Synchronization Primitives pt 1. (ch. 30+31+32)

Operating Systems

Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Condition Variables

- Many cases a thread wishes to wait until a certain **condition**
- e.g., waiting for another thread to complete
  - Often called a `join()`

- Shared variable: works, but hugely inefficient

How should a thread wait for a condition?

Condition Variables

- **Waiting** on a condition
  - Thread puts itself in a queue until some state of execution
- **Signaling** on a condition
  - Some other thread can wake waiting thread

Condition Variables

- **Waiting** on a condition
  - Thread puts itself in a queue until some state of execution
- **Signaling** on a condition
  - Some other thread can wake waiting thread

- Name is a bit misleading
  - More of a queue
  - We are responsible for the actual "condition"

Definitions and Routines

- Declare a condition variable:

```
pthread_cond_t cv;
```

- Operations:

```
// wait:
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
// signal:
pthread_cond_signal(pthread_cond_t *c);
```

- Wait call takes **mutex** as a parameter
  - Caller must be its **owner** (have it locked)
  - Releases the lock, puts caller to sleep
  - On wake up, re-acquires lock and returns

Parent Waiting For Child

- Two threads:
  - **Parent**:
    - Creates child thread
    - Waits on CV until child completes
  - **Child**:
    - Prints a message ("child")
    - Wakes parent by signaling on CV

Parent Waiting For Child

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
3
4  void* child(void* arg) {
5      printf("child\n");
6      thr_exit();
7      return NULL;
8  }
9  int main(void) {
10     printf("parent: begin\n");
11     pthread_t p;
12     pthread_create(&p, NULL, child, NULL);
13     thr_join();
14     printf("parent: end\n");
15     return 0;
16 }
```

Parent Waiting For Child

```
1 void thr_exit() {
2     pthread_cond_signal(&c);
3 }
4 void thr_join() {
5     pthread_mutex_lock(&m);
6     pthread_cond_wait(&c, &m);
7     pthread_mutex_unlock(&m);
8 }
```

- Why might this code fail?

Parent Waiting For Child

```
1 void thr_exit() {
2     pthread_cond_signal(&c);
3 }
4 void thr_join() {
5     pthread_mutex_lock(&m);
6     pthread_cond_wait(&c, &m);
7     pthread_mutex_unlock(&m);
8 }
```

- Why might this code fail?
  - Child runs immediately
  - Will signal, but no thread asleep on CV
  - Parent runs, calls wait and gets stuck
  - Solution?

Parent Waiting For Child

```
1 void thr_exit() {
2     pthread_cond_signal(&c);
3 }
4 void thr_join() {
5     pthread_mutex_lock(&m);
6     pthread_cond_wait(&c, &m);
7     pthread_mutex_unlock(&m);
8 }
```

- Why might this code fail?
  - Child runs immediately
  - Will signal, but no thread asleep on CV
  - Parent runs, calls wait and gets stuck
  - Solution? use `done` variable

Parent Waiting For Child

```
1  int done = 0;
2  void thr_exit() {
3      done = 1;
4      pthread_cond_signal(&c);
5  }
6  void thr_join() {
7      pthread_mutex_lock(&m);
8      if (done == 0)
9          pthread_cond_wait(&c, &m);
10     pthread_mutex_unlock(&m);
11 }
```

- Why might this code fail?

Parent Waiting For Child

```
1  int done = 0;
2  void thr_exit() {
3      done = 1;
4      pthread_cond_signal(&c);
5  }
6  void thr_join() {
7      pthread_mutex_lock(&m);
8      if (done == 0)
9          pthread_cond_wait(&c, &m);
10     pthread_mutex_unlock(&m);
11 }
```

- Why might this code fail?
  - Parent calls join, sees `done=0`
  - Interrupted just before wait, context switch to child
  - Child sets done, signal is lost, parent is stuck again
  - Solution?

Parent Waiting For Child

```
1  int done = 0;
2  void thr_exit() {
3      done = 1;
4      pthread_cond_signal(&c);
5  }
6  void thr_join() {
7      pthread_mutex_lock(&m);
8      if (done == 0)
9          pthread_cond_wait(&c, &m);
10     pthread_mutex_unlock(&m);
11 }
```

- Why might this code fail?
  - Parent calls join, sees `done=0`
  - Interrupted just before wait, context switch to child
  - Child sets done, signal is lost, parent is stuck again
  - Solution? hold lock while signaling

Parent Waiting For Child

```
1  int done = 0;
2  void thr_exit() {
3      pthread_mutex_lock(&m);
4      done = 1;
5      pthread_cond_signal(&c);
6      pthread_mutex_unlock(&m);
7  }
8  void thr_join() {
9      pthread_mutex_lock(&m);
10     while (done == 0)
11         pthread_cond_wait(&c, &m);
12     pthread_mutex_unlock(&m);
13 }
```

- Additionally, check variable in a loop
  - Condition variable may signal unexpectedly
  - Also crucial for more than 2 threads

Covering Conditions

- Memory allocator implementation
- Assume zero bytes are free:
  - Thread A calls `allocate(100)`
  - Thread B calls `allocate(10)`
  - Both A and B wait on the condition
  - Thread C calls `free(50)`

Covering Conditions

- Memory allocator implementation
- Assume zero bytes are free:
  - Thread A calls `allocate(100)`
  - Thread B calls `allocate(10)`
  - Both A and B wait on the condition
  - Thread C calls `free(50)`
    - Which waiting thread wakes up?

Covering Conditions

- Memory allocator implementation
- Assume zero bytes are free:
  - Thread A calls `allocate(100)`
  - Thread B calls `allocate(10)`
  - Both A and B wait on the condition
  - Thread C calls `free(50)`
    - Which waiting thread wakes up?
- Solution: wake up all waiting threads
  - Use `pthread_cond_broadcast()`
  - Performance cost: too many threads might be woken

Producer / Consumer

- Also: **bounded buffer** problem
- **Producer**
  - Produces data items
  - Wishes to place items in a buffer
- **Consumer**
  - Grabs items out of the buffer
  - Consumes items in some way
- e.g., web server consumes HTTP requests (work queue)

Producer / Consumer

- Also used in pipes:
  - `grep foo file.txt | wc -l`
  - `grep` - output lines from `file.txt` containing `foo`
  - `wc -l` - output number of lines from input
  - Shell redirects `grep` standard output to a **pipe**
    - Created by the `pipe` system call
  - Other end connected to standard input of `wc`
  - `grep` - producer, `wc` - consumer
- In-kernel bounded buffer

Producer / Consumer

```
1  int buffer;
2  int count = 0; // initially, empty
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9  int get() {
10     assert(count == 1);
11     count = 0;
12     return buffer;
13 }
```

First Attempt

- What are the two problems?

