Further Thread Synchronization

Outline

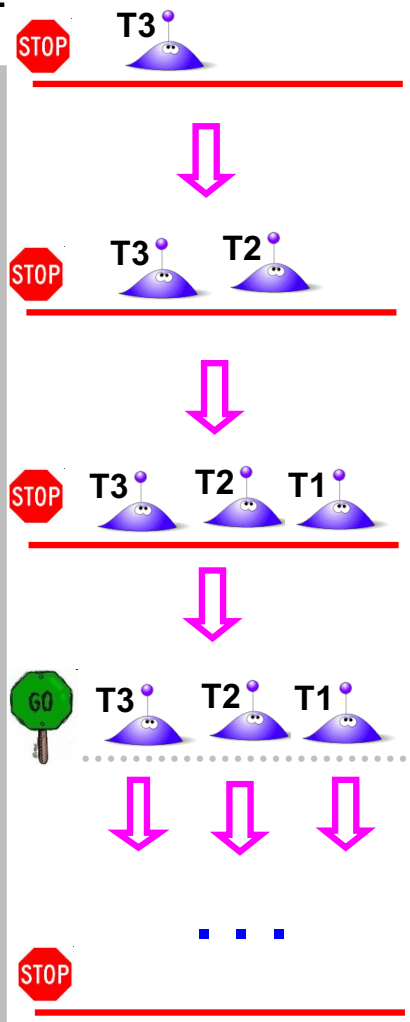
- Further Lock-based Thread Synchronization
  - **Barriers**
  - Condition variables

Barriers

- A **barrier** is a synchronization point at which a certain number of threads must arrive before any participating thread can continue.
- Participating threads call pthread_barrier_wait(). Participating threads block until the specified number of threads have called pthread_barrier_wait().

```
pthread_t thr[3];
pthread_barrier_t mybarrier;

void * tfunc (void * arg) {
  int i;
  for (i=1; i<MAX_GENERATIONS; i++) {
    // participate computing generation i from generation i-1:
    ...
    pthread_barrier_wait(&mybarrier); // wait until all 3
                                      // threads arrive at the
  }                                   // barrier.
}

int main() {
  int i, j;
  pthread_barrier_init(&mybarrier, NULL, 3);
  for (j=0; j<3; j++) {
    pthread_create(&thr[j], NULL, tfunc, (void *)j);
  }
  pthread_exit(NULL);// Exit the main thread.
}
```

Here we set up a barrier for 3 threads.



Barriers (cont.)

