Department of Computer Science
University of Bristol

COMS20001 - Concurrent Computing

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

Lecture 13

Sharing Memory: condition variables and semaphores in C

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

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

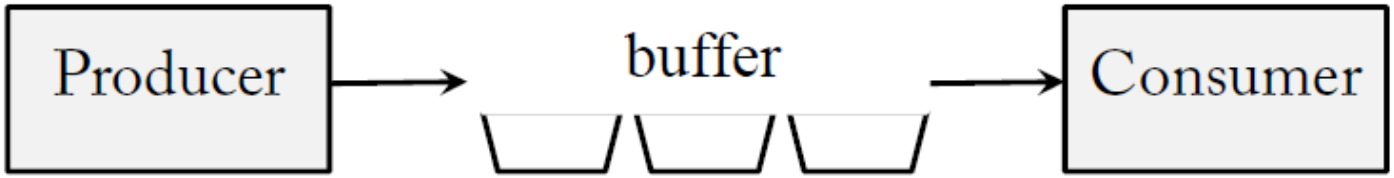
Busy Waiting vs. Suspension

- **'Busy Waiting' (also known as Spinning) …**
  a thread repeatedly checks to see if a condition is true

- Peterson's Algorithm uses 'Busy Waiting'

```
...
  while (
    (interested[1]==true) &&
    (turn == 0)) {
    // busy wait
  }
...
```

- **Consider a Producer-Consumer System with Limited Buffer and Permanent Operation**



→ explore suspension instead of busy waiting

Semaphore Implementation using Scheduling

```
class SemaphoreT {
  int        count;
  QueueType queue;
  public:
  SemaphoreT(int howMany);
  void P();
  void V();
}

SemaphoreT::SemaphoreT(
  int howMany) {
  count = howMany;
}

SemaphoreT::P() {
  if (count <= 0)
    sleep(queue);
  count--;
}

SemaphoreT::V() {
  count++;
  wakeup(queue);
}
```

- Need to implement all these methods as Critical Sections themselves !!!!

- **sleep(queue)**

```
thread_current.state = sleeping;
queue.enter(thread_current);
schedule;
```

- **wakeup(queue)**

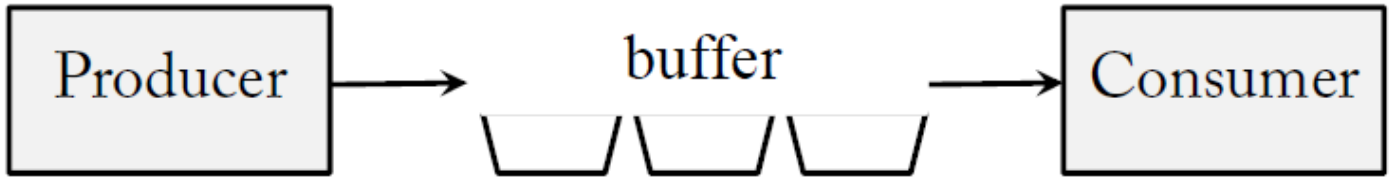
```
thread_current.state = ready;
switch_to(queue.take);
```

Producer-Consumer System using Semaphores

achieves mutual exclusion

```
...
// PRODUCER FRAGMENT
...

void produce() {
  while (true) {
    item = produce_item();
    noOfEmpty.P(); //wait&decr buffer
    critSec.P();   //enter critSec

    // send to buffer
    enter_item(item);

    critSec.V();   //leave critSec
    noOfFull.V();  //incr items
  }
}
```

```
...
// CONSUMER FRAGMENT
...

void consume() {
  while (true) {
    noOfFull.P(); //wait&decr item
    critSec.P();  //enter critSec

    // receive from buffer
    item = remove_item();

    critSec.V();   //leave critSec
    noOfEmpty.V(); //incr free buffer
    consume_item(item);
  }
}
```



Introduction

- Semaphores as mutexes
- Producer consumer problem revisited with semaphores and FIFO queue
- “Programming” exercise

