Concurrent Algorithms
Lecture 7 - Monitors

Eugene Kenny

eugkenny.lit@gmail.com

Limerick Institute of Technology

November 20th 2014

Outline

- limitations of semaphores
- monitors
- example: bounded buffer with monitors
- comparison of semaphores and monitors
- monitors and Java
- Readers and Writers problem
- selective mutual exclusion problems
- scheduling policies
  - readers' preference
  - writers' preference
  - a fair solution
- monitor solutions

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);

A monitor implements a shared data structure together with the (coarsegrained 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 monitors 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 monitors 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

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

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 ...
```

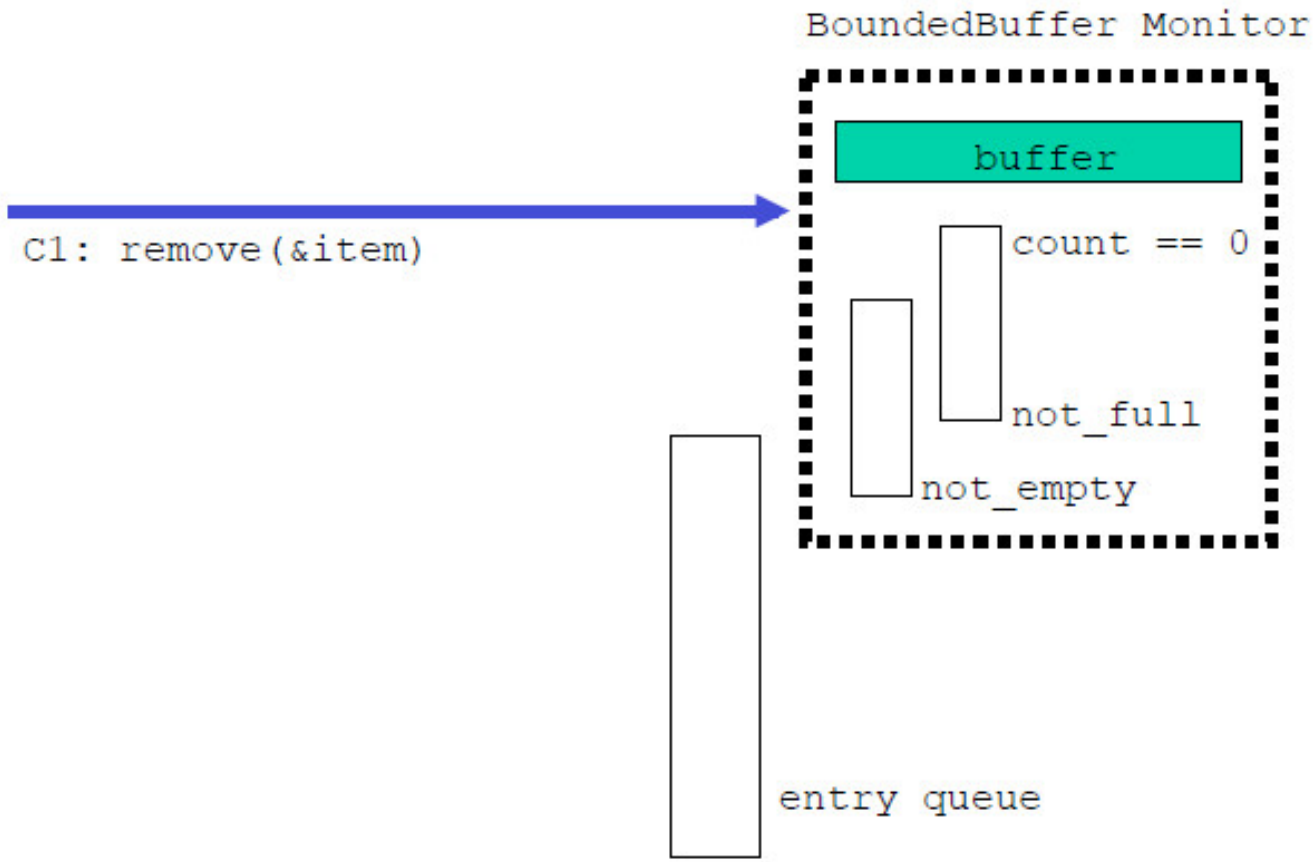
Bounded Buffer with Monitors