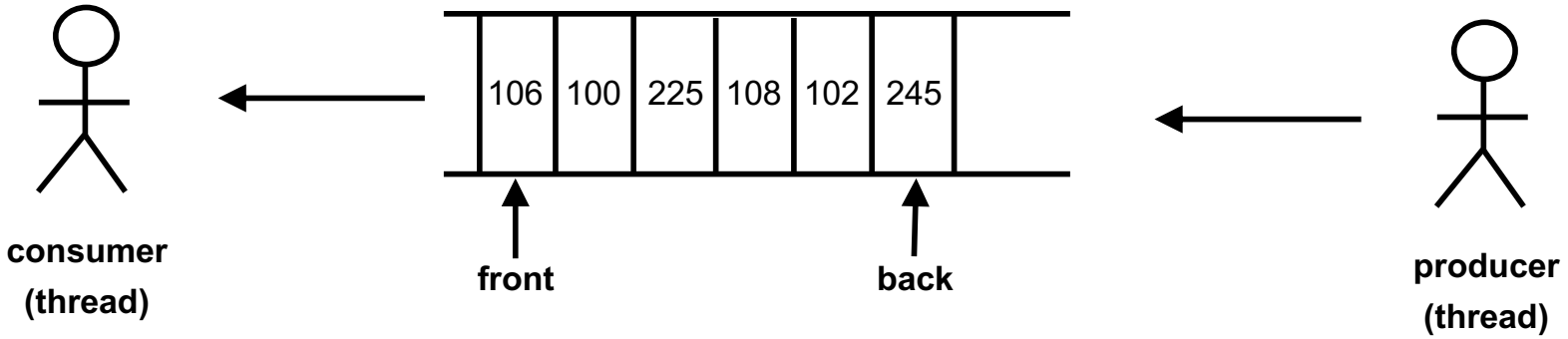
- Barriers are of type **`pthread_barrier_t`**.
- Use **`pthread_barrier_init()`** to initialize a barrier.
  - The first argument is a pointer to a barrier variable.
  - The second argument is a pointer to barrier attributes.
    - Use **`NULL`** for default attributes, which is sufficient in our case.
  - The third argument specifies the number of threads that are needed to open the barrier.
    - Example on the previous slide: 3 means
      - the first 2 threads that call pthread_barrier_wait() will block.
      - the third thread calling pthread_barrier_wait() will open the barrier. All three threads continue execution.
- Use pthread_barrier_wait() to block on the barrier.

```
pthread_barrier_t mybarrier;

pthread_barrier_init(&mybarrier, NULL, number_of_threads);

pthread_barrier_wait(&mybarrier);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Outline

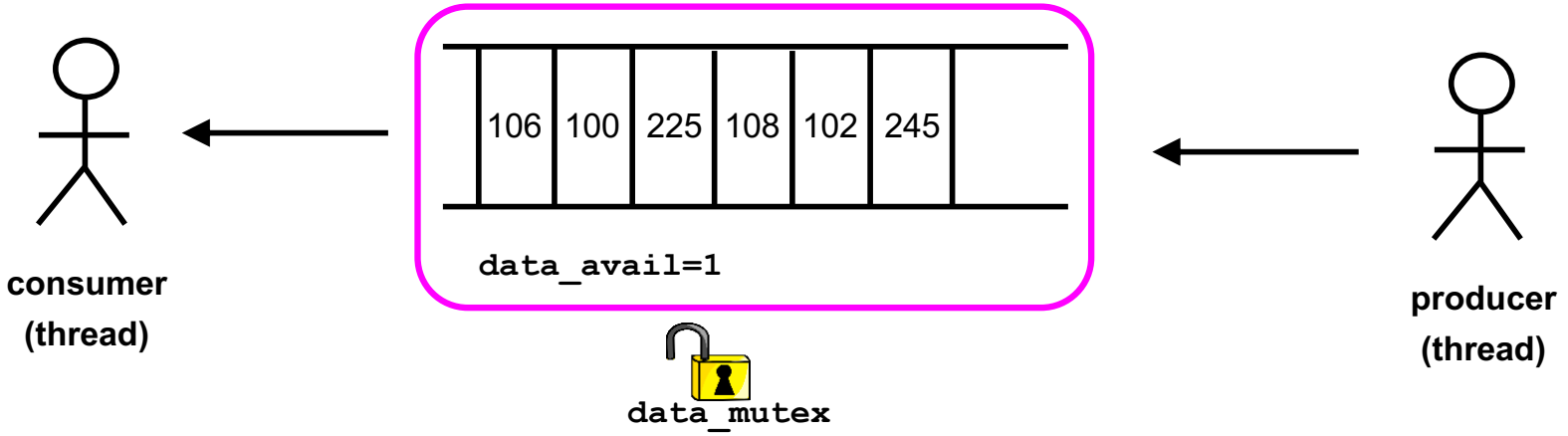
- Further Lock-based Thread Synchronization
  - Barriers ✓
  - **Condition variables** ← next

Producer-Consumer Problem



- Producer inserts data into queue (back side)
- Consumer reads data from queue (front side)
- Consumer can only read data when queue not empty.
- Producer can only insert data into queue when queue not full.
  - Synchronizing the producer on the “not full” condition works same as synchronizing the consumer on “not empty”. Synchronizing the producer on “not full” is left out to keep the example simple.

Producer-Consumer Synchronization



- Global variable **`data_avail`** is used to tell the consumer that data is available.
  - data_avail = 1 means “data is available”.
  - data_avail = 0 means “queue is empty”.
- The queue data-structure and variable **`data_avail`** are shared between the producer and consumer thread.
- To avoid race conditions we use mutex “**`data_mutex`**” to synchronize access.

Without Condition Variables

```
// shared global variables:
int data_avail = 0;
pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *)
{
      Produce data

      pthread_mutex_lock(&data_mutex);

      Insert data into queue;
      // tell consumer that data is available:
      data_avail = 1;

      pthread_mutex_unlock(&data_mutex);

}
```

- Producer inserts data into queue
- Global variable **`data_avail`** is used to tell the consumer that data is available.
- The queue and variable **`data_avail`** are shared and must be protected by a mutex.

Without Condition Variables (cont.)

