G52CON: Concepts of Concurrency

Lecture 9 Monitors

Brian Logan
School of Computer Science
bsl@cs.nott.ac.uk

Outline of this lecture

- solution to semaphores exercise
- limitations of semaphores
- monitors
- example: bounded buffer with monitors
- comparison of semaphores and monitors
- monitors and Java

Problems with semaphores

Semaphores can be used to solve simple mutual exclusion and condition synchronisation problems. However ...

- they are *low-level*: omitting a single ***V*** operation is likely to lead to deadlock; omitting a ***P*** operation may lead to a violation of mutual exclusion
- they are *unstructured*: synchronisation code is typically dispersed throughout the whole program, rather than localised in well-defined regions
- they *confuse conceptually distinct operations*: the same primitives are used to implement mutual exclusion and condition sychronisation

Monitors as abstract data types

A *monitor* is an abstract data type representing a shared resource.

- semaphores can be used to *control access* to a shared resource;
- monitors *encapsulate* the shared resource (usually—we'll look at another approach in the next lecture);

A monitor implements a shared data structure together with the (coarse-grained atomic) operations which manipulate the data structure.

Monitors

Monitors have four components:

- a set of *private variables* which represent the state of the resource;
- a set of *monitor procedures* which provide the public interface to the resource;
- a set of *condition variables* used to implement condition synchronisation; and
- *initialisation code* which initialises the private variables.

Monitor procedures

Monitor procedures manipulate the values of the private monitor variables:

- only the names of monitor procedures are visible outside the monitor
- the only way a process can read or change the value of a private monitor variable is by calling a monitor procedure
  - the private monitor variables are shared by all the monitor procedures
  - monitor procedures may also have their own local variables—each procedure call gets its own copy of these
- statements within monitor procedures (or initialisation code) may not access variables declared outside the monitor (unless passed as arguments to a monitor procedure)

Condition variables

Condition variables are used to delay a process that can't safely execute a monitor procedure until the monitor's state satisfies some boolean condition:

- condition variables are not visible outside the monitor and the only access to them is via *special monitor operations* within monitor procedures

- like semaphores, their values can't be tested or assigned to directly even by the monitor procedures

Synchronisation

Synchronisation within monitors is achieved using monitor procedures and condition variables:

- mutual exclusion is implicit—monitor procedures by definition execute with mutual exclusion;
- condition synchronisation must be programmed explicitly using *condition variables*.

Mutual exclusion

At most *one* instance of *one* monitor procedure may be active in a monitor at a time:

- if one process is executing a monitor procedure and another process calls a procedure of the same monitor, the second process blocks and is placed on the *entry queue* for the monitor
- when the process in the monitor completes its monitor procedure call, mutual exclusion is passed to a blocked process on the entry queue
- entry queues are usually defined to be FIFO, so the first process to block will be the next to one to enter the monitor.

Critical sections

This solves the critical section problem

- if all the shared state in the system is held in private monitor variables; and
- all communication between processes is via calls to monitor procedures; then
- any accesses to any part of the shared state by any process is guaranteed to be mutually exclusive of any other accesses to that part of the state by other processes.

Different classes of critical section can be implemented using a different monitor for each class

Condition synchronisation

The value of a condition variable is a *delay queue* of blocked processes waiting on a condition

- if a call to a monitor procedure can't proceed until the monitor's state satisfies some boolean condition, the process that called the monitor procedure *waits* on the corresponding condition variable

- when another process executes a monitor procedure that makes the condition true, it *signals* to the process(es) waiting on the condition variable

Condition variables are like semaphores used for condition synchronisation

© Brian Logan 2014

Operations on condition variables

We assume that the following operations are defined for a condition variable `v`:

- **`wait`**`(v)` wait at the end of the delay queue for `v`
- **`signal`**`(v)` wake the process at the front of the delay queue for `v` and continue
- **`signal_all`**`(v)` wake all the processes on the delay queue for `v` and continue
- **`empty`**`(v)` true if the delay queue for `v` is empty

© Brian Logan 2014

The `wait` operation

If a process can’t proceed, it blocks on a condition variable `v` by executing:

- `wait(v);`

The blocked process relinquishes exclusive access to the monitor and is appended to the end of the delay queue for `v`.

The **`signal`** operation

Processes blocked on a condition variable `v` are woken up when some *other* process performs a **`signal`** operation on the variable:

- **`signal`**`(v);`

This awakens the process at the front of the delay queue. If the delay queue for `v` is empty, **`signal`** does nothing.

This is *unlike* semaphores, where if no process was waiting on the semaphore, a ***V*** operation increments the semaphore.

Signalling disciplines

When a monitor procedure calls **`signal`** on a condition variable, it wakes up the first blocked process in the delay queue waiting on the condition.

- *Signal and Wait:* the signaller waits until some later time and the signalled process executes now.

- *Signal and Continue:* the signaller continues and the signalled process executes at some later time.

The examples in this lecture use the *signal and continue* signalling discipline.

Bounded Buffer with monitors

```
monitor BoundedBuffer {
    // Private variables …
    Object buf = new Object[n];
    integer out = 0,        // index of first full slot
            in = 0,         // index of first empty slot
            count = 0;      // number of full slots

    // Condition variables ...
    condvar not_full,       // signalled when count < n
            not_empty;      // signalled when count > 0

    // continued ...
```

Bounded Buffer with monitors 2

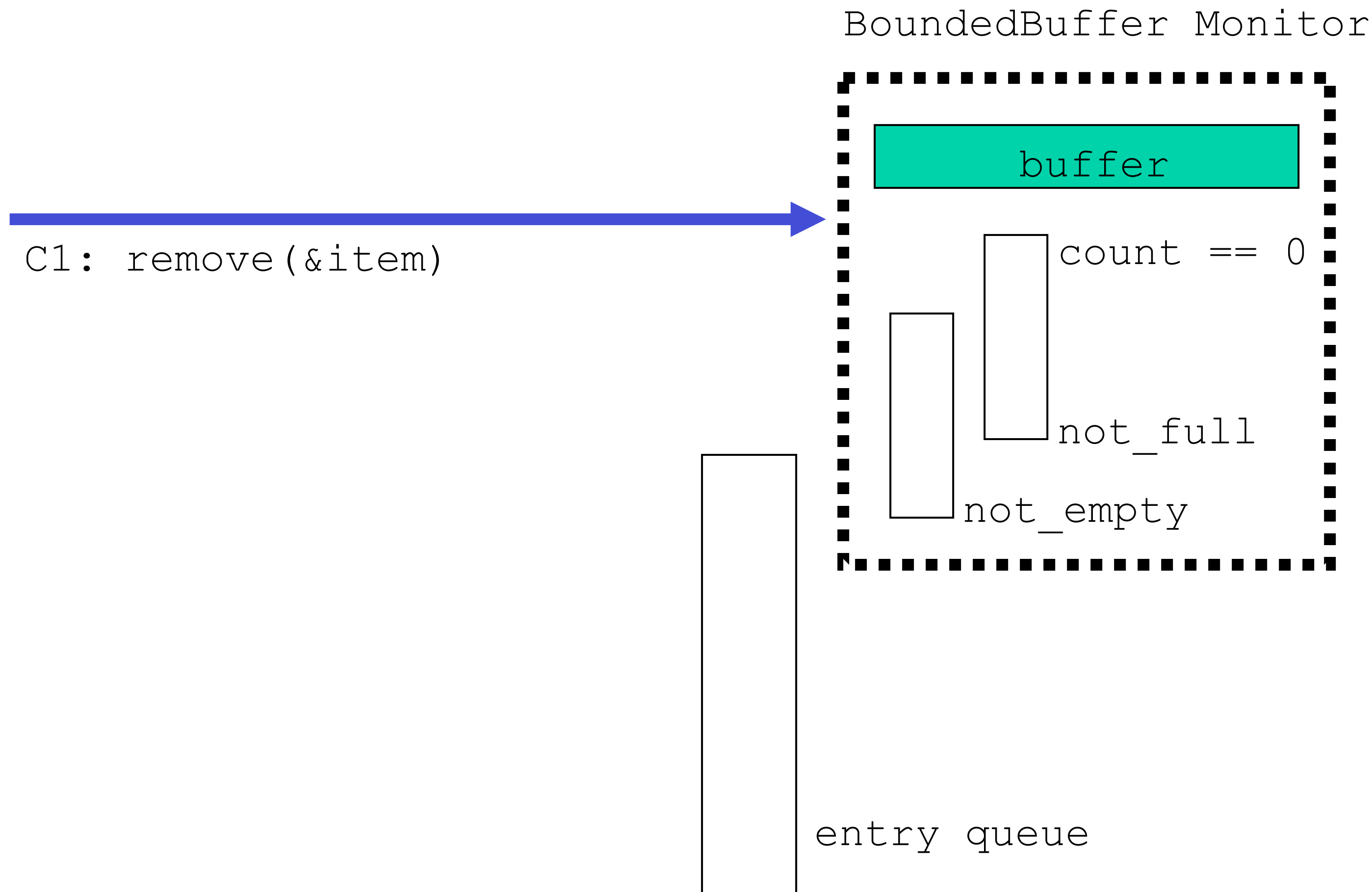
```
// Monitor procedures ...
//(signal & continue signalling discipline)
procedure append(Object data) {
    while(count == n) {
        wait(not_full);
    }
    buf[in] = data;
    in = (in + 1) % n;
    count++;
    signal(not_empty);
}

// continued ...
```

