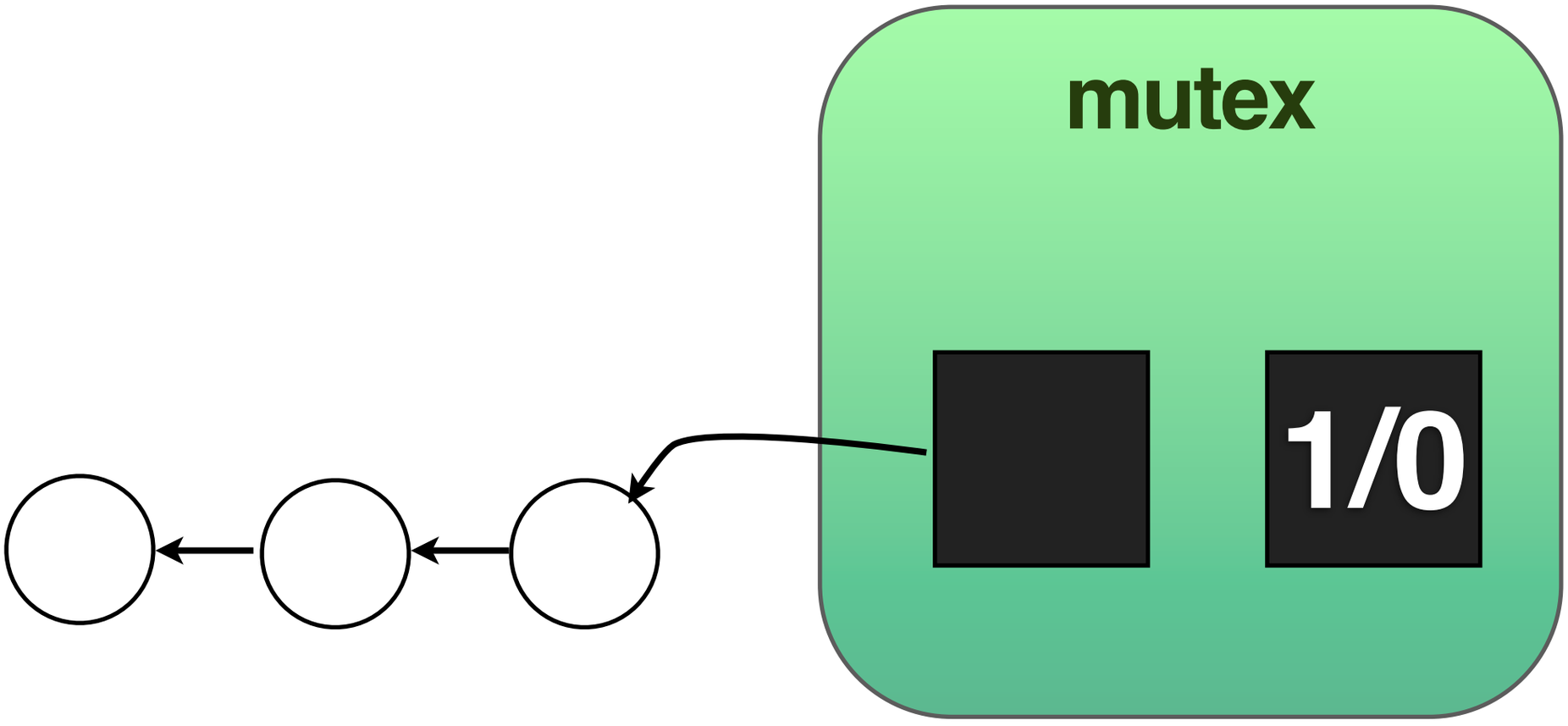
```
while (1)
    think
    state[ i ] = HUNGRY
    if state[ L ] == EAT || state[ R ] == EAT
        down(semaphore[ i ])
    state[ i ] = EAT
    eat
    state[ i ] = THINK
    if state[L] == HUNGRY && state[LL] != EAT
        up(semaphore[ L ])
    if state[R] == HUNGRY && state[RR] != EAT
        up(semaphore[ R ])
```

```
while (1)
    think
    state[ i ] = HUNGRY
    if state[ i ] == HUNGRY && state[ L ] != EAT && state[ R ] != EAT
        up(semaphore[ i ])
        state[ i ] = EAT
    down(semaphore[ i ])
    eat
    state[ i ] = THINK
    if state[L] == HUNGRY && state[LL] != EAT && state[ LR ] != EAT
        up(semaphore[ L ])
        state[ L ] = EAT
    if state[R] == HUNGRY && state[RL] != EAT && state[RR] != EAT
        up(semaphore[ R ])
        state[ L ] = EAT
```