```
1   void *consumer(void *)
2   {
3       int GotItem = 0;
4       while( GotItem == 0 )
5       {
6           pthread_mutex_lock(&data_mutex);
7           if ( data_avail == 1 ) {
8               Fetch_data_item_from_queue();
9               if ( queue is empty )
10               data_avail = 0;
11              GotItem = 1;
12          }
13          pthread_mutex_unlock(&data_mutex);
14      }
15      consume_data();
16  }
```

- Problem: the consumer must spin until data becomes available.
  - aquire **data_mutex**
  - check **data_avail**
  - release **data_mutex**
- Once **data_avail** set to 1 by producer, consumer can extract data item from queue.
  - set **GotItem=1** to exit spin loop.
- We would like a solution were the consumer blocks until data becomes available.
  - Better then consuming CPU cycles through busy waiting!

Note: Spinning means to circle in the loop from lines 4-14 until a data item was fetched from the queue.

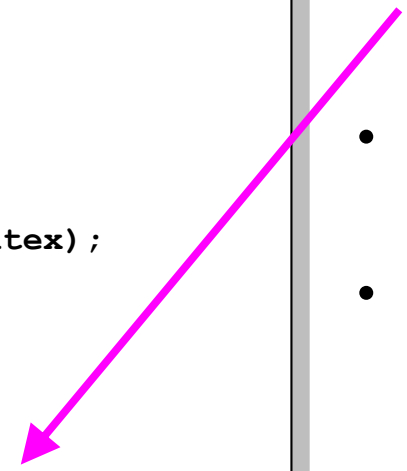
Condition Variables

- Waiting and signaling on condition variables:

  - pthread_cond_wait(condition, mutex)
    - Blocks the thread until the specific condition is signaled.
    - Must be called with mutex locked!
    - Automatically releases the mutex while it waits.
    - Upon return of function (condition has been signaled), mutex is locked again.
  - pthread_cond_signal(condition)
    - Wake up (at least) one thread waiting on the condition variable.
    - Must be called after mutex is locked, and must unlock mutex thereafter.
  - pthread_cond_broadcast(condition)
    - Used when multiple threads blocked at the condition.
    - Wake up all threads blocked at the condition.
      - ‘thundering herd’ syndrome if only one can get resource and others have to go to go back to blocking state.

With Condition Variables

```
int data_avail = 0;
pthread_mutex_t data_mutex =
        PTHREAD_MUTEX_INITIALIZER;
pthread_cont_t data_cond =
        PTHREAD_COND_INITIALIZER;

void *producer(void *)
{
    Produce data

    pthread_mutex_lock(&data_mutex);


    Insert data into queue;
    data_avail = 1;

    pthread_cond_signal(&data_cond);

    pthread_mutex_unlock(&data_mutex);