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

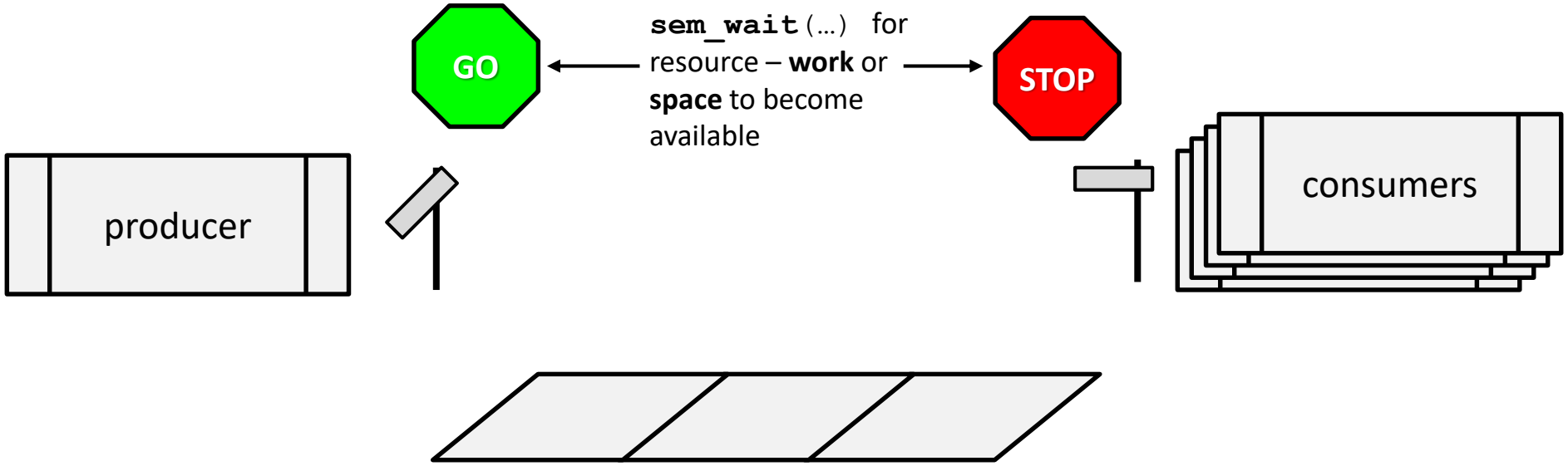
Semaphores

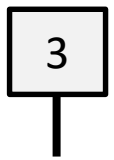
```
//When you set up your semaphore – probably global and set up in main()
err = sem_init(&s_data_lock, 0, 1)
if (err) { // deal with it }

…

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 }
```

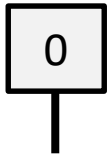
Queue – 3 slots (initial state)





spaces

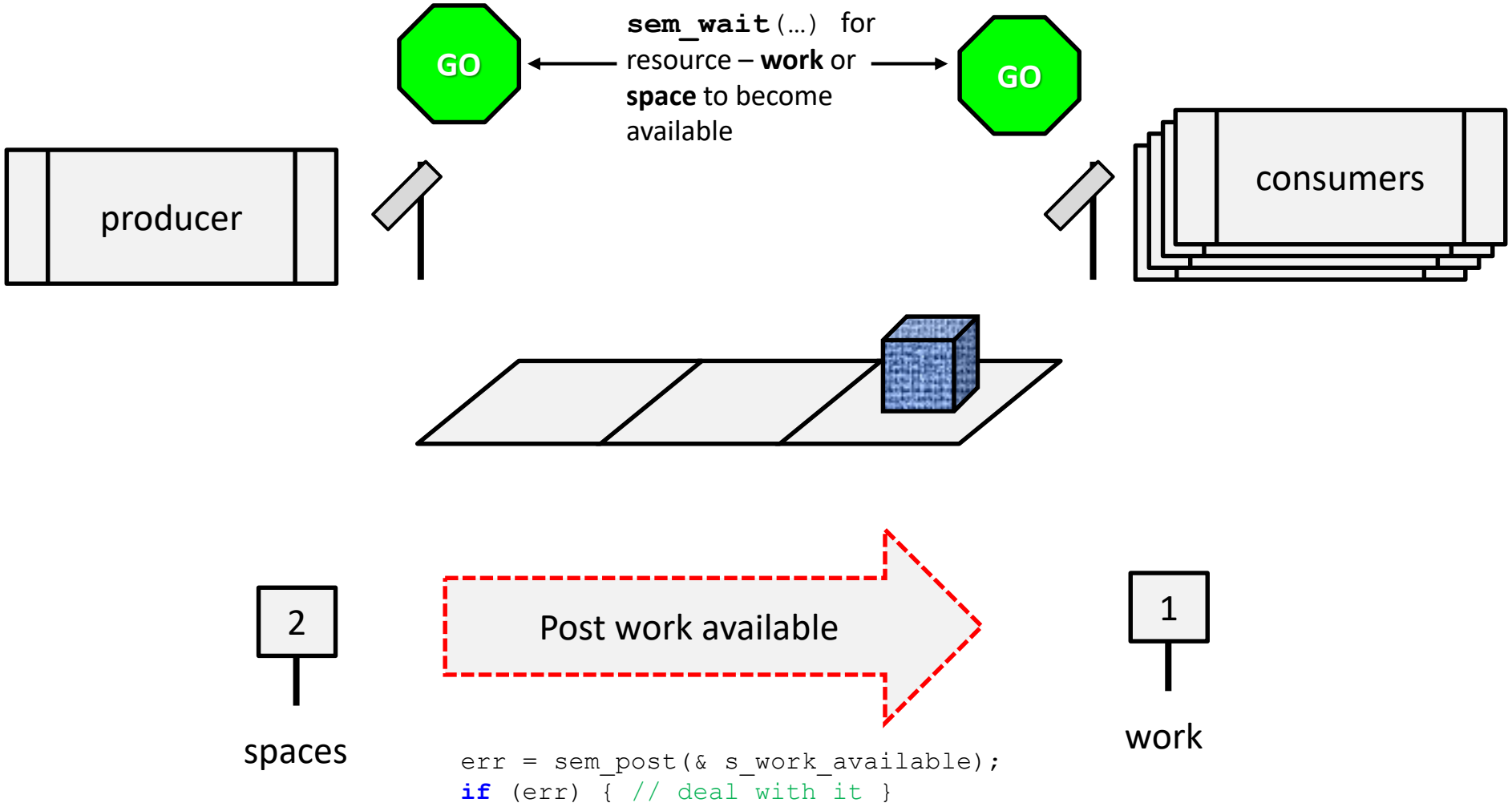
```
err = sem_init(&s_space_available, 0, 3)
if (err) { // deal with it }

…
```



