# INF113: Semaphores

Kirill Simonov

24.10.2025

# Locks and conditional variables

- Lock a.k.a mutex: "blocks" a part of code when in use by a thread
- Conditional variable: can either wait for a specific condition or signal the condition
  - Always used together with a lock
- **Next:** A simpler-to-use primitive that can model both*

```
pthread_mutex_t mutex;

void *mythread(void *arg) {
  ...
  pthread_mutex_lock(&mutex);
  counter = counter + 1;
  pthread_mutex_unlock(&mutex);
  ...
}
```

mutex

```
void thr_exit() {
  pthread_mutex_lock(&m);
  done = 1;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}

void thr_join() {
  pthread_mutex_lock(&m);
  while (done == 0)
    pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}
```

conditional variable

# Semaphore: Interface

- Semaphore holds an integer **value**

```
#include <semaphore.h>
sem_t s;
```

- Has to be initialized, the value is then set to the argument

```
sem_init(&s, 0, value);
```

- `sem_wait` decrements value and waits if negative

```
int sem_wait(sem_t *s) {
  decrement the value of semaphore s
  if value of semaphore s is negative, wait
}
```

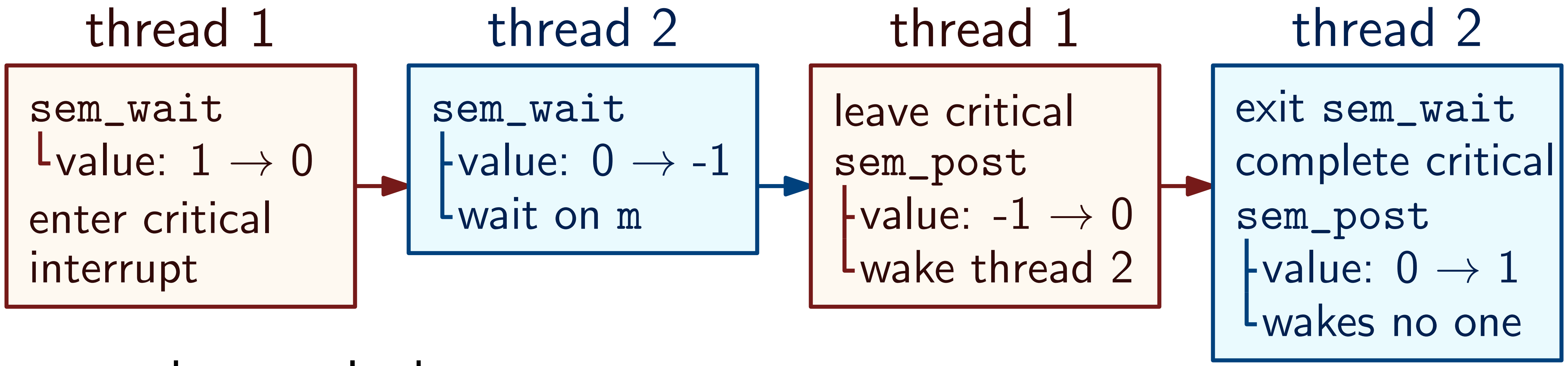
- `sem_post` increments value and sends a signal

```
int sem_post(sem_t *s) {
  increment the value of semaphore s
  if there are threads waiting, wake one
}
```

assume these are done atomically

# Semaphore as a lock

- Simply wrap the critical section in `sem_wait` and `sem_post`




- Binary semaphore = lock

```
sem_t m;
sem_init(&m, 0, 1);
...
sem_wait(&m);
// critical section here
sem_post(&m);
```

```
int sem_wait(sem_t *s) {
  decrement value
  if negative, wait
}

int sem_post(sem_t *s) {
  increment value
  wake a waiting thread
}
```