© Brian Logan 2014

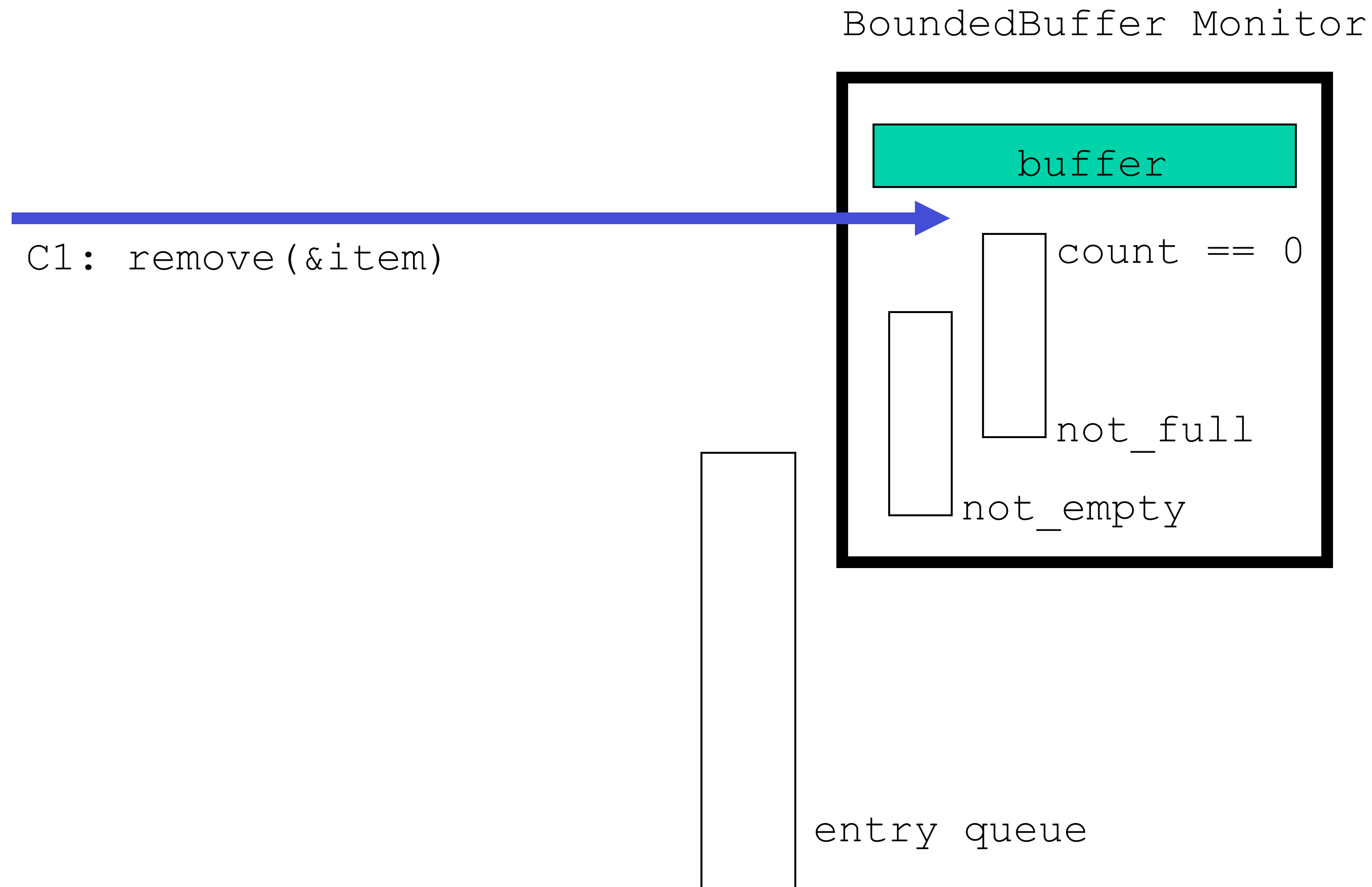
Bounded Buffer with monitors 3

```
    procedure remove(Object &item) {
        while(count == 0) {
            wait(not_empty);
        }
        item = buf[out];
        out = (out + 1) %n;
        count--;
        signal(not_full);
    }
}
```

