Dekker's Algorithm

January 16, 2019

- Critical sections
- Implementing critical sections
- Critical sections in low level code — A challenge

Critical Sections

1: Critical sections

1.1: Introduction

- non-critical sections — both (any number of) processes may run in parallel
- critical section — execution of critical sections may not overlap, e.g. resource allocation

In the abstract:

```
non-critical section;
begin critical section {// pre-protocol
   critical actions;
}  end critical section // post-protocol
non-critical section;
```

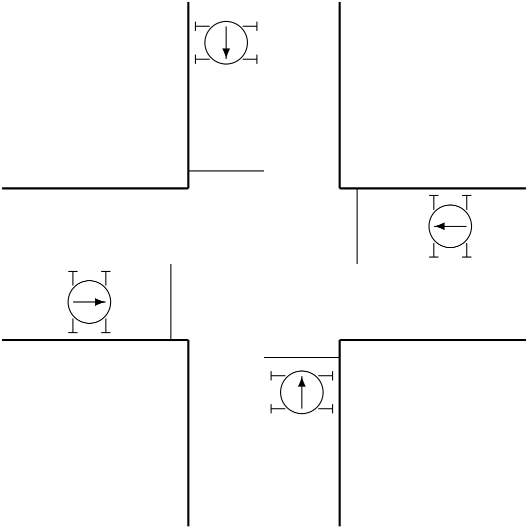
Properties

1.2: Properties required

- Mutual exclusion
- No deadlock
- No starvation
- Liveness
- Loosely connectedness

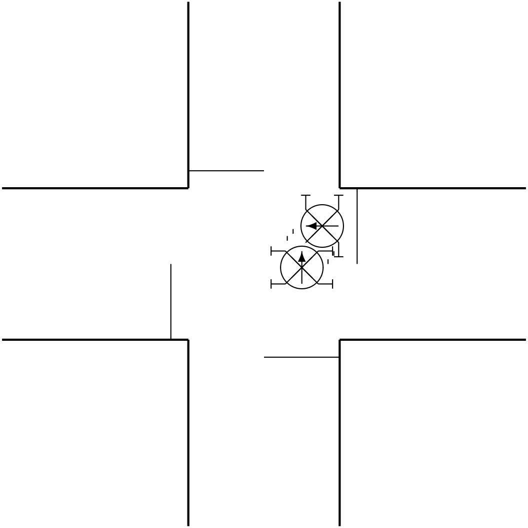
Properties

An analogy for concurrent processes



Mutual Exclusion

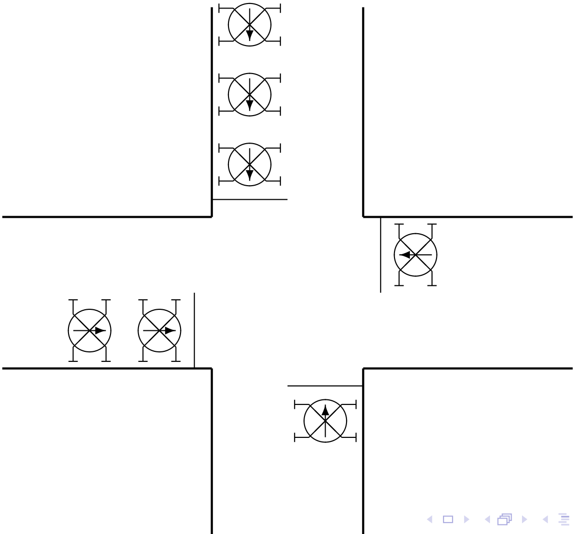
1.2.1: Mutual exclusion



Deadlock

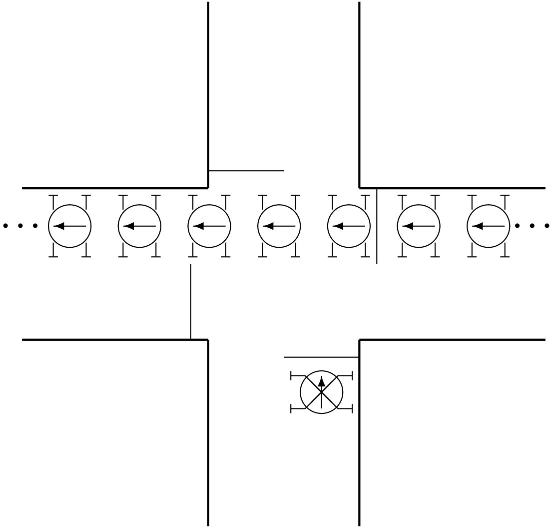
1.2.2: Deadlock

An example (*priorité à droite/voorrang van rechts*):



Starvation

1.2.3: Starvation



Liveness

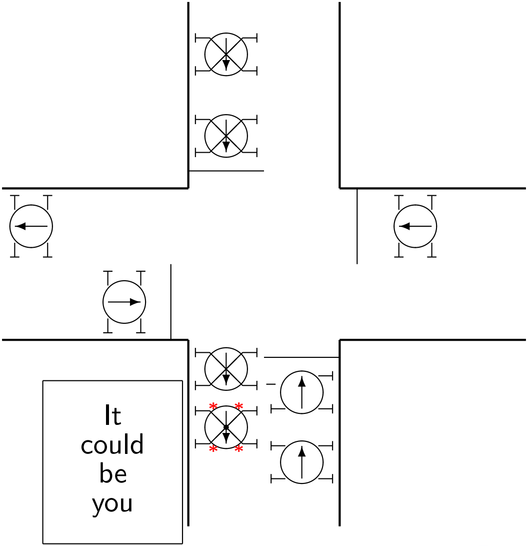
1.2.4: Liveness

- system is live if there is no chance of deadlock or starvation
- proc(p): chance that process p will be able to proceed.

$$
\begin{aligned}
\text{deadlock:} &\quad \text{proc(p)} = \mathbf{0} \\
\text{starvation:} &\quad \mathbf{0} < \text{proc(p)} < \mathbf{1} \\
\text{liveness:} &\quad \text{proc(p)} = \mathbf{1}
\end{aligned}
$$

Loosely Connectedness

1.2.5: Loosely connected processes



Implementation

2: Implementing critical sections

2.1: Reynold's criterium

> *The result of any instruction with at most one* critical reference *is locally deterministic.*
>
> *Critical reference:*
> *A read or write action on a shared variable.*

Note: this is not a *law* but an *assumption*.