```
1  pthread_cond_t cond;
2  pthread_mutex_t mutex;
3
4  void produce(int i) {
5      pthread_mutex_lock(&mutex);
6      if (count == 1)
7          pthread_cond_wait(&cond, &mutex);
8      put(i);
9      pthread_cond_signal(&cond);
10     pthread_mutex_unlock(&mutex);
11 }
12 int consume() {
13     pthread_mutex_lock(&mutex);
14     if (count == 0)
15         pthread_cond_wait(&cond, &mutex);
16     int tmp = get();
17     pthread_cond_signal(&cond);
18     pthread_mutex_unlock(&mutex);
19     return tmp;
20 }
```

First Attempt

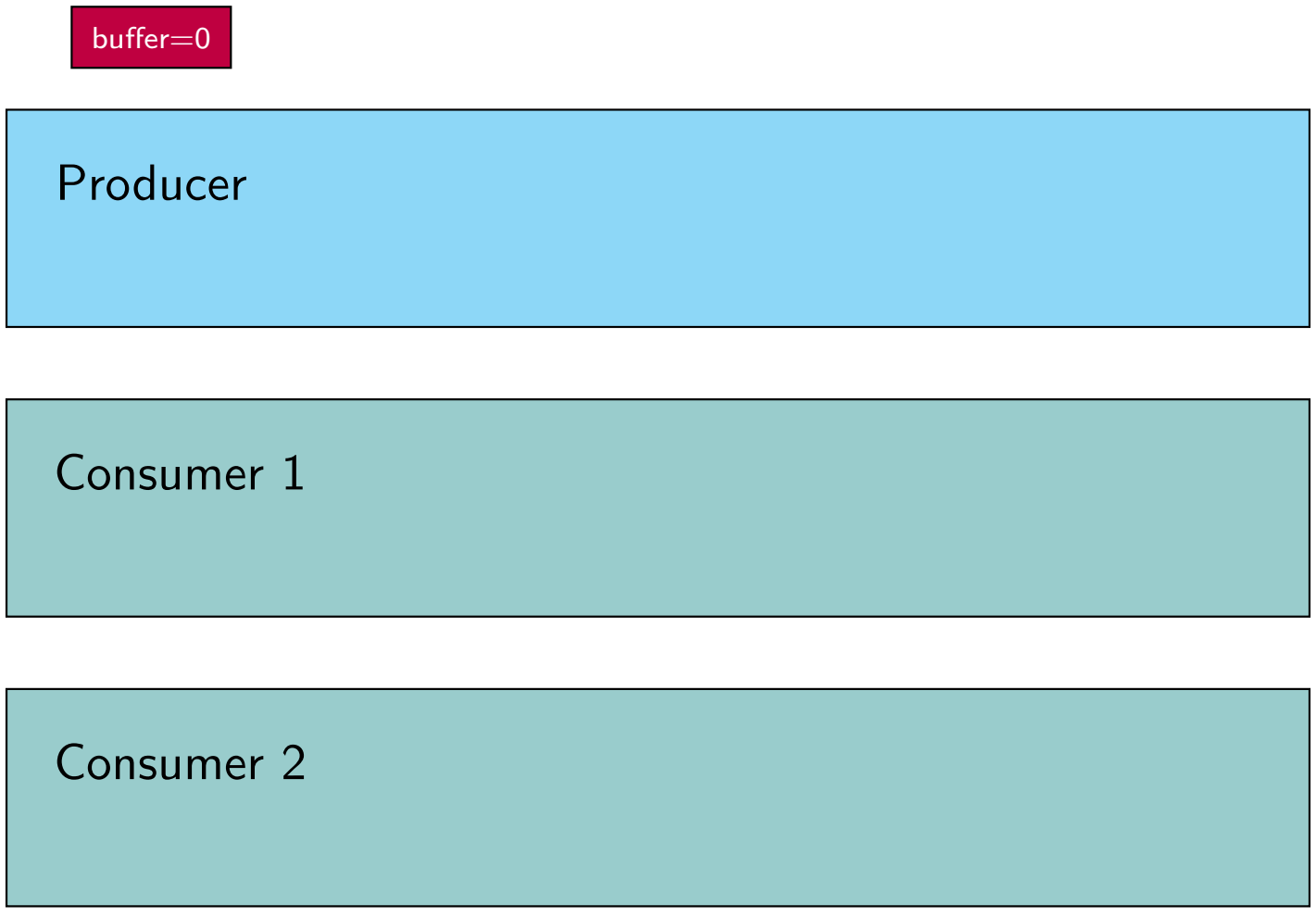
- More than one consumer / producer

First Attempt

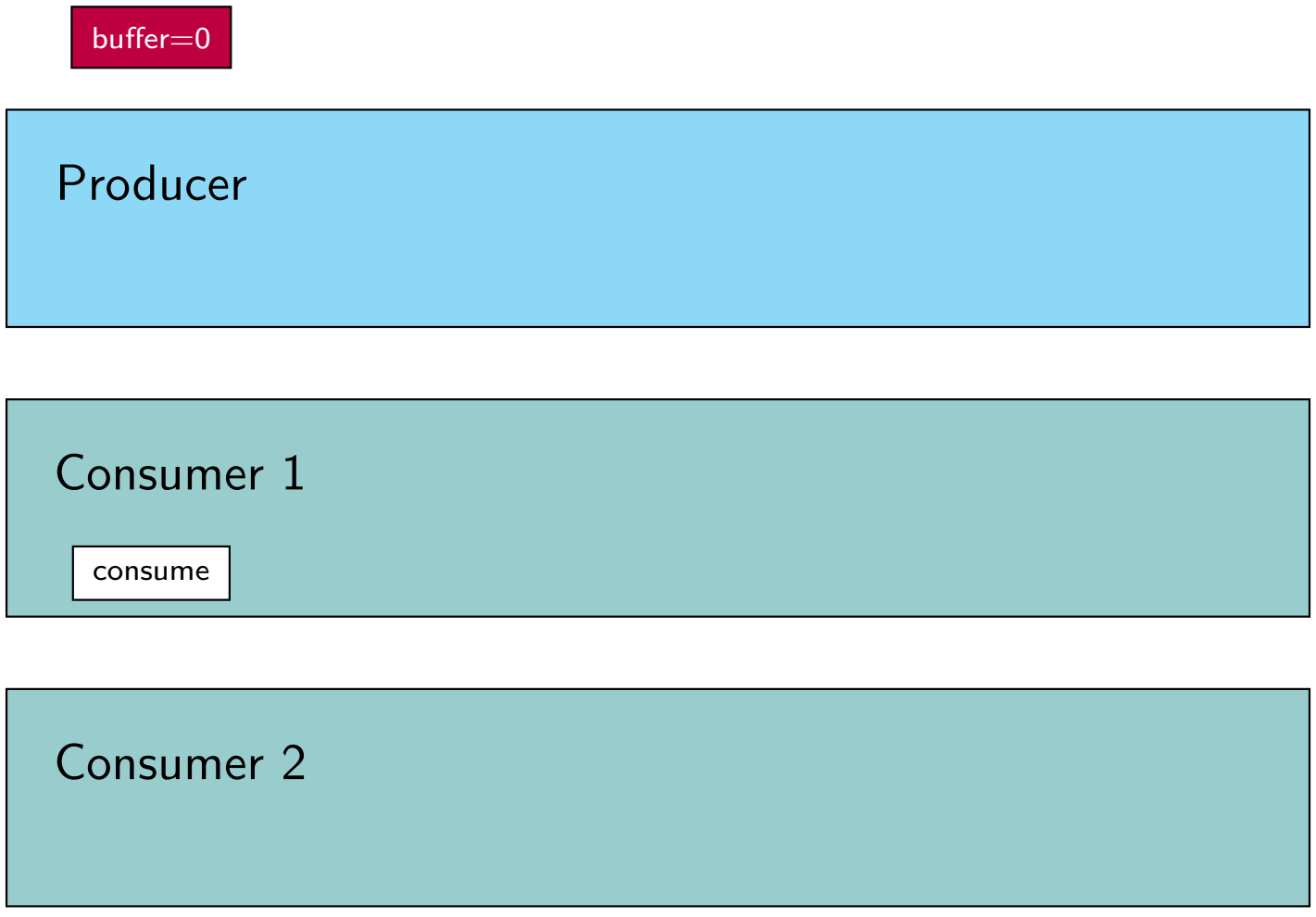
- More than one consumer / producer
- Two consumers:
  - No `while` on CV in `consume()`
  - Can consume when empty!
  - (and the same for `produce()`)

