Department of Computer Science
University of Bristol

# COMS20001 - Concurrent Computing

www.cs.bris.ac.uk/Teaching/Resources/COMS20001

Lecture 14

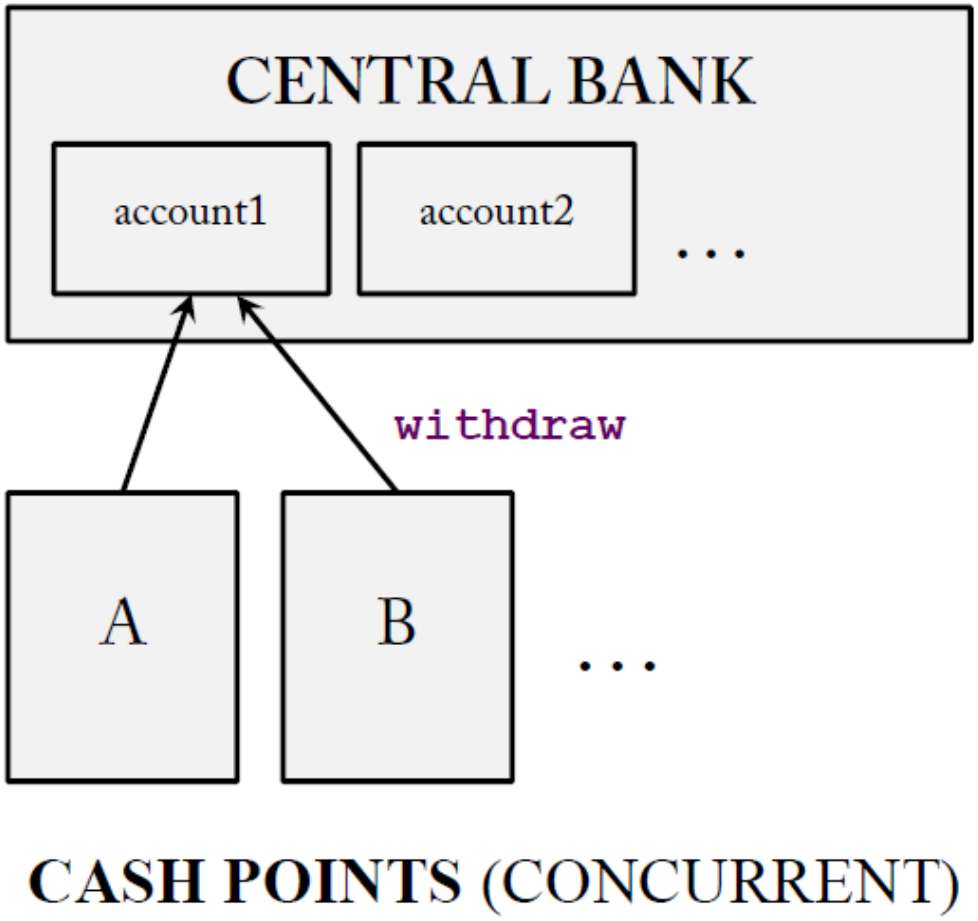
## Sharing Memory, Locks Critical Sections, condition variables and semaphores

Sion Hannuna | hannuna@cs.bris.ac.uk
Tilo Burghardt | tilo@cs.bris.ac.uk
Dan Page | daniel.page@bristol.ac.uk

# Race Conditions: Bank Example (in C)

```
...
// shared global memory
// at bank
int account1 = 99;
int account2 = ...
...
```

```
...
// code fragment for
// withdrawing cash
...
int withdraw1( int amount ) {
  if (amount <= account1) {
    account1 -= amount;
    return 1;
  } else return 0;
}
...
```



# Race Conditions: Concurrent Withdrawels 2

```
// shared global memory held at bank
int account1 = 99;
```

```
// started from CASH POINT A
 withdraw1(20);
```

```
// started from CASH POINT B
 withdraw1(90);
```

```
...
// THREAD AT CASH POINT A

  if (amount <= account1) {
```

```
    account1 -= amount;
```

```
    return 1;
  } else return 0; ...
```

```
...
// THREAD AT CASH POINT B
```

```
  if (amount <= account1) {
```

```
    account1 -= amount;
    return 1;
  } else return 0;
```

```
...
```


£110 withdrawn

# Race Conditions: Critical Section

**Critical Sections are…**

code fragments that interact with a shared resource and should not be accessed by more than one thread at any one time.

```
...
// shared global memory held at bank
int account1 = 99;
int account2 = ...
...
// code fragment for withdrawing cash
...
int withdraw1( int amount ) {
  if (amount <= account1) {
    account1 -= amount;
    return 1;
  } else return 0;
}
...
```

# Shared memory – key concepts

- Critical sections
  - Where shared resources could be accessed by multiple threads simultaneously
- Mutex locks
  - Used to protect critical sections of code - specifically to prevent a **race condition**
- Condition variables
  - Used to put threads to sleep and wake them up again – help to avoid **busy waiting**
  - Require an associated **predicate**
- Semaphores
  - Can be used to perform different tasks depending on their initial value
    - For example, a **binary semaphore** can do the same job as a mutex lock

# Semaphores

A semaphore has a value ≥ 0. Operations:

**post** – increase the value by 1 (atomically).

**wait** – if > 0, decrease by 1 (atomically), otherwise wait until this becomes possible.

Semaphores can be used for different purposes depending on their initial value.

# Mutual exclusion

Semaphore with initial value 1: mutual exclusion.

Wait before entering a critical section; post when you exit it.

This way only one thread can be in the section at any one time.

Warning: there is no “double post” protection.

# Code for semaphore mutexes

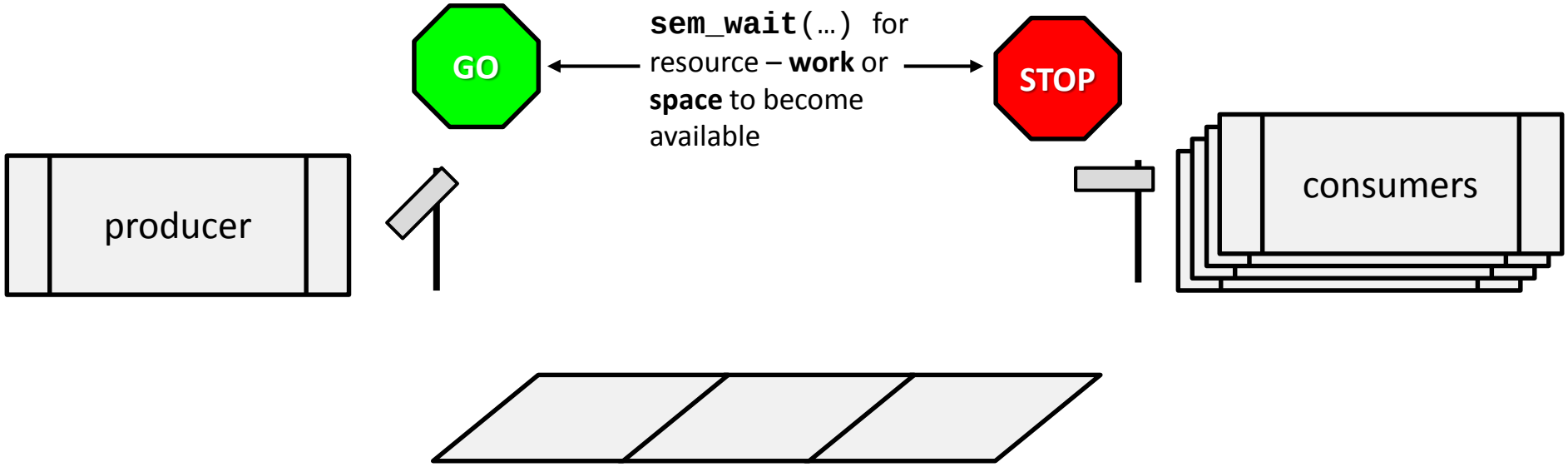
## Semaphores

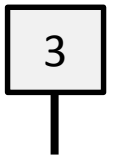
```
//When you set up your semaphore –
probably global and set up in main()
err = sem_init(&s_data_lock, 0, 1)
if (err) { // deal with it }

…

err = sem_wait(&s_data_lock);
if (err) { // deal with it }

// Critical section

err = sem_post(&s_data_lock);
if (err) { // deal with it }
```