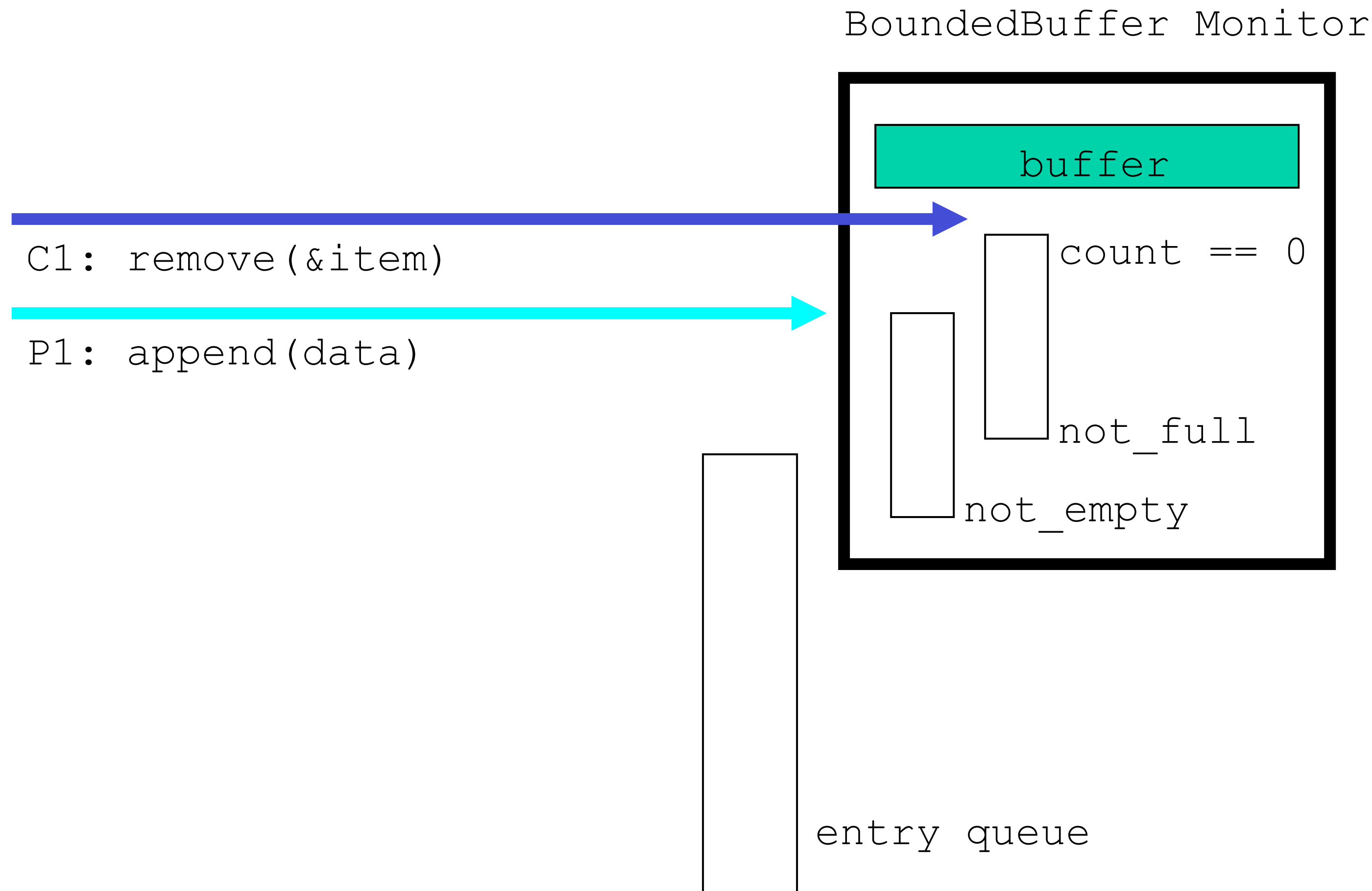
An example trace 1



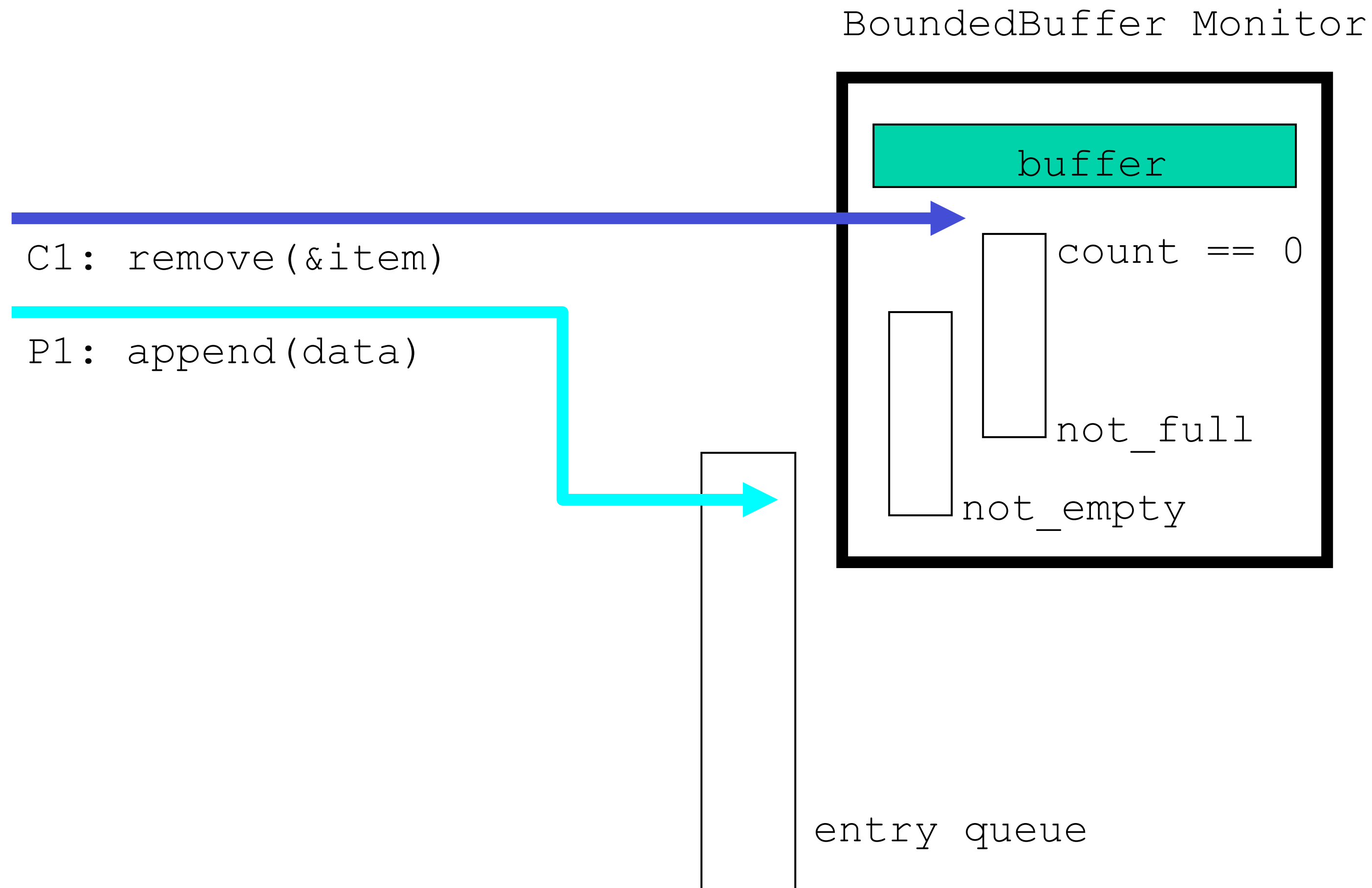
An example trace 2



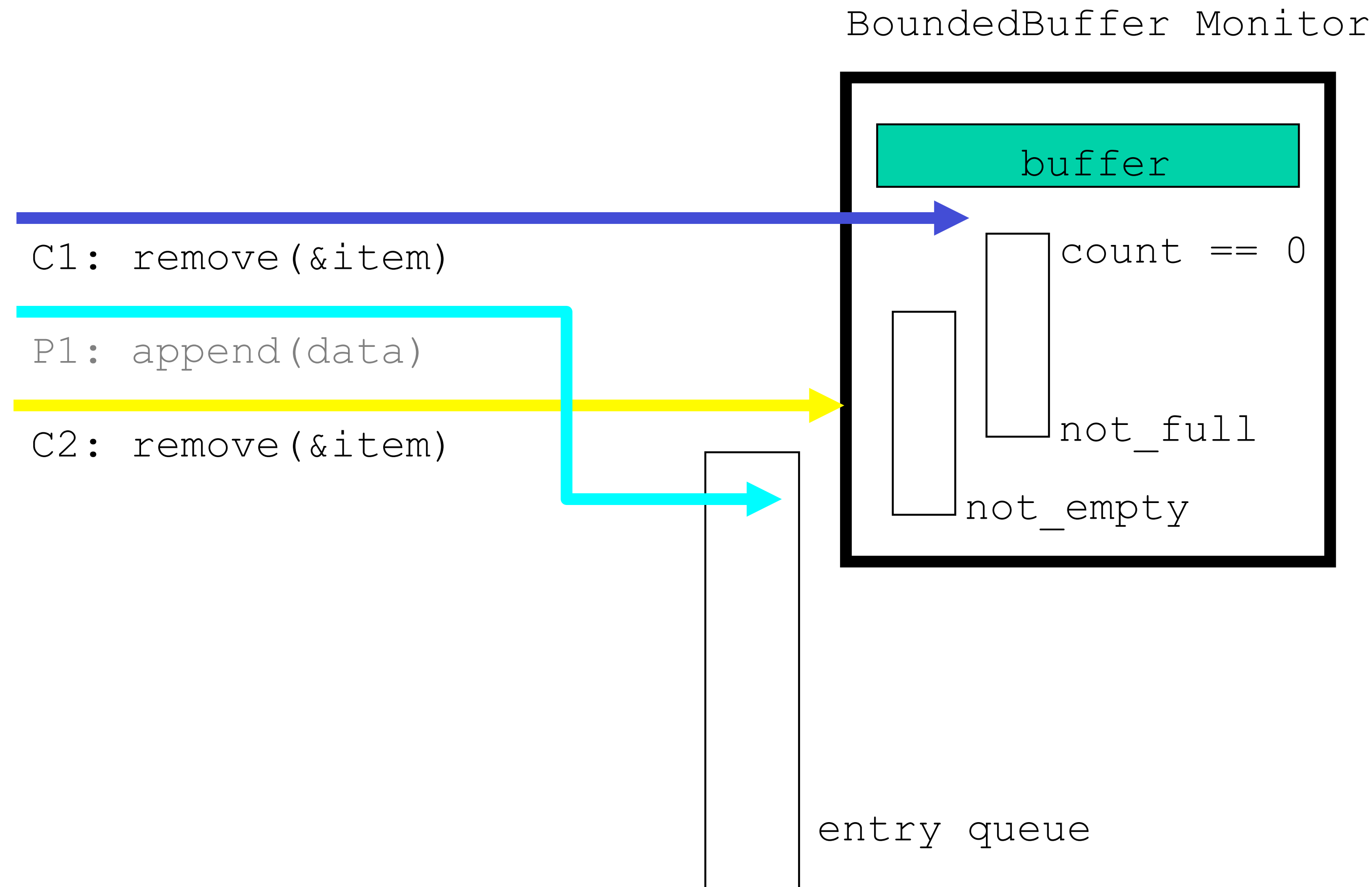
An example trace 3



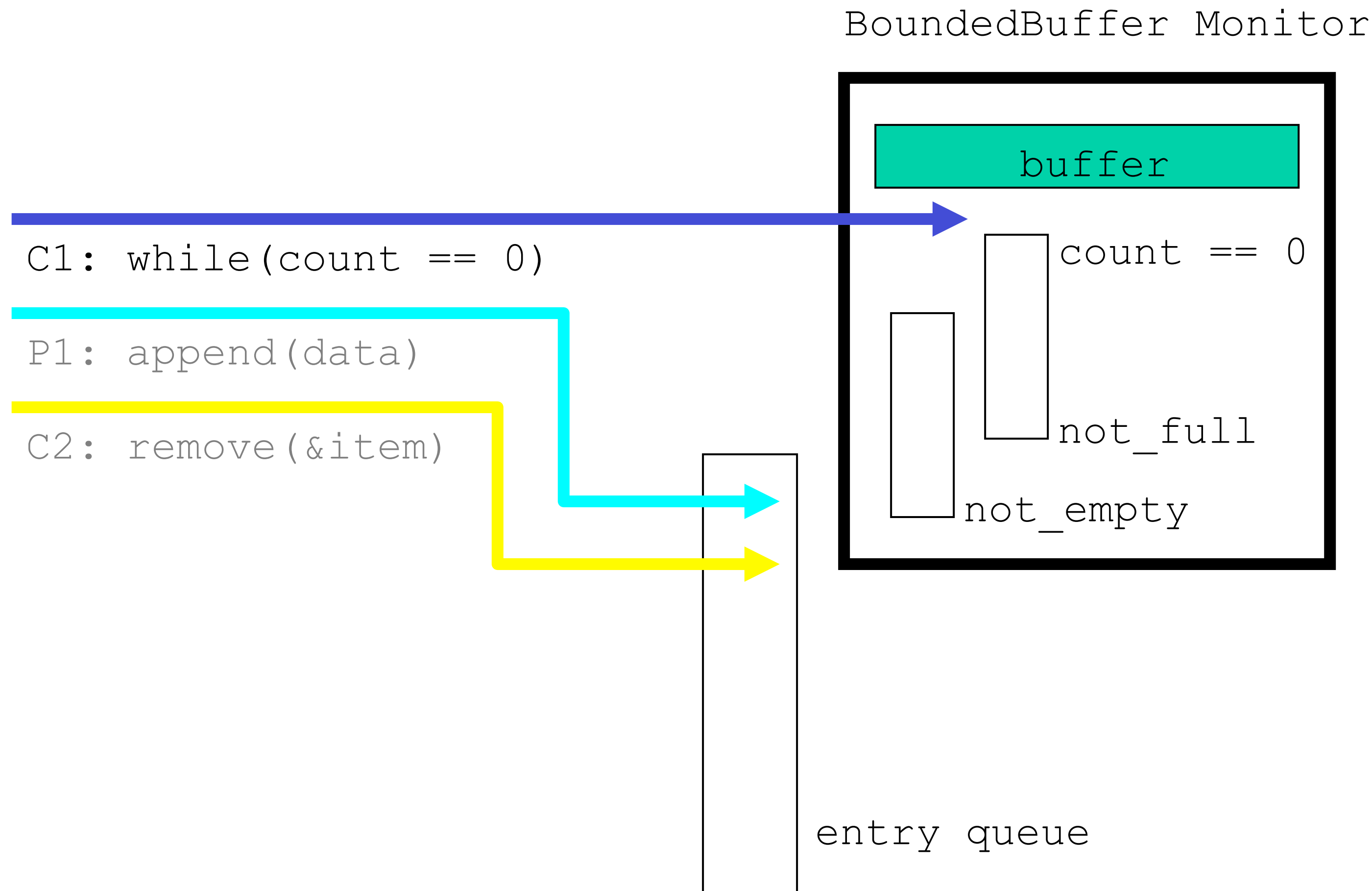
An example trace 4



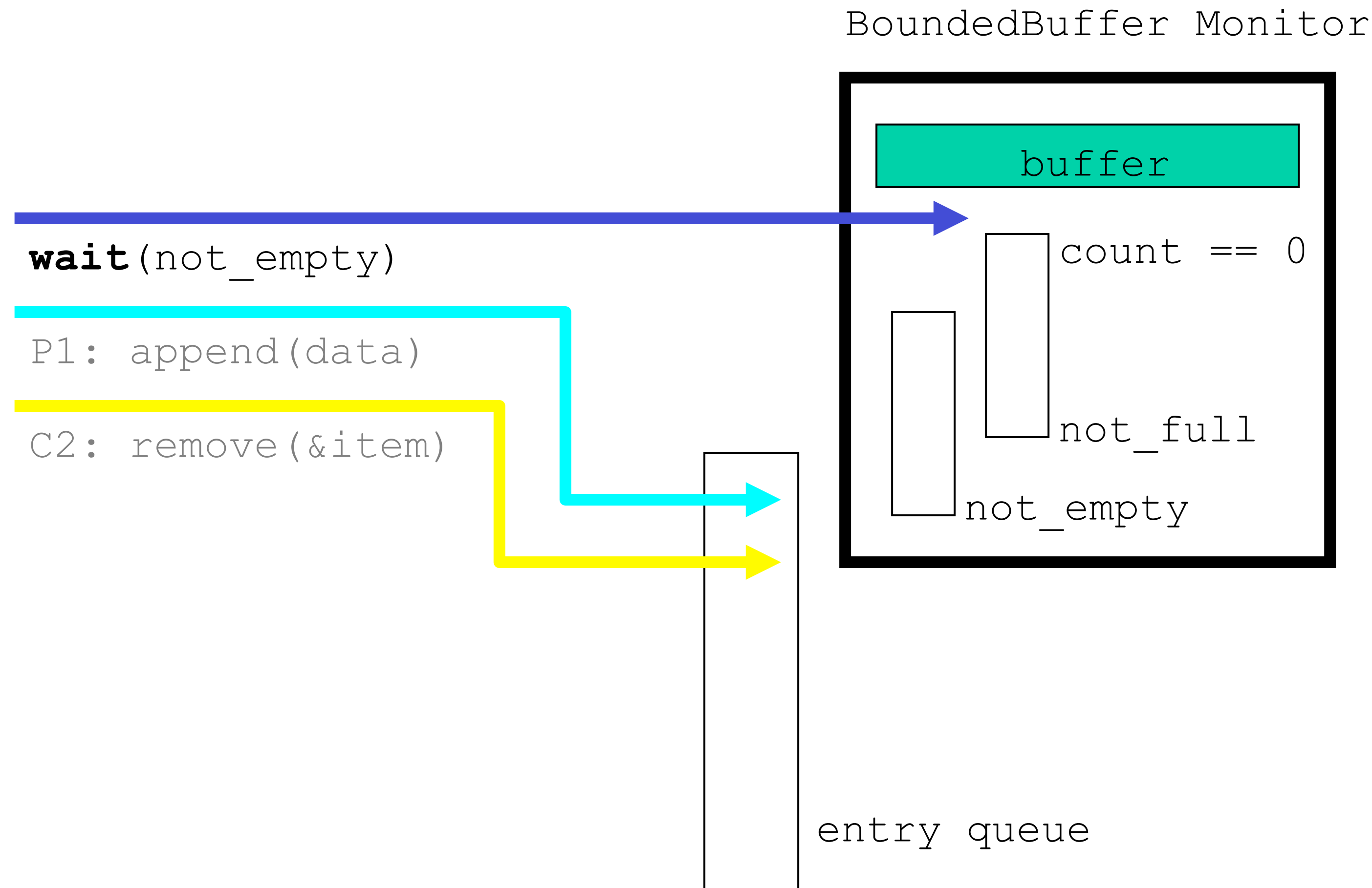
An example trace 5



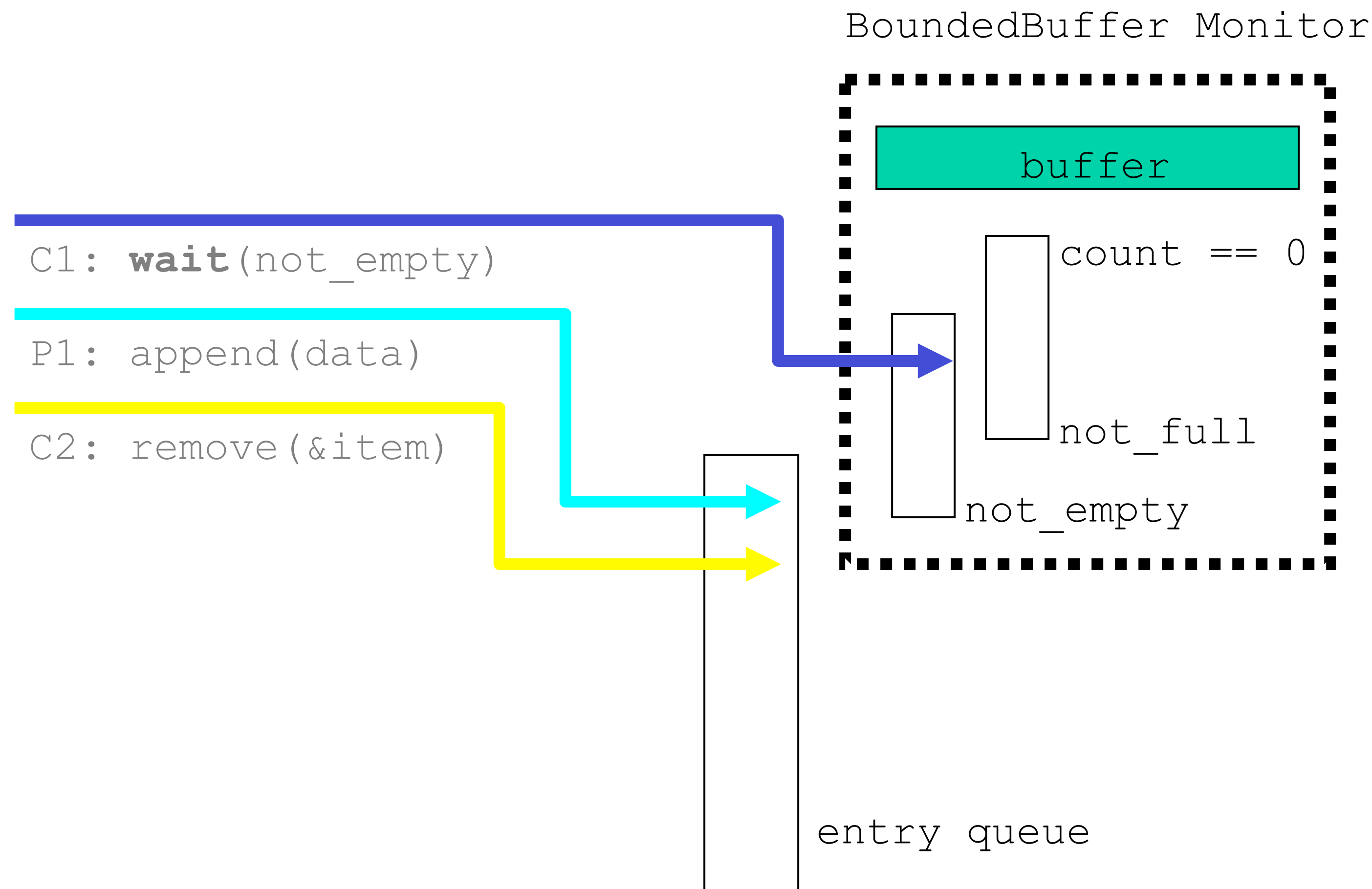
An example trace 6