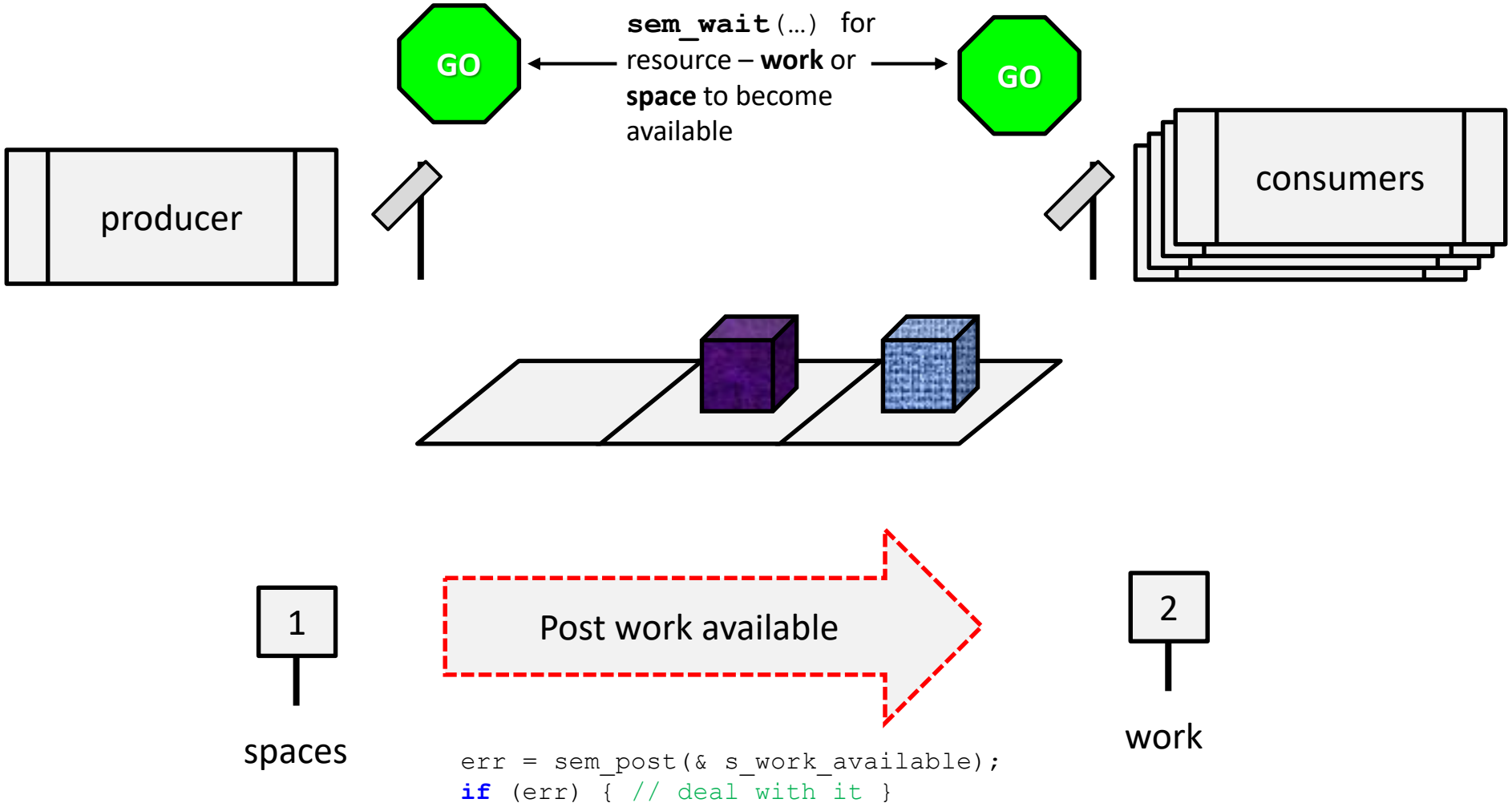
work

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

…
```

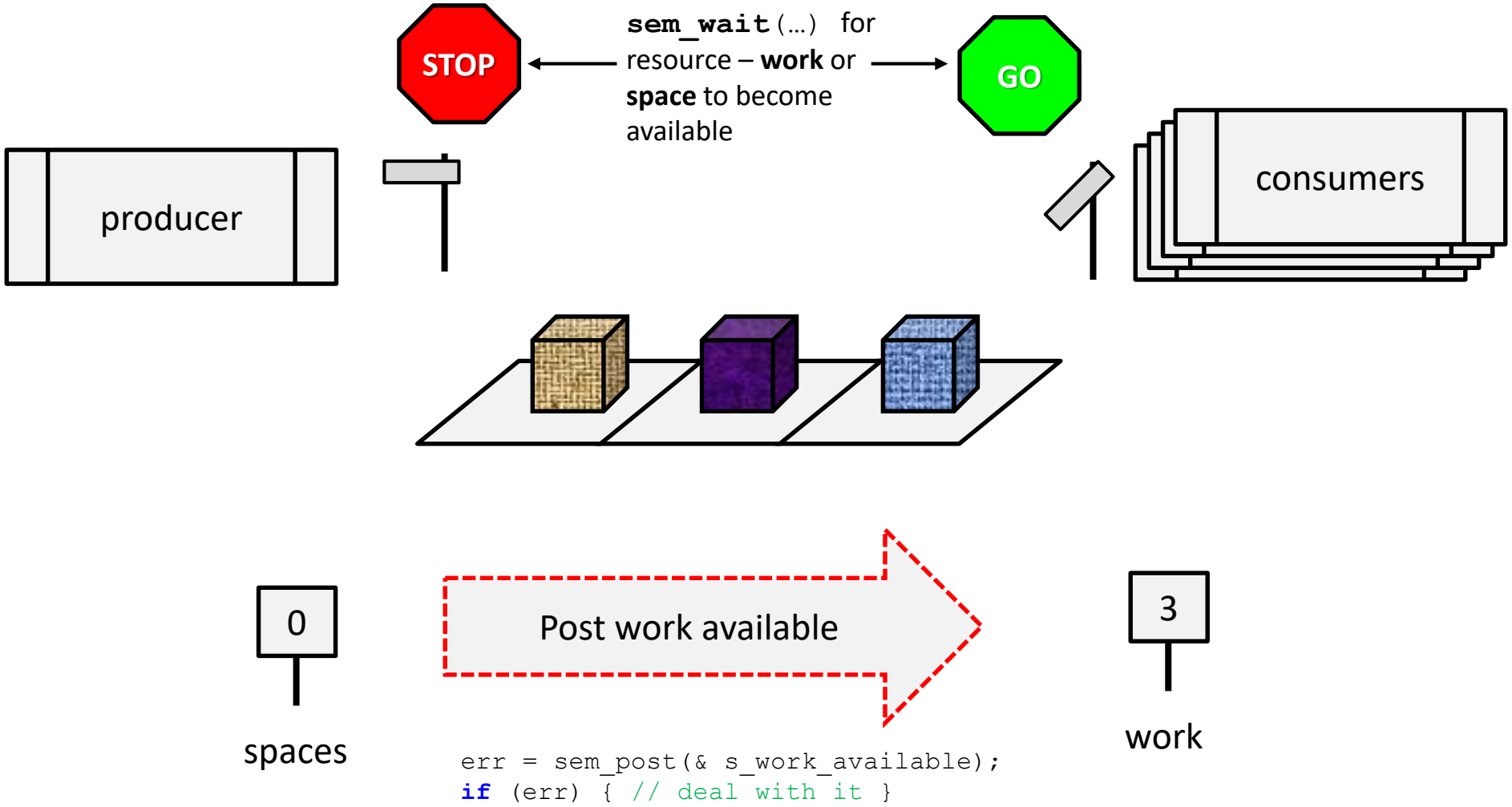
Queue – 3 slots (work added)



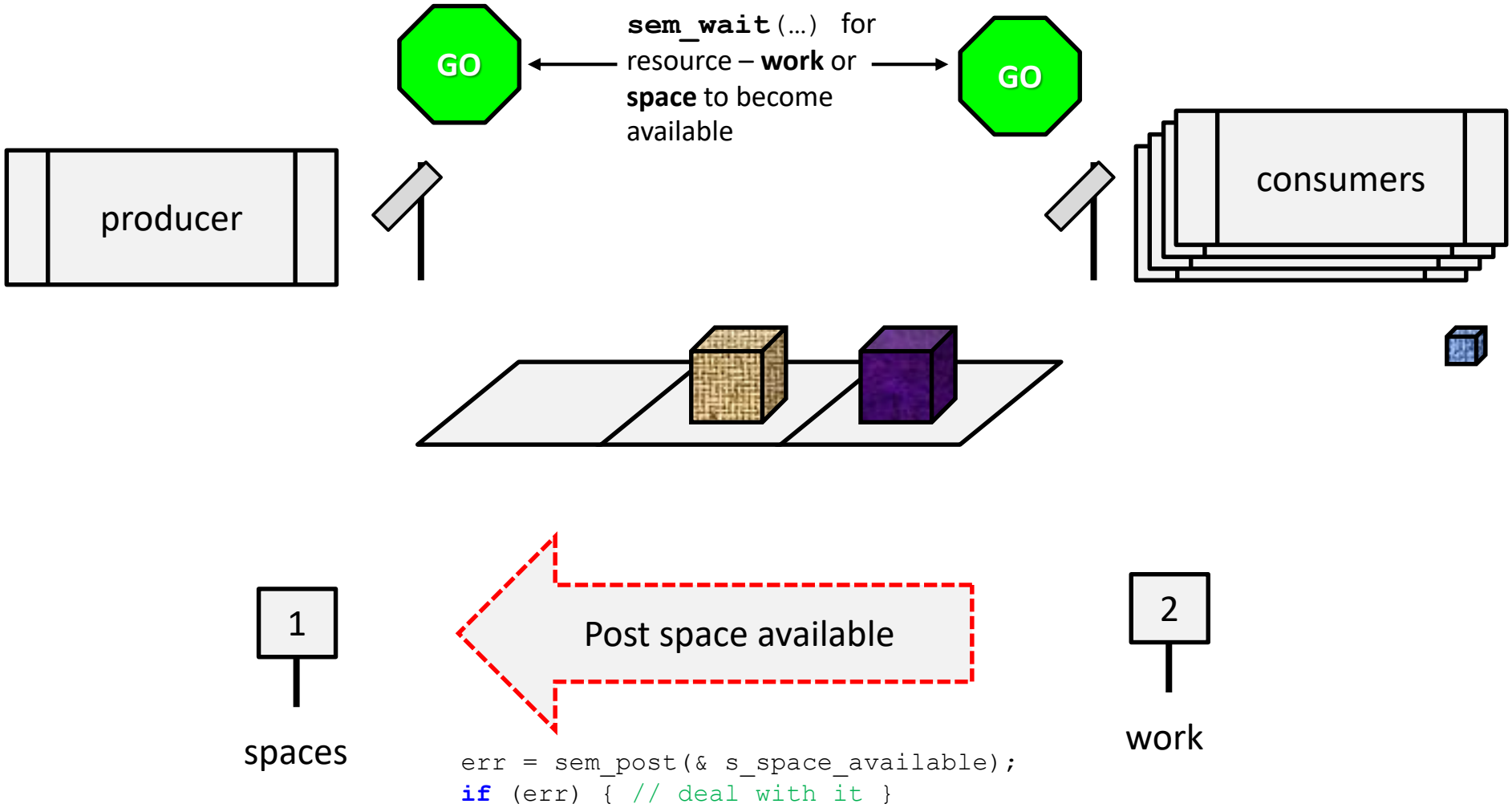
Queue – 3 slots (work added)



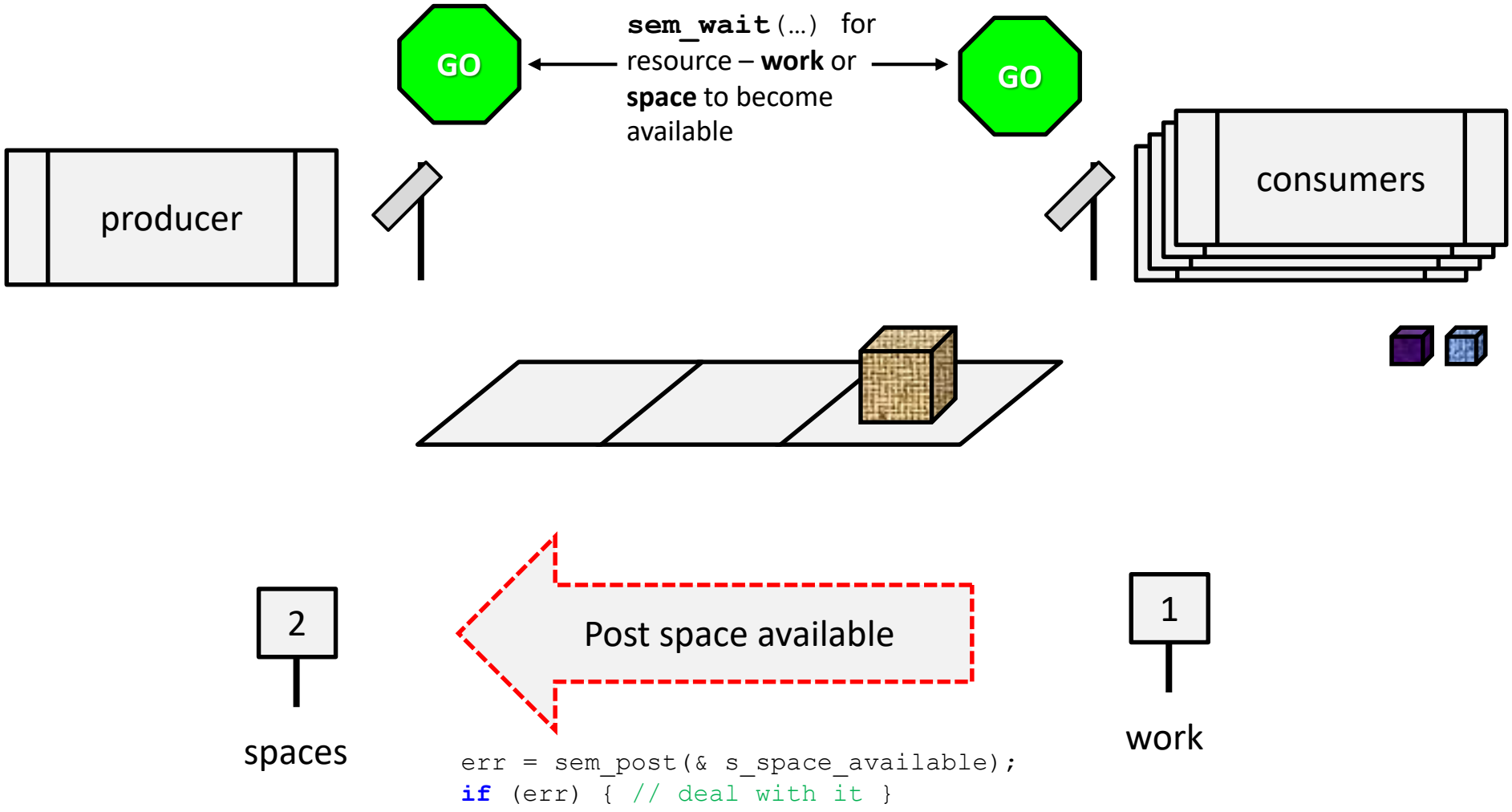
Queue – 3 slots (work added)



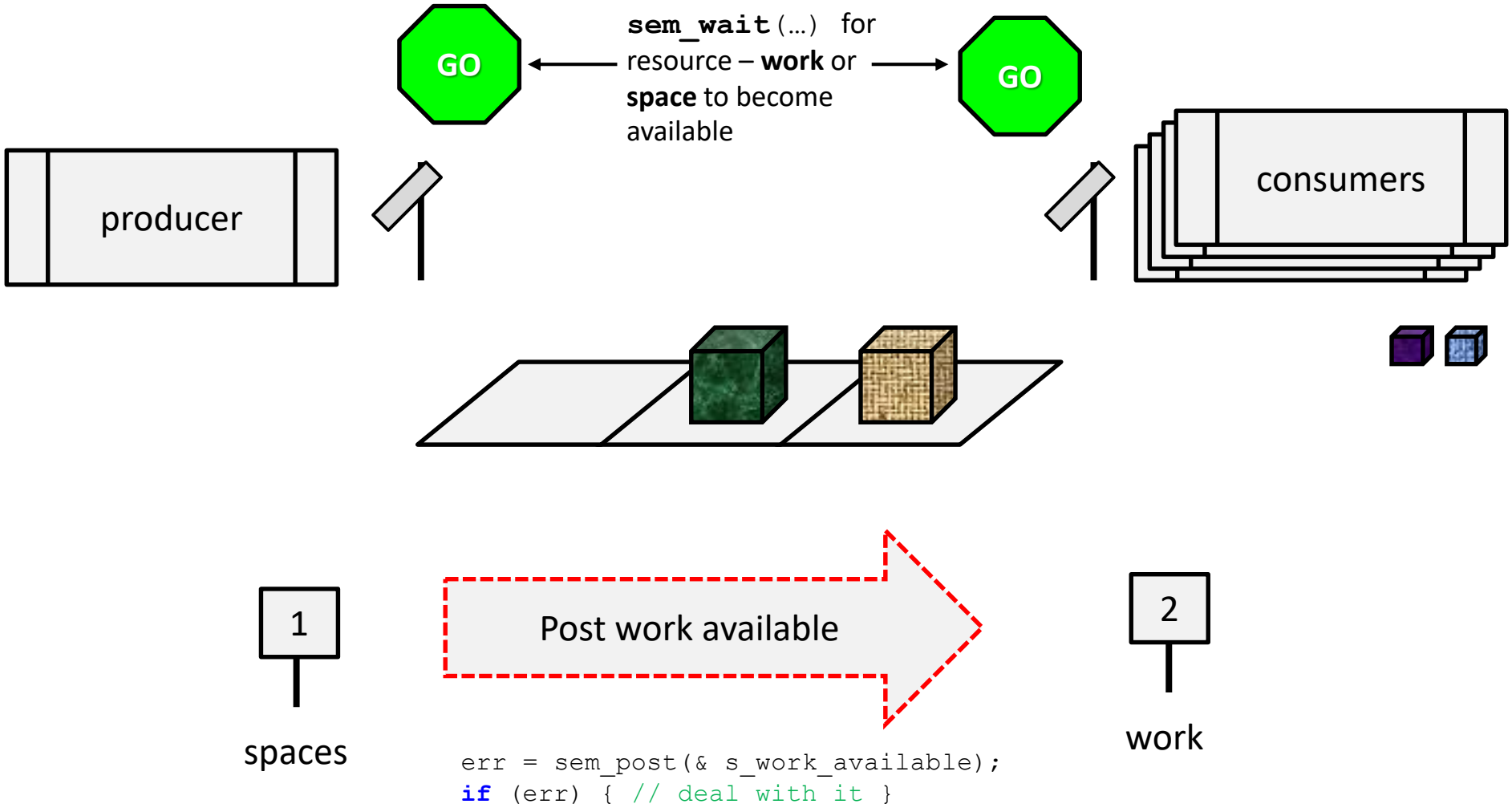
Queue – 3 slots (work done)



Queue – 3 slots (work done)



Queue – 3 slots (work added)