reminder

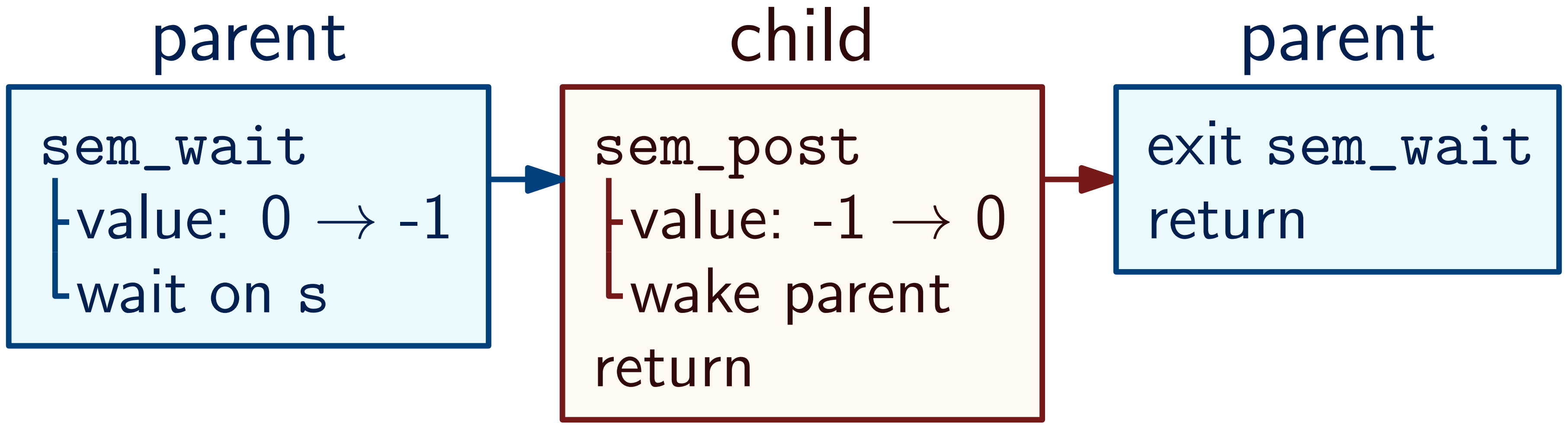
# Semaphores for ordering

- Example: Parent thread waits for the child thread to finish

```
sem_t s;

void *child(void *arg) {
  sem_post(&s);
  return NULL;
}

int main() {
  sem_init(&s, 0, 0);
  pthread_t c;
  pthread_create(&c, NULL,
    child, NULL);
  sem_wait(&s);
  return 0;
}
```




```
int sem_wait(sem_t *s) {
  decrement value
  if negative, wait
}

int sem_post(sem_t *s) {
  increment value
  wake a waiting thread
}
```

reminder

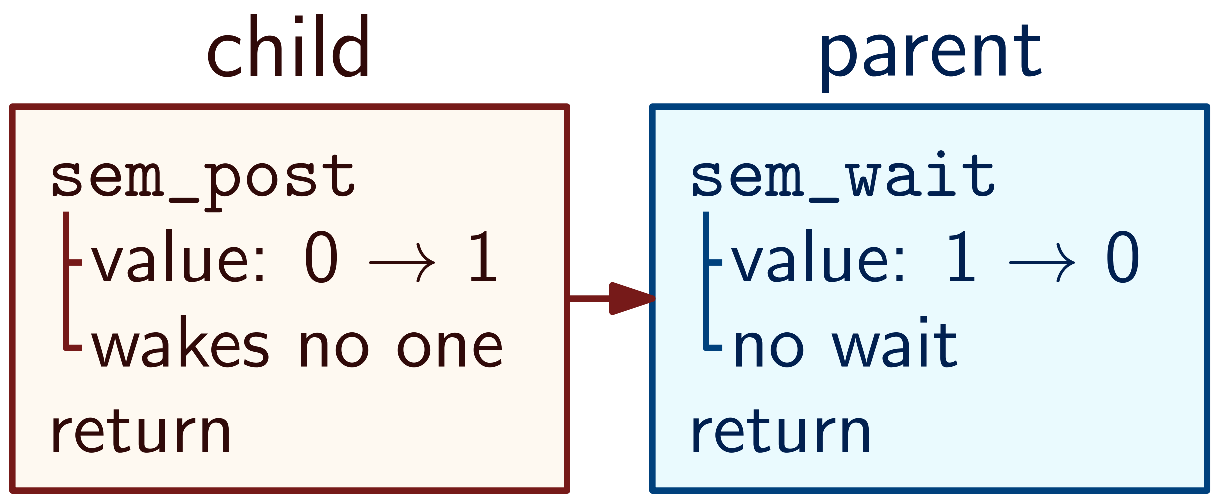
# Semaphores for ordering

- Example: Parent thread waits for the child thread to finish

```c
sem_t s;

void *child(void *arg) {
  sem_post(&s);
  return NULL;
}

int main() {
  sem_init(&s, 0, 0);
  pthread_t c;
  pthread_create(&c, NULL,
    child, NULL);
  sem_wait(&s);
  return 0;
}
```