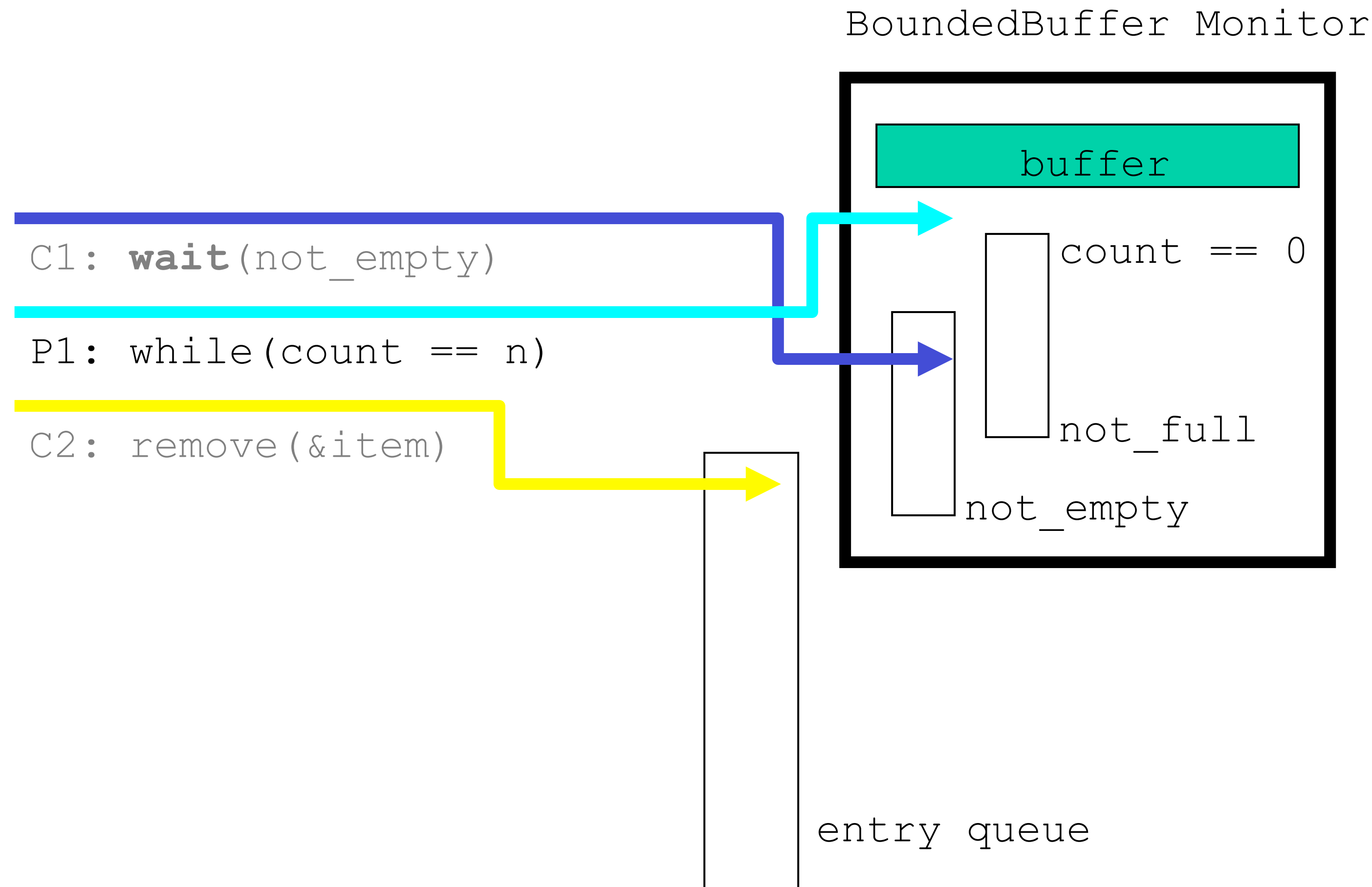
An example trace 7



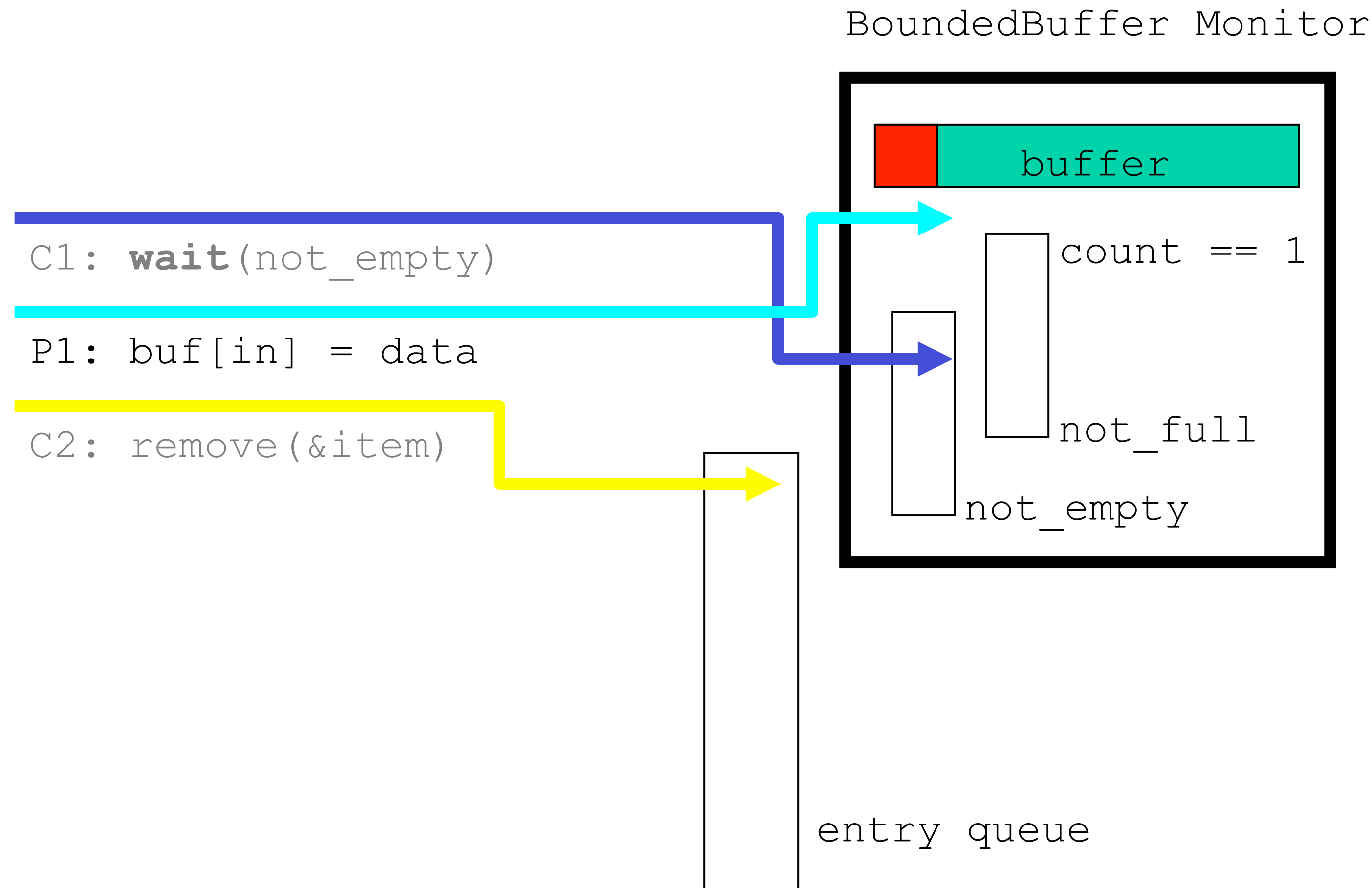
An example trace 8



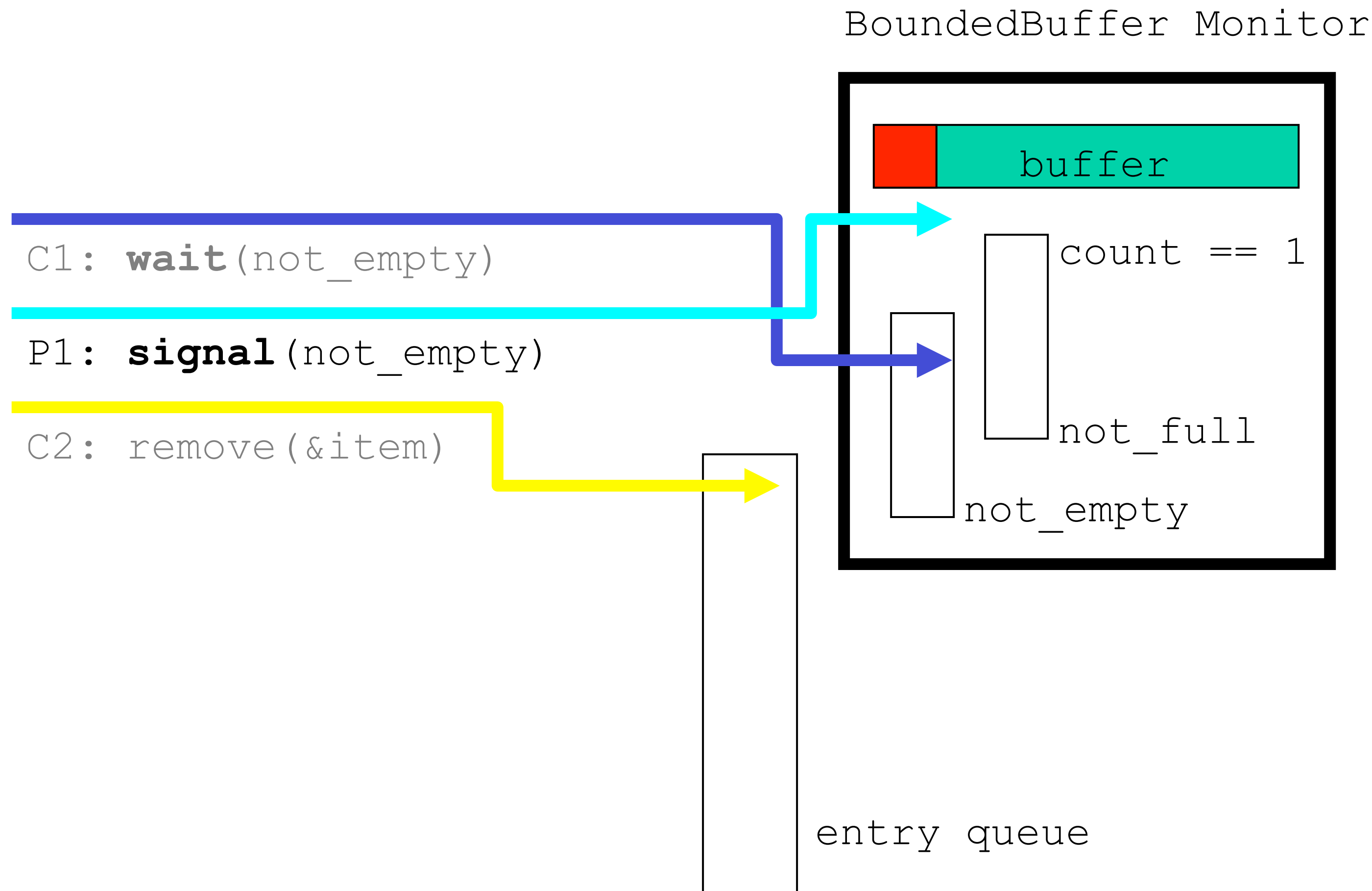
An example trace 9



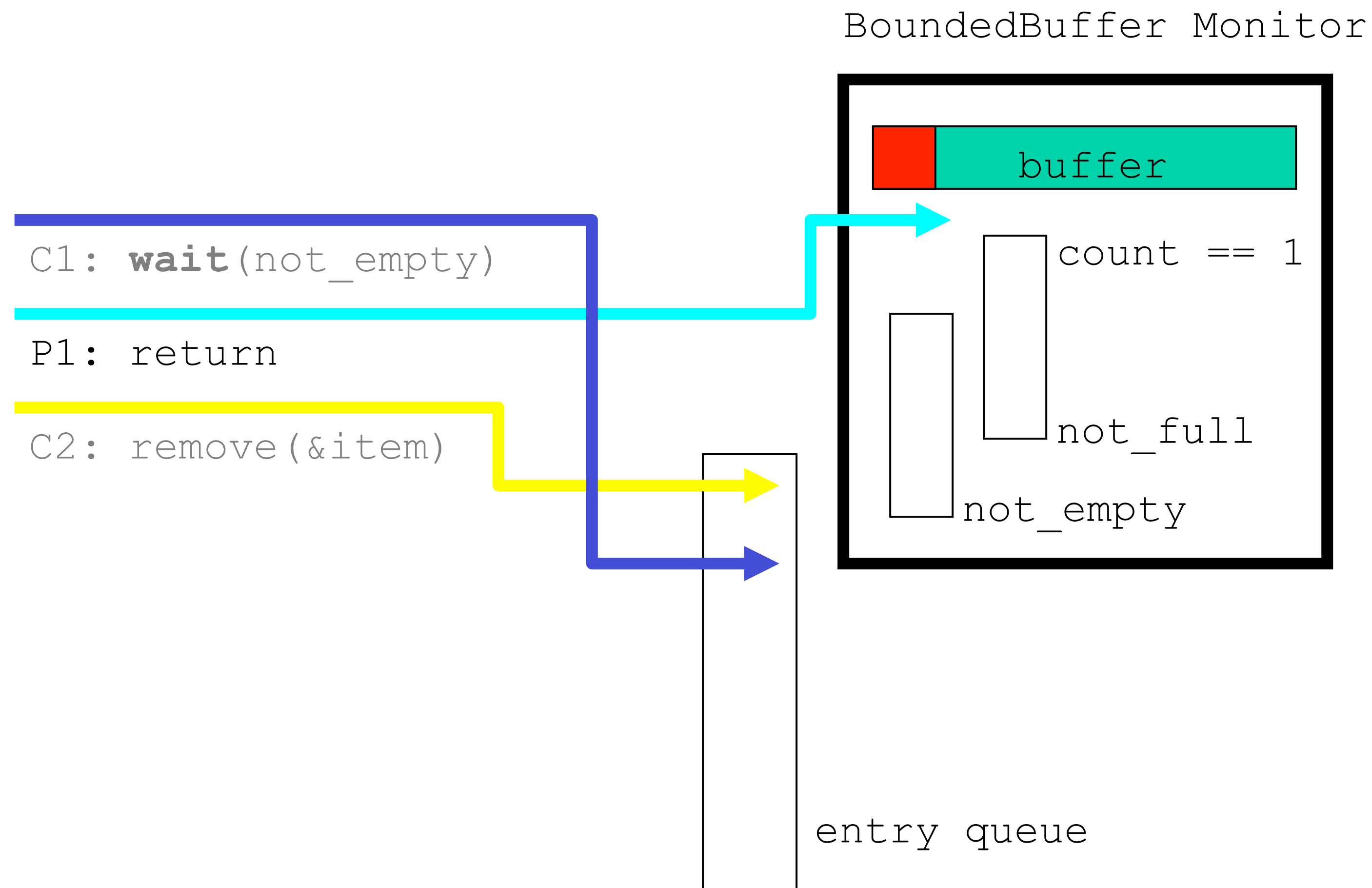
An example trace 10



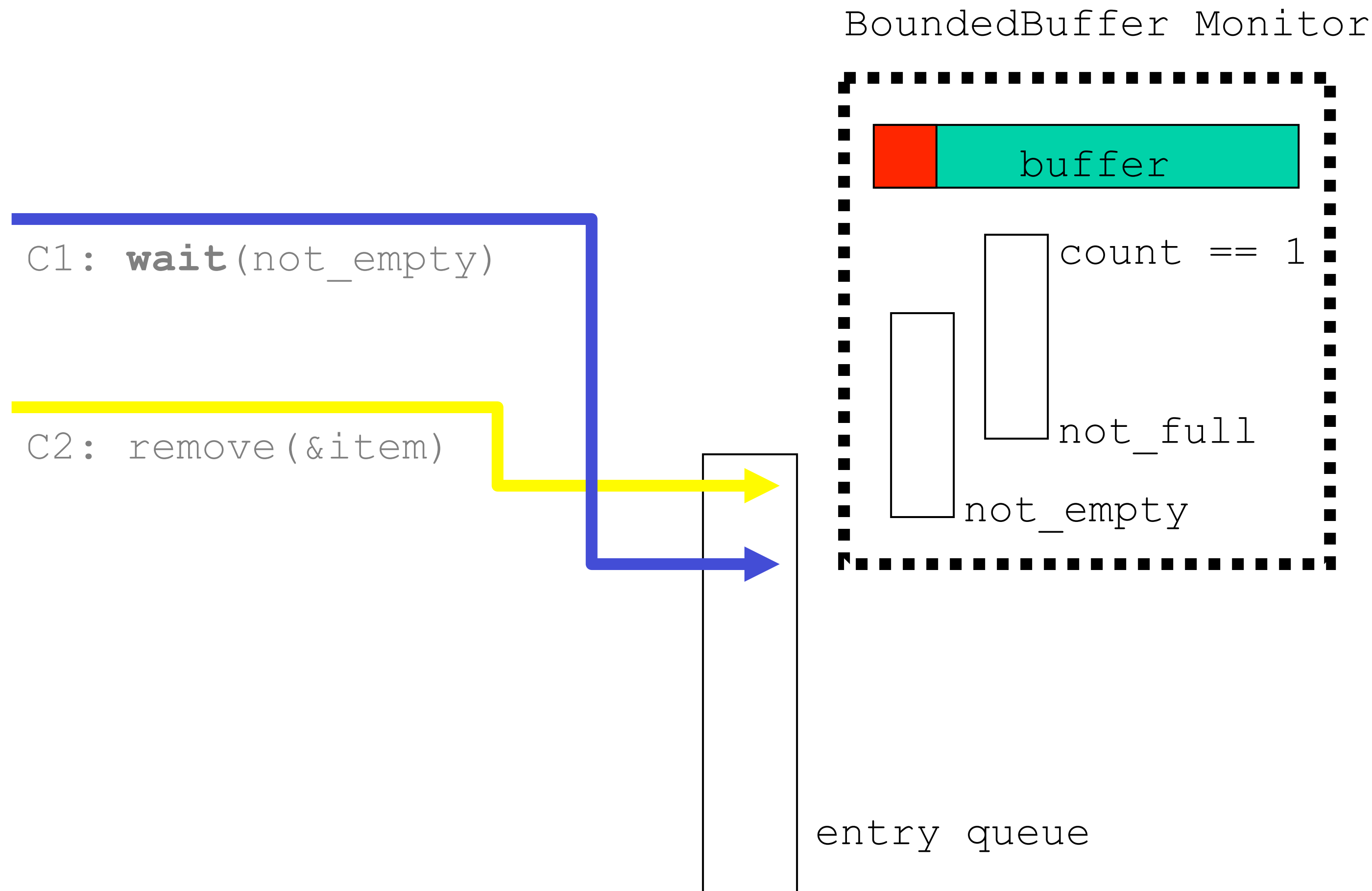
An example trace 11



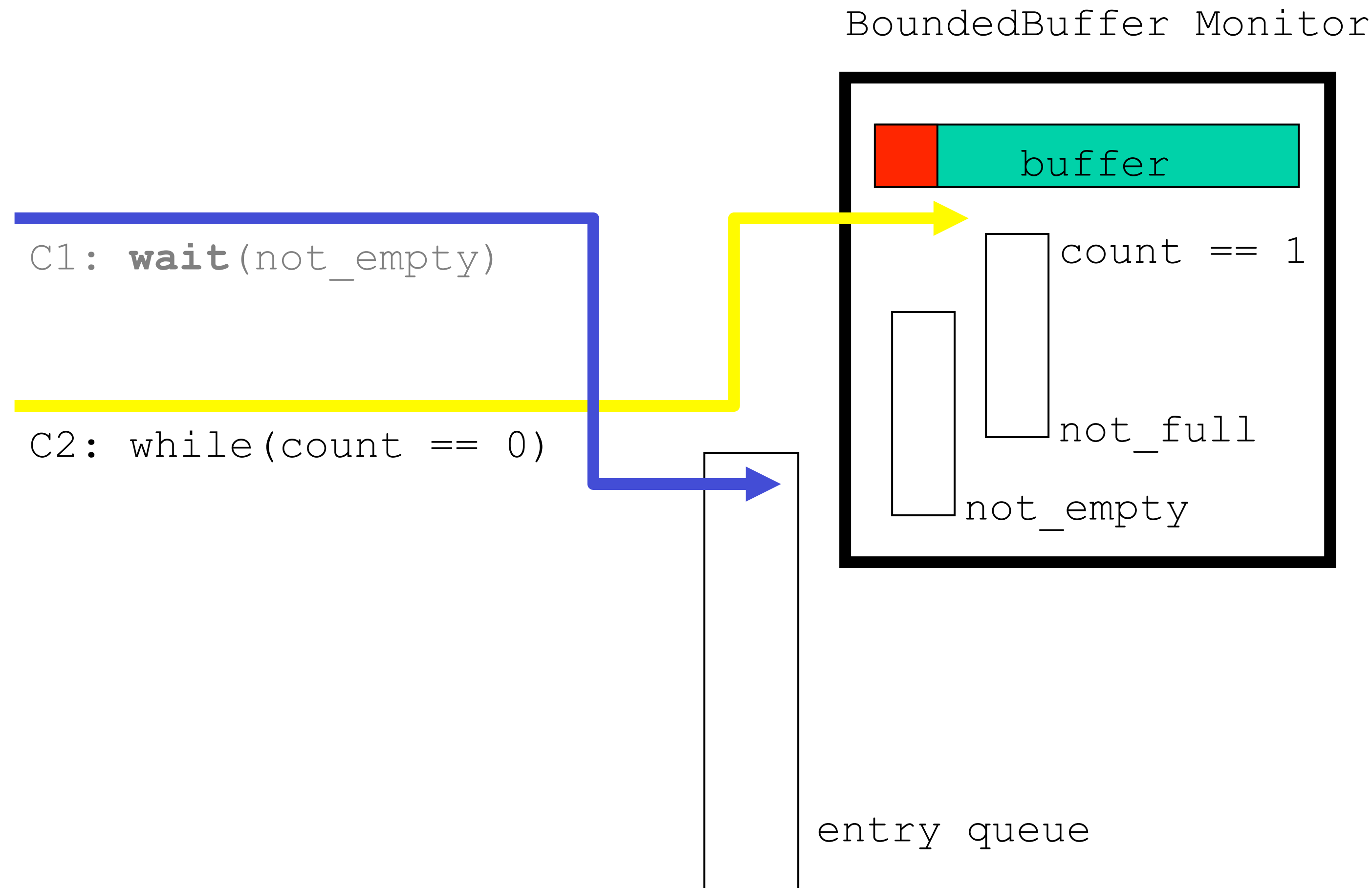