}
```



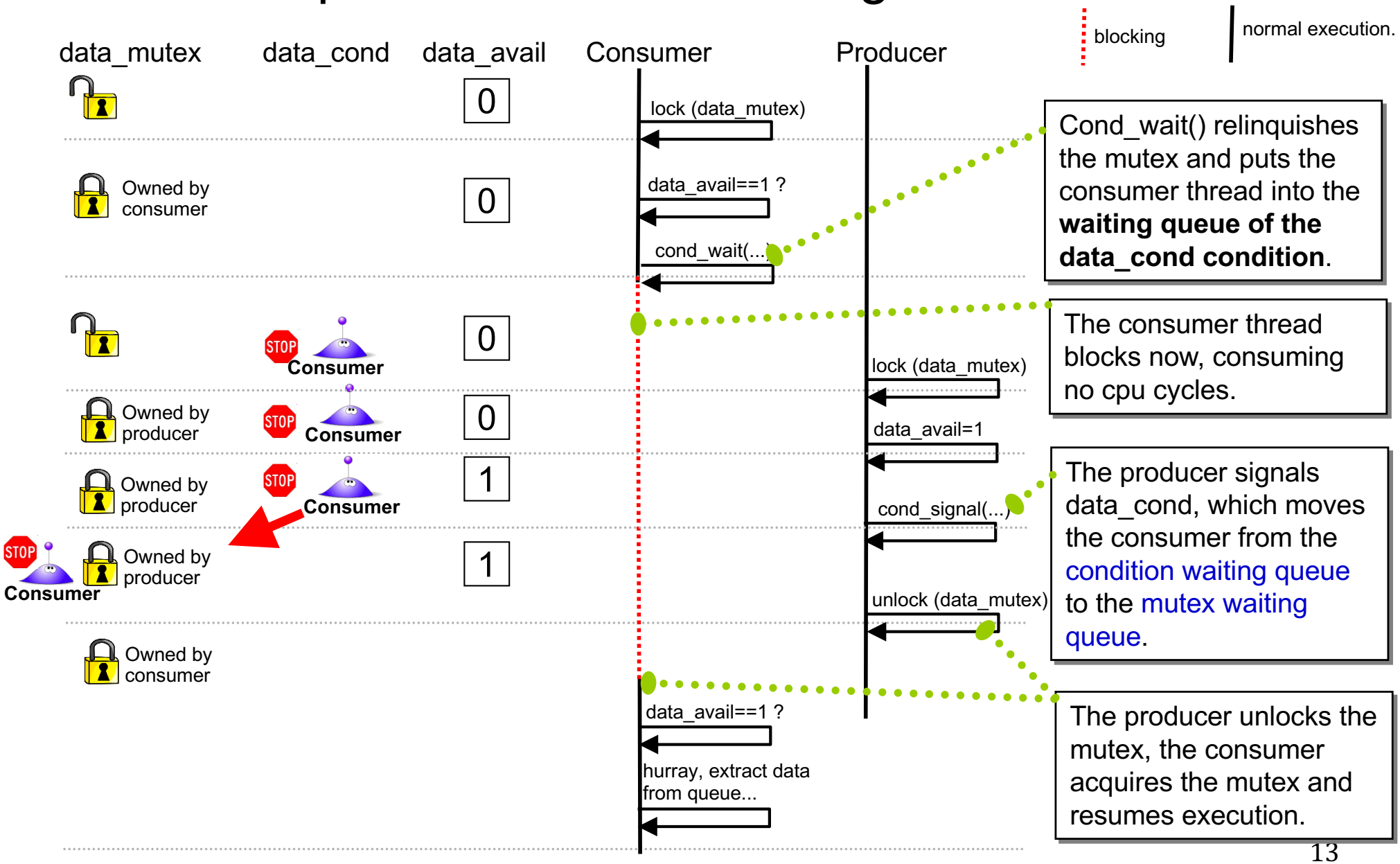
- Variable **`data_cond`** is a condition variable.
- Producer uses **`data_cond`** to signal consumer that new data is available.
- Will unblock a blocking consumer.
- Has no effect if no consumer is blocking.
  - The signal is '**lost**' (which is ok if nobody is waiting).

With Condition Variables (cont.)

```
void *consumer(void *)
{
  pthread_mutex_lock(&data_mutex);

  while( data_avail == 0 ) {
    // sleep on condition variable:
    pthread_cond_wait(&data_cond, &data_mutex);
  }

  // woken up, execute critical section:
  Extract data from queue;
  if (queue is empty)
     data_avail = 0;

  pthread_mutex_unlock(&data_mutex);