## Mutexes

```
//When you set up your mutex –
probably global
pthread_mutex_t lock =
PTHREAD_MUTEX_INITIALIZER;

…

err = pthread_mutex_lock(&lock);
if (err) { // deal with it }

// Critical section

err = pthread_mutex_unlock(&lock);
if (err) { // deal with it }
```

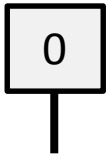
# Queue (semaphores) – 3 slots (initial state)




spaces

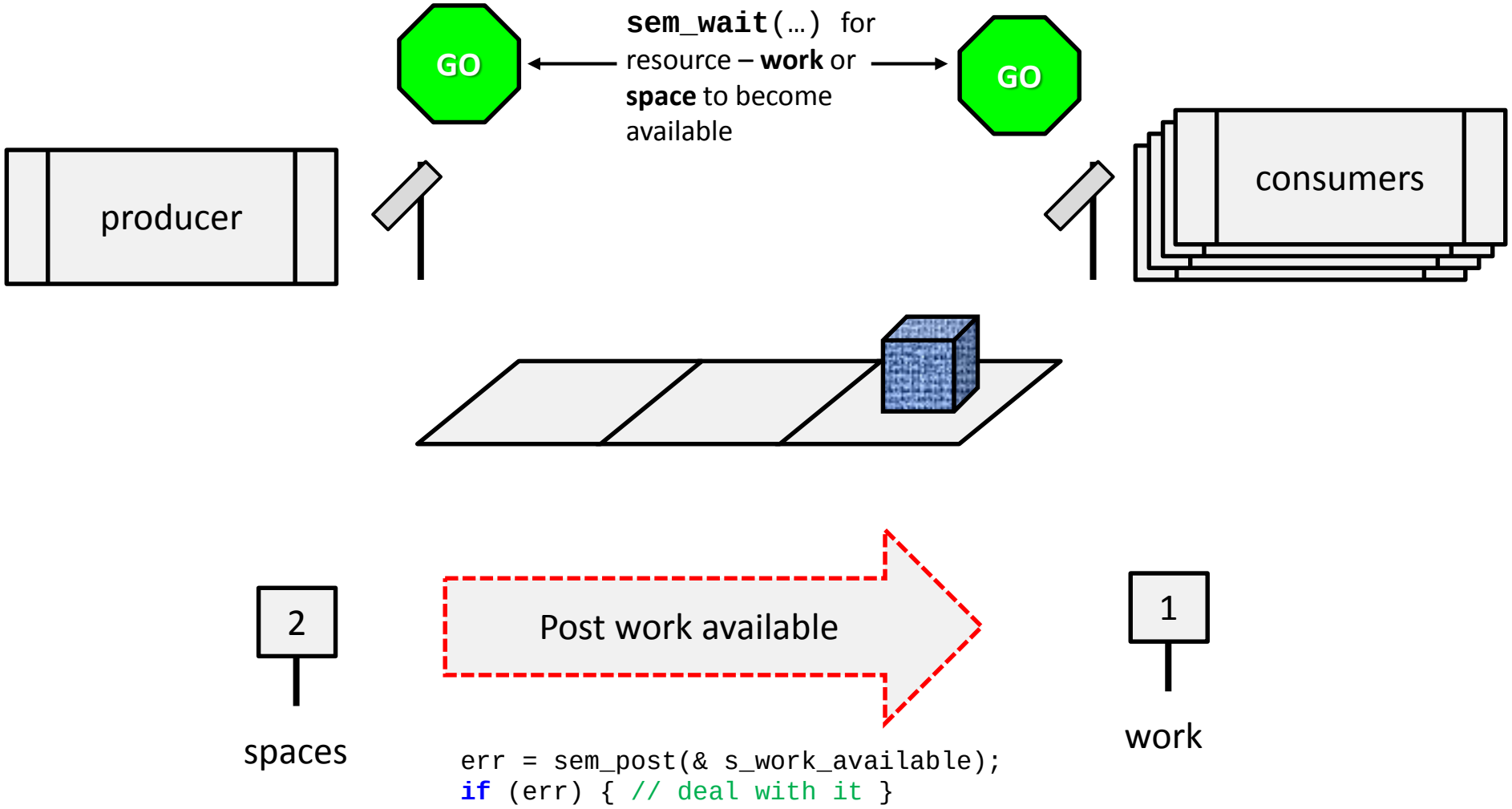
```
err = sem_init(&s_space_available, 0, 3)
if (err) { // deal with it }

…
```



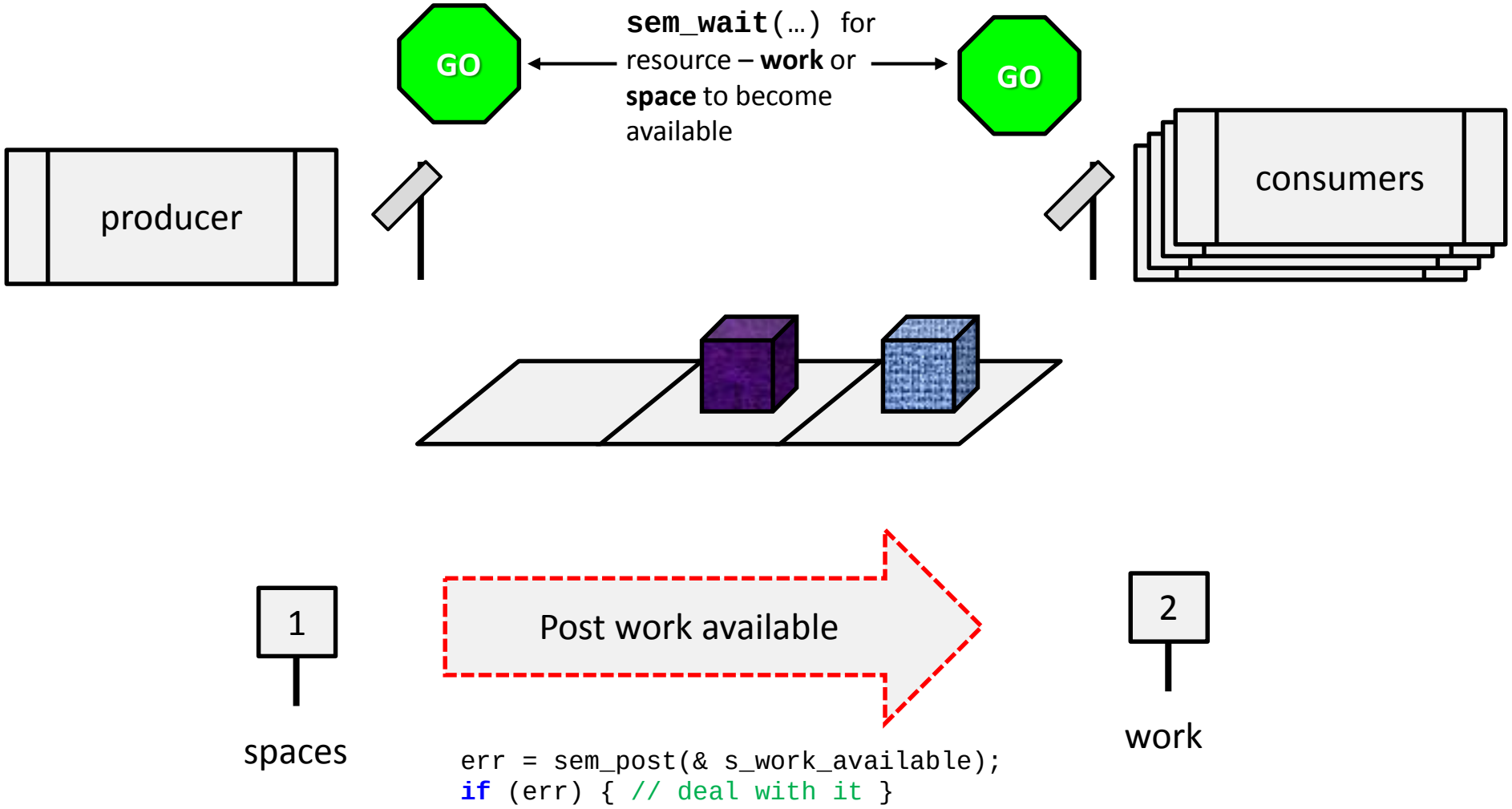

work

```
err = sem_init(&s_work_available, 0, 0)
if (err) { // deal with it }

…
```