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, &mutex);
    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

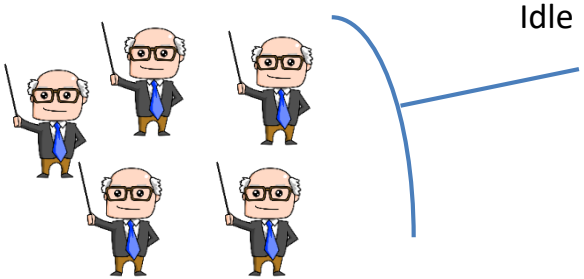
```
while (1) {
  wait(spaces_avail);
  wait(q_lock);
  put_item();
  post(work_avail);
  post(q_lock);
}
```

Consumer

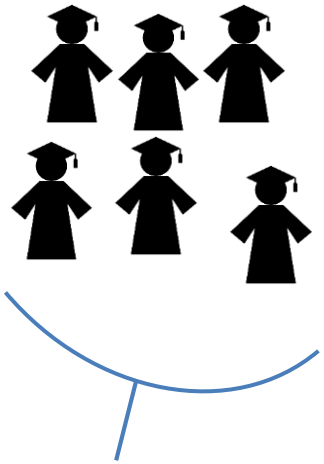
```
while (1) {
  wait(work_avail);
  wait(q_lock);
  get_item();
  post(spaces_avail);
  post(q_lock);
}
```

Student marker simulation

Initial state



Idle markers!



Students panic for random time …