```
1 int consume() {
2     pthread_mutex_lock(&mutex);
3     while (count == 0)
4         pthread_cond_wait(&cond, &mutex);
5     int tmp = get();
6     pthread_cond_signal(&cond);
7     pthread_mutex_unlock(&mutex);
8     return tmp;
9 }
```

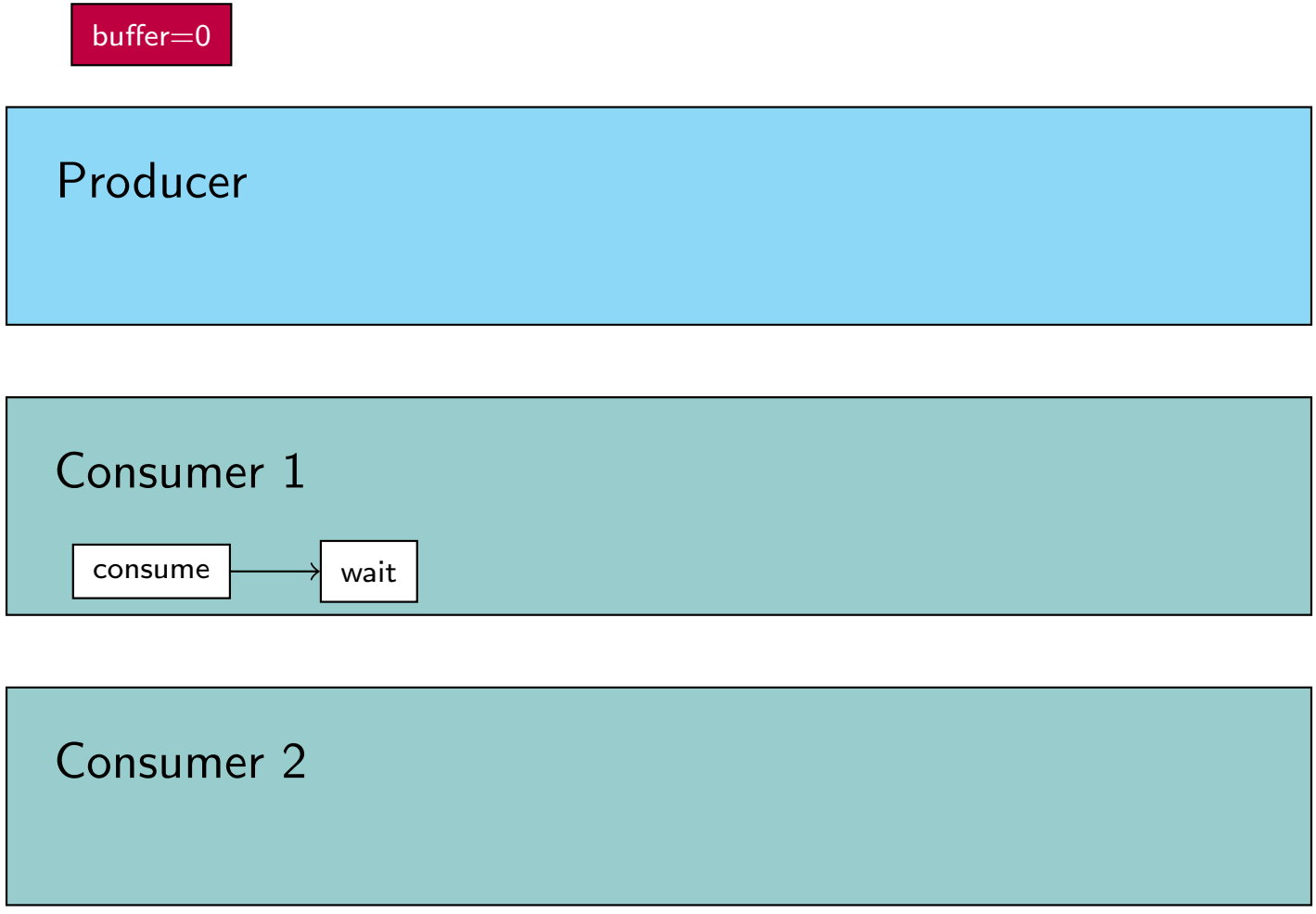
First Attempt



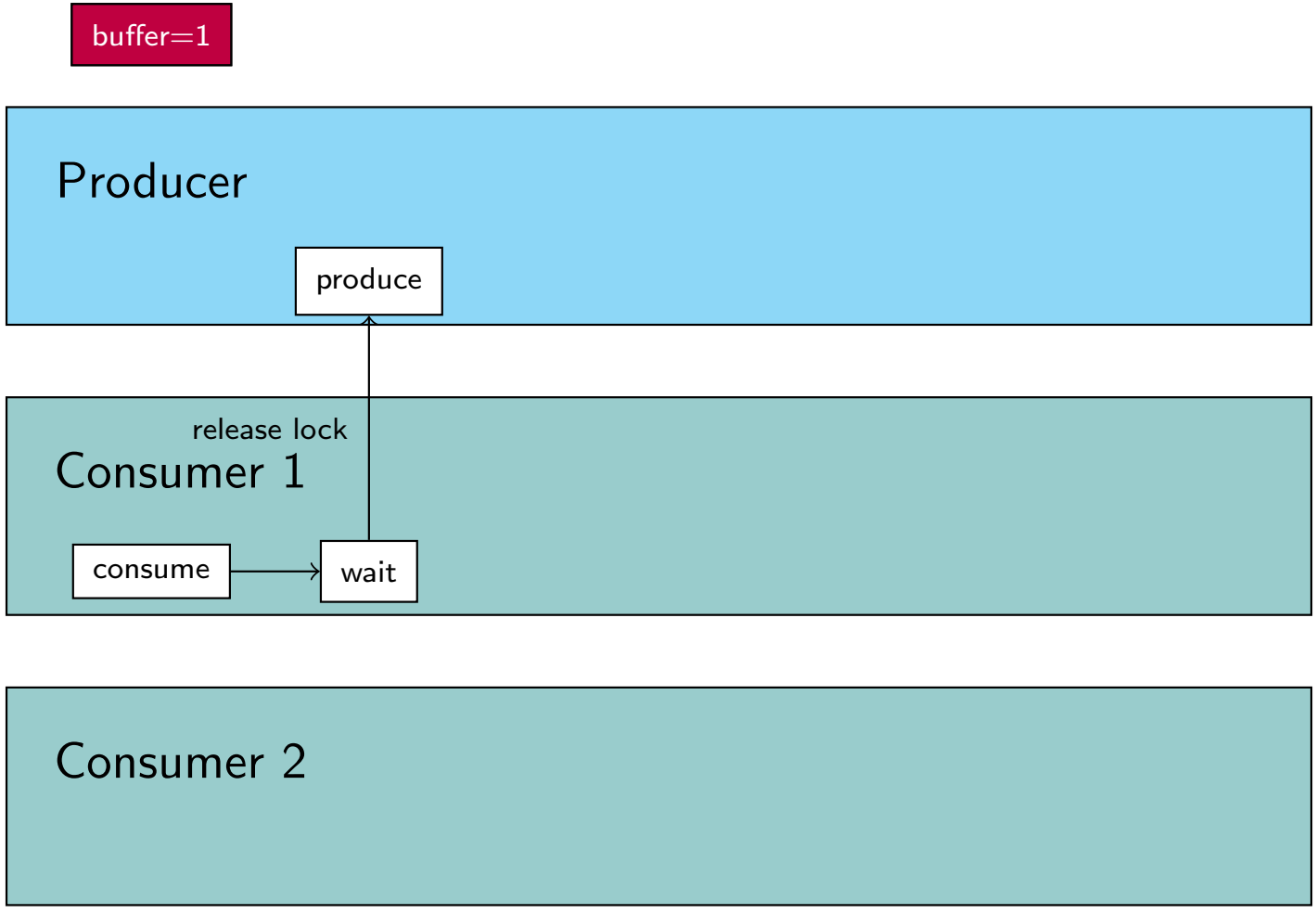
First Attempt



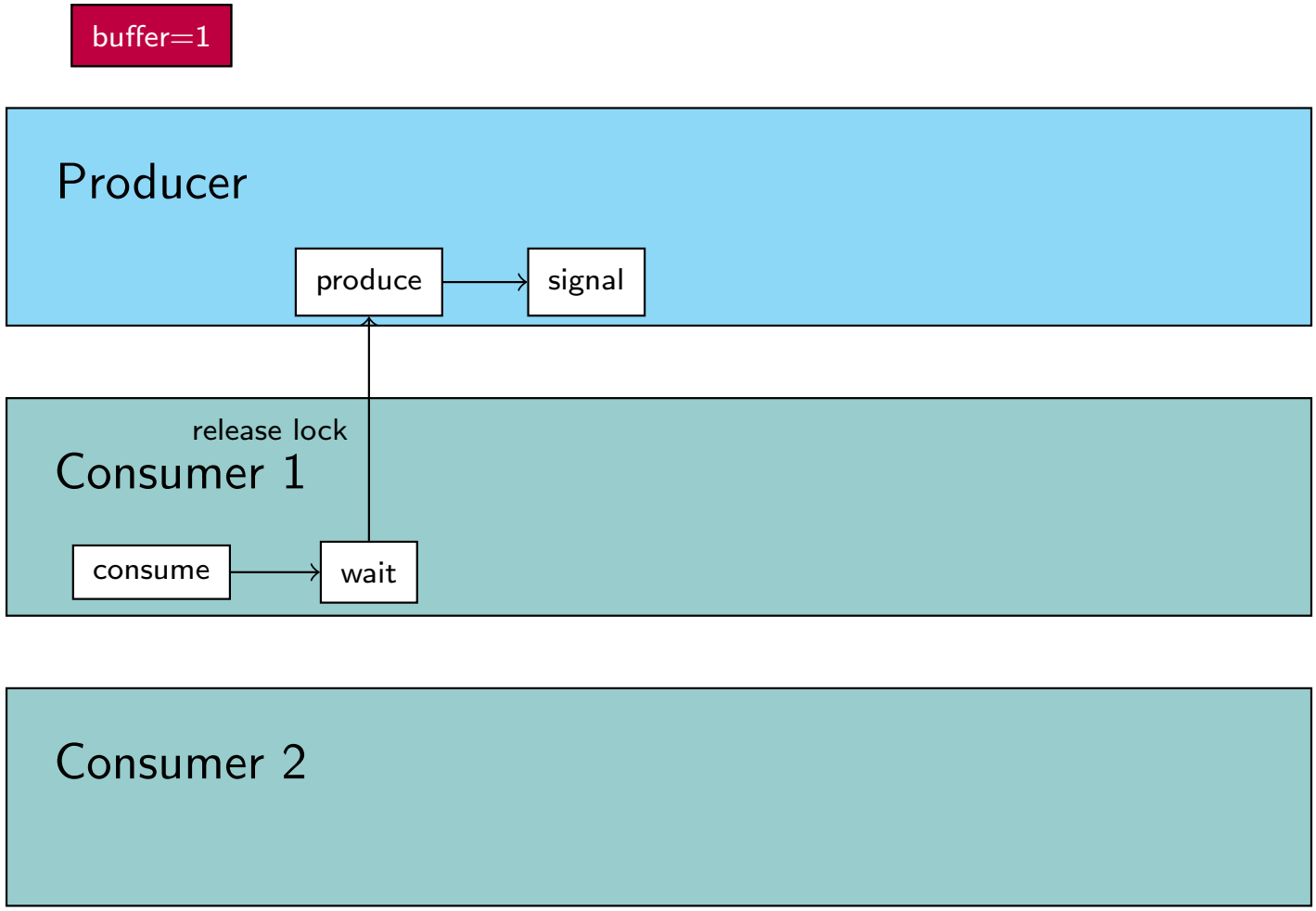
First Attempt



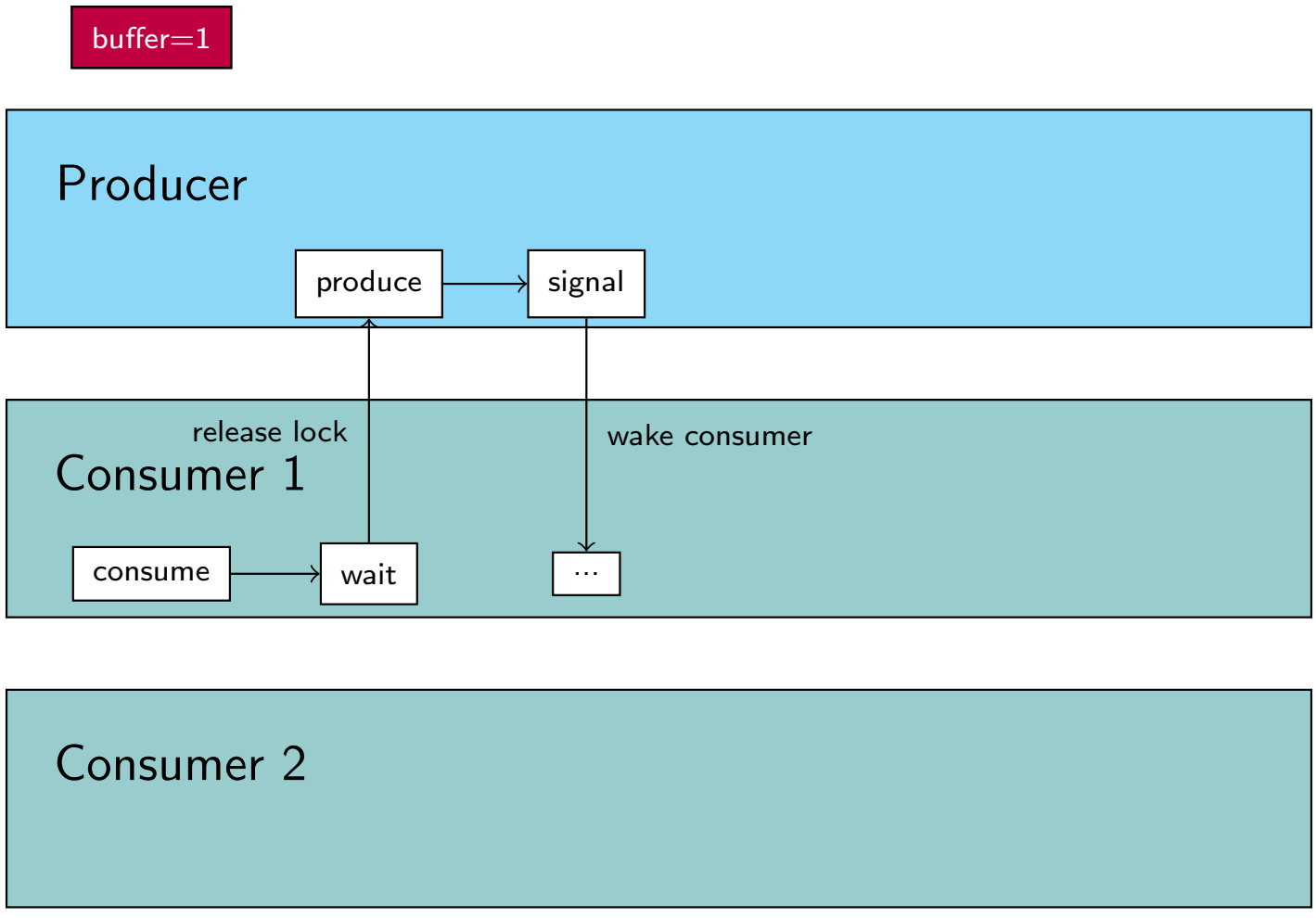