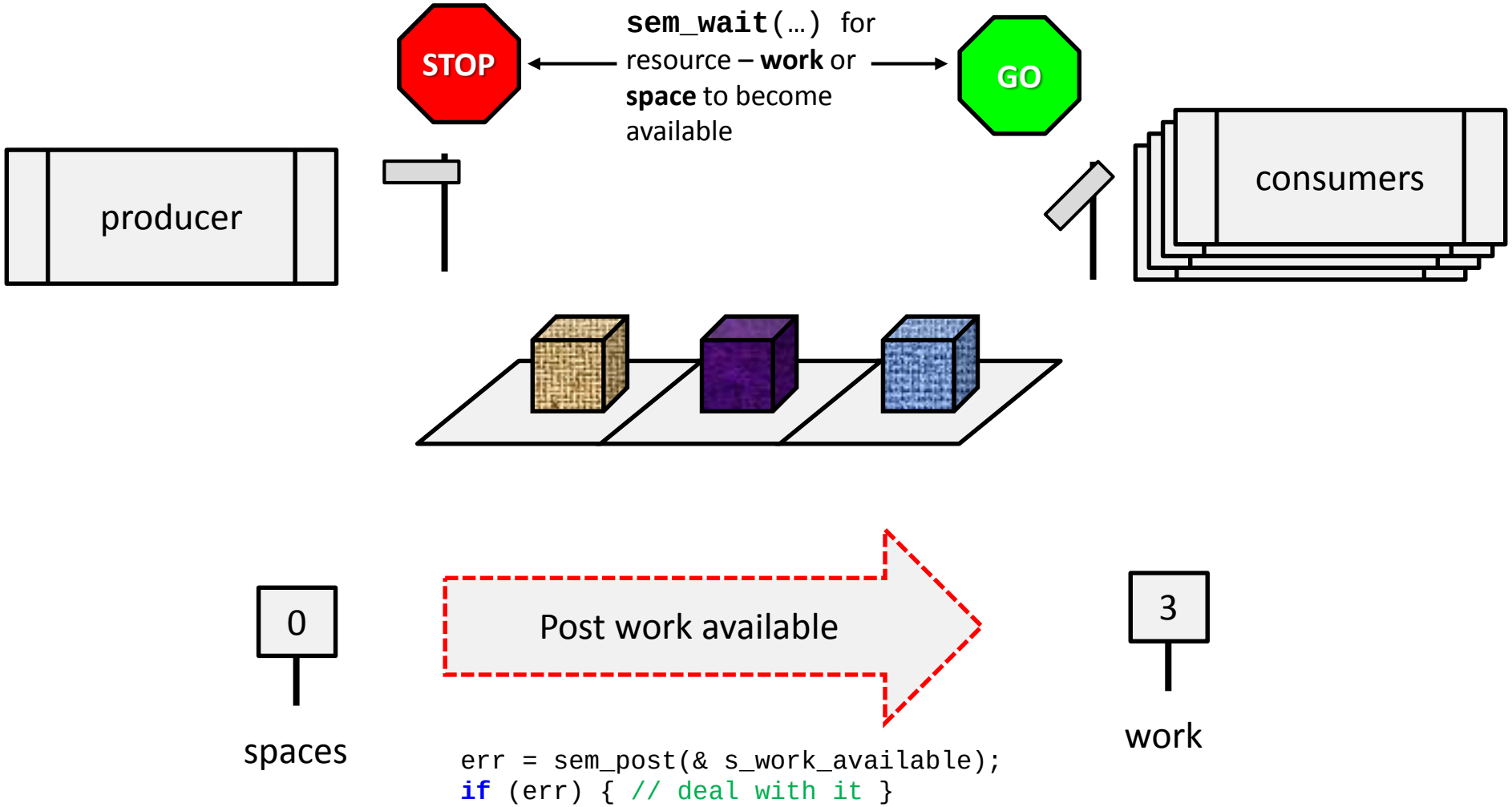
# Queue (semaphores) – 3 slots (work added)



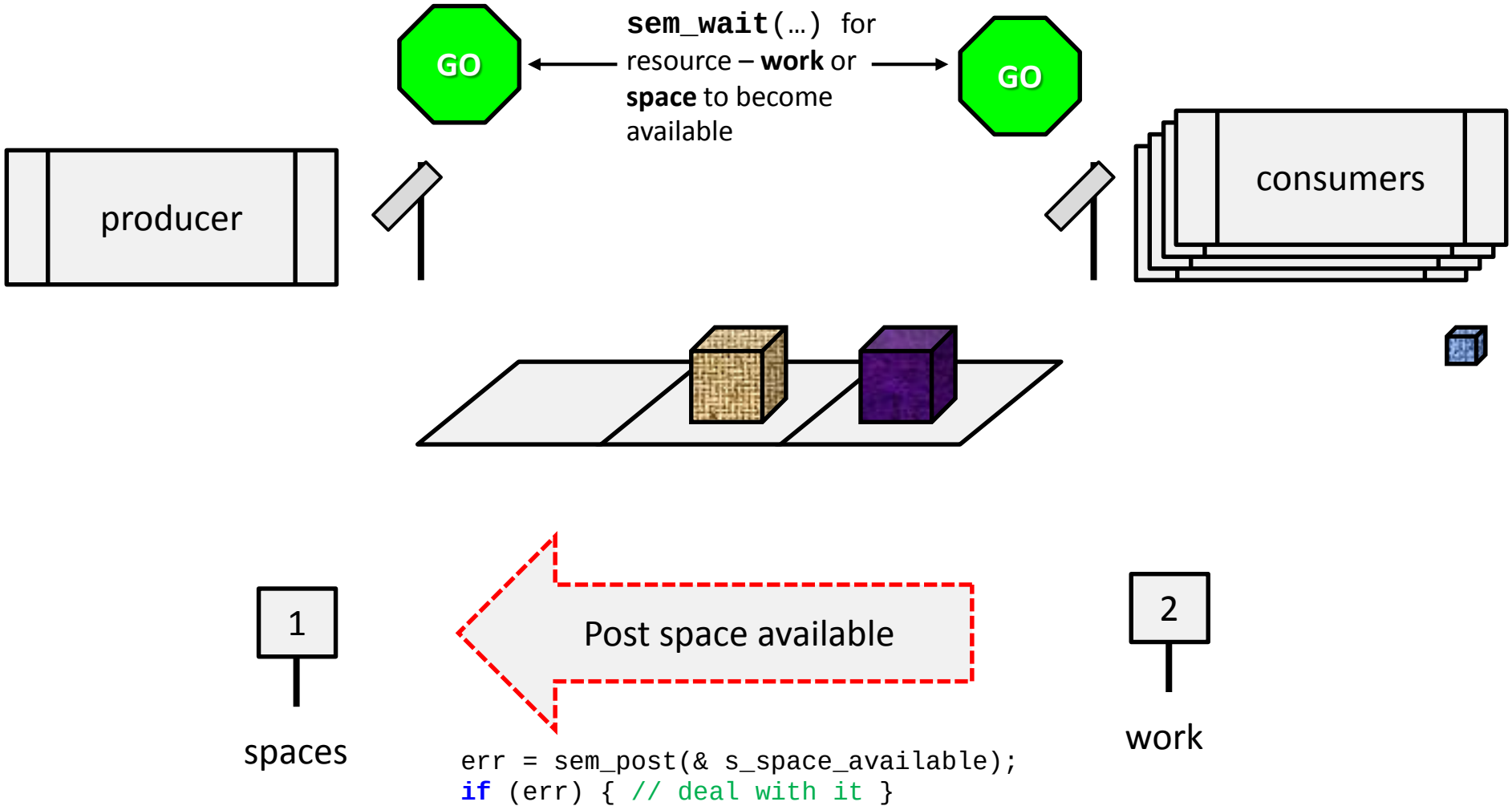
# Queue (semaphores) – 3 slots (work added)