The Model

2.2: The model

- Two processes communicating via shared variables.
- The processes are interleaved.
- Reynold's criterium is satisfied.
- Any sequence of time slices is possible.

General Structure

General structure

```
public class Process extends Thread {

    private int id; // local variable

    public Process(int id) {
        this.id = id;
    }

    public void run() {
        while (true) {
            pre-protocol;
            critical section;
            post-protocol;
        }  // end while loop
    }  // end method
}
```

Taking Turns

2.3: The attempts
2.3.1: First attempt: Taking turns

```
// shared variable
private static int turn = 0;

public void run() {
    while (true) {
        while (turn != id); // wait loop — do nothing
        critical section;
        turn = (id + 1) % 2; // set turn to other process's id
    }
}
```

Taking Turns

- Mutual exclusion — yes, actions on `turn` are atomic. `turn` must be 0 or 1.
- Deadlock — no, each process takes turns
- Starvation — no, each process takes turns
- Liveness — yes, no deadlock or starvation
- Loosely connected — no, if process dies outside CS other process is stuck.
  Also — slowest process determines speed.

Indicating Presence

2.3.2: Attempt 2: Indicating presence

```
// shared variables
private static boolean[] procInCS = {false,false} ;

public void run() {
    while (true) {
        while (procInCS[(id+1)%2]); // wait loop
        procInCS[id] = true;
        critical section;
        procInCS[id] = false;
    }
}
```

Indicating Presence

No mutual exclusion:

	procInCS[0]	procInCS[1]
	false	**false**
proc(0): ?procInCS[1]	**false**	**false**
proc(1): ?procInCS[0]	**false**	**false**
proc(0): procInCS[0] = **true**	**true**	**false**
proc(1): procInCS[1] = **true**	**true**	**true**

Indicating Presence

Other properties:

- Deadlock — no, there is always a chance that `procInCS[id]` will become **false**.
- Starvation — yes, if one process rushes round the loop before the other can test `procInCS[id]`.
- Liveness — no, since there can be starvation
- Loosely connected — yes, if a process dies outside the critical section (including protocols) its `procInCS[id]` will be false.

Indicating Intention

2.3.3: Attempt 3: Indicating intention

```
// shared variables
private static boolean[] procReqCS = {false,false} ;

public void run() {
    while (true) {
        procReqCS[id] = true;
        while (procReqCS[(id+1)%2]); // wait loop
        critical section;
        procReqCS[id] = false;
    }
}
```

Indicating Intention

Deadlock:

	procReqCS[0]	procReqCS[1]
	false	false
proc(0): procReqCS[0] = true	true	false
proc(1): procReqCS[1] = true	true	true
proc(0): ?procReqCS[1]	true	true
proc(1): ?procReqCS[0]	true	true

Indicating Intention

- Mutual exclusion — yes, process sets `procReqCS[id]` before test.
- Starvation — yes, again a process can rush round.
- Liveness — no, deadlock and starvation.
- Loosely connected — yes, outside CS `procReqCS[id]` is **false**

Indicating and Rescinding Intention

2.3.4: Attempt 4: Indicating and rescinding intention

```
public void run() {
    while (true) {
        procReqCS[id] = true;
        while (procReqCS[(id+1) % 2]) {// wait loop
            procReqCS[id] = false; // rescind intention
            procReqCS[id] = true; // reinstate intention
        }
        critical section;
        procReqCS[id] = false;
    }
}
```

Indicating and Rescinding Intention

- Mutual exclusion — yes, see previous attempt.
- Deadlock — no, the previous scenario can no longer occur.
- Starvation — yes, similar to previous attempt.
- Liveness — no, starvation.
- Loosely connected — yes.

Dekker's Algorithm

2.3.5: Attempt 5: Dekker's algorithm

A combination of attempts 1 and 4. Uses

```
// shared variables
private static int turn = 0;
private static boolean[] procReqCS = {false,false} ;
```

The `run` method...

Dekker's Algorithm

```
while (true) {
    procReqCS[id] = true;
    while (procReqCS[(id+1) % 2]) {
        if (turn == (id+1) % 2) {// other process has priority
            procReqCS[id] = false; // rescind intention
            while (turn != id) ; // wait for priority
            procReqCS[id] = true; // reinstate intention
        }
    }
    critical section;
    turn = (id+1) % 2; procReqCS[id] = false;
}
```

Dekker's Algorithm

- Mutual exclusion — yes.
- Deadlock — no.
- Starvation — no.
- Liveness — yes.
- Loosely connected — yes.

Critical sections in low level code

3: Critical sections in low level code — A challenge

```
      SUB :a:  :b:    ; programme set up
      JMP LWY :c:
:a:   SUB :d:  :a:
:b:   JMP LWY :d:     ; end of programme set up

:c:                   ; start of concurrent code
                      ; . . . non-critical section
:d:   SUB :d:  :a:    ; preprotocol
                      ; . . . critical section
      ADD :d:  :a:    ; postprotocol
                      ; . . . non-critical section
```

End of Dekker's algorithm lecture