An example trace 12



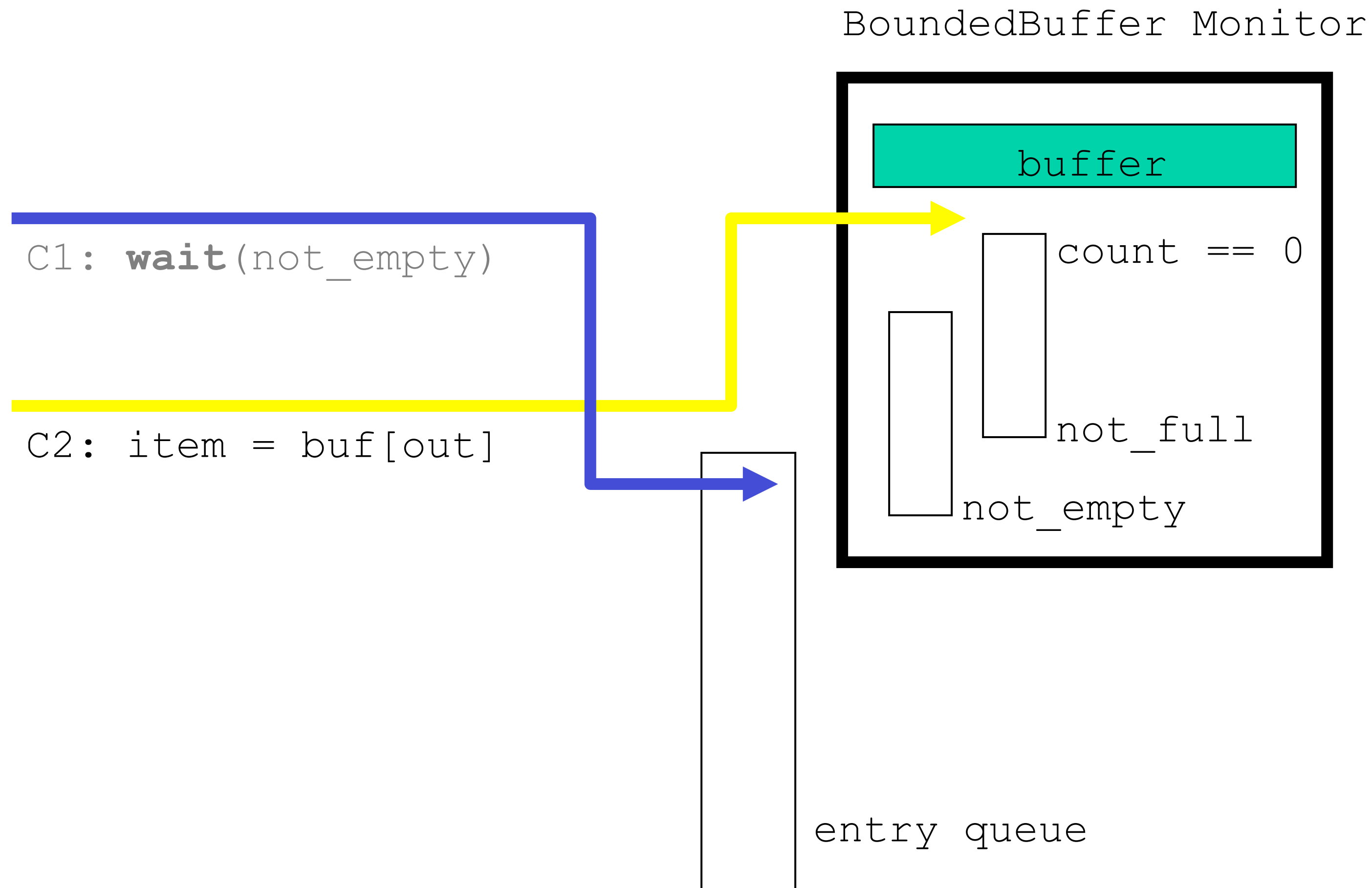
An example trace 13



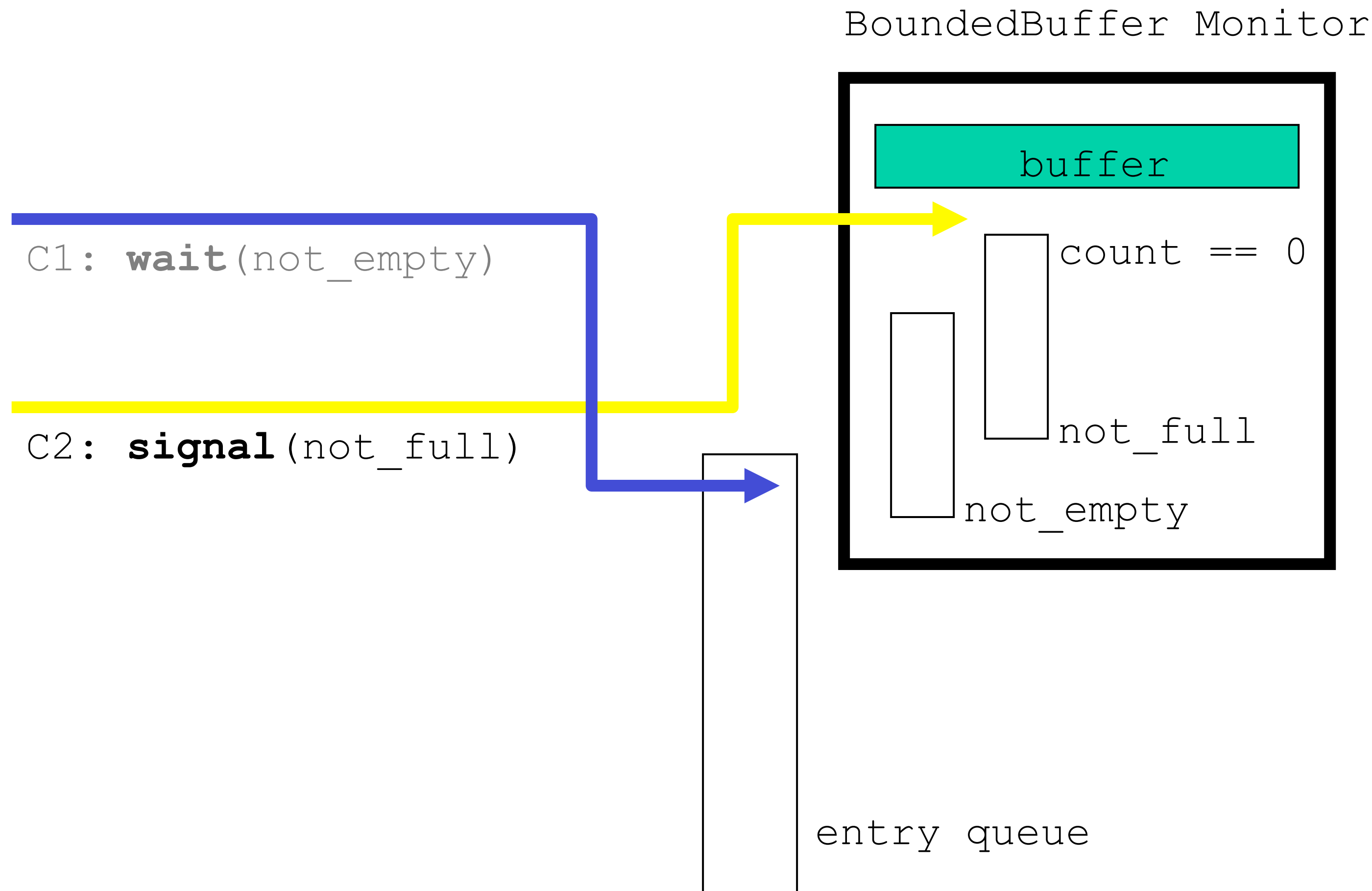
An example trace 14



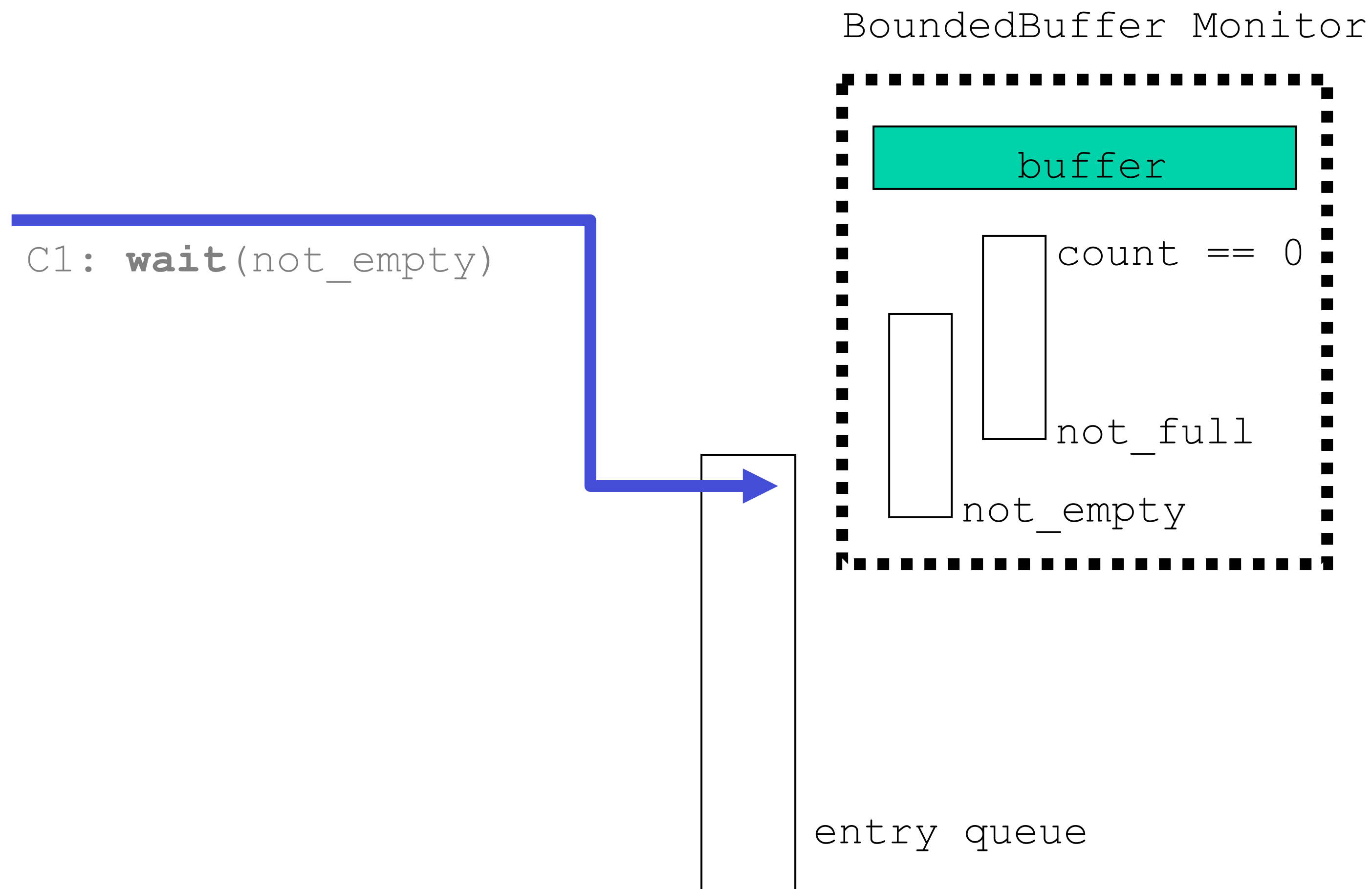
An example trace 15



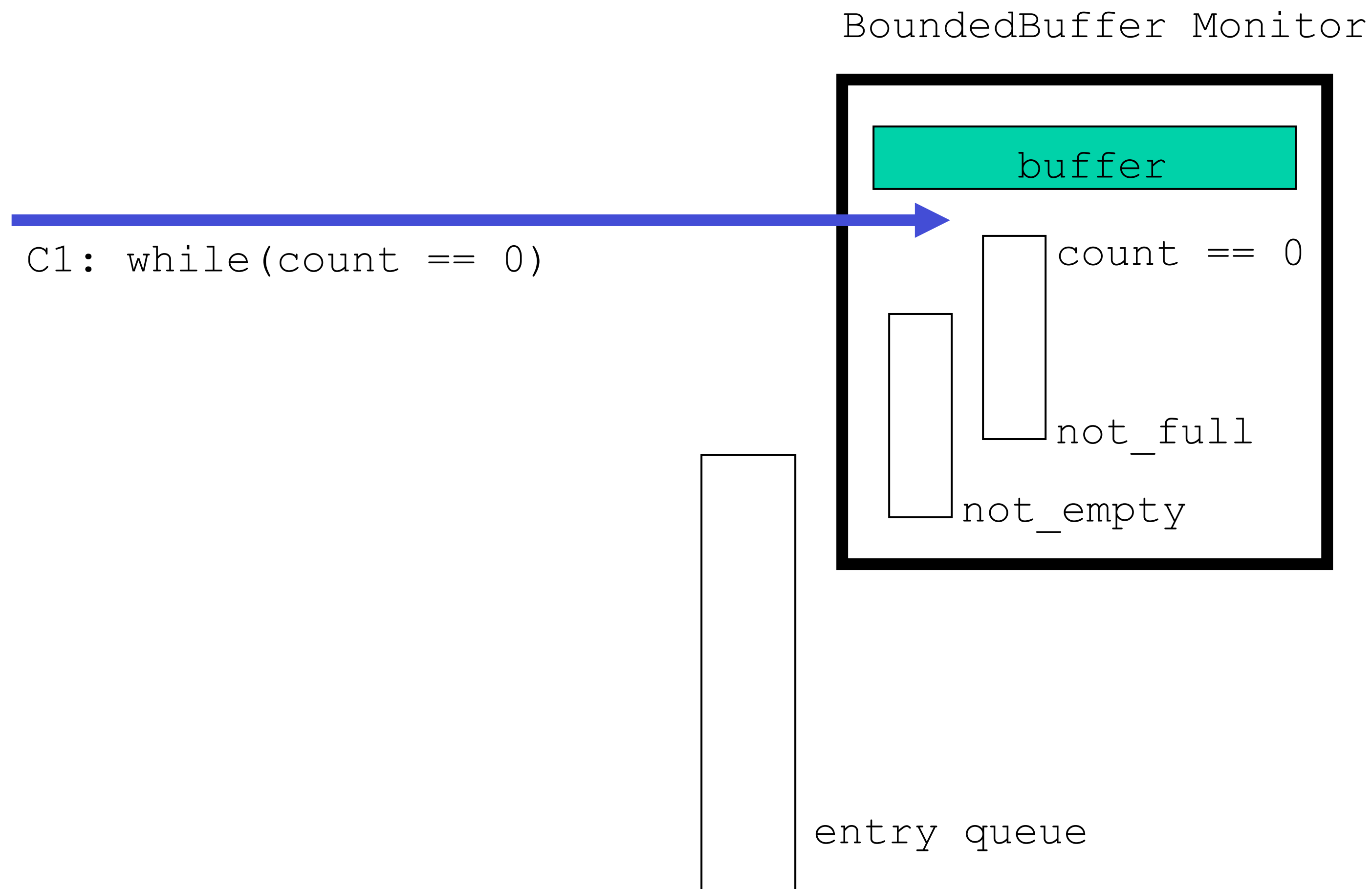
An example trace 16



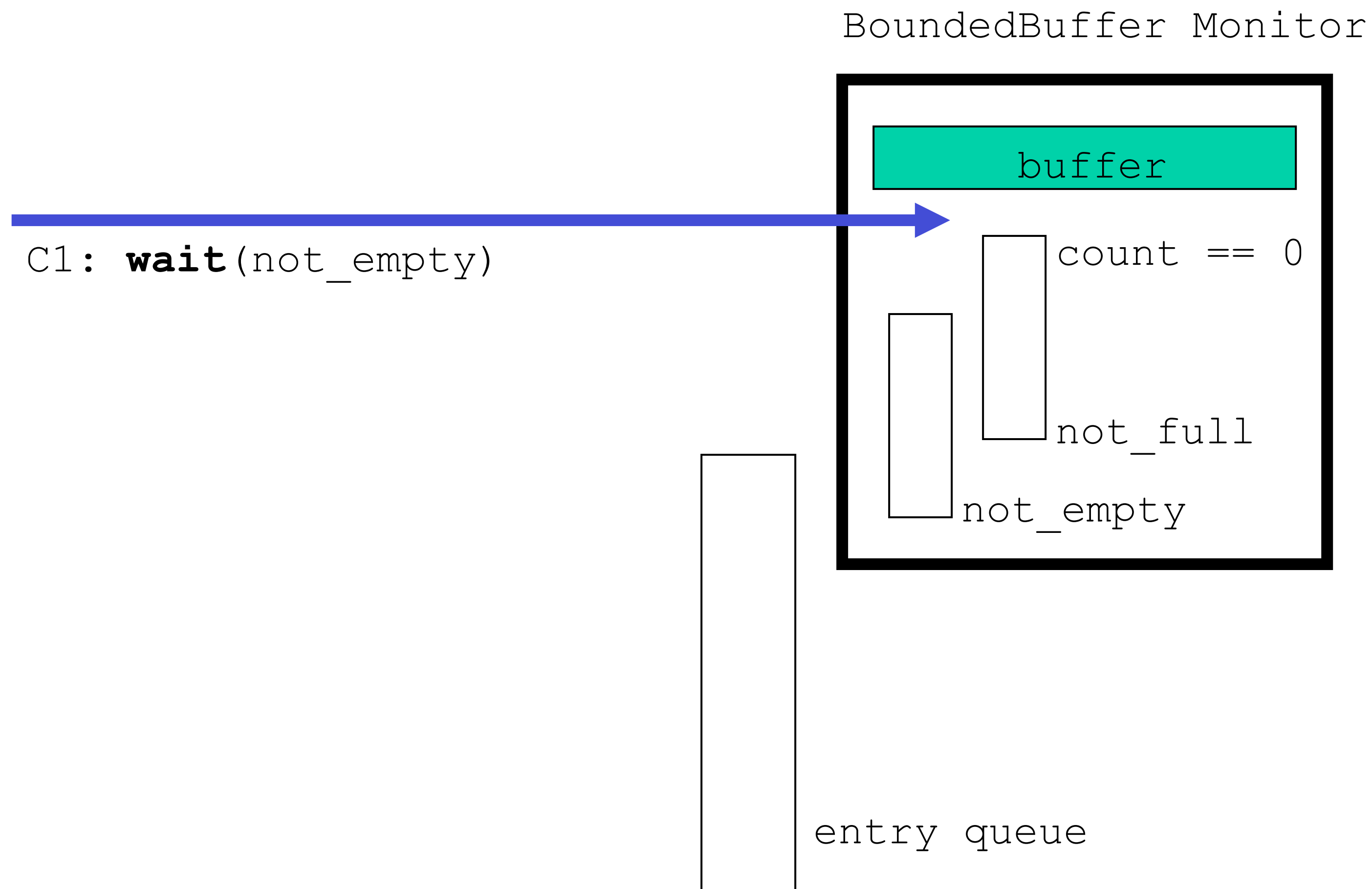
An example trace 17



An example trace 18