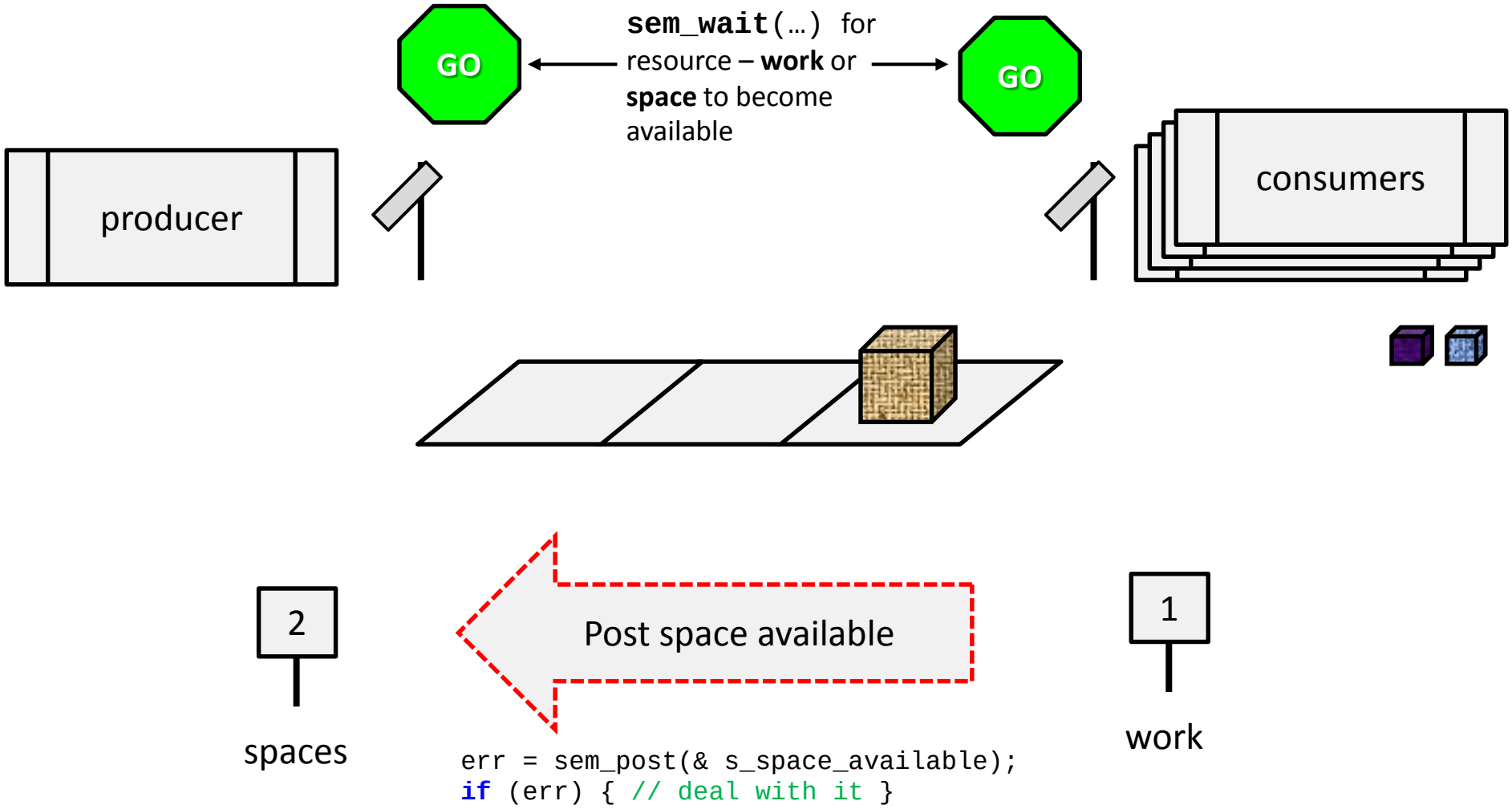
# Queue (semaphores) – 3 slots (work added)



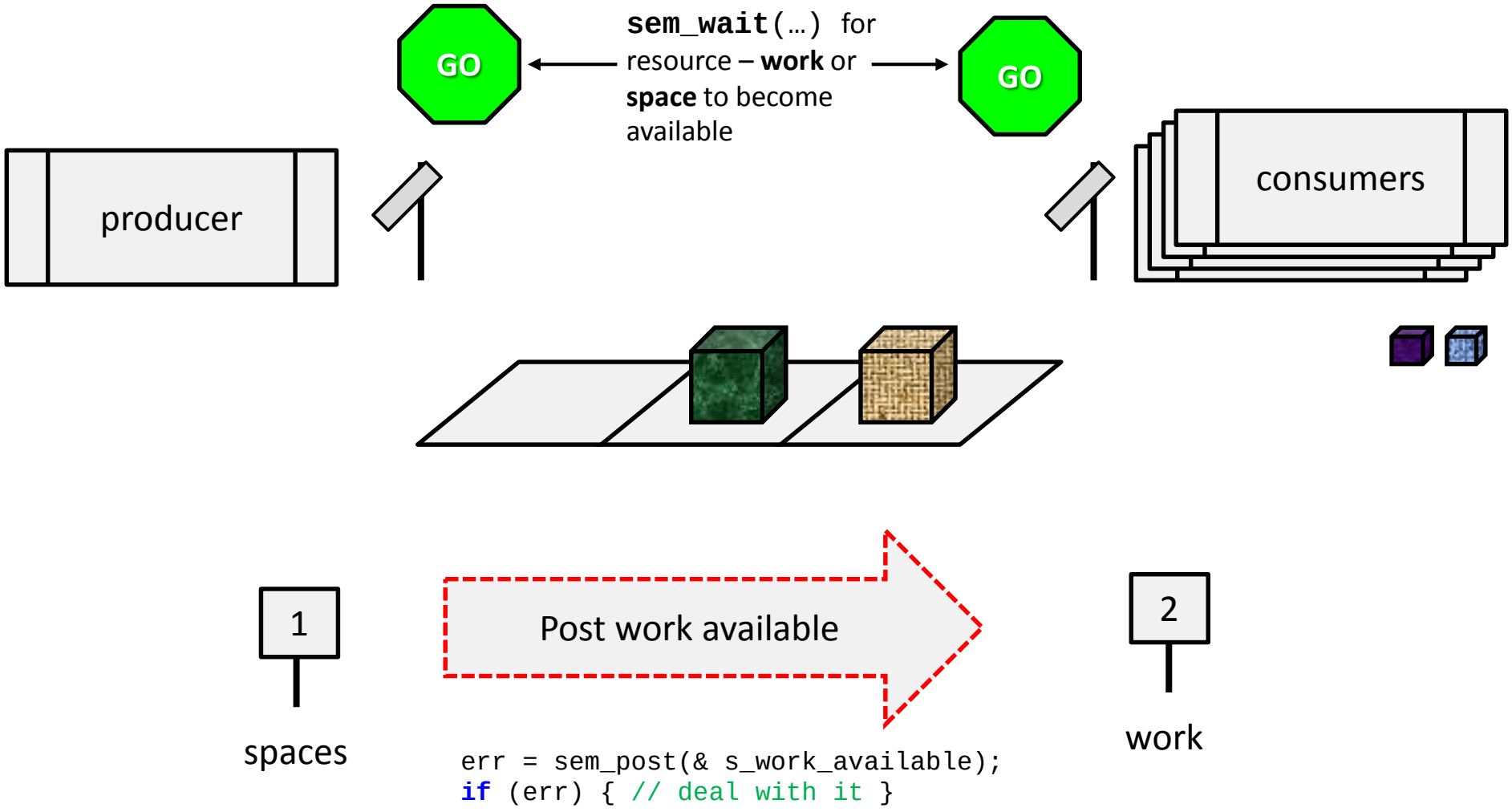
# Queue (semaphores) – 3 slots (work done)