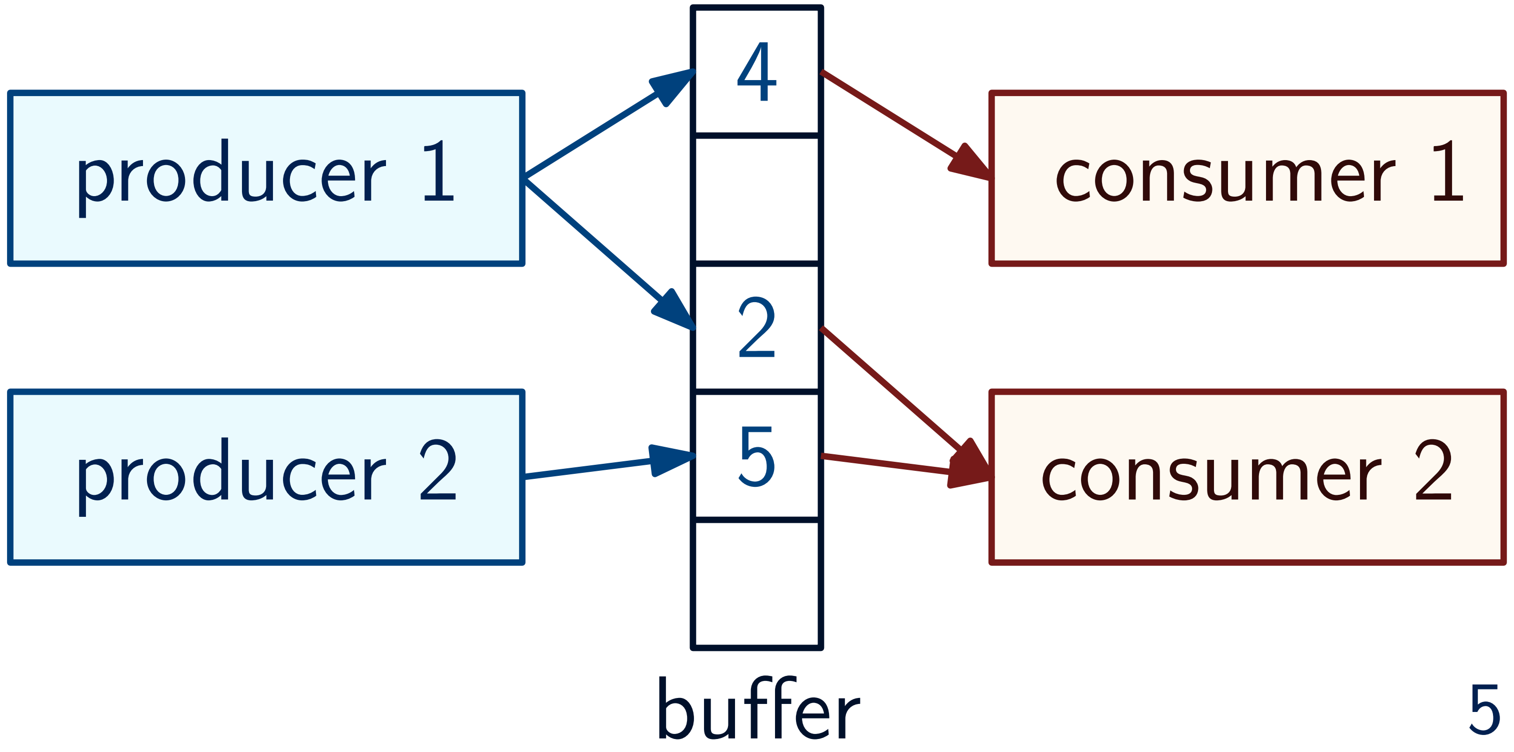

```c
int sem_wait(sem_t *s) {
  decrement value
  if negative, wait
}

int sem_post(sem_t *s) {
  increment value
  wake a waiting thread
}
```

reminder

# Reminder: producer/consumer

- **Producer** threads generate data items and place them in the buffer
- **Consumers** pick the items from the buffer and process them
- Examples:
  - Multi-threaded web server
  - `grep foo file.txt | wc -l`
- Now with semaphores!



# Producer/consumer: Template

- Need to add waiting/signaling upon put/get

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    put(i);
  }
}

void *consumer(void *arg) {
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

wait on empty
signal full

wait on full
signal empty

template producer and consumer

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;

void put(int value) {
  buffer[fill_ptr] = value;
  fill_ptr = (fill_ptr + 1) % MAX;
}

int get() {
  int tmp = buffer[use_ptr];
  use_ptr = (use_ptr + 1) % MAX;
  return tmp;
}
```

helper functions

# Two semaphores

- Add one semaphore for each of the two conditions:
- Initially, `MAX` "empty" resources, and 0 "full" resources
- Every producer cycle moves one from "empty" to "full"

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);     empty--
    put(i);
    sem_post(&full);      full++
  }
}
```

```
void *consumer(void *arg) {
  while (1) {
    sem_wait(&full);   full--
    int tmp = get();
    sem_post(&empty);  empty++
    printf("%d\n", tmp);
  }
}

int main() {
  ...
  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);
  ...
}
```