```
// 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 ...
```

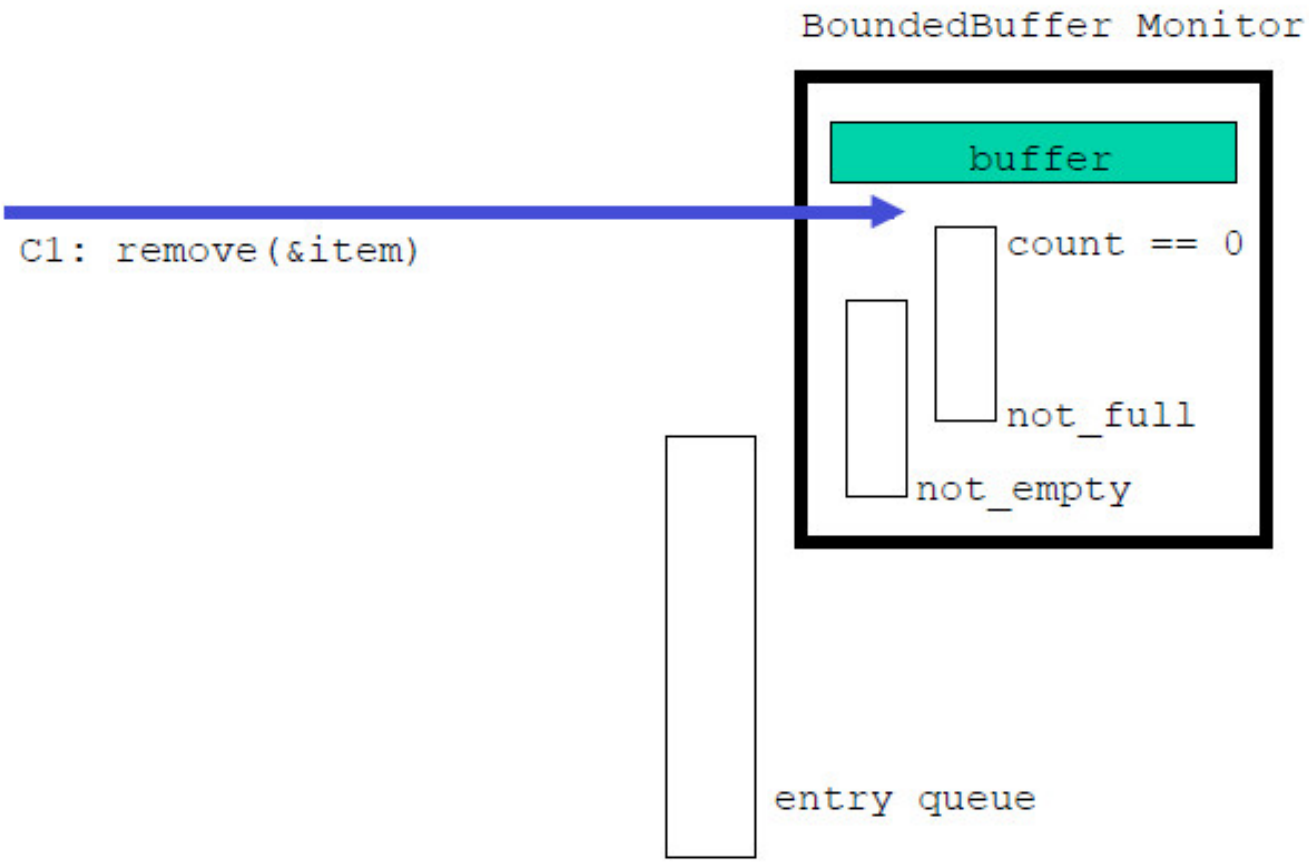
Bounded Buffer with Monitors

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

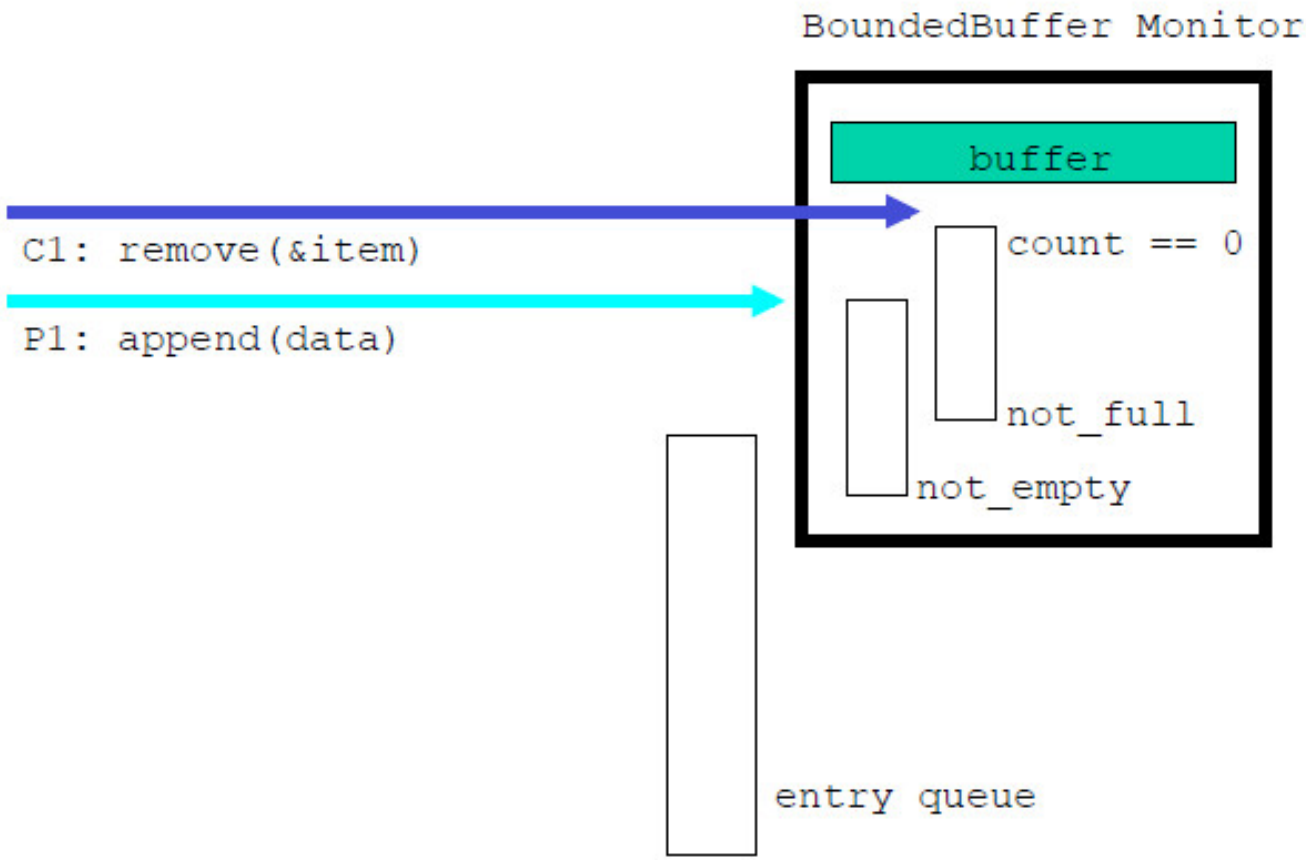
An Example Trace



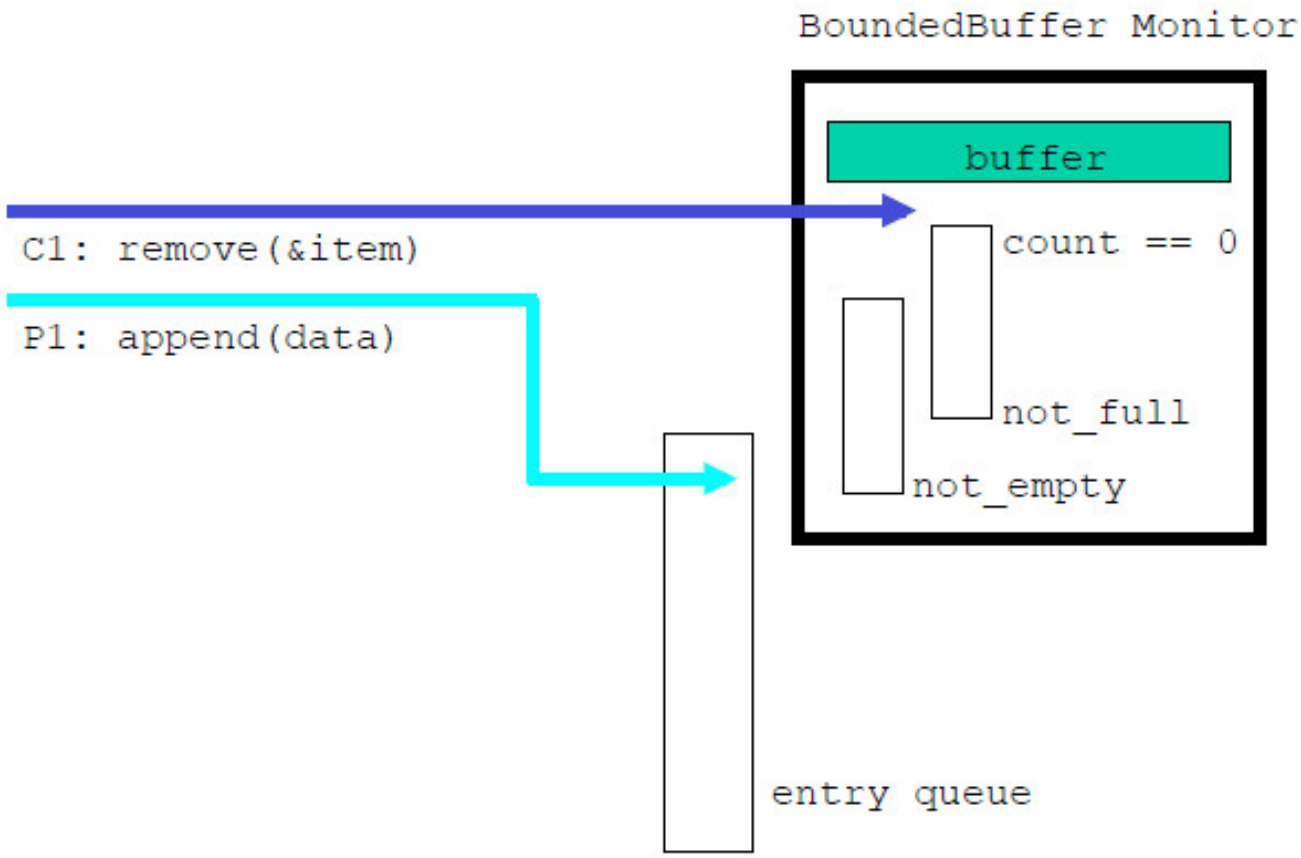
An Example Trace



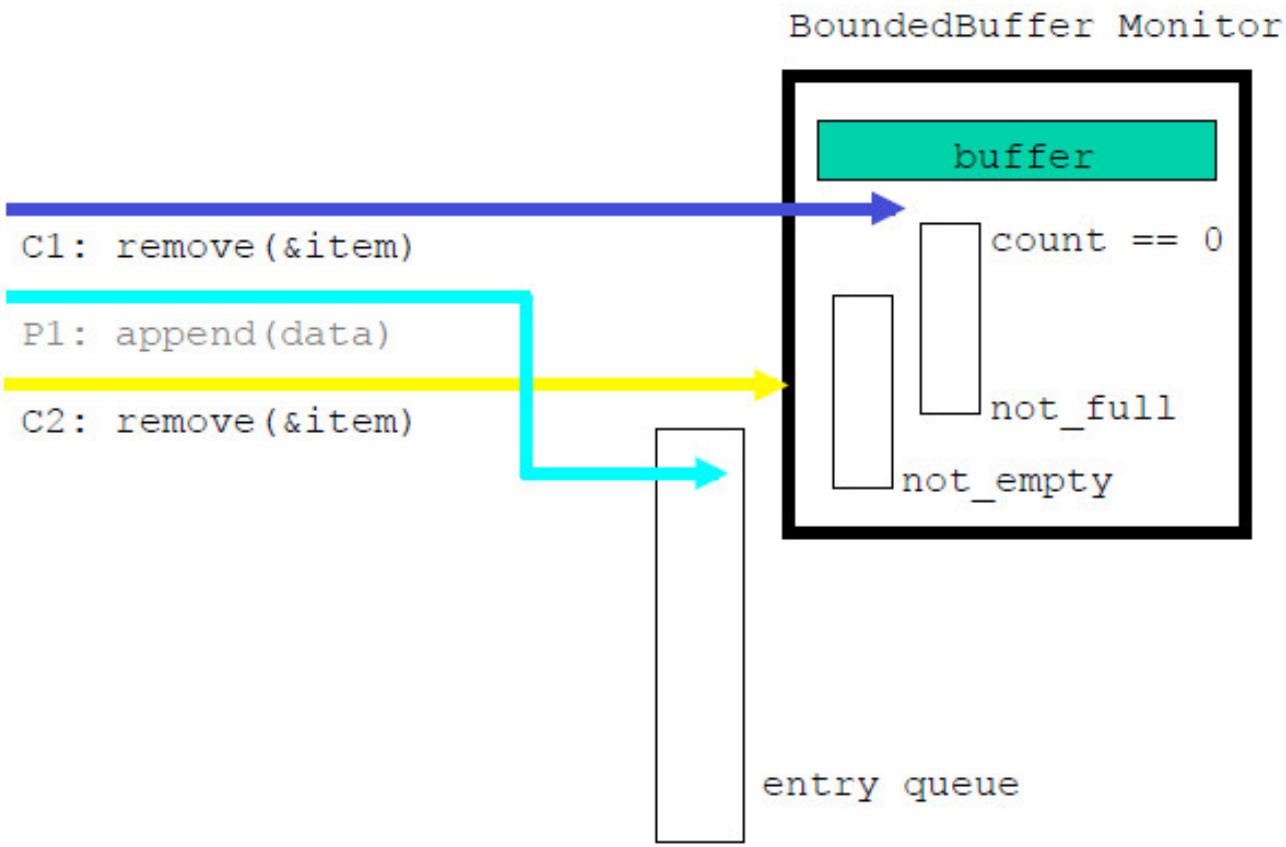
An Example Trace



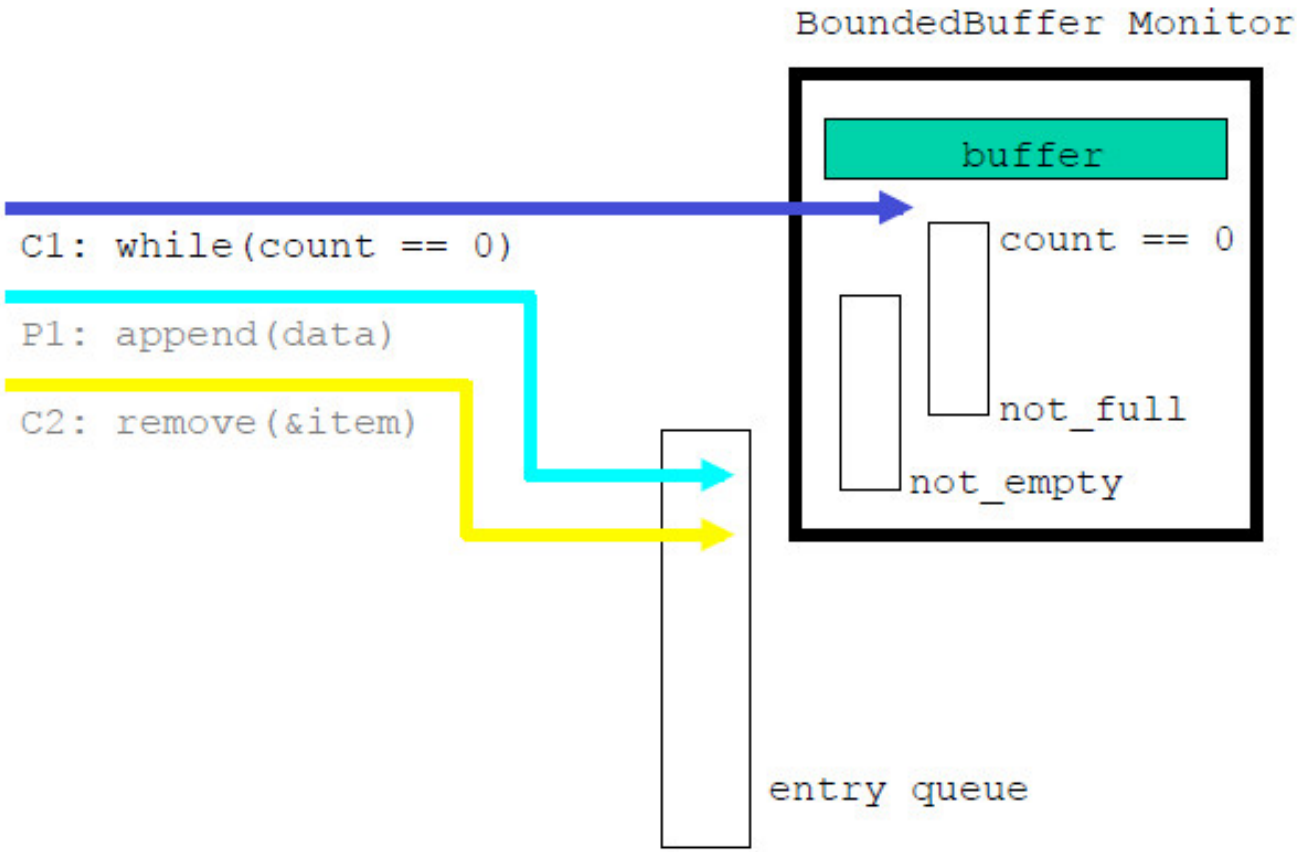
An Example Trace



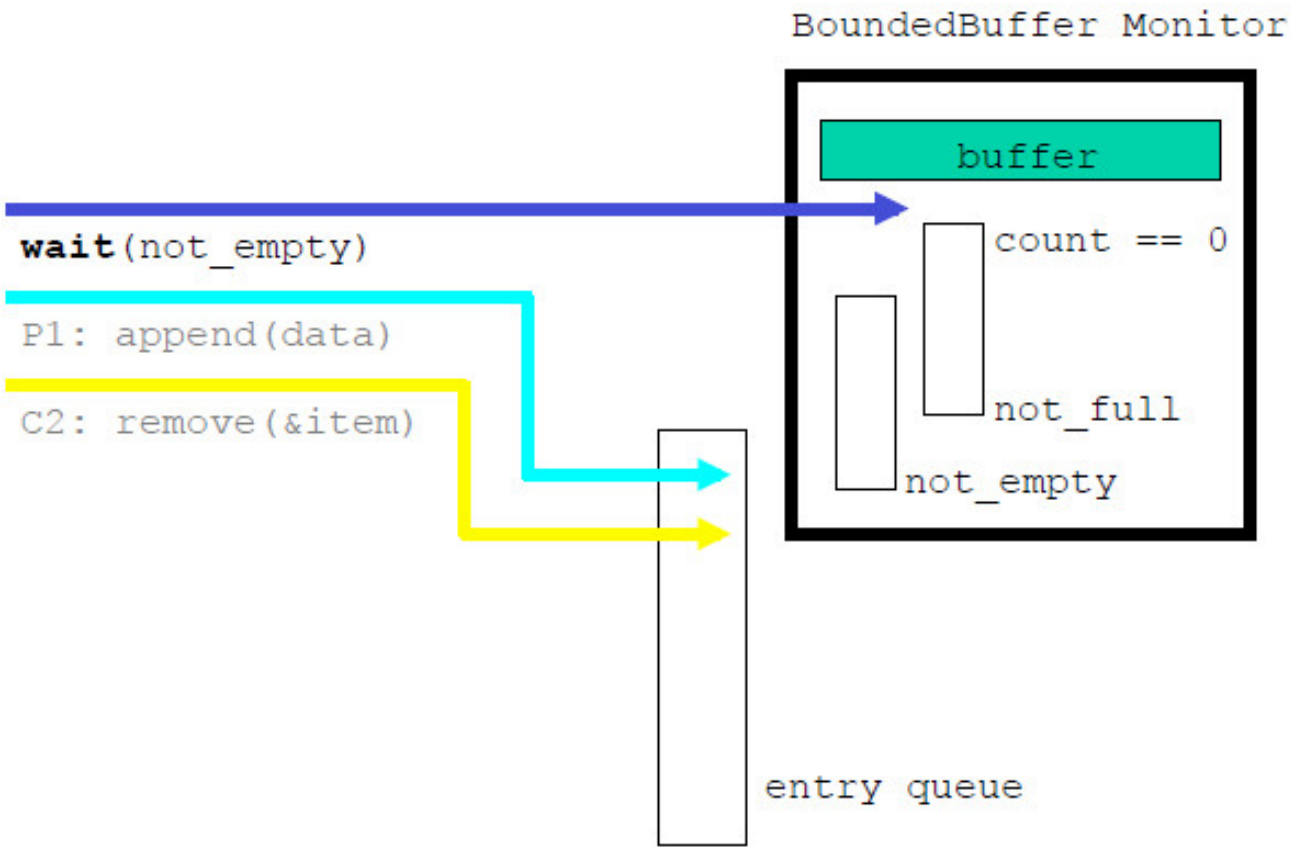
An Example Trace



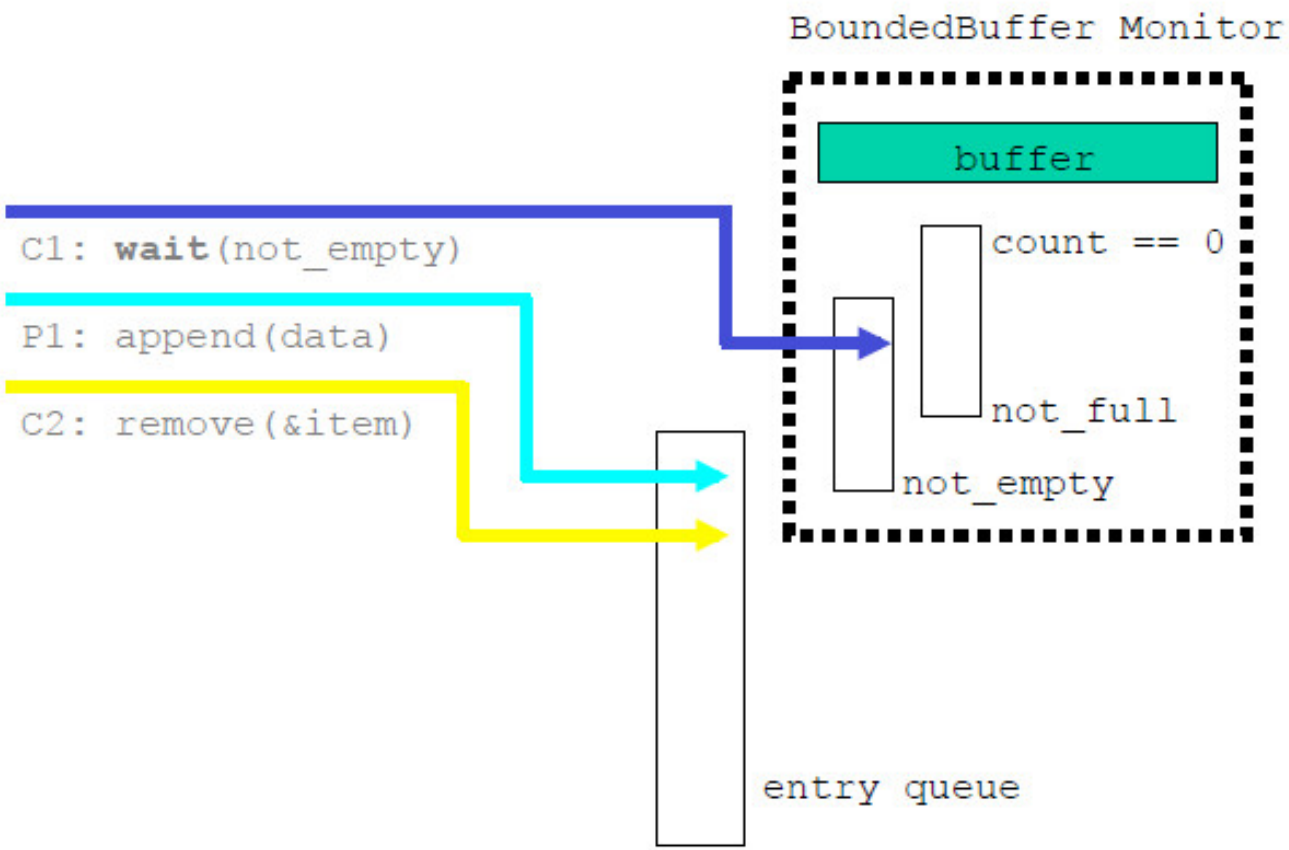
An Example Trace



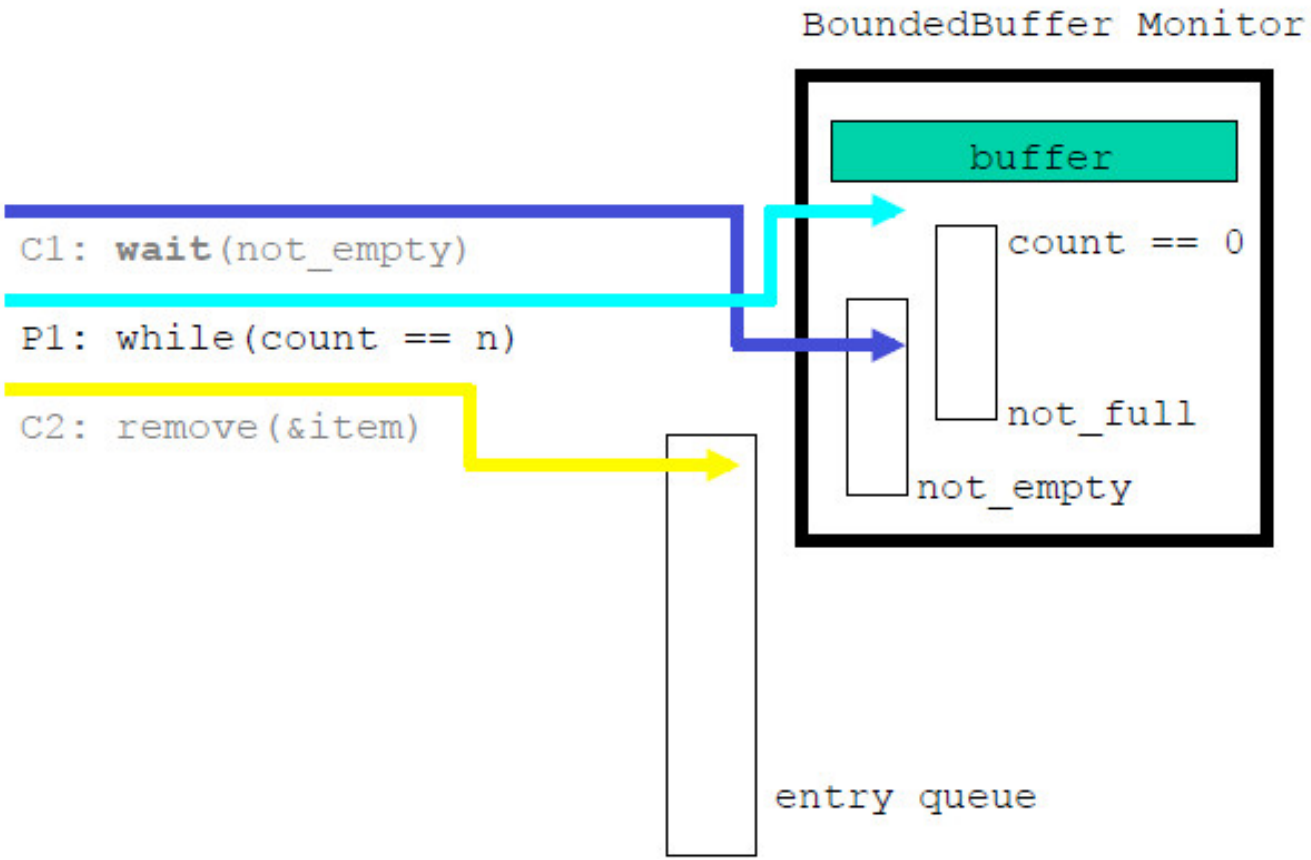
An Example Trace



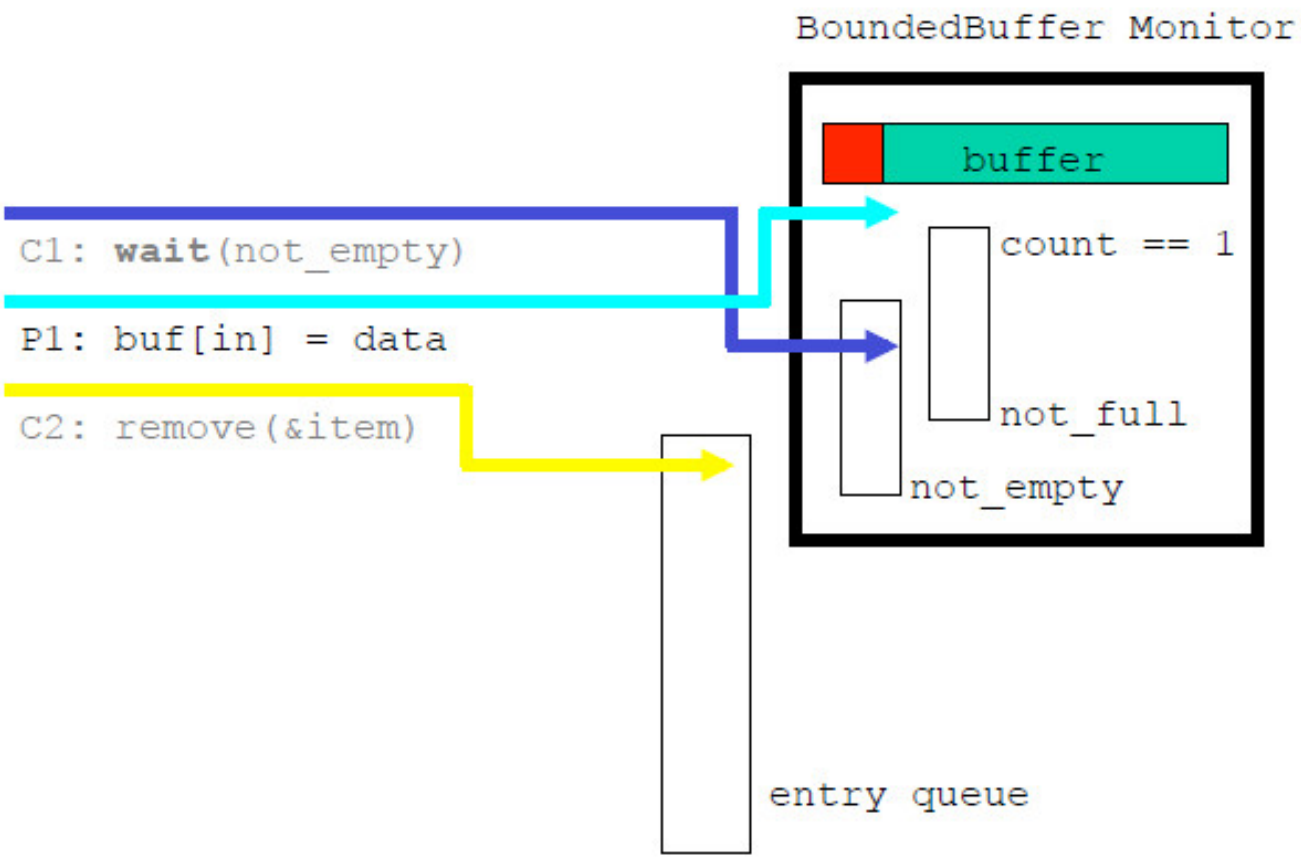
An Example Trace



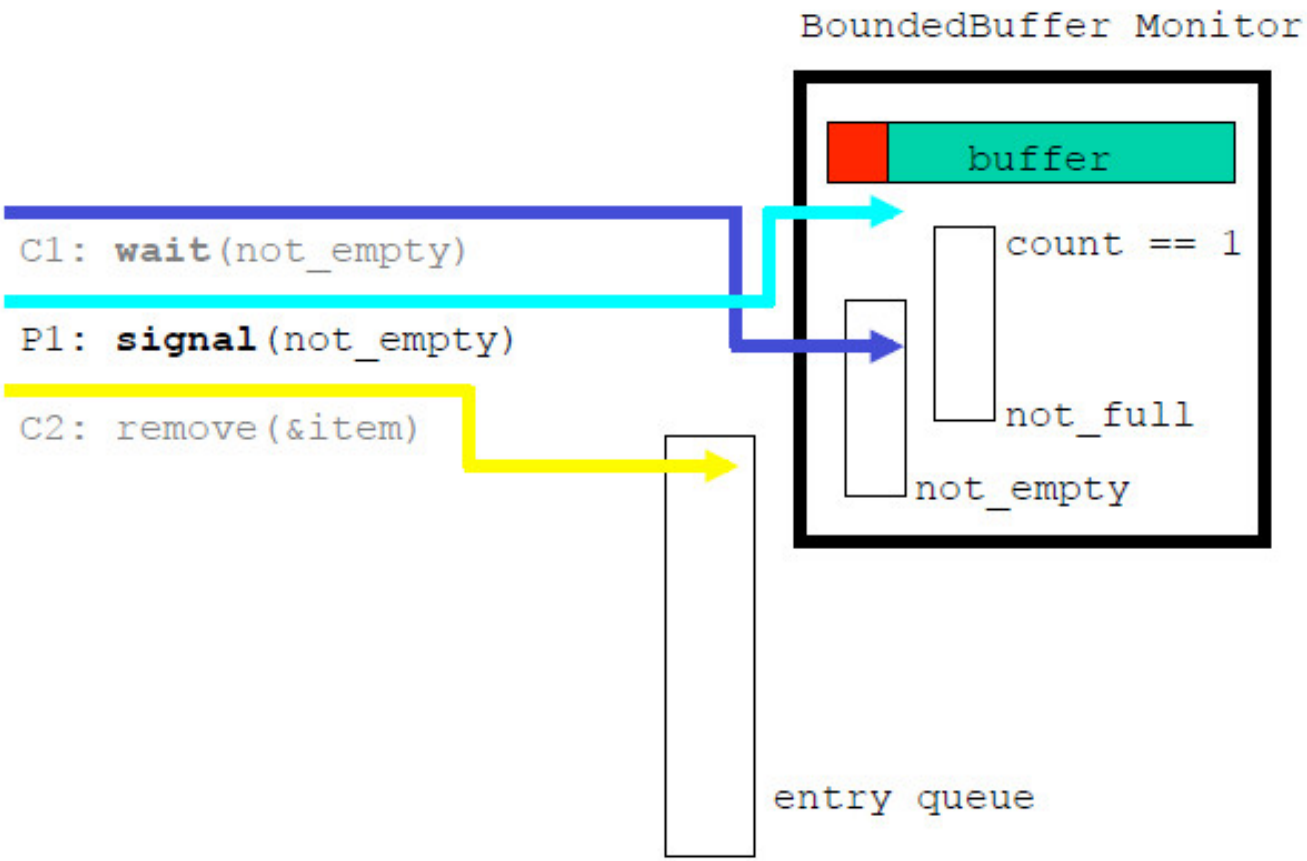
An Example Trace



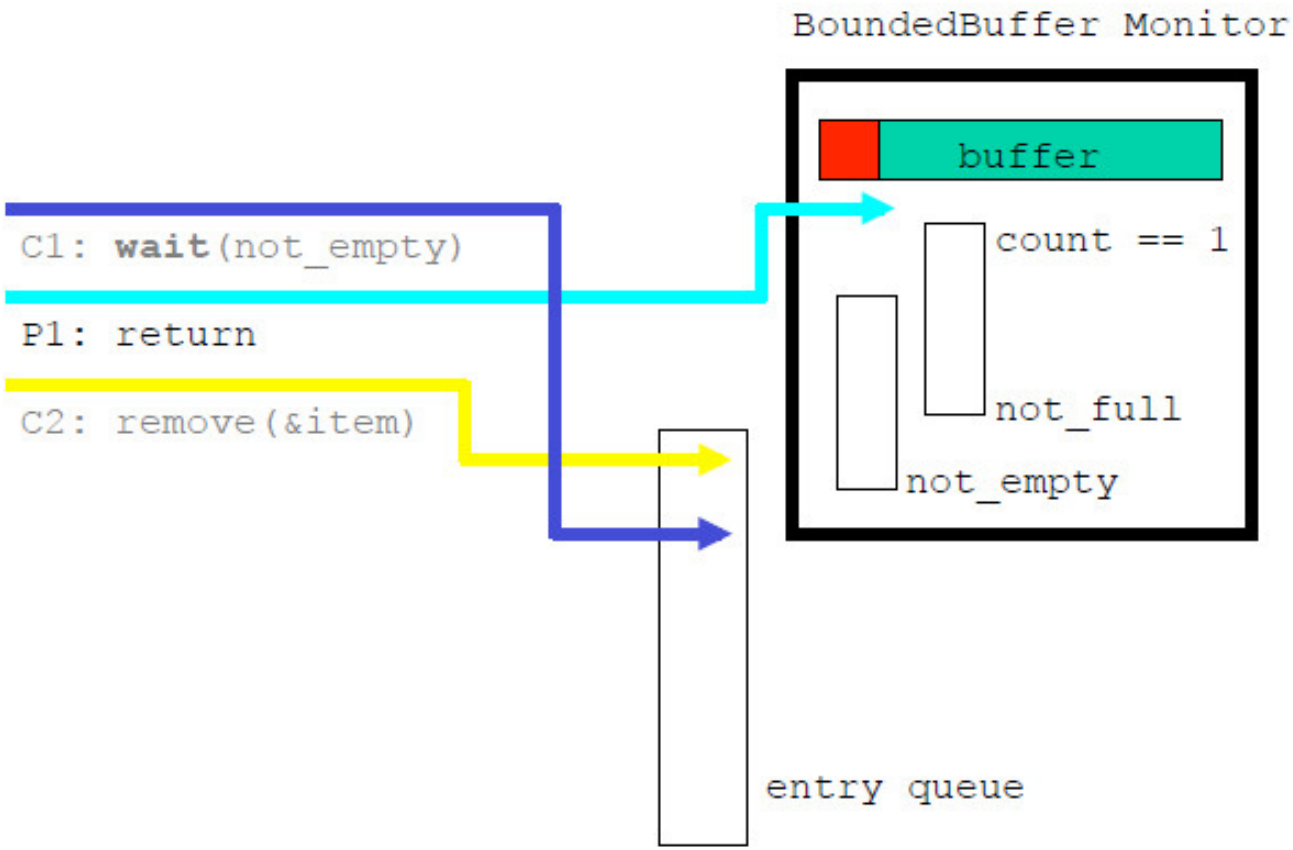
An Example Trace



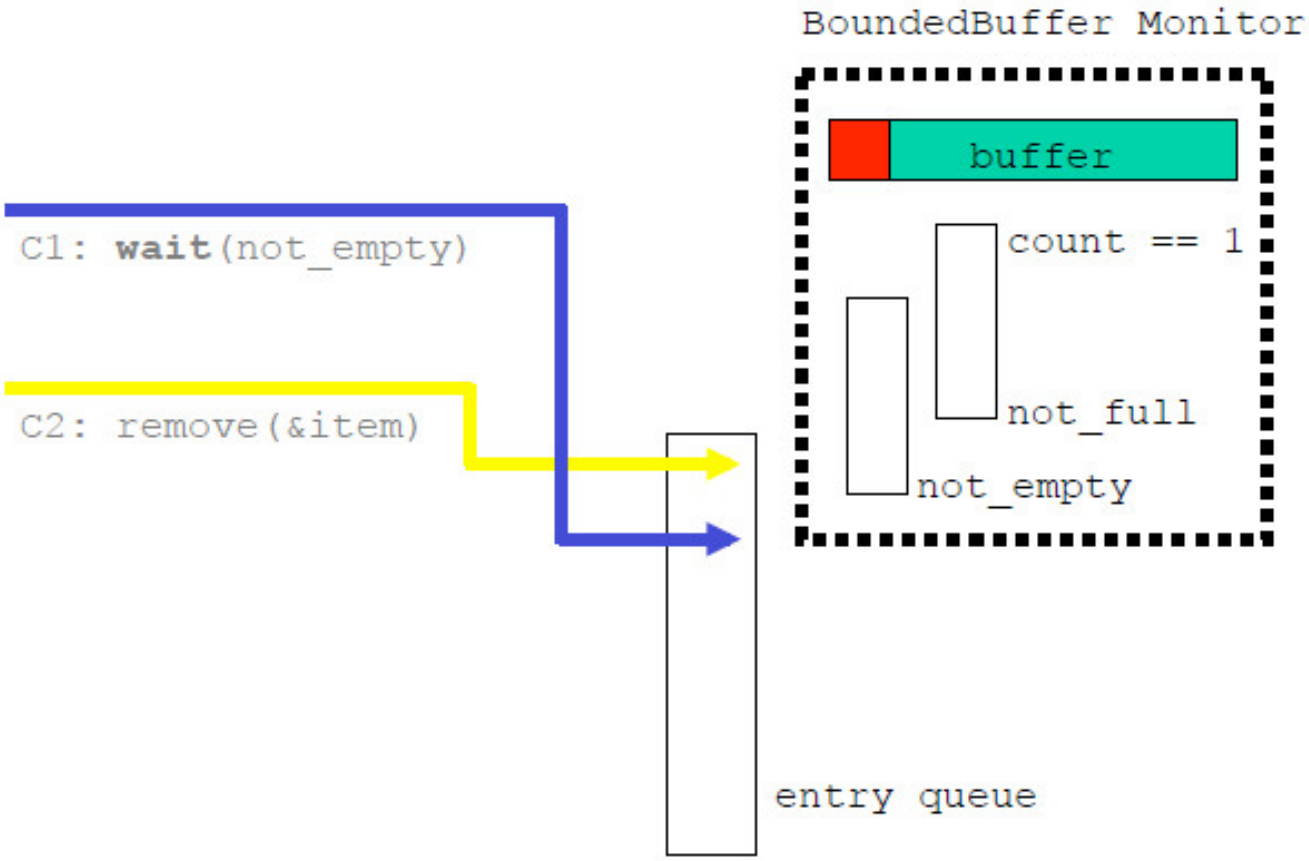
An Example Trace



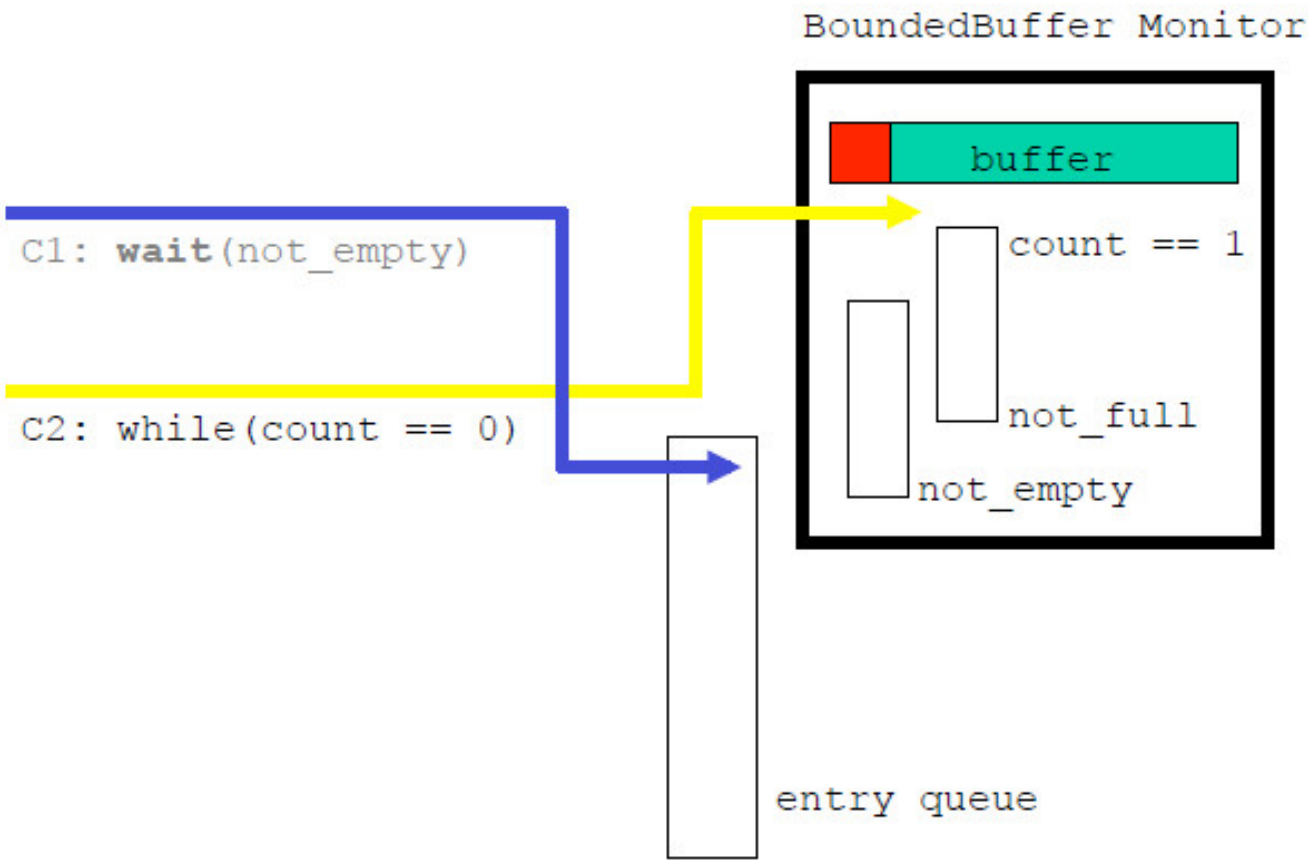
An Example Trace



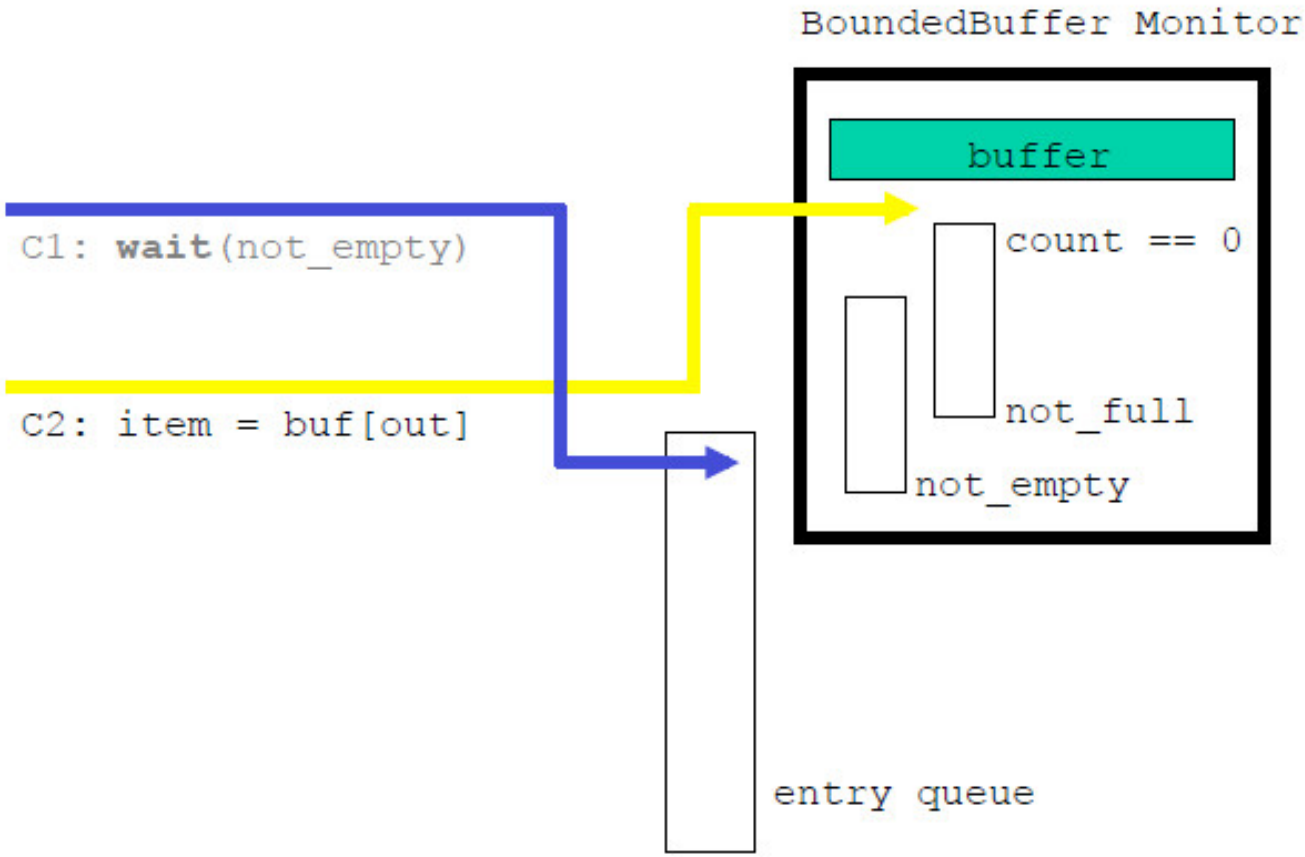
An Example Trace



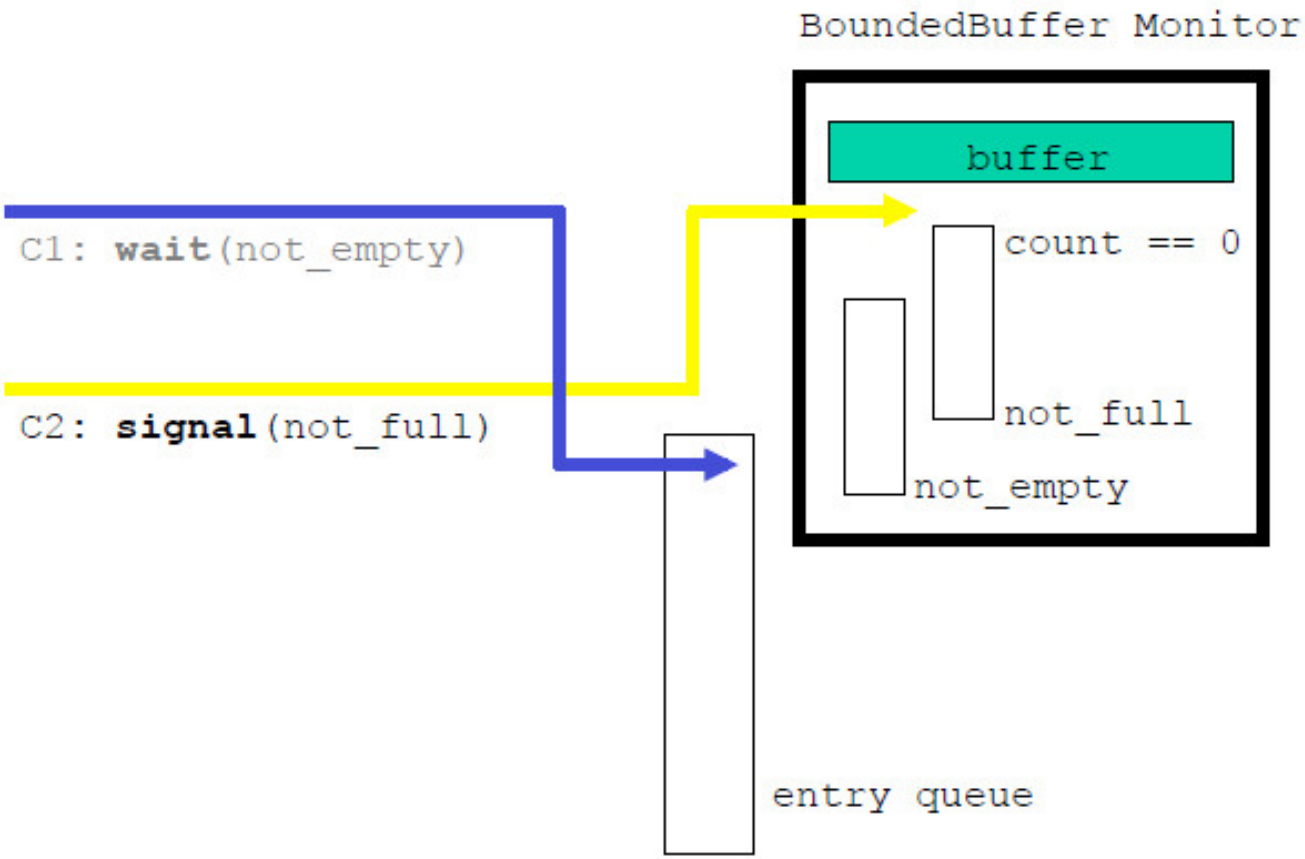
An Example Trace



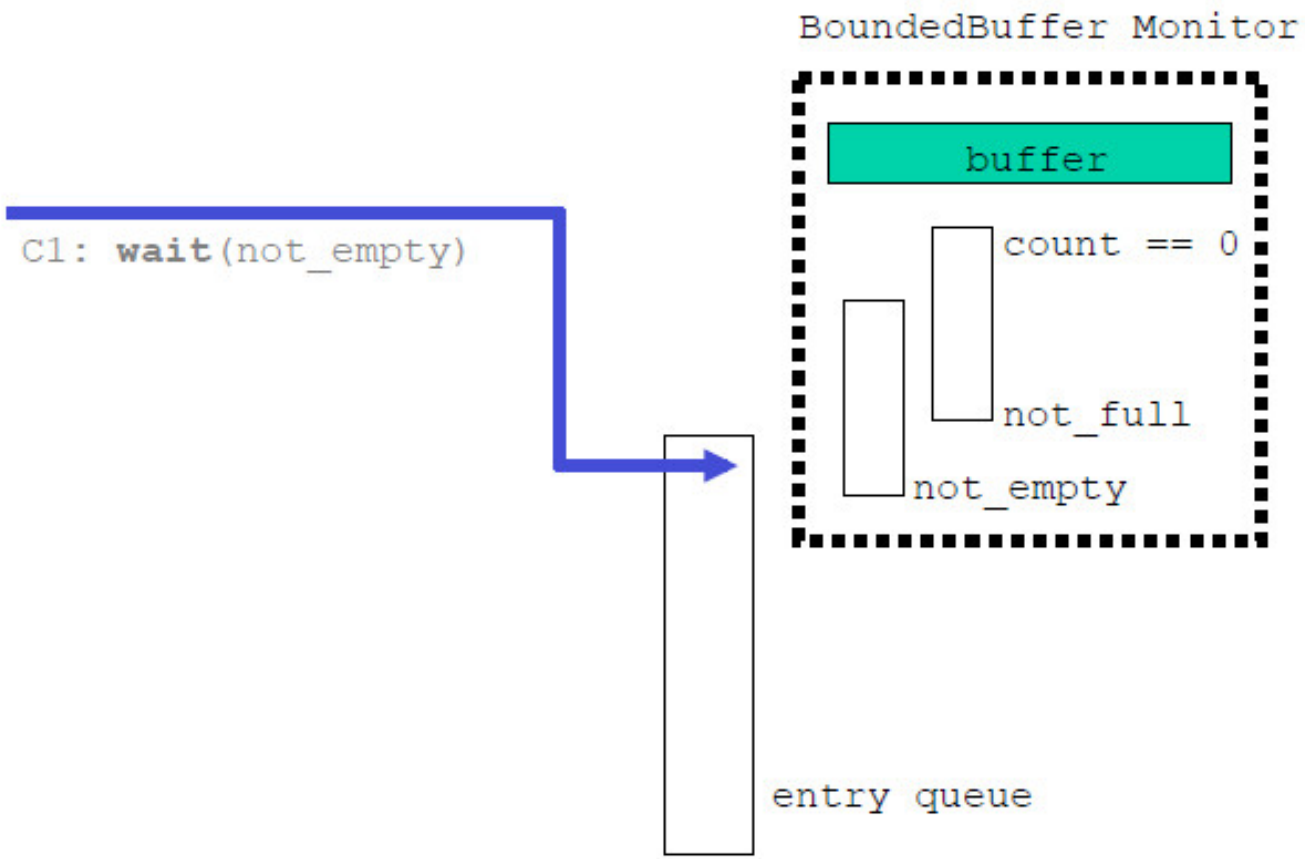
An Example Trace



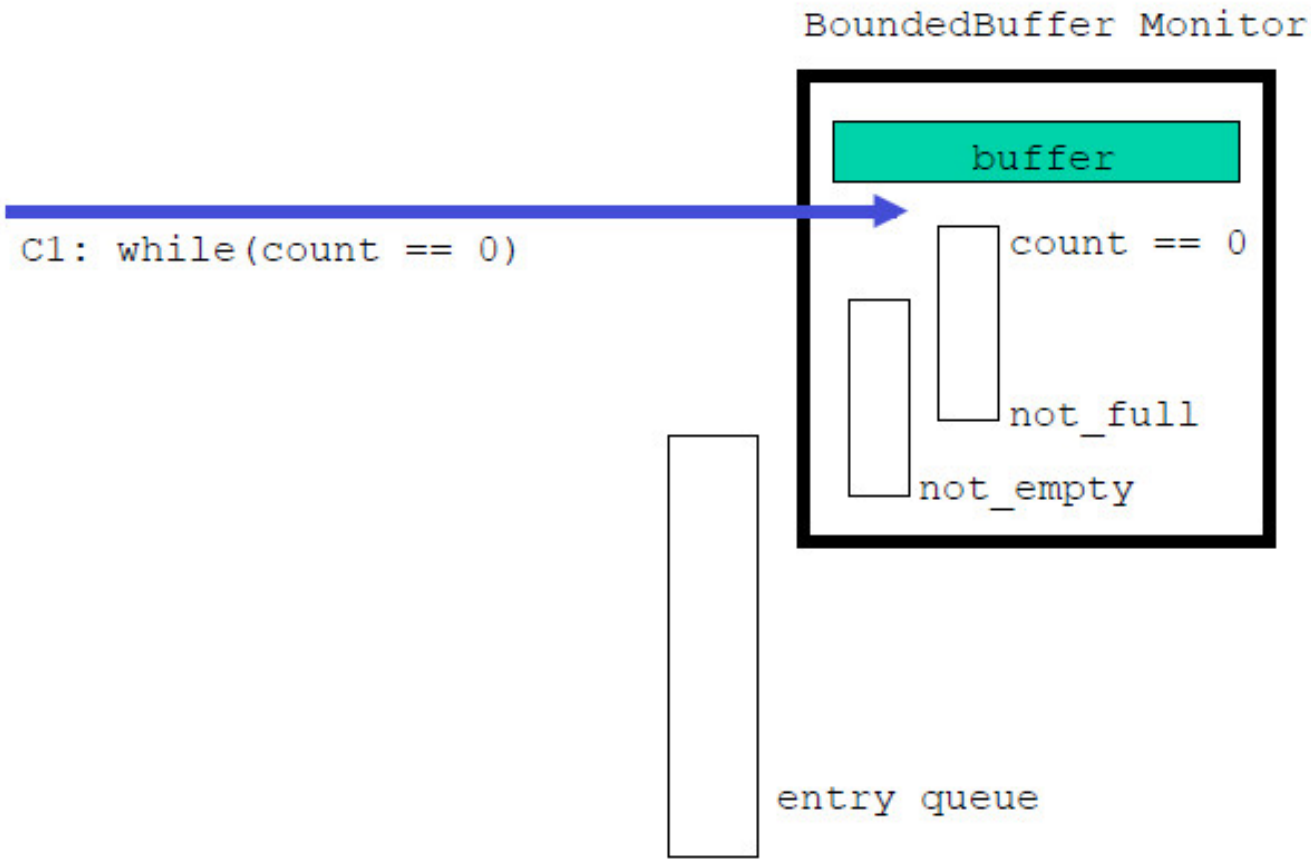
An Example Trace



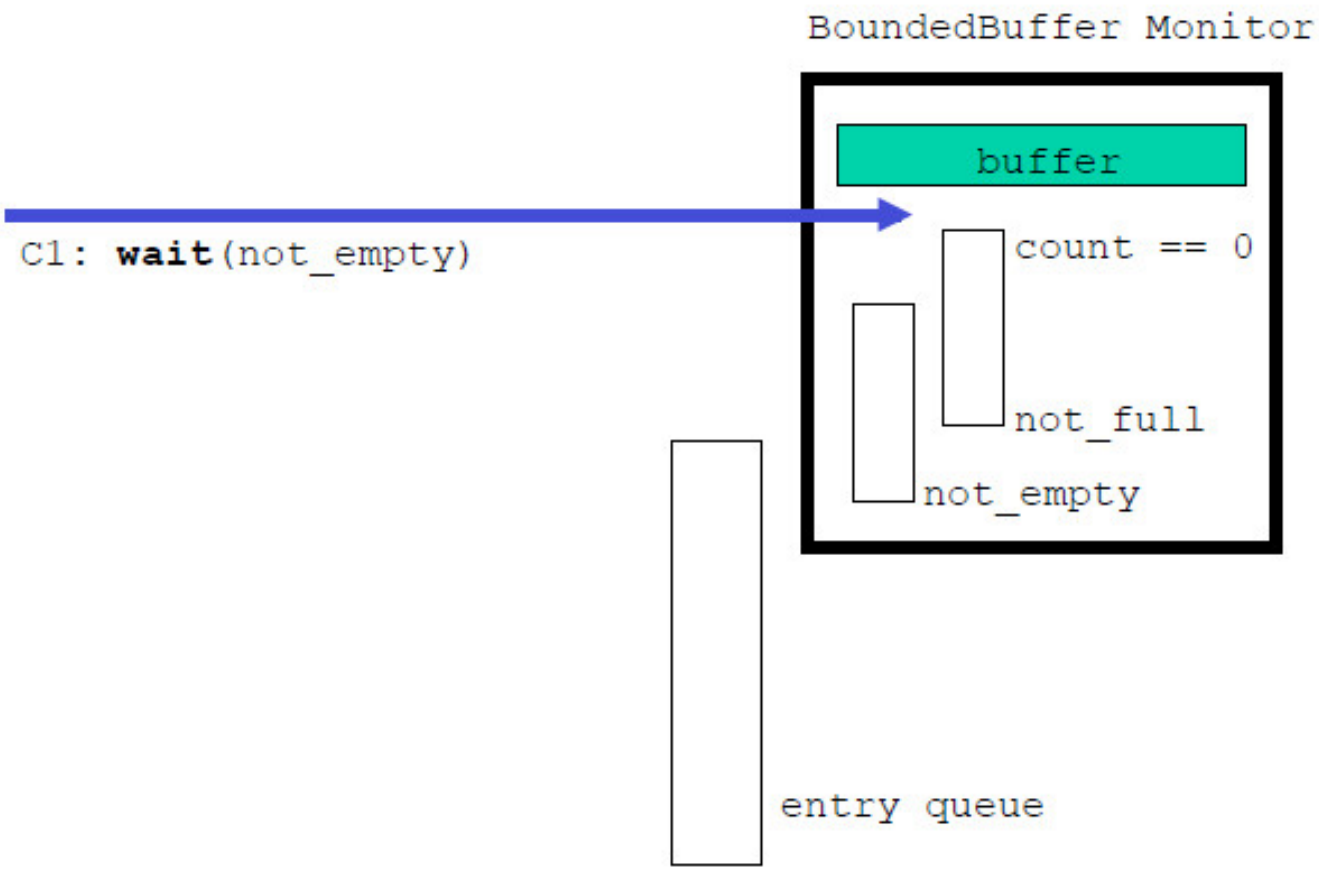
An Example Trace