# Queue (semaphores) – 3 slots (work done)



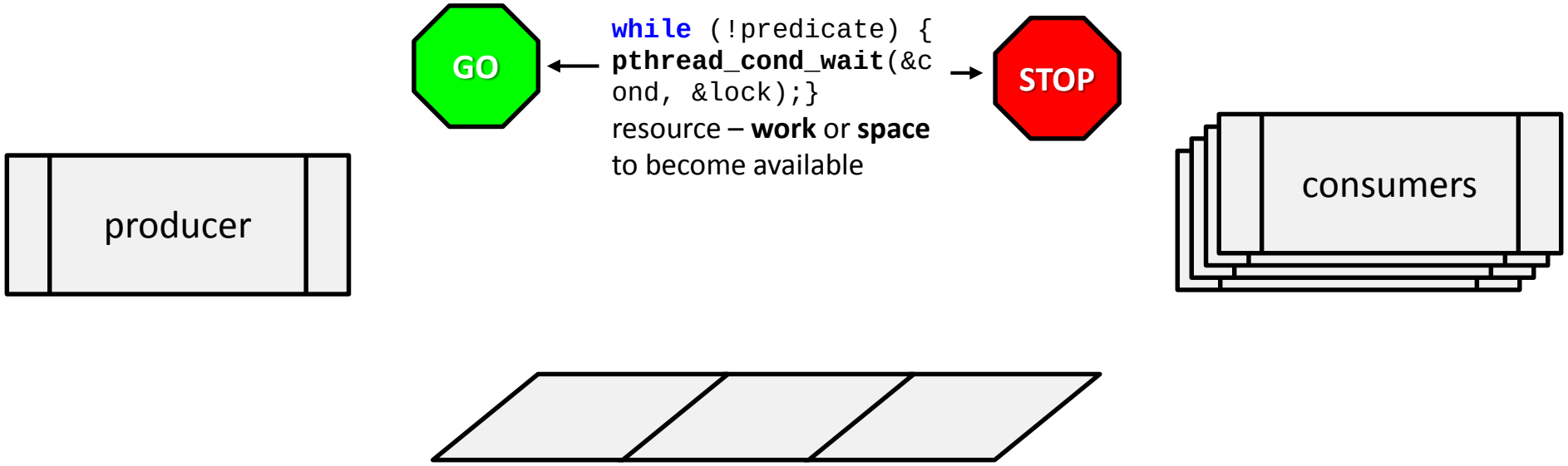
# Queue (semaphores) – 3 slots (work added)



# Producer consumer with condition variables

Condition variables and mutexes may be used to achieve the same result as semaphores in the previous example

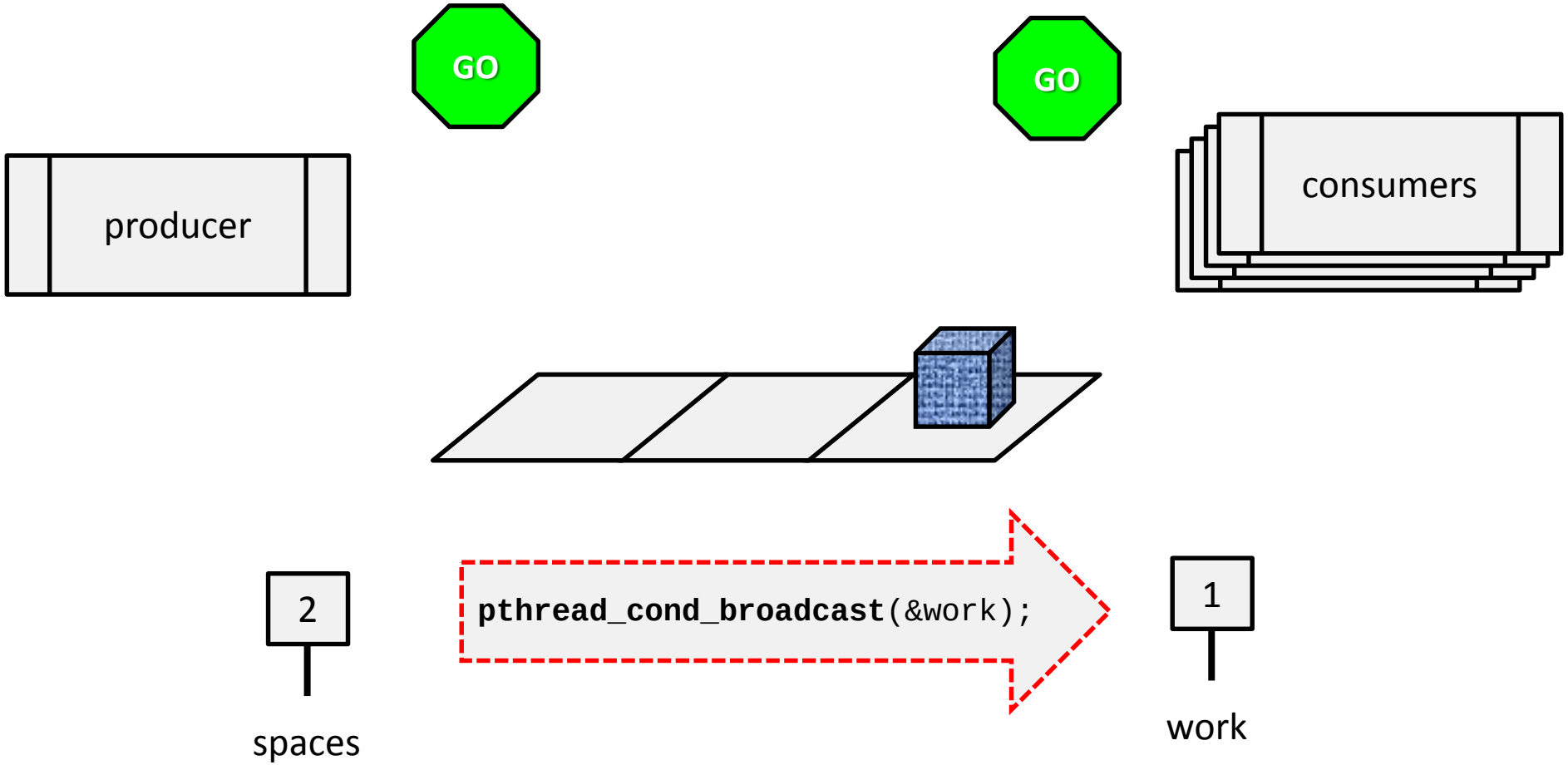
# Queue – 3 slots (initial state)




```
pthread_mutex_lock(&lock);
while (bufferfull) {
    pthread_cond_wait(&space, &lock);
}
```

```
pthread_mutex_lock(&lock);
while (bufferempty) {
    pthread_cond_wait(&work, &lock);
}
```

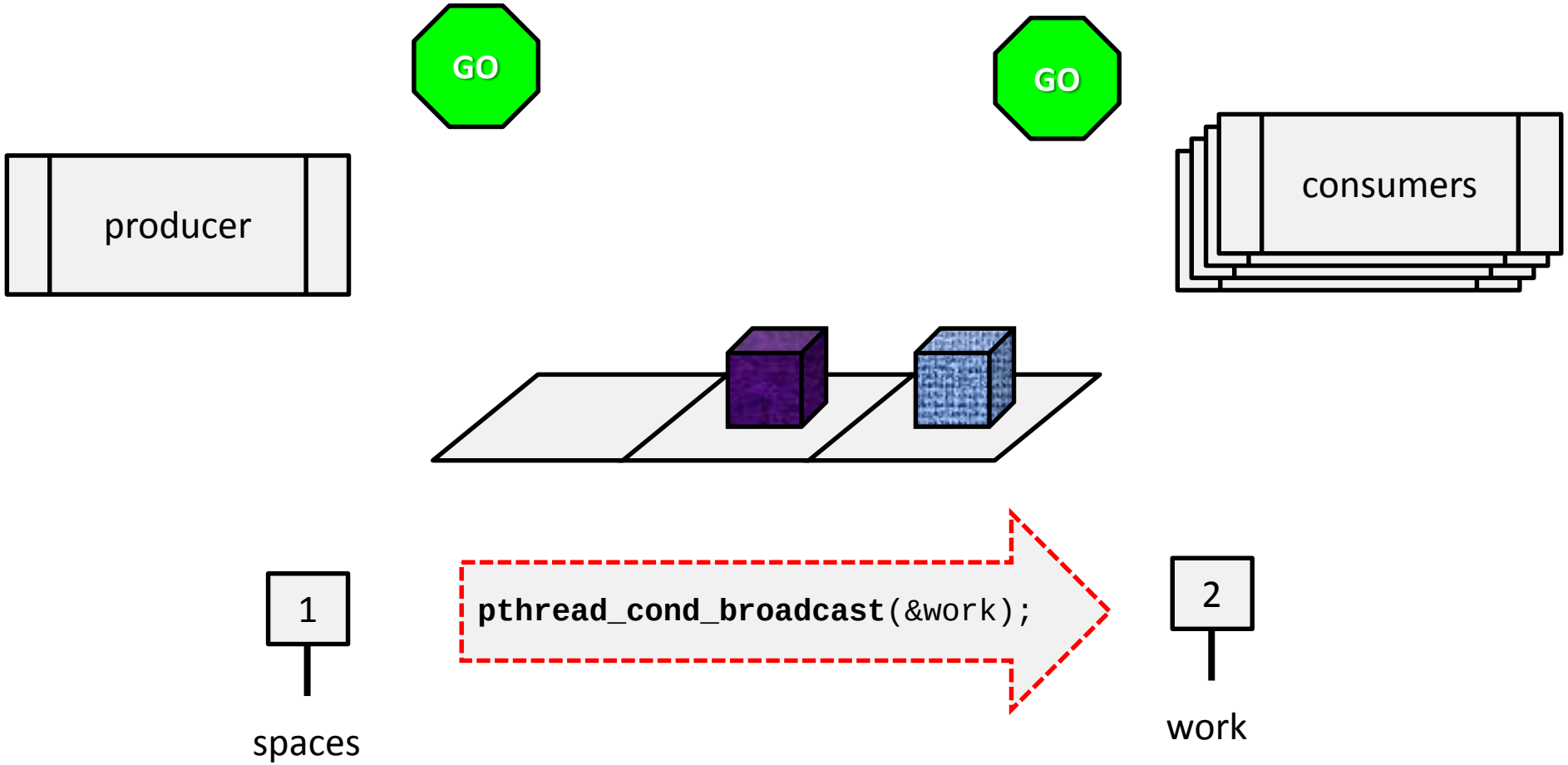
# Queue – 3 slots (work added)




```
pthread_mutex_lock(&lock);
while (bufferfull) {
    pthread_cond_wait(&space, &lock);
}
```

```
pthread_mutex_lock(&lock);
while (bufferempty) {
    pthread_cond_wait(&work, &lock);
}
```