# Two semaphores: Example 1

- One producer, one consumer, `MAX = 1`

**consumer**

```
sem_wait full
├value: 0 → -1
└wait on full
```

→

**producer**

```
sem_wait empty
└value: 1 → 0
put(0)
sem_post full
├value: -1 → 0
└wake full
```

→

**producer**

```
sem_wait empty
├value: 0 → -1
└wait on empty
```

→

**consumer**

```
exit sem_wait
get()
sem_post empty
├value: -1 → 0
└wake empty
```

```c
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);
    put(i);
    sem_post(&full);
  }
}
```

```c
void *consumer(void *arg) {
  while (1) {
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

# Two semaphores: Example 2

- Two producers, one consumer, `MAX = 1`

**consumer**
sem_wait full
value: 0 → -1
wait on full

→ **producer 1**
sem_wait empty
value: 1 → 0
put(0)
interrupt

→ **producer 2**
sem_wait empty
value: 0 → -1
wait on empty

→ **producer 1**
sem_post full
value: -1 → 0
wake full

→ **consumer**
exit sem_wait
get()
sem_post empty
value: -1 → 0
wake empty

- Producers may run in parallel, but only one gets through `sem_wait` since the value is one

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);
    put(i);
    sem_post(&full);
  }
}
```

```
void *consumer(void *arg) {
  while (1) {
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

# Two semaphores: Example 3

- Two producers, one consumer, `MAX = 2`

| consumer | producer 1 | producer 2 |
|---|---|---|
| sem_wait full<br>value: 0 → -1<br>wait on full | sem_wait empty<br>value: 2 → 1<br>put(0)<br>interrupt | sem_wait empty<br>value: 1 → 0<br>put(0)<br>... |

possible interleaving

```
p1: buffer[0] = 0;
p2: buffer[0] = 0;
p1: fill_ptr = 1;
p2: fill_ptr = 1;
```

one data item lost

- The producers both enter put without a lock!

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);
    put(i);
    sem_post(&full);
  }
}
```

```
void *consumer(void *arg) {
  while (1) {
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

# Two semaphores and a mutex

- Let us add a lock—another binary semaphore—to guard the critical section
- Only one thread at a time can access the buffer
- But now we can have a deadlock

consumer

```
sem_wait mutex
 value: 1 → 0
sem_wait full
 value: 0 → -1
 wait on full
```

producer

```
sem_wait mutex
 value: 0 → -1
 wait on mutex
```

```c
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(i);
    sem_post(&full);
    sem_post(&mutex);
  }
}
```