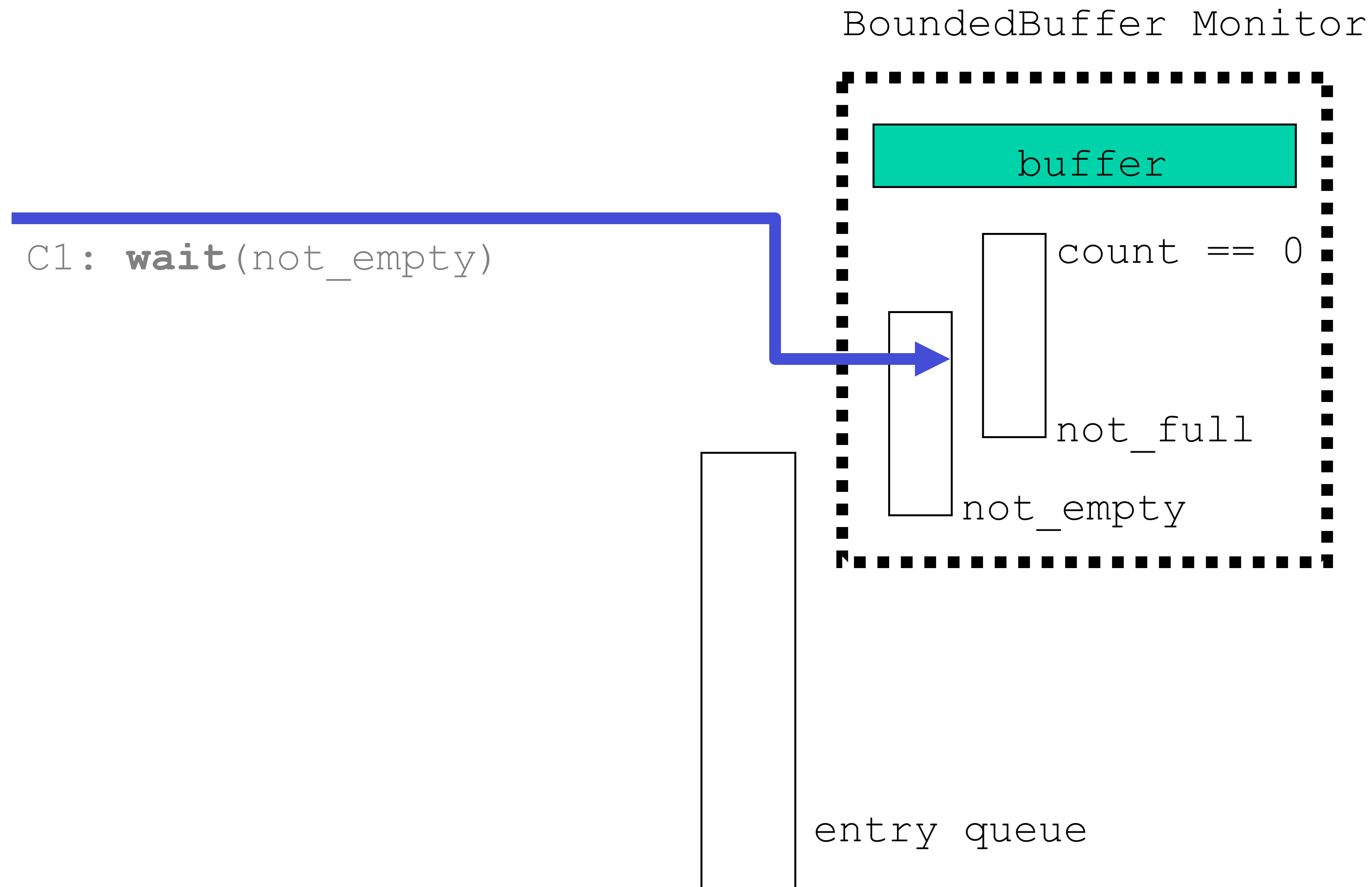


An example trace 19



© Brian Logan 2014

An example trace 20



© Brian Logan 2014

Semaphores and monitors

Semaphores and monitors have the same expressive power:

- monitors can be used to simulate semaphores; and
- semaphores can be used to simulate monitors.

What we gain with monitors is a higher level of abstraction.

Monitors & Java

Monitors form the basis of Java's support for (shared memory) concurrency:

- *mutual exclusion* can be implemented in Java using the `synchronized` keyword
- a synchronized method (or block) is executed under mutual exclusion with all other synchronized methods on the same object

- Java provides basic operations for *condition synchronisation*: `wait(),notify(),notifyAll()`
- each Java object has a single (implicit) condition variable and delay queue, the *wait set*
- Java uses the *signal and continue* signalling discipline

© Brian Logan 2014

The next lecture

Monitors II

Suggested reading:

- Andrews (2000), chapter 5;
- Ben-Ari (1982), chapter 5;
- Burns & Davies (1993), chapter 7, sections 7.4–7.9;