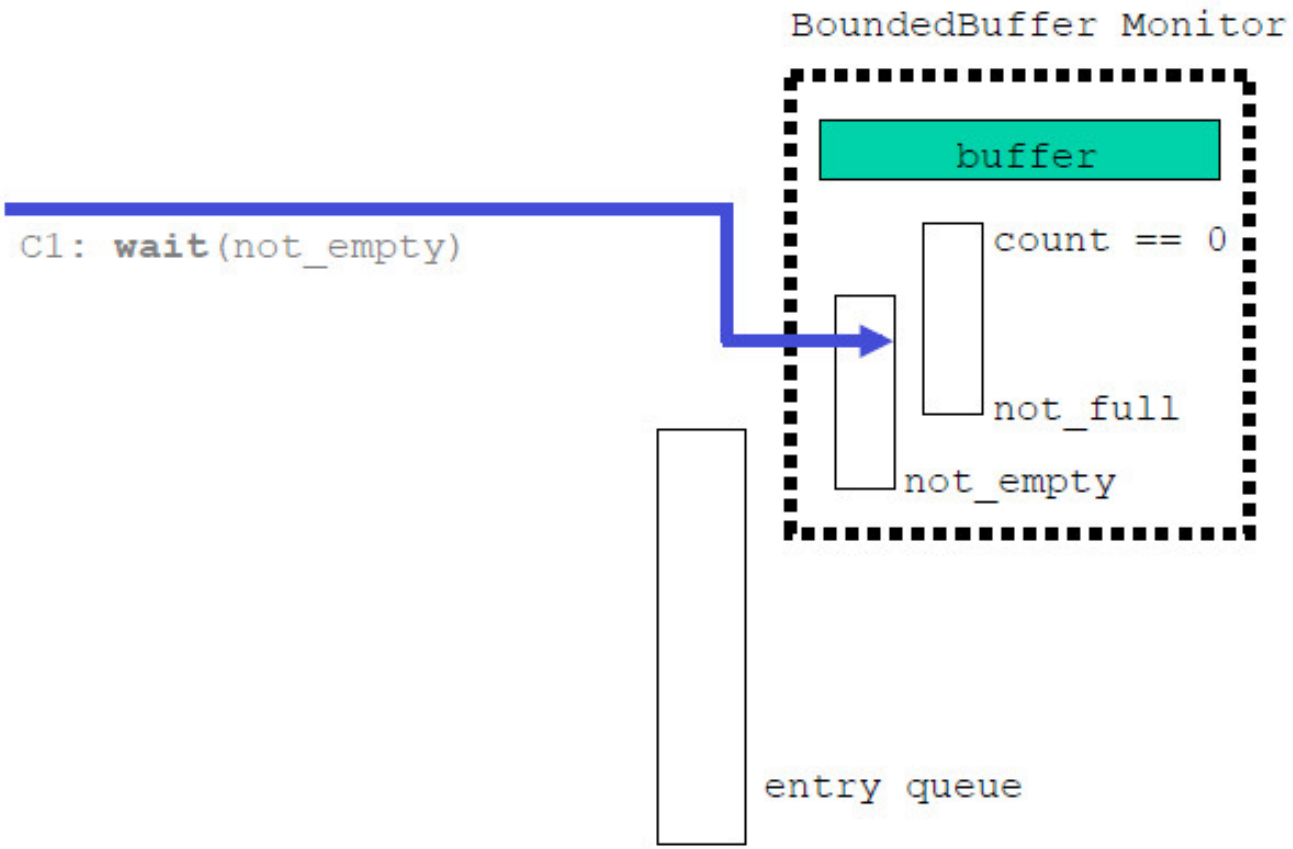
An Example Trace



An Example Trace



An Example Trace



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 Javas 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

Readers and Writers

Given a shared file and a collection of processes that need to access and update the file:

- **reader** processes only need to read the file
- **writer** processes need to both read and write the file

We assume that

- the file is initially in a consistent state
- each read or write executed in isolation transforms the file into a new consistent state.

Scope of the Problem

The Readers and Writers problem is an abstraction of a common class of concurrency problems:

- the *'file'* can be any shared resource, e.g., an area of memory, or a database
- *'reading'* and *'writing'* can be any operations:
  - *reading* doesn't change the resource – readers can be allowed to proceed concurrently
  - *writing* must be mutually exclusive of all readers and all other writers

Selective Mutual Exclusion

- a *critical section* is a section of code belonging to a process that accesses shared variables, where, to ensure correct behaviour, the critical section must be given mutually exclusive access to the shared variables