```c
void *consumer(void *arg) {
  while (1) {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    printf("%d\n", tmp);
  }
}
```

# Full solution

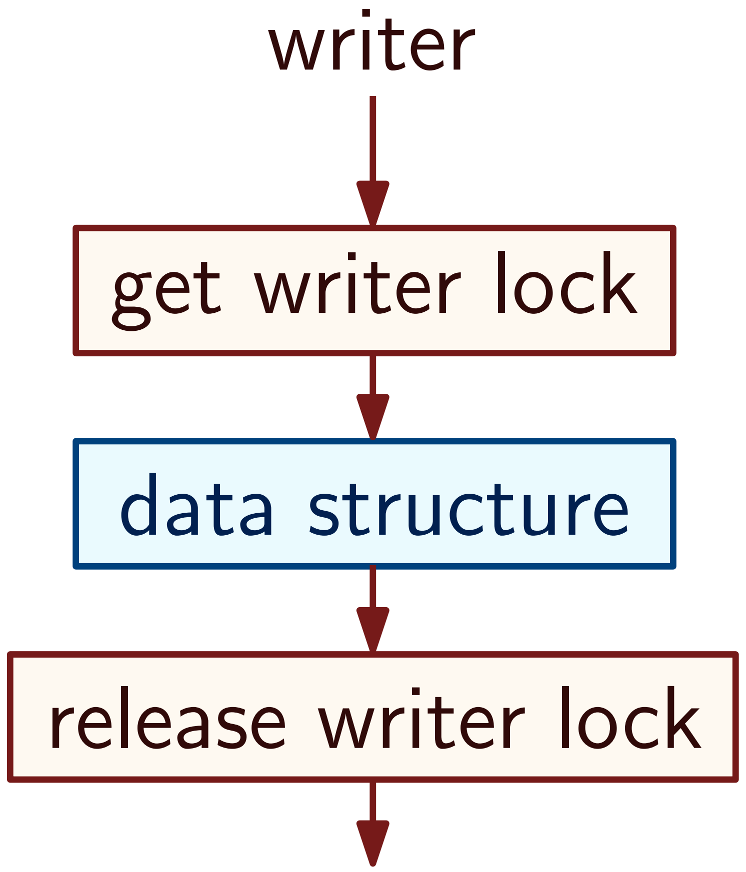
- Put lock on the critical section only
- Now, a thread cannot go into sleep while holding `mutex`
- Finally, a working solution for producer/consumer with semaphores only

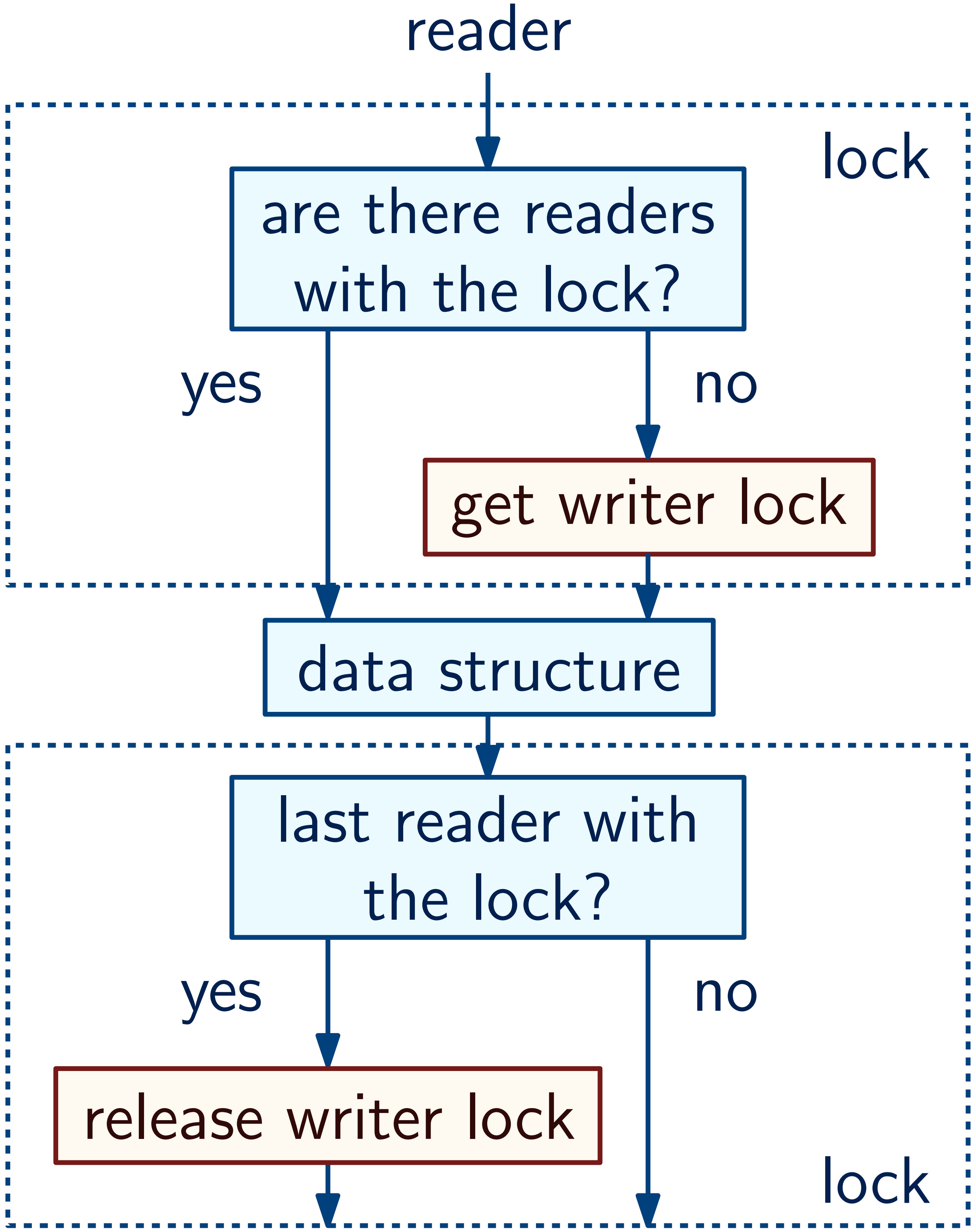
```c
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);
    sem_wait(&mutex);
    put(i);
    sem_post(&mutex);
    sem_post(&full);
  }
}
```

```c
void *consumer(void *arg) {
  while (1) {
    sem_wait(&full);
    sem_wait(&mutex);
    int tmp = get();
    sem_post(&mutex);
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

# Reader-writer lock

- Assume a data structure that many threads read from, but only a few write to
- Reader threads can go in simultaneously
- Writer thread must block all else



# Reader-writer lock: Implementation

- Keep two locks and a counter

```
void init() {
  readers = 0;
  sem_init(&lock, 0, 1);
  sem_init(&writelock, 0, 1);
}
```

writer

```
void acquire_writelock() {
    sem_wait(&writelock);
}

void release_writelock() {
    sem_post(&writelock);
}
```