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

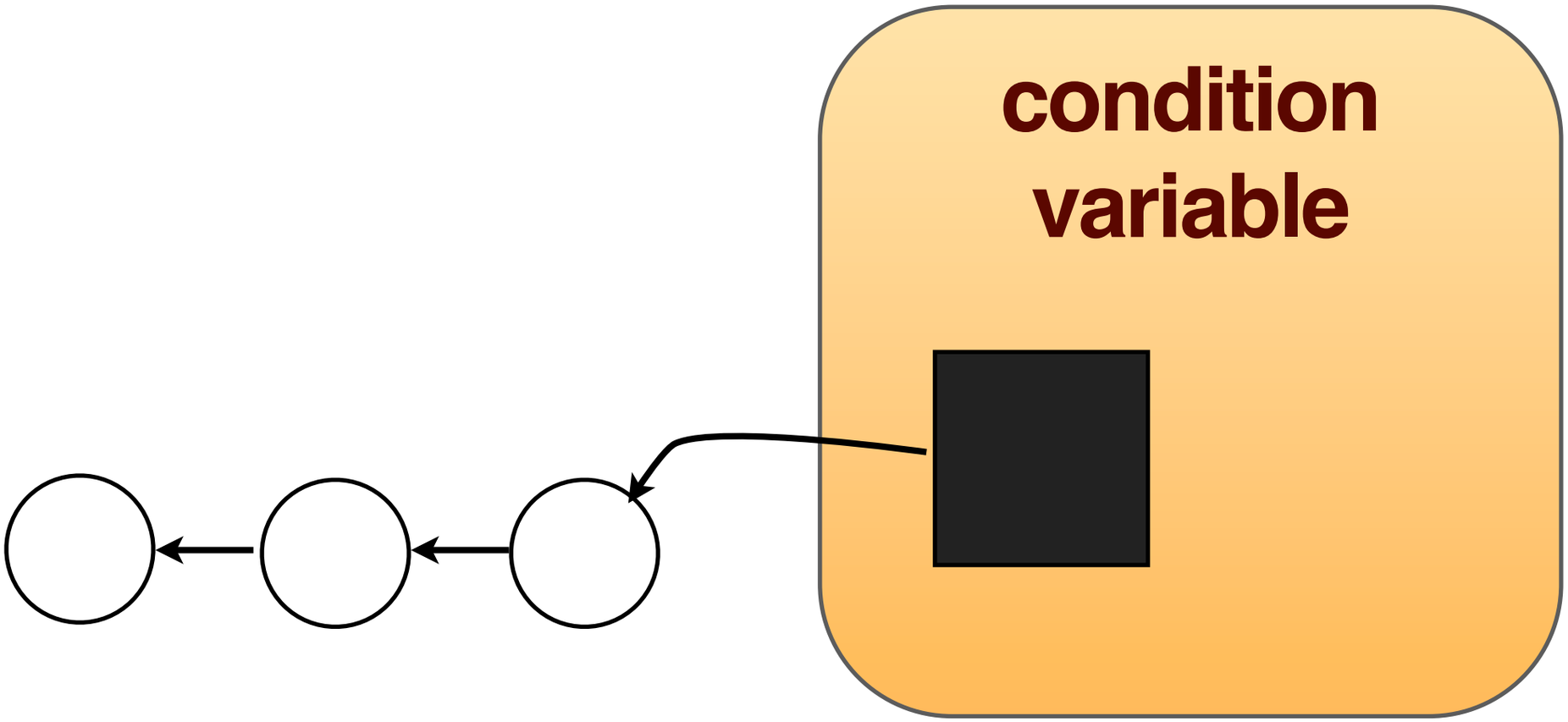
```
while (1)
    think
    down(mutex)
    state[ i ] = HUNGRY
    test( i )
    up(mutex)
    down(semaphore[ i ])
    eat
    down(mutex)
    state[ i ] = THINK
    test( L )
    test( R )
    up(mutex)
```

the **mutex** abstraction



lock / unlock

the
condition variable
abstraction



wait / signal

POSIX threads in C

```
#include <pthread.h>
```

```
gcc a.c -lpthread
```

```
pthread_create(..)
pthread_exit(..)
pthread_join(..)
pthread_yield(..)
```

demo

```
pthread_mutex_init(..)
pthread_mutex_lock(..)
pthread_mutex_unlock(..)
pthread_mutex_trylock(..)
pthread_mutex_destroy(..)
```

```
pthread_cond_init(..)
pthread_cond_wait(..)
pthread_cond_signal(..)
pthread_cond_broadcast(..)
pthread_cond_destroy(..)
```