- *mutual exclusion* is the requirement that, at any given time, at most one process is executing its critical section
- mutual exclusion applies between critical sections

In the Readers and Writers problem, reading need not be not mutually exclusive of reading, but writing must be mutually exclusive of reading and writing.

An Example

A simple library catalogue has two kinds of users:

- *borrowers* who use the catalogue to search for books or to find out whether a particular book is on loan (readers); and
- *library staff* who update the catalogue when new books are added to the library stock or to record which books are on loan (writers).

Updating the Library Catalogue

If we allow all users unrestricted access to the catalogue, a borrower may see the catalogue when it is an inconsistent state, e.g.:

- a borrower is looking for a book which has just arrived at the library
- a librarian is updating the catalogue to include the book, e.g., by copying and editing an existing entry
- the borrower sees an inconsistent state of the catalogue, e.g., the entry contains an incorrect shelf number.

Correctness vs Efficiency

The Readers and Writers problem is a special case of the general problem of a number of processes all of which may *both* read and write to a file:

- any solution to the general problem will also be a *correct* solution to the Readers and Writers problem
- however (much) more *efficient* solutions are possible for the Readers and Writers problem

In general, an efficient solution depends on the specifics of the problem.

A Simple Solution

One solution to the readers and writers problem would be to make all accesses to the file mutually exclusive:

- at most one process, whether reader or writer would be able to access the file at a time.
- this is simple and correct but, in general, the performance of such an approach is unacceptable, e.g., in the case of the library catalogue it would mean that only one reader could search the catalogue at a time.

Synchronisation Requirements

To ensure correctness and for maximum efficiency (concurrency) and we require that:

- if a writer is writing to the file, no other writer may write to the file and no reader may read it; and
- any number of readers may simultaneously read the file.

Readers' Preference Protocol

Given a sequence of read and write requests which arrive in the following order:

$$R_1 R_2 W_1 R_3 \ldots$$

in a *Readers' Preference* protocol: R_3 takes priority over W_1

$$R_1 R_2 R_3 W_1 \ldots$$

Writers' Preference Protocol

Given a sequence of read and write requests which arrive in the following order:

$$W_1 W_2 R_1 W_3 \ldots$$

in a *Writers' Preference* protocol: W3 takes priority over R1

$$W_1 W_2 W_3 R_1 \ldots$$

Fairness

Both Readers' Preference and Writers' Preference never allow a reader and a writer or two writers to access the file at the same time

- however they are not *fair* solutions:
- for example, with Readers' Preference, as long as a single reader is active, no writer can gain access but other readers are allowed in.
  - if new readers arrive in rapid succession, W_1 will be indefinitely delayed
  - the librarian would have to wait until the last user was finished with the catalogue before they could update it.

A Fair Solution

A *fair* solution to the Reader's and Writer's problem:

- if there are waiting writers then a new reader is required to wait for the termination of a write; and
- if there are readers waiting for the termination of a write, they have priority over the next write.

Approaches to the Problem

As a (selective) *mutual exclusion* problem – classes of processes compete for access to the file:

- reader processes compete with writers; and
- individual writer processes compete with readers and with each other.

As a *condition synchronisation* problem:

- reader processes must wait until no writers are accessing the file;
- writer processes must wait until there are no readers or other writers accessing the file.

A Monitor Solution

Although the file is shared, we cant encapsulate it in a monitor, since the readers couldnt then access it concurrently:

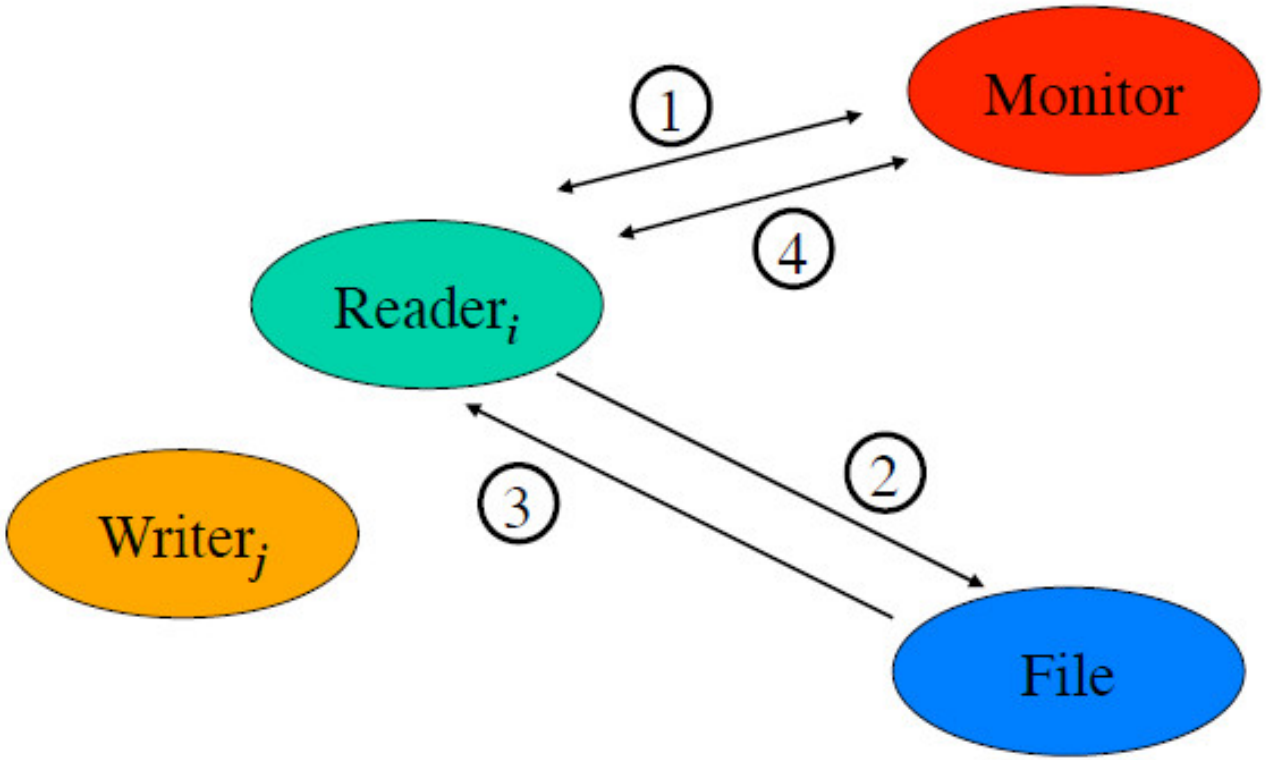
- instead the monitor is used to *arbitrate* access to the file – the file itself is global to the readers and writers.
- this basic structure is often employed in monitor based programs.

An Arbitration Monitor

The arbitration monitor grants permission to access the file:

- processes inform the monitor when they want access to the file (access requests) and when they are finished (release requests)
- with two kinds of access request (read and write) and two release requests (read and write), the monitor has four procedures:
  - `startRead()`
  - `endRead()`
  - `startWrite()`
  - `endWrite()`

Monitor Arbitration



ReadersWriters Monitor