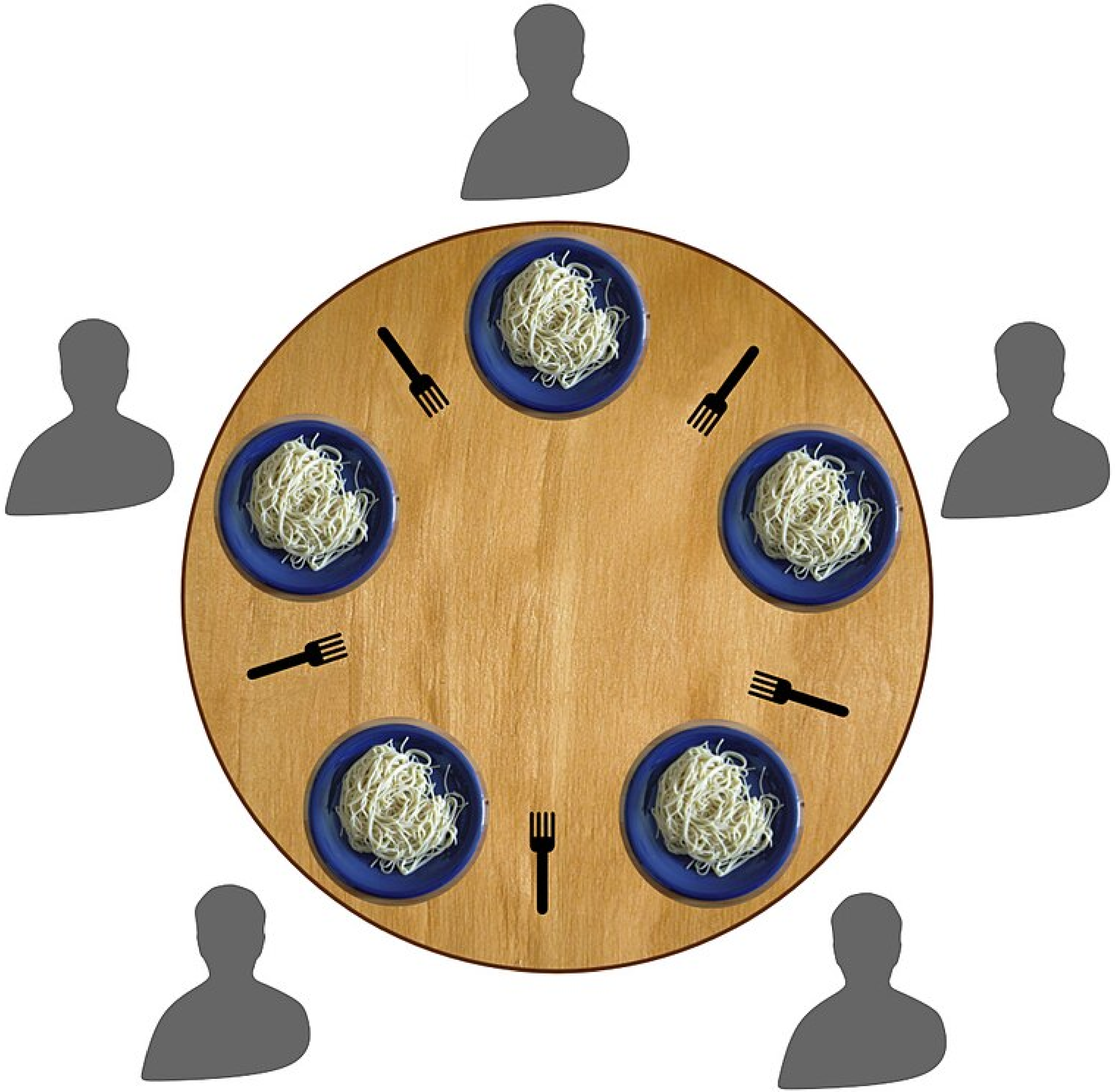
reader

```
void acquire_readlock() {
    sem_wait(&lock);
    readers++;
    if (readers == 1)
      sem_wait(&writelock);
    sem_post(&lock);
}

void release_readlock() {
    sem_wait(&lock);
    readers--;
    if (readers == 0)
      sem_post(&writelock);
    sem_post(&lock);
}
```

# Dining philosophers

- Five philosophers seat around the table
- Each has a plate, and there is a fork between each pair of plates
- Philosophers alternate between thinking and eating
- A philosopher can only eat if both adjacent forks are free
- **Task:** Design an algorithm so that no philosopher starves