S = number of students
M = number of markers
K = number of markers required to mark each student
N = number of students marked by each marker
T = total time is session
D = time to mark one presentation

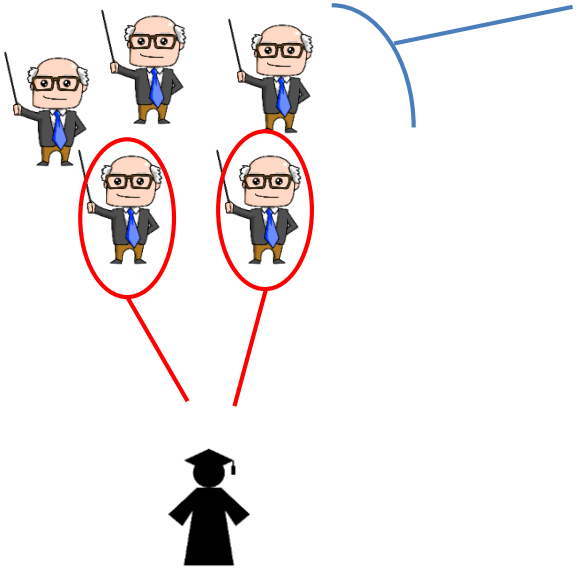
Task description

A student thread should execute the following steps:

- 1. Panic for a random time between 0 and (T-D-1) minutes.
- 2. Enter the lab.
- 3. Grab K markers when they are idle.
- 4. Do a demo for D minutes.
- 5. Exit from the lab.