  consume_data();
}
```

- Consumer acquires lock.
- Checks **`data_avail`** for available data.
- If no data is available, the consumer blocks using pthread_cond_wait().
  - will relinquish the mutex!
  - otherwise producer could not produce!
- Once the producer signals **`data_cond`** and relinquishes the mutex, the consumer will be unblocked.
  - Holds the mutex again!

Example: Consumer blocking on condition



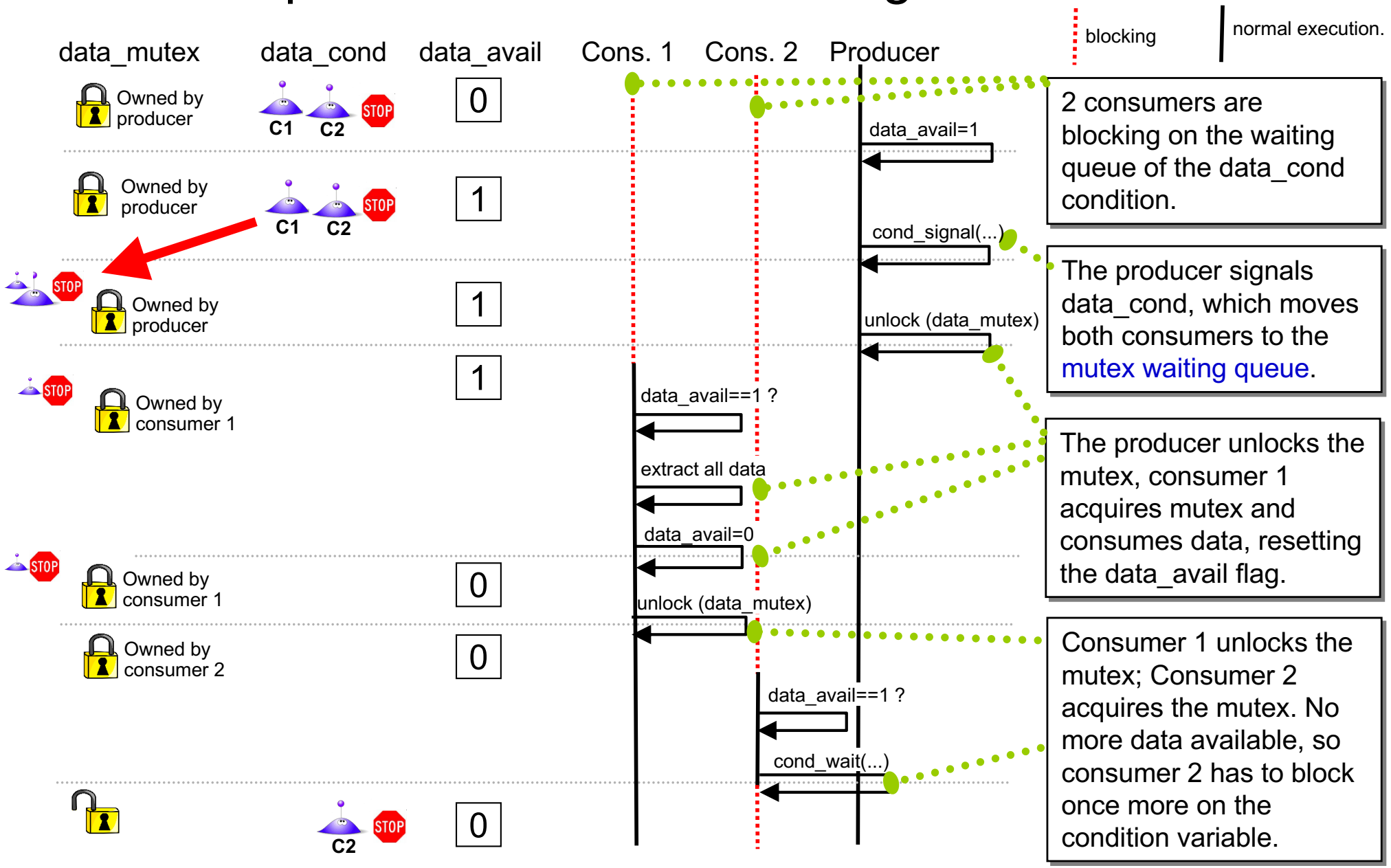
With Condition Variables (cont.)

```
void *consumer(void *)
{
  pthread_mutex_lock(&data_mutex);

  while( data_avail == 0 ) {
    // sleep on condition variable:
    pthread_cond_wait(&data_cond, &data_mutex);
  }

  // woken up, execute critical section:
  Extract data from queue;
  if (queue is empty)
     data_avail = 0;

  pthread_mutex_unlock(&data_mutex);

  consume_data();
}
```

- The while loop is needed in case of >1 consumers.
- If >1 consumers wake up, only one will get the mutex and consume the data item.
- Thereafter the next consumer might acquire the mutex.
  - Data item is consumed now!!
- **Re-check** of condition necessary!

Note:
- The producer-consumer example requires a second condition variable to signal the buffer empty condition to the producer.
- This has been omitted here for brevity.

Example: 2 consumers blocking on condition



Outline

- Further Lock-based Thread Synchronization ✓
  - Barriers ✓
  - Condition variables ✓