cc wiki

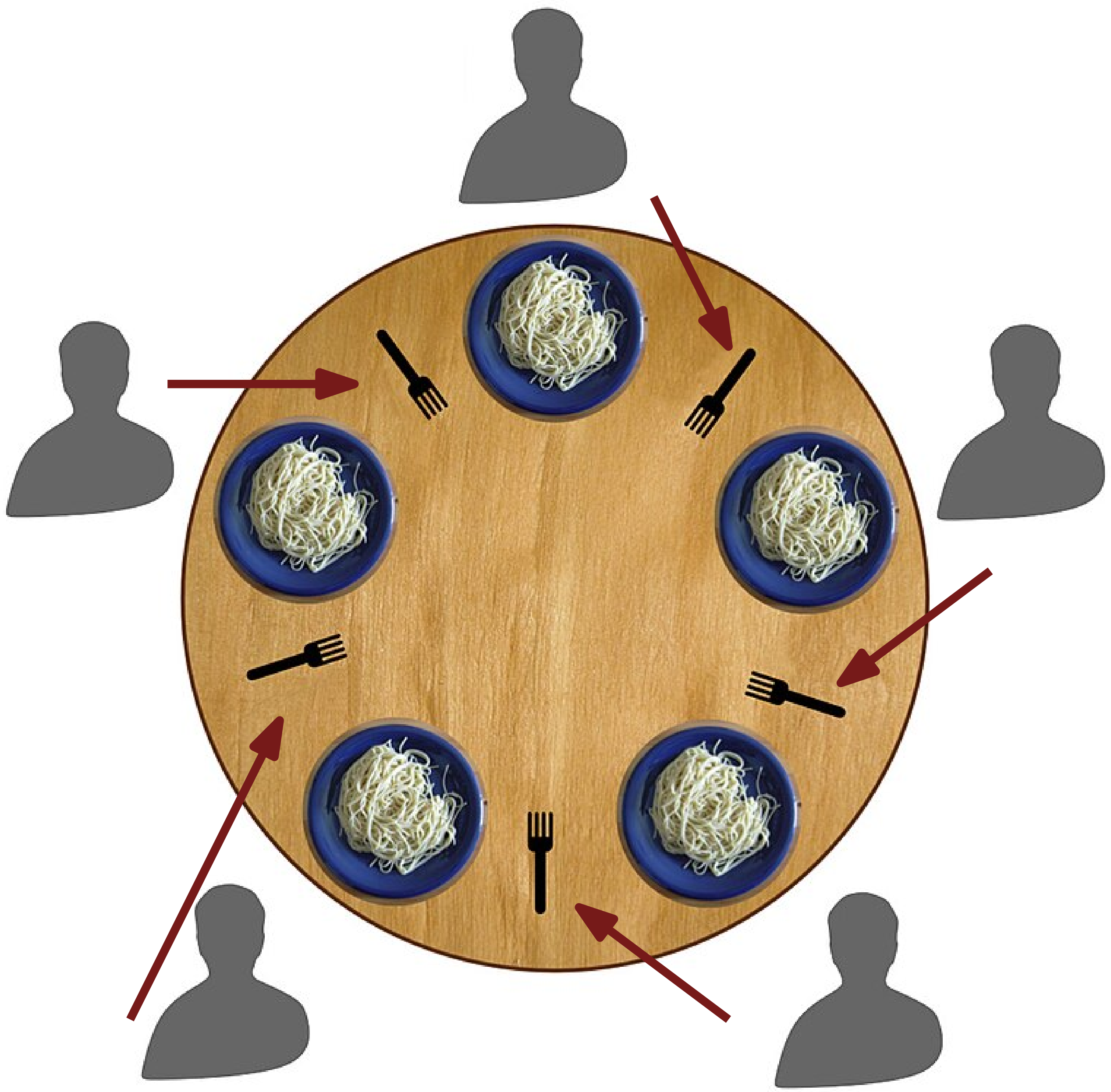
# Dining philosophers: Attempt

- Eat whenever both forks are available

```
while (1) {
  think();
  get_forks(p);
  eat();
  put_forks(p);
}
```

```
void get_forks(int p) {
  sem_wait(&forks[left(p)]);
  sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
  sem_post(&forks[left(p)]);
  sem_post(&forks[right(p)]);
}
```