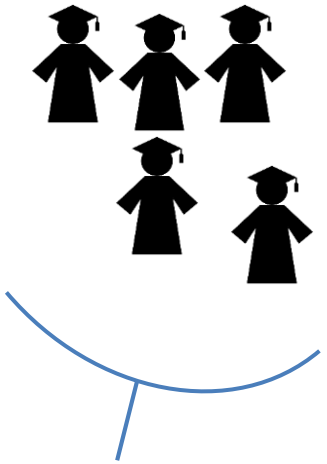
A marker thread executes the following steps:

- 1. Enter the lab.
- 2. Repeat (N times):
- a. Wait to be grabbed by a student.
- b. Wait for the student's demo to begin.
- c. Wait for the student's demo to finish.
- 3. Exit from the lab.

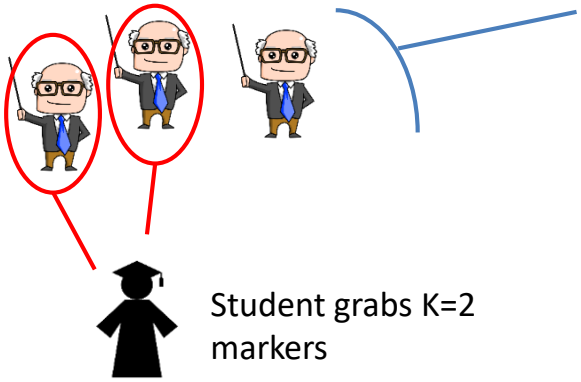


Idle markers!

Student grabs K=2 markers

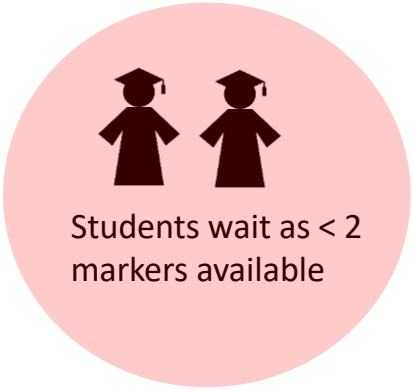


Students panic for random time …

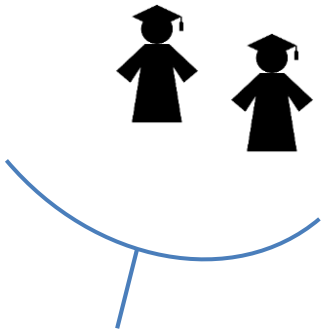