First Attempt



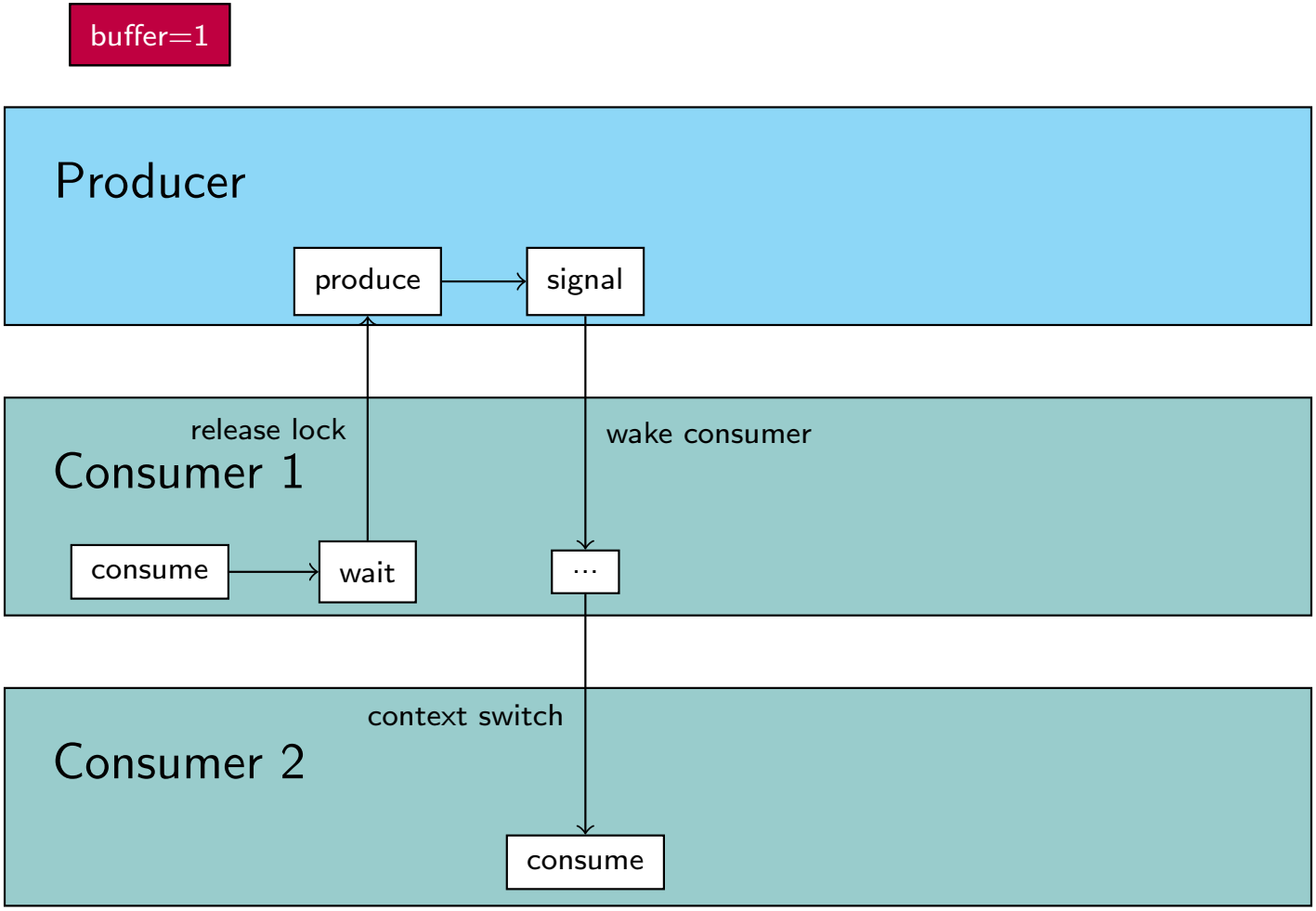
First Attempt



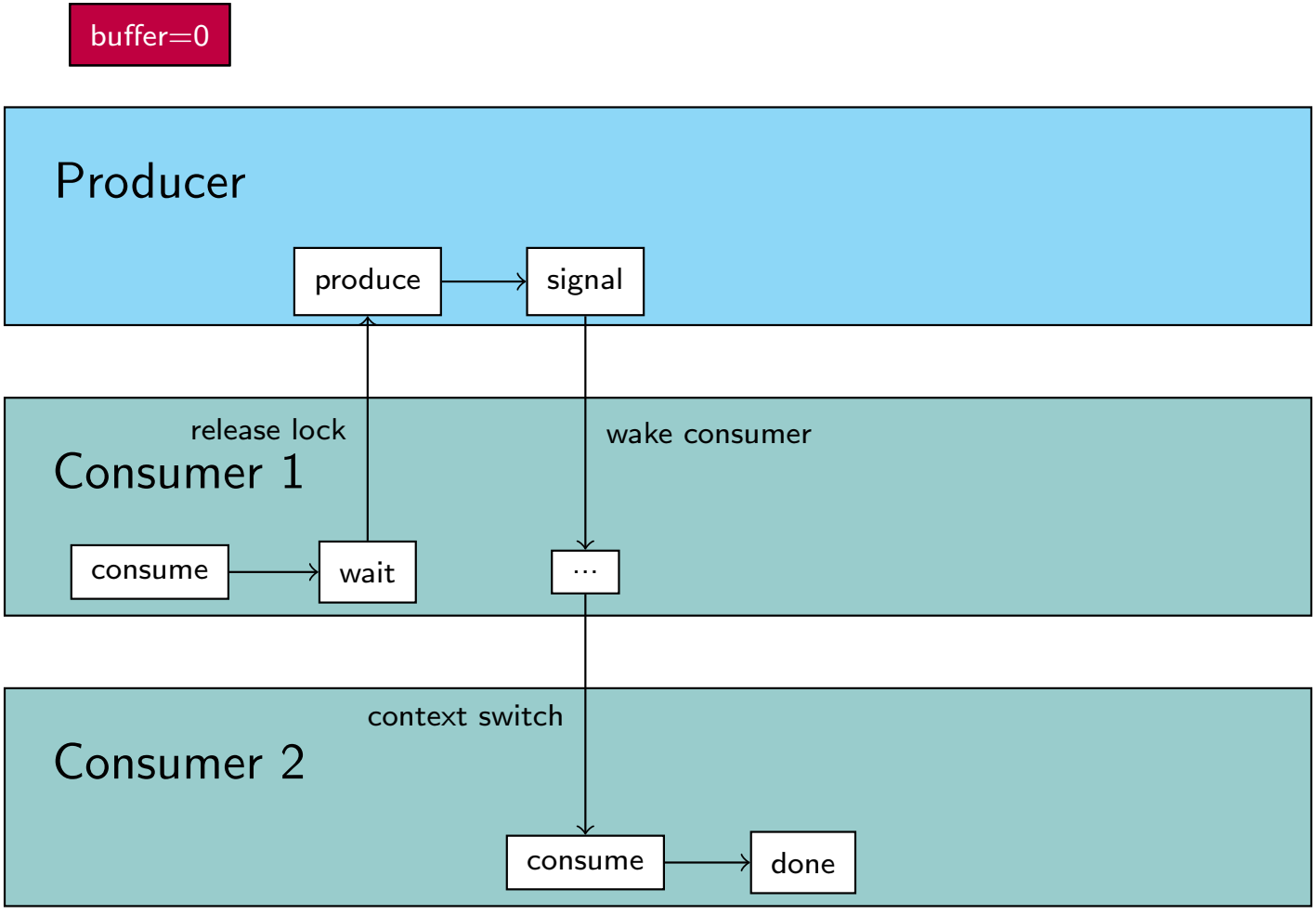
First Attempt



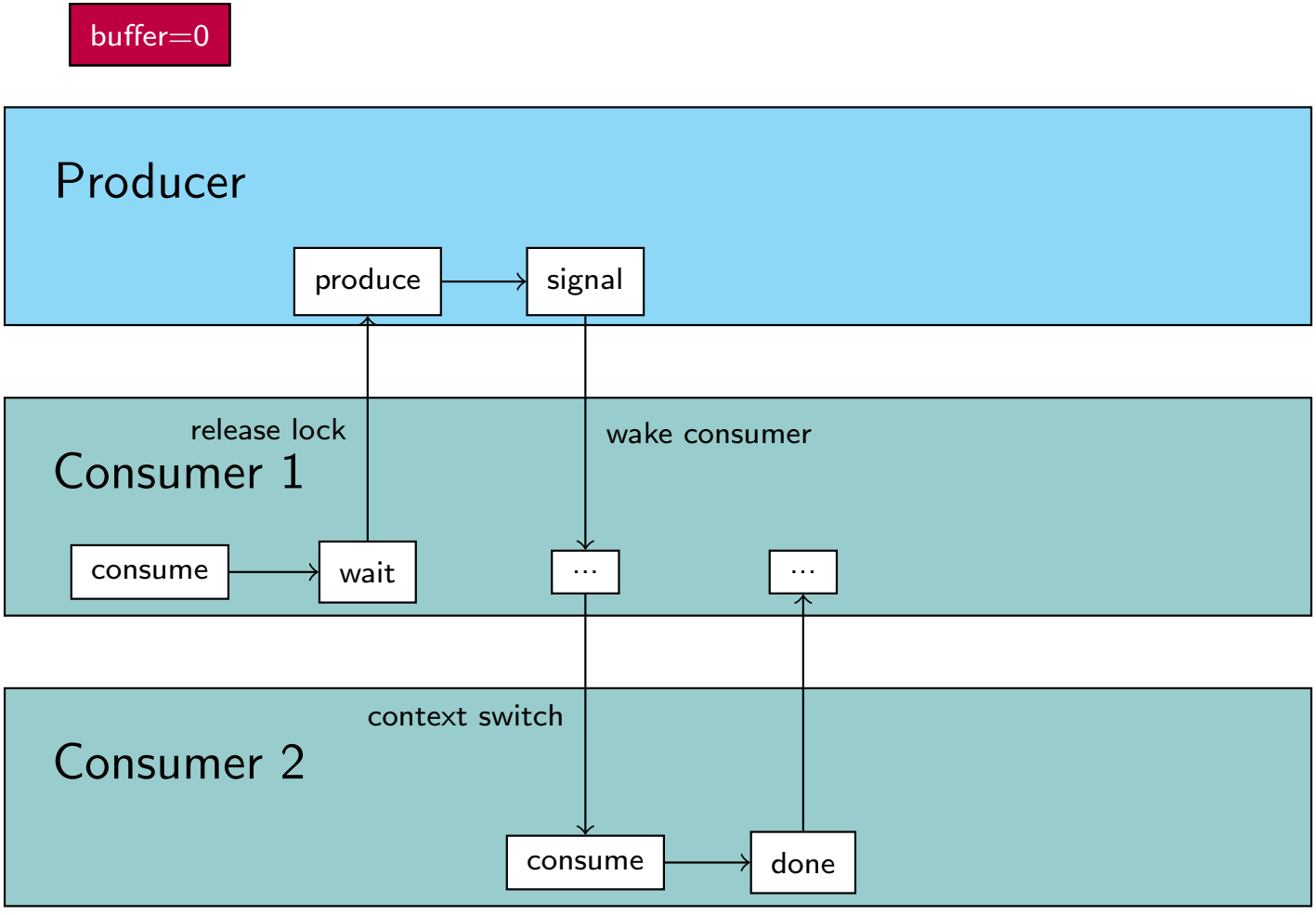
First Attempt



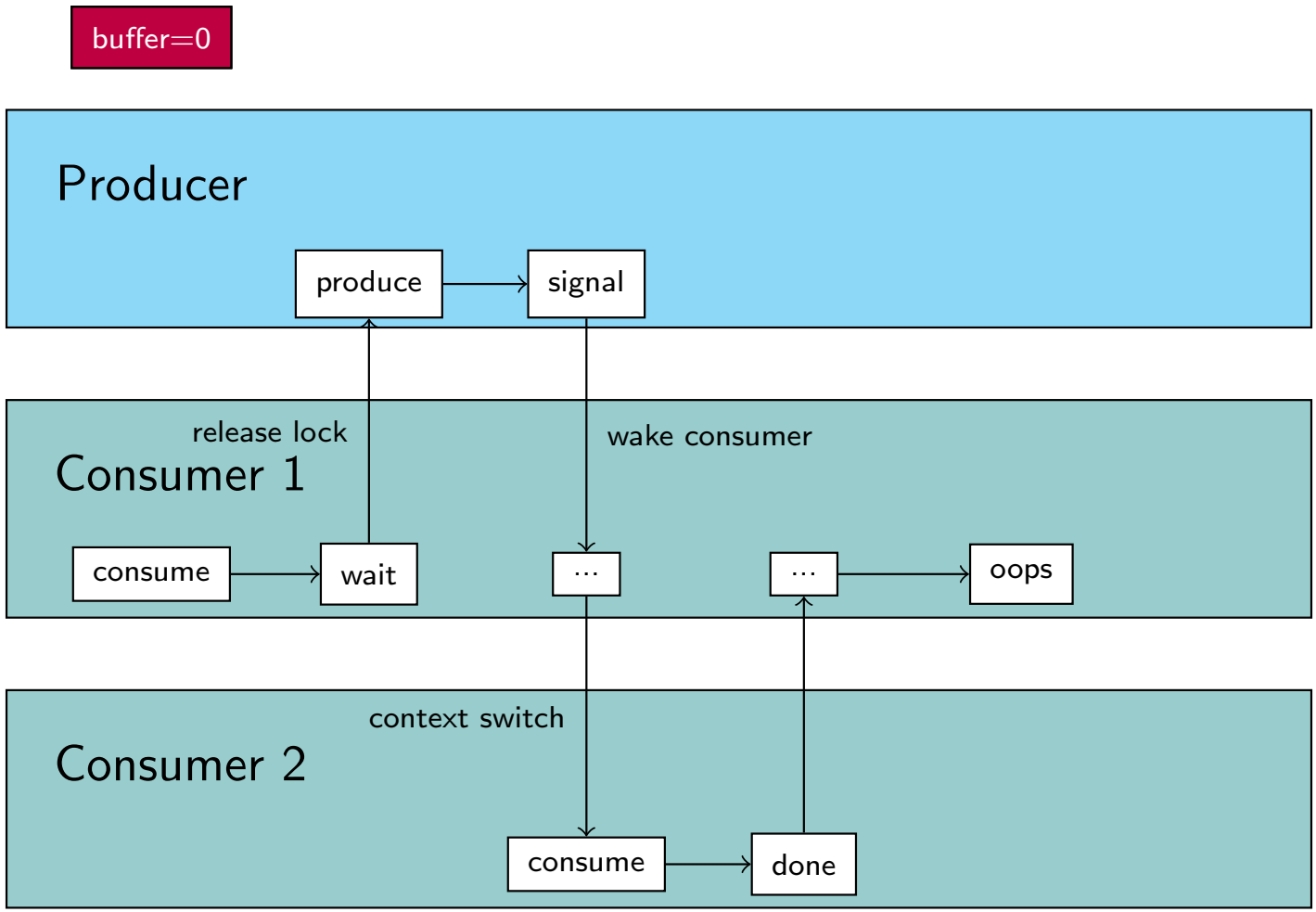
First Attempt



First Attempt



First Attempt



First Attempt

- Only one condition variable!
  - Consumer might wake another consumer
  - Producer might wake another producer
- Solution?

First Attempt

- Only one condition variable!
  - Consumer might wake another consumer
  - Producer might wake another producer
- Solution? use **two** condition variables
  - Producer threads wait on `empty`
  - Consumer threads wait on `full`