cc wiki

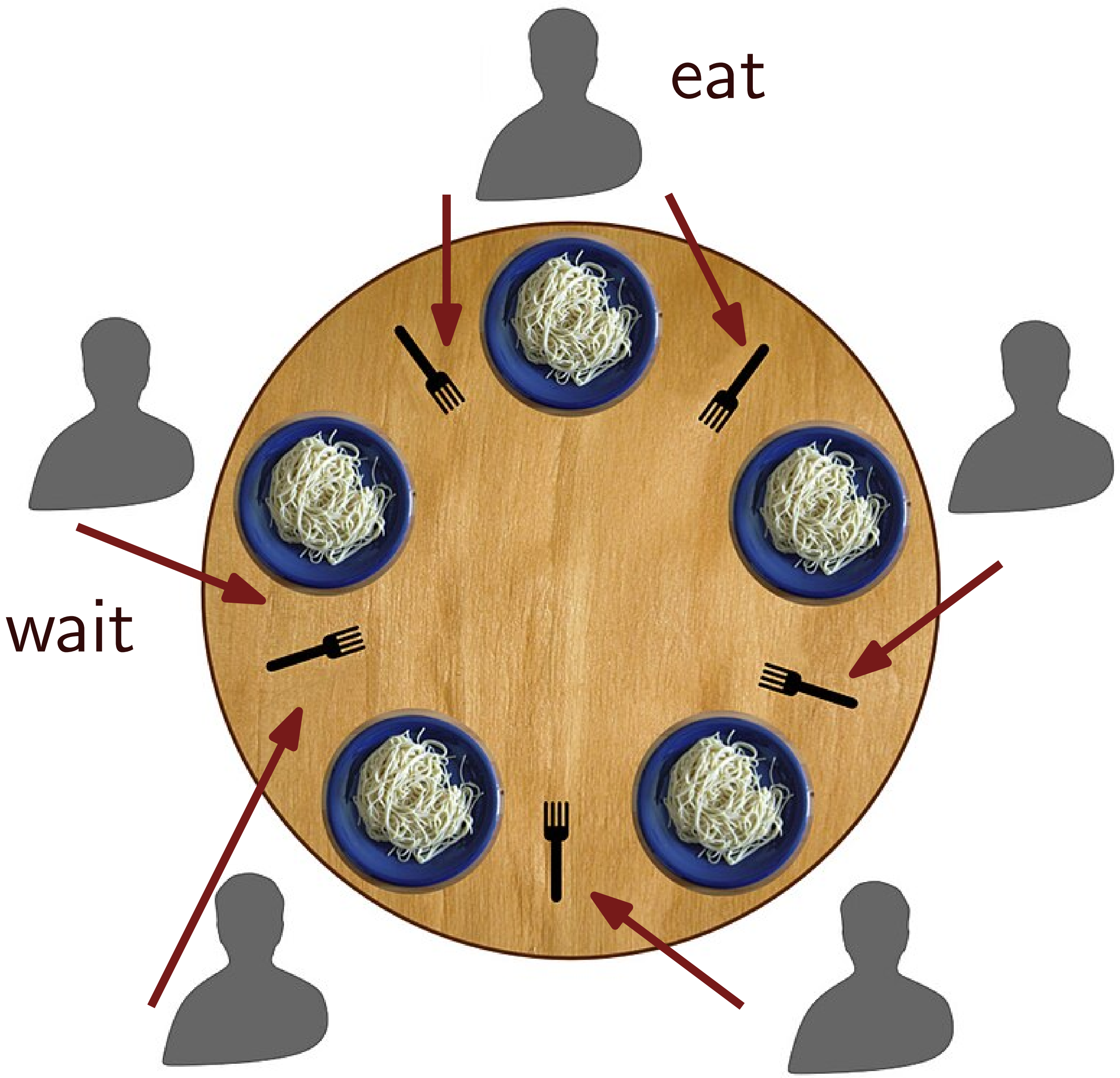
deadlock—each has one fork, no one lets go

# Dining philosophers: Solution

- We need to break the symmetry

```
void get_forks(int p) {
  if (p == 4) {
    sem_wait(&forks[right(p)]);
    sem_wait(&forks[left(p)]);
  } else {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
  }
}

void put_forks(int p) {
  sem_post(&forks[left(p)]);
  sem_post(&forks[right(p)]);
}
```




cc wiki

# Throttling

- We might want to limit access even without a race condition
- Example: memory-intensive section
  - 100 threads allocate 1GB each—out of memory, everything goes to swap
  - Only 5 threads allowed at a time—smooth performance
- Solvable by semaphores:

```
sem_t s;
sem_init(&s, 0, 5);
...
sem_wait(&m);
// at most 5 threads at a time
sem_post(&m);
```

# Implementing semaphores

- Store value, a lock and a conditional variable:

```
int value;
pthread_cond_t cond;
pthread_mutex_t lock;

init(int v) {
  value = v;
  pthread_cond_init(&cond);
  pthread_mutex_init(&lock);
}
```

```
void wait() {
  pthread_mutex_lock(&lock);
  while (value <= 0)
    pthread_cond_wait(&cond, &lock);
  s->value--;
  pthread_mutex_unlock(&lock);
}
```

```
void post() {
  pthread_mutex_lock(&lock);
  value++;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&lock);
}
```

# Summary

- Often, semaphore is the only primitive you need to use
  - Implements both “locking” and “ordering” mechanics
  - But, it is challenging to derive conditional variables in full generality
- Locks and conditional variables enable semaphore’s implementation
- Sometimes more general synchronization primitives are more costly
  - A general semaphore vs a lock
  - A read-write lock vs a lock
- See Chapter 31 for more details, and Chapters 32–33 for extra background
- Next two weeks: Storage