G52CON:
Concepts of Concurrency

Lecture 10 Monitors II

Brian Logan

School of Computer Science

bsl@cs.nott.ac.uk

Outline of this lecture

- Readers and Writers problem

- selective mutual exclusion problems

- scheduling policies

  – readers' preference

  – writers' preference

  – a fair solution

- monitor solutions

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: W_3 takes priority over R_1

$$W_1 \ W_2 \ W_3 \ R_1 \ \ldots$$

Fairness

Both Readers' Preference and Writers' Preference never allow a reader and and 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 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 can't encapsulate it in a monitor, since the readers couldn't 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
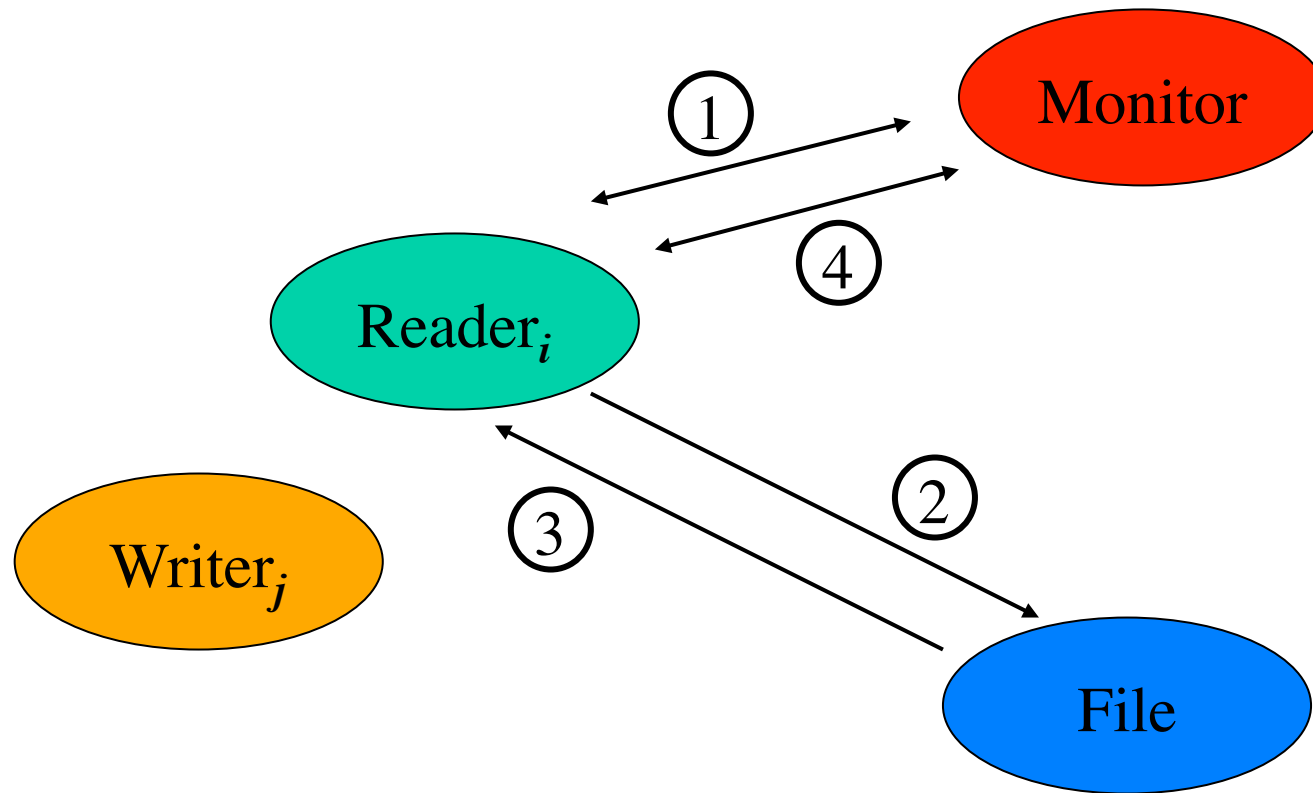
ReadersWriters monitor

```
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 1

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


    condvar okToRead,
            okToWrite;


// monitor procedures follow ...
```

Readers' Preference 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);
}}
```

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 1

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


    condvar okToRead,
            okToWrite;


// monitor procedures follow ...
```

Writers' Preference 2

```
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 1

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


    condvar okToRead,
            okToWrite;


// monitor procedures follow ...
```

A fair solution 2

```
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

The next lecture

Synchronisation in Java

Suggested reading:

- Lea (2000), chapter 2.

Sun Java Tutorial, Synchronizing Threads

`docs.oracle.com/javase/tutorial/essential/concurrency/sync.html`