```
monitor ReadersWriters {
    boolean writing = false;
    integer readers = 0,
            waitingReaders = 0;
            watitingWriters = 0;

    condvar okToRead,
            okToWrite;

// monitor procedures follow ...
```

Solution 1

```
procedure startRead() {
    while (writing or
           waitingWriters > 0) {
        wait(okToRead);
    }
    readers++;
}
```

```
procedure endRead() {
    readers--;
    if (readers == 0)
        signal(okToWrite);
}
```

```
procedure startWrite() {
    if (writing or readers > 0 or
        waitingWriters > 0) {
        waitingWriters++;
        wait(okToWrite);
        waitingWriters--;
    }
    writing = true;
}
```

```
procedure endWrite() {
    writing = false;
    if (waitingWriters > 0)
        signal(okToWrite);
    else
        signal_all(okToRead);
}}
```

Solution 2

```
procedure startRead() {
    if (writing) {
        waitingReaders++;
        wait(okToRead);
        waitingReaders--;
    }
    readers++;
}

procedure endRead() {
    readers--;
    if (readers == 0)
        signal(okToWrite);
}
```

```
procedure startWrite() {
    while (writing or
           readers > 0 or
           waitingReaders > 0) {
        wait(okToWrite);
    }
    writing = true;
}

procedure endWrite() {
    writing = false;
    if (waitingReaders > 0)
        signal_all(okToRead);
    else
        signal(okToWrite);
}}
```

Solution 3

```
procedure startRead() {
    if (writing or
        waitingWriters > 0) {
        waitingReaders++;
        wait(okToRead);
        waitingReaders--;
    }
    readers++;
}

procedure endRead() {
    readers--;
    if (readers == 0)
        signal(okToWrite);
}
```

```
procedure startWrite() {
    if (writing or readers > 0 or
        waitingReaders > 0 or
        waitingWriters > 0) {
        waitingWriters++;
        wait(okToWrite);
        waitingWriters--;
    }
    writing = true;
}

procedure endWrite() {
    writing = false;
    if (waitingReaders > 0)
        signal_all(okToRead);
    else
        signal(okToWrite);
}}
```

Exercise

- which solution gives preference to readers
- which solution gives preferences to writers
- which solution is 'fair'

Readers' Preference

```
monitor ReadersPreference {
    boolean writing = false;
    integer readers = 0,
            waitingReaders = 0;

    condvar okToRead,
            okToWrite;

// monitor procedures follow ...
```

Readers' Preference

```
procedure startRead() {
    if (writing) {
        waitingReaders++;
        wait(okToRead);
        waitingReaders--;
    }
    readers++;
}

procedure endRead() {
    readers--;
    if (readers == 0)
        signal(okToWrite);
}
```

```
procedure startWrite() {
    while (writing or
           readers > 0 or
           waitingReaders > 0) {
        wait(okToWrite);
    }
    writing = true;
}

procedure endWrite() {
    writing = false;
    if (waitingReaders > 0)
        signal_all(okToRead);
    else
        signal(okToWrite);
}}
```

Signal and Continue and **while/ if**

- waiting readers are signalled when a writer finishes
- this moves the waiting readers from the `okToRead` delay queue to the monitor entry queue
- writer then releases the monitor lock
  - newly arrived reader will proceed as `writing` is false
  - reader(s) on the entry queue will return from `wait`, decrementing `waitingReaders` and incrementing `readers`
  - newly arrived writer will wait, either because there are active or waiting readers
- waiting readers do not need to recheck their delay condition

Signal and Continue and **while/ if**

- when the last reader finishes, it signals to one waiting writer, *w*
- writer *w* moves from the `okToWrite` delay queue to the monitor entry queue
- if another reader arrives before *w* enters the monitor, it will set `readers > 0`
- if another writer arrives before *w* enters the monitor, it will set `writing` to true
- causing *w* to wait again when it rechecks its delay condition

Writers' Preference

```
monitor WritersPreference {
    boolean writing = false;
    integer readers = 0,
            waitingWriters = 0;

    condvar okToRead,
            okToWrite;

// monitor procedures follow ...
```

Writers' Preference

```
procedure startRead() {
    while (writing or
           waitingWriters > 0) {
        wait(okToRead);
    }
    readers++;
}
```

```
procedure endRead() {
    readers--;
    if (readers == 0)
        signal(okToWrite);
}
```

```
procedure startWrite() {
    if (writing or readers > 0 or
        waitingWriters > 0) {
        waitingWriters++;
        wait(okToWrite);
        waitingWriters--;
    }
    writing = true;
}
```

```
procedure endWrite() {
    writing = false;
    if (waitingWriters > 0)
        signal(okToWrite);
    else
        signal_all(okToRead);
}}
```

A Fair Solution

```
monitor ReadersWriters {
    boolean writing = false;
    integer readers = 0,
            waitingReaders =0,
            waitingWriters = 0;

    condvar okToRead,
            okToWrite;

// monitor procedures follow ...
```

A Fair Solution

```
procedure startRead() {
    if (writing or
        waitingWriters > 0) {
        waitingReaders++;
        wait(okToRead);
        waitingReaders--;
    }
    readers++;
}

procedure endRead() {
    readers--;
    if (readers == 0)
        signal(okToWrite);
}
```

```
procedure startWrite() {
    if (writing or readers > 0 or
        waitingReaders > 0 or
        waitingWriters > 0) {
        waitingWriters++;
        wait(okToWrite);
        waitingWriters--;
    }
    writing = true;
}

procedure endWrite() {
    writing = false;
    if (waitingReaders > 0)
        signal_all(okToRead);
    else
        signal(okToWrite);
}}
```

Efficiency

- in practice, exclusive writes and concurrent reads are not sufficient to gain the necessary performance in large database systems
- the internal structure of the database must be used to limit the area to which a write lock is imposed
- this area then behaves as if it were supporting a readers and writers protocol