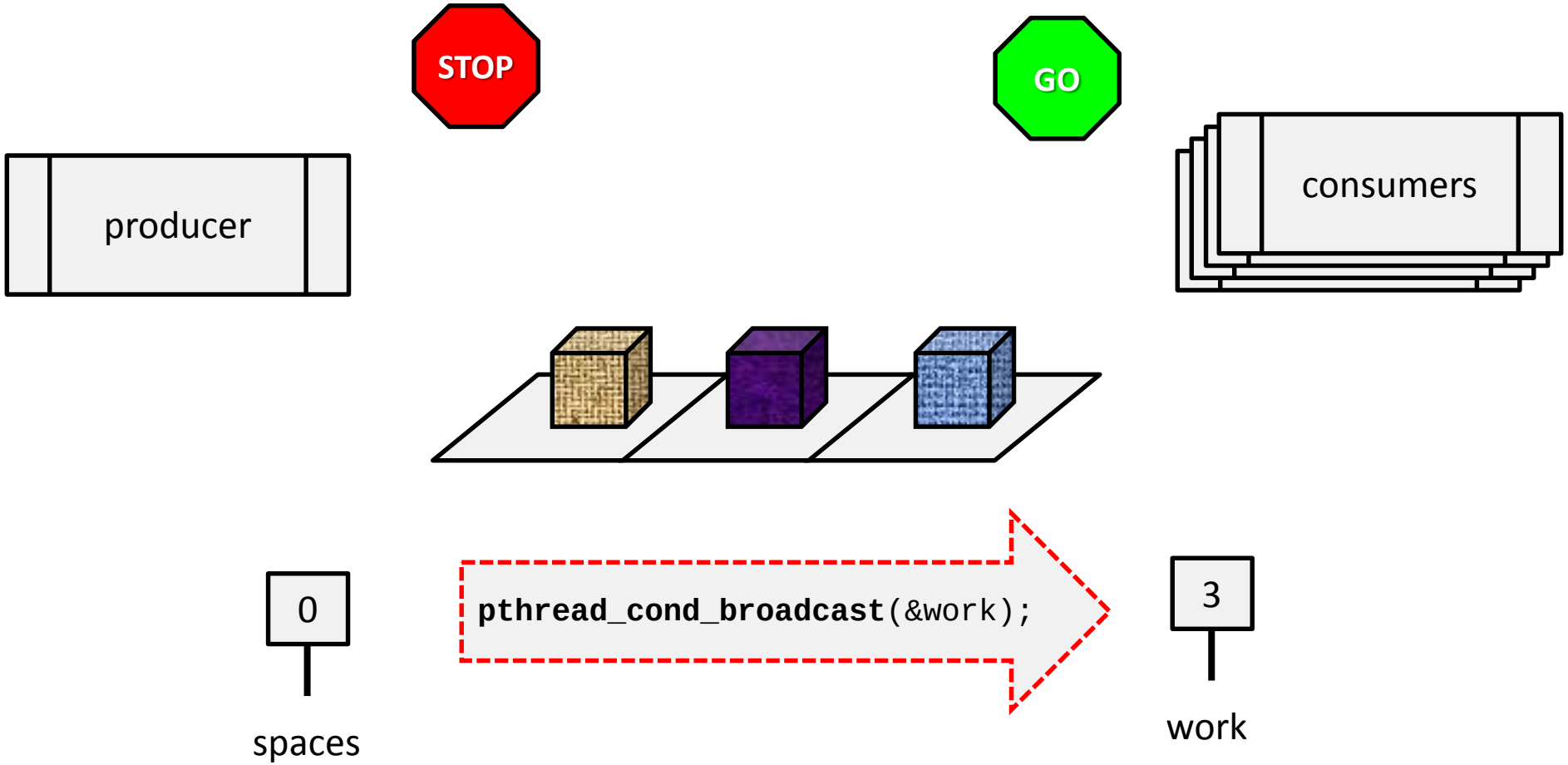
# Queue – 3 slots (work added)




```
pthread_mutex_lock(&lock);
while (bufferfull) {
    pthread_cond_wait(&space, &lock);
}
```

```
pthread_mutex_lock(&lock);
while (bufferempty) {
    pthread_cond_wait(&work, &lock);
}
```

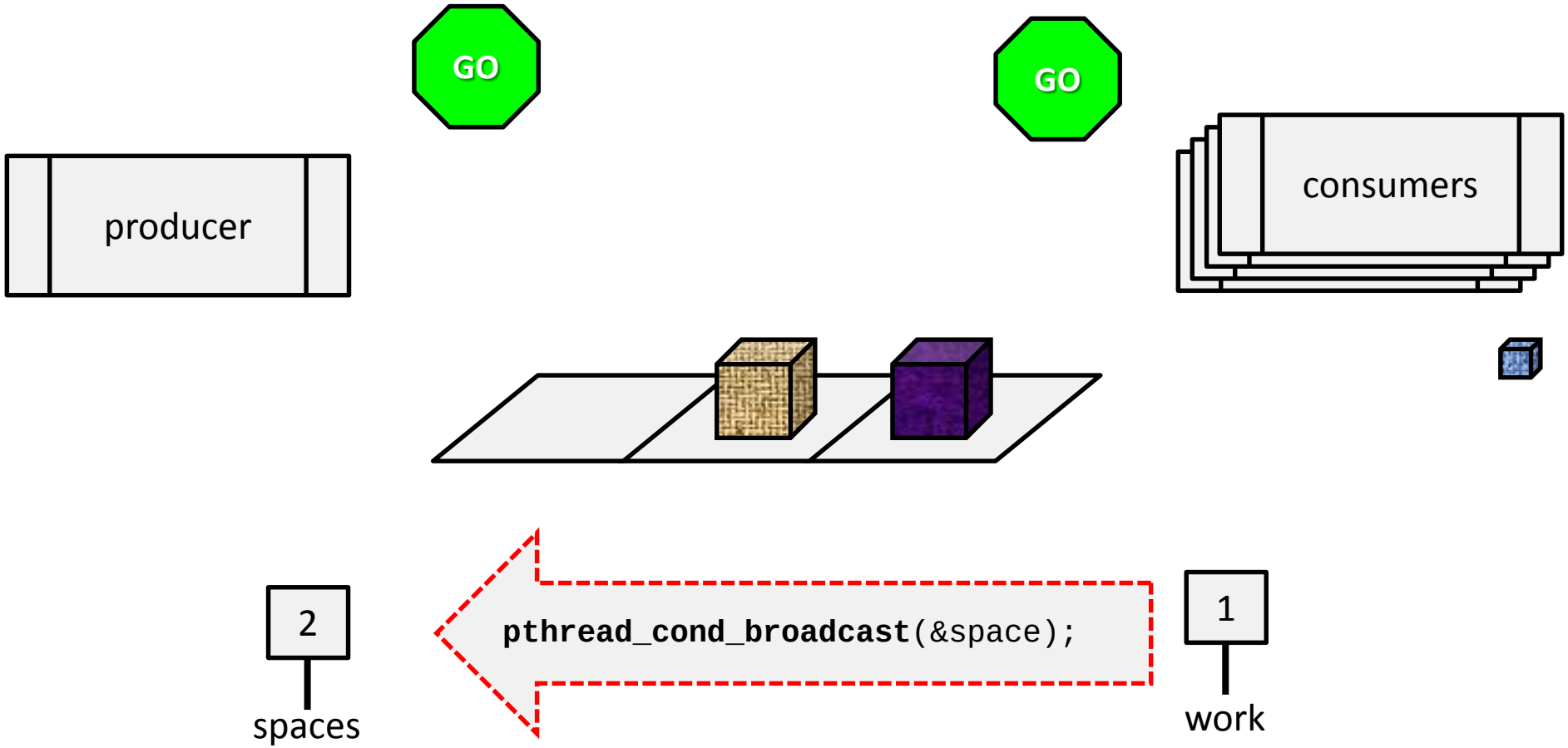
# Queue – 3 slots (work added)




```
pthread_mutex_lock(&lock);
while (bufferfull) {
    pthread_cond_wait(&space, &lock);
}
```

```
pthread_mutex_lock(&lock);
while (bufferempty) {
    pthread_cond_wait(&work, &lock);
}
```