Idle marker!

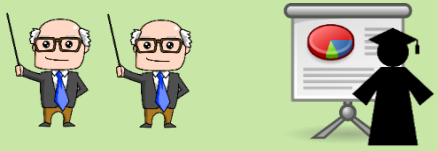
Student grabs K=2 markers



Students wait as < 2 markers available



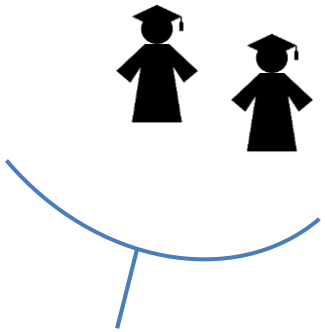
Students panic for random time …



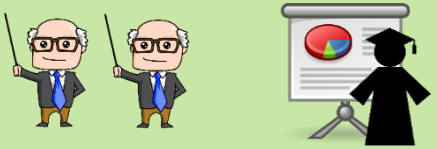
Do a demo for D minutes

Idle marker!

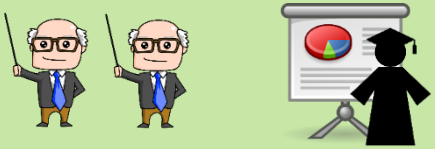
Students wait as < 2 markers available



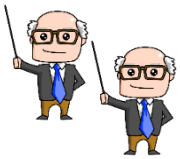
Students panic for random time …



Do a demo for D minutes



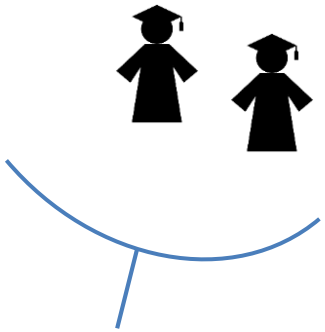
Do a demo for D minutes



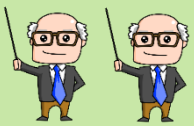
Markers available again

Idle marker!

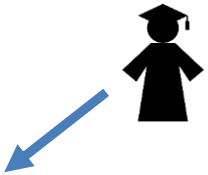
Students wait as < 2 markers available



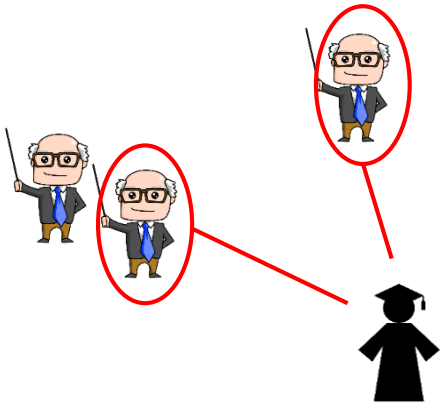
Students panic for random time …



Do a demo for D minutes



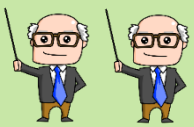
Student exits lab



Student grabs K=2 markers

Students wait as < 2 markers available

Students panic for random time …



Do a demo for D minutes