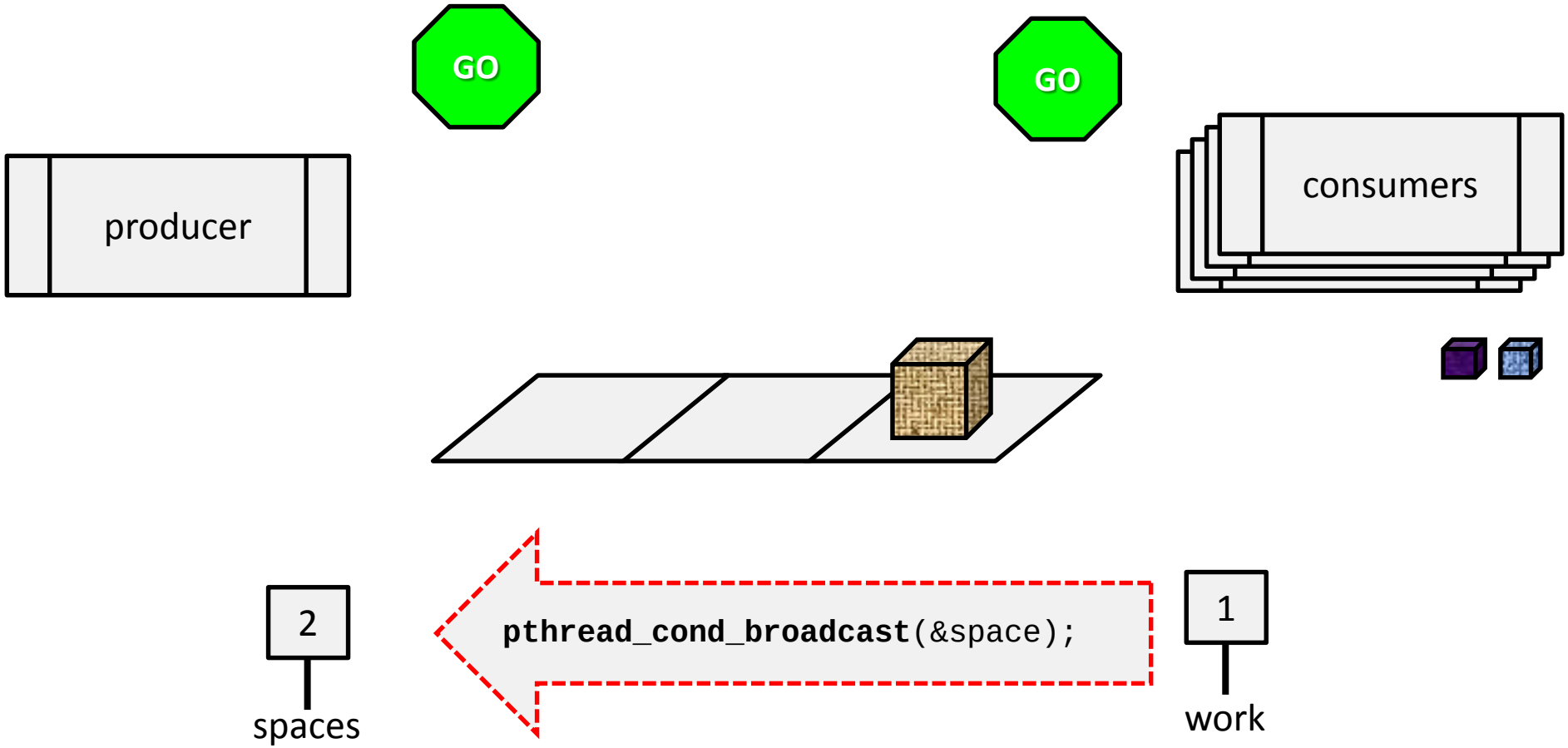
# Queue – 3 slots (work done)




```
pthread_mutex_lock(&lock);
while (bufferfull) {
    pthread_cond_wait(&space, &lock);
}
```

```
pthread_mutex_lock(&lock);
while (bufferempty) {
    pthread_cond_wait(&work, &lock);
}
```

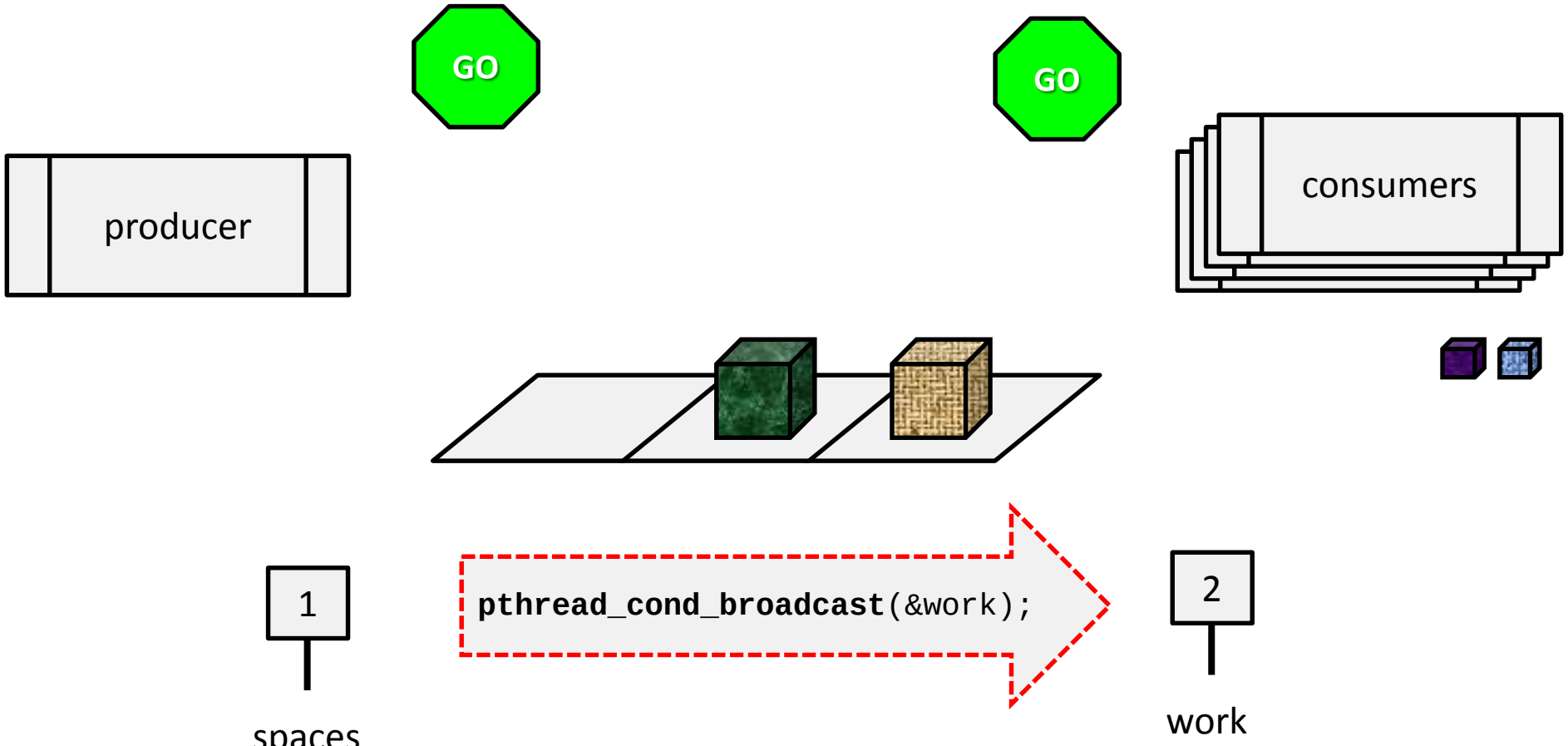
# Queue – 3 slots (work done)




```
pthread_mutex_lock(&lock);
while (bufferfull) {
    pthread_cond_wait(&space, &lock);
}
```

```
pthread_mutex_lock(&lock);
while (bufferempty) {
    pthread_cond_wait(&work, &lock);
}
```

# Queue – 3 slots (work added)




```
pthread_mutex_lock(&lock);
while (bufferfull) {
    pthread_cond_wait(&space, &lock);
}
```

```
pthread_mutex_lock(&lock);
while (bufferempty) {
    pthread_cond_wait(&work, &lock);
}
```

# Condition wait vs semaphores

## Conditions + mutex

```
// lock the mutex
err = pthread_mutex_lock(&lock);
if (err) { // deal with it }


while (!predicate) {
    err = pthread_cond_wait(&space, &lock);
    if (err) { // deal with it }
}

// Critical section (producer)

err = pthread_cond_broadcast(&work);
if (err) { // deal with it }

err = pthread_mutex_unlock(&lock);
if (err) { // deal with it }
```

## Semaphores

```
// wait for space to put new work
err = sem_wait(& s_space_available);
if (err) { // deal with it }

err = sem_wait(&s_data_lock);
if (err) { // deal with it }

// Critical section (producer)

err = sem_post(& s_work_available);
if (err) { // deal with it }

err = sem_post(&s_data_lock);
if (err) { // deal with it }
```

# Producer-consumer

Producer

```
while (1) {
  wait(spaces_avail);
  wait(q_lock);
  put_item();
  post(work_avail);
  post(q_lock);
}
```

Consumer

```
while (1) {
  wait(work_avail);
  wait(q_lock);
  get_item();
  post(spaces_avail);
  post(q_lock);
}
```

# Semaphores for multiple client server application

Accepted request queue

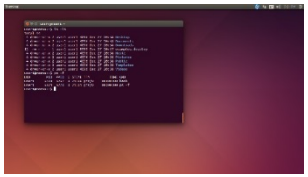
```
while(true){
  /*Wait non-empty
  request queue*/
  /*Pop request from
  queue and signal main
  loop */
  /*Interact with client
  and key value server*/
}
```

Worker threads

```
while(true){
  /*Wait for client request
  on either socket - poll
  */
  if(control){
    /*Handle control request
    immediately*/
  }
  else if(data){
    /*Wait for non-full
    request queue*/
    /*Push request onto queue
    and signal workers*/
  }
}
```

Main loop

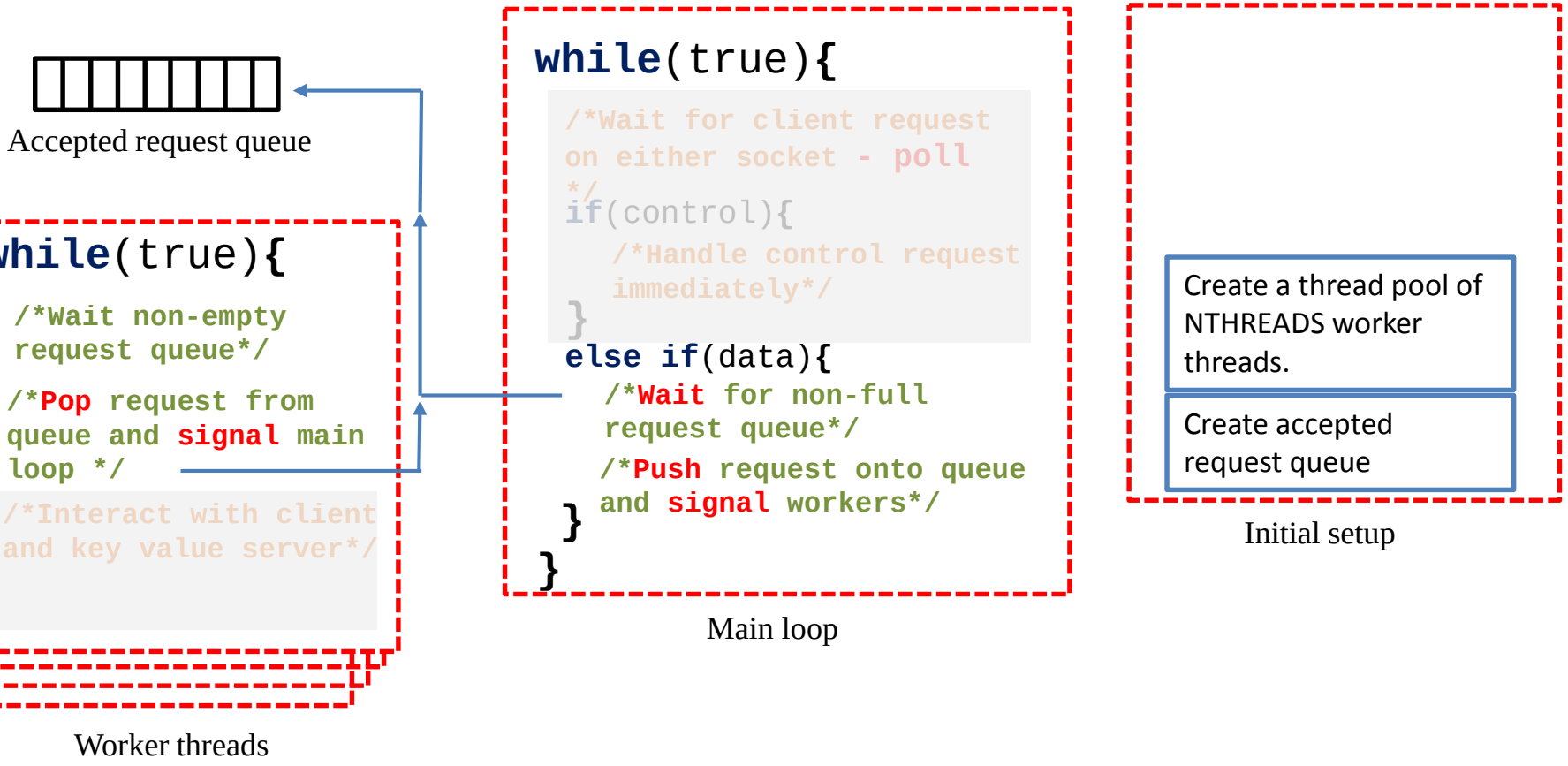
Key value DB



Client on telnet

- Set up sockets for two way communication between clients and server, by “binding” to CPORT and DPORT
- Create a thread pool of NTHREADS worker threads.
- Create accepted request queue

Initial setup


Accepted request queue
while(true){
/*Wait non-empty request queue*/
/*Pop request from queue and signal main loop */
/*Interact with client and key value server*/
}
Worker threads
while(true){
/*Wait for client request on either socket - poll */
if(control){
/*Handle control request immediately*/
}
else if(data){
/*Wait for non-full request queue*/
/*Push request onto queue and signal workers*/
}
}
Main loop
Create a thread pool of NTHREADS worker threads.
Create accepted request queue
Initial setup

# Shared memory – key concepts

- Critical sections
  - Where shared resources could be accessed by multiple threads simultaneously
- Mutex locks
  - Used to protect critical sections of code - specifically to prevent a **race condition**
- Condition variables
  - Used to put threads to sleep and wake them up again – help to avoid **busy waiting**
  - Require an associated **predicate**
- Semaphores
  - Can be used to perform different tasks depending on their initial value
    - For example, a **binary semaphore** can do the same job as a mutex lock

# Shared memory – other concepts

- Peterson's algorithm
  - How it achieves mutual exclusion
  - Disadvantages
- Test and set in hardware
- Producer consumer problem
- Demands on Critical Sections
  - Security
  - Liveness
  - Fairness