Concurrency Control

COMP3231 Operating Systems

2004/S2

WHAT IS CONCURRENCY CONTROL?

Concurrency appears in many contexts:

- ➜ Multi-threading: concurrent threads share an address space
- ➜ Multi-programming: concurrent processes execute on a uniprocessor
- ➜ Multi-processing: concurrent processes on a multiprocessor
- ➜ Distributed processing: concurrent processes executing on multiple nodes connected by a network

What is Concurrency Control?

Concurrency appears in many contexts:

- ➜ Multi-threading: concurrent threads share an address space
- ➜ Multi-programming: concurrent processes execute on a uniprocessor
- ➜ Multi-processing: concurrent processes on a multiprocessor
- ➜ Distributed processing: concurrent processes executing on multiple nodes connected by a network

Concurrency is also used in different forms:

- ➜ Multiple applications (multiprogramming)
- ➜ Structured application (application is a set of concurrent threads or processes)
- ➜ Operating-system structure (OS is a set of threads or processes)

Concurrent processes (threads) need special support:

- ➜ Communication among processes
- ➜ Allocation of processor time
- ➜ Sharing of resources
- ➜ Synchronisation of multiple processes

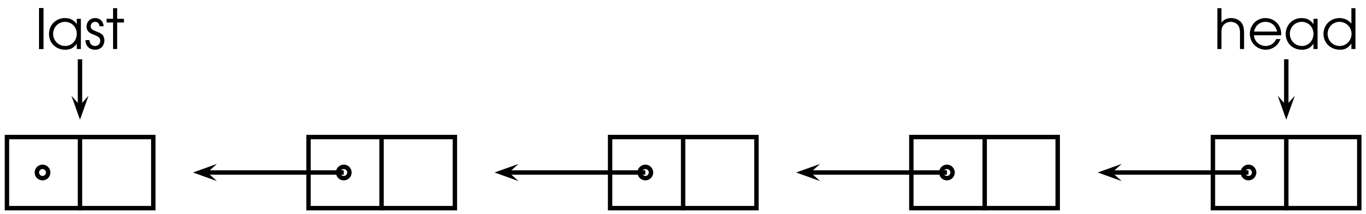
Concurrent processes (threads) need special support:

- ➜ Communication among processes
- ➜ Allocation of processor time
- ➜ Sharing of resources
- ➜ Synchronisation of multiple processes

Concurrency can be dangerous to the unwary programmer:

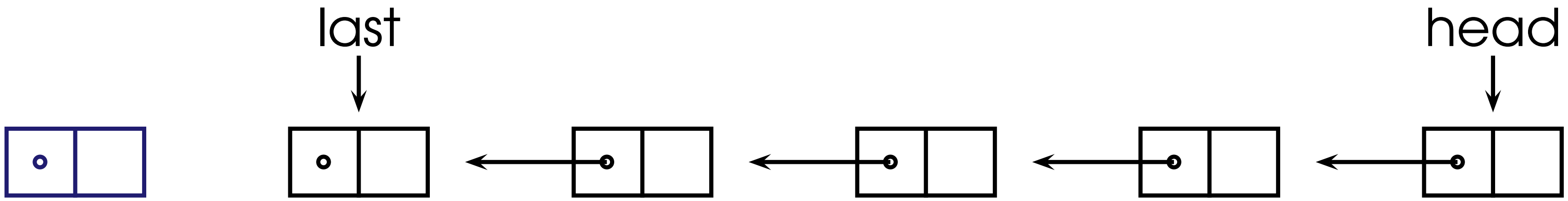
- ➜ Sharing global resources (order of read and write operations)
- ➜ Management of allocation of resources (danger of deadlock)
- ➜ Programming errors difficult to locate (Heisenbugs)

Concurrent access to a global queue



:

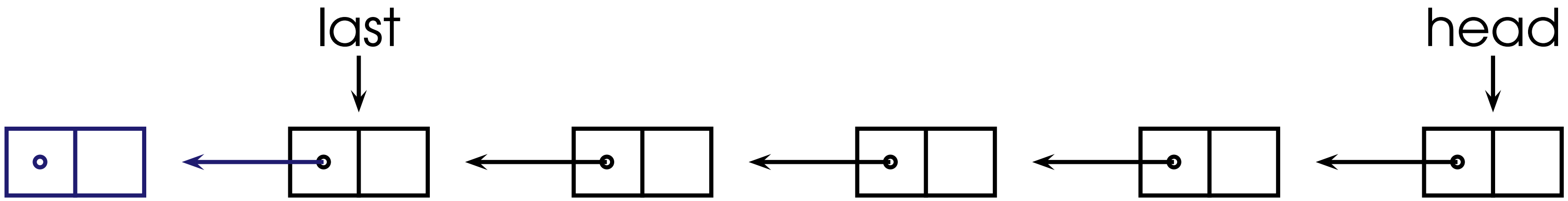
CONCURRENT ACCESS TO A GLOBAL QUEUE



Inserting:

① create new object

Concurrent access to a global queue

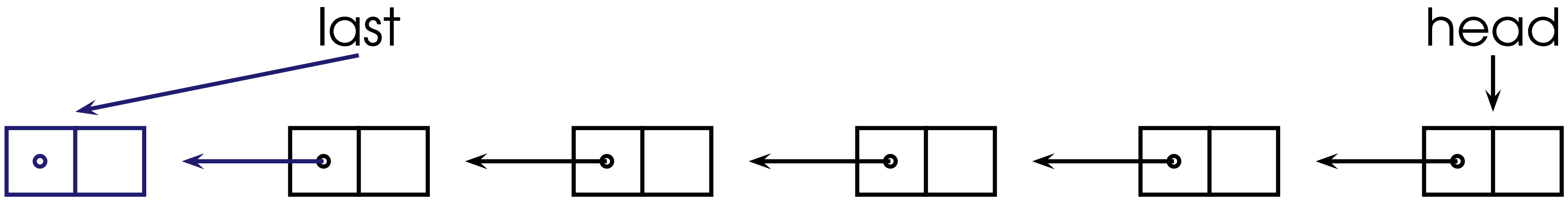


Inserting:

① create new object

② set `last->next` to `&new`

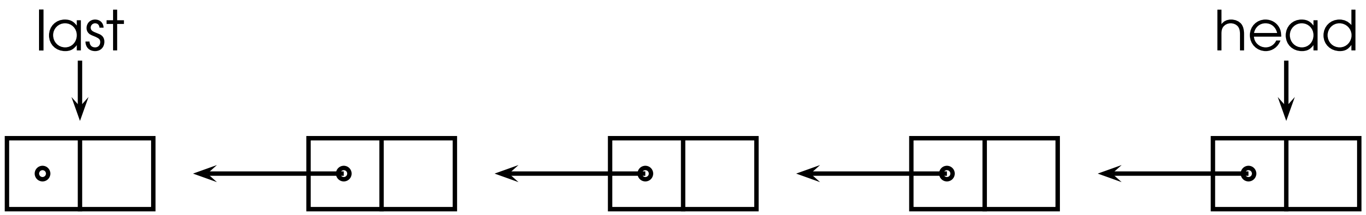
Concurrent access to a global queue



Inserting:

1. create new object
2. set `last->next` to `&new`
3. set `last` to `&new`

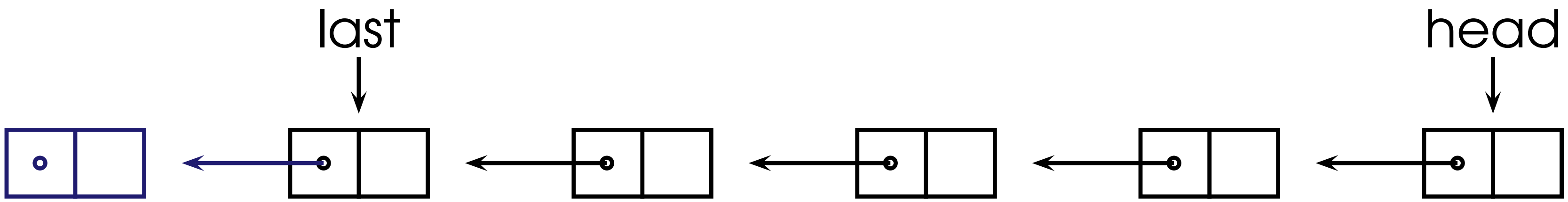
CONCURRENT ACCESS TO A GLOBAL QUEUE



Thread A:

Thread B:

Concurrent access to a global queue

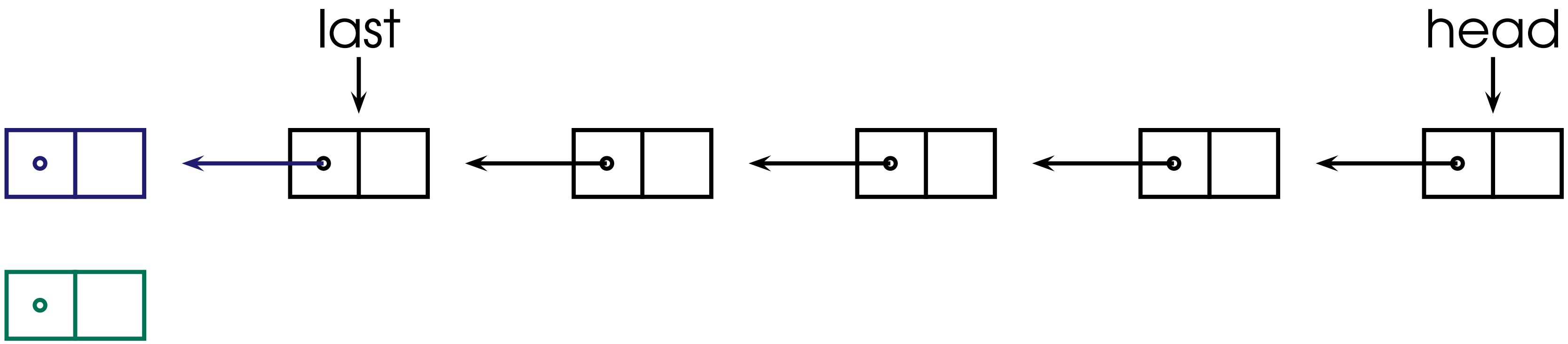


Thread A:

- ➜ create new object
- ➜ set `last->next` to `&new`

Thread B:

Concurrent access to a global queue



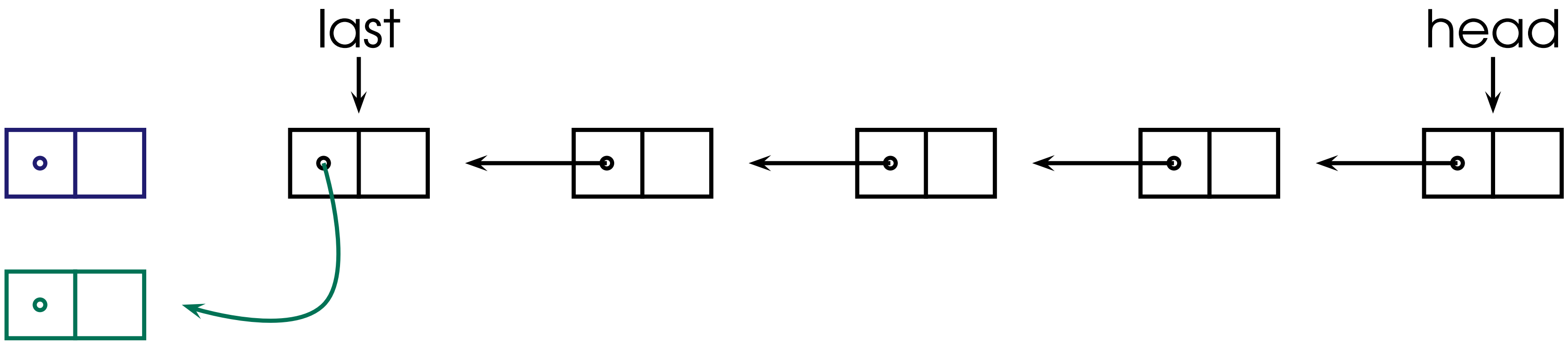
Thread A:

- ➜ create new object
- ➜ set `last->next` to `&new`

Thread B:

- ➜ create new object

CONCURRENT ACCESS TO A GLOBAL QUEUE



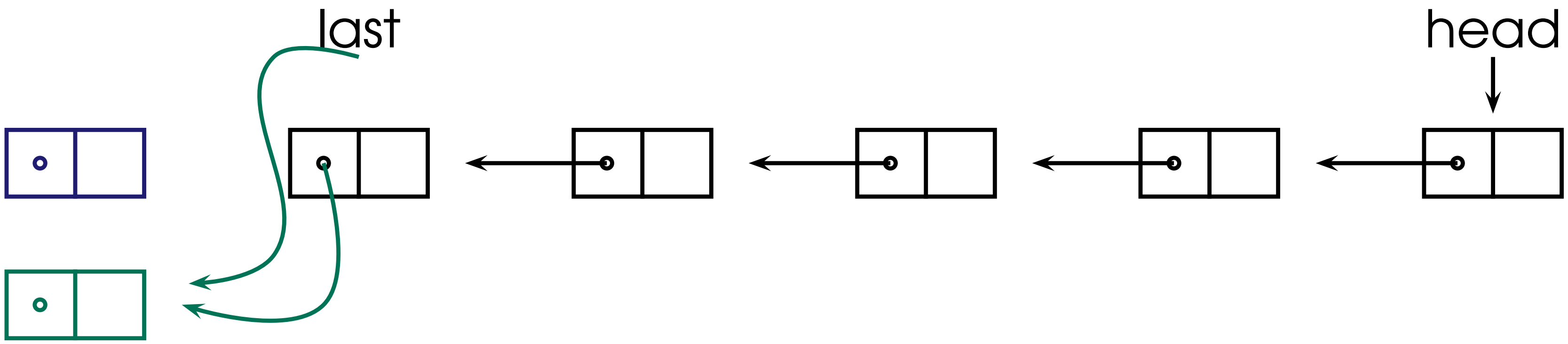
Thread A:

- create new object
- set `last->next` to `&new`

Thread B:

- create new object
- set `last->next` to `&new`

Concurrent access to a global queue



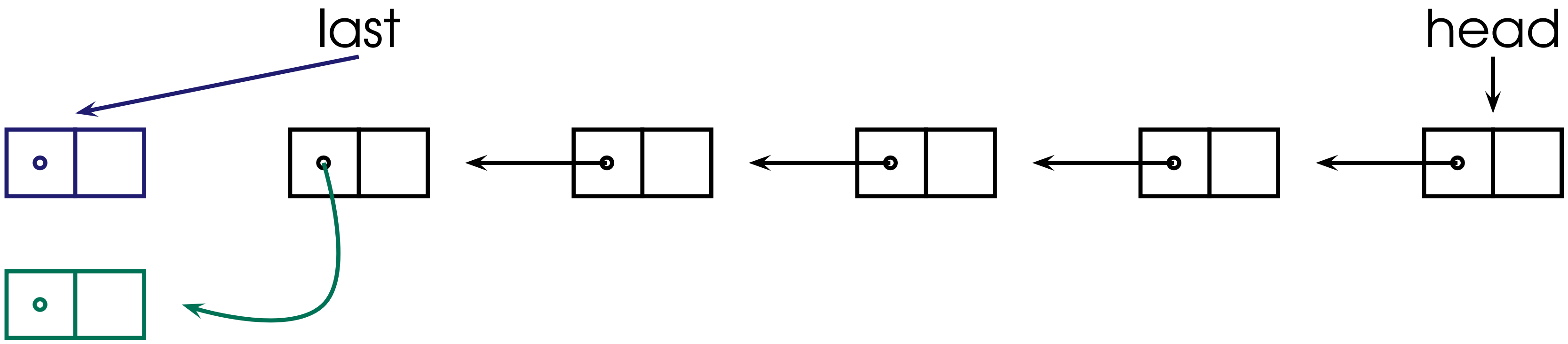
Thread A:

- create new object
- set `last->next` to `&new`

Thread B:

- create new object
- set `last->next` to `&new`
- set `last` to `&new`

Concurrent access to a global queue



Thread A:

- create new object
- set `last->next` to `&new`

Thread B:

- create new object
- set `last->next` to `&new`
- set `last` to `&new`

Thread A:

- set `last` to `&new`

We can get the same problem with truly parallel threads:

Thread A	Thread B
⋮	⋮
create new object	⋮
⋮	create new object
⋮	`last->next = &new`
`last->next = &new`	⋮
`last = &new`	⋮

We can get the same problem with truly parallel threads:

Thread A	Thread B
⋮	⋮
create new object	⋮
⋮	create new object
⋮	`last->next = &new`
`last->next = &new`	⋮
`last = &new`	⋮

Lessons learned:

We can get the same problem with truly parallel threads:

Thread A	Thread B
⋮	⋮
create new object	⋮
⋮	create new object
⋮	`last->next = &new`
`last->next = &new`	⋮
`last = &new`	⋮

Lessons learned:

➜ We have to control access to shared resource (such as shared variables)

We can get the same problem with truly parallel threads:

Thread A	Thread B
⋮	⋮
create new object	⋮
⋮	create new object
⋮	`last->next = &new`
`last->next = &new`	⋮
`last = &new`	⋮

Lessons learned:

- ➜ We have to control access to shared resource (such as shared variables)
- ➜ We can do this effectively by controlling access to the code utilising those shared resources ⇒ critical sections

Concurrent access to a global queue

Only one thread at a time should have write access to the queue:

➜ Thread A creates new object, sets `last->next` pointer

CONCURRENT ACCESS TO A GLOBAL QUEUE

Only one thread at a time should have write access to the queue:

- ➜ Thread A creates new object, sets `last->next` pointer
- ➜ Thread A is suspended

Concurrent access to a global queue

Only one thread at a time should have write access to the queue:

- Thread A creates new object, sets `last->next` pointer
- Thread A is suspended
- Thread B is scheduled, calls `insert`, but since Thread A is currently in `insert`, has to wait

Concurrent access to a global queue

Only one thread at a time should have write access to the queue:

- ➜ Thread A creates new object, sets `last->next` pointer
- ➜ Thread A is suspended
- ➜ Thread B is scheduled, calls `insert`, but since Thread A is currently in `insert`, has to wait
- ➜ Thread A is resumed, the data structure is in the same state as it was when it was suspended.

Concurrent access to a global queue

Only one thread at a time should have write access to the queue:

- ➜ Thread A creates new object, sets `last->next` pointer
- ➜ Thread A is suspended
- ➜ Thread B is scheduled, calls `insert`, but since Thread A is currently in `insert`, has to wait
- ➜ Thread A is resumed, the data structure is in the same state as it was when it was suspended.
- ➜ Thread A completes operation

Concurrent access to a global queue

Only one thread at a time should have write access to the queue:

- ➜ Thread A creates new object, sets `last->next` pointer
- ➜ Thread A is suspended
- ➜ Thread B is scheduled, calls `insert`, but since Thread A is currently in `insert`, has to wait
- ➜ Thread A is resumed, the data structure is in the same state as it was when it was suspended.
- ➜ Thread A completes operation
- ➜ Thread B is allowed to execute `insert`

Concurrency Control

➜ Processes can

- compete for resources
  - Processes may not be aware of each other
  - execution must not be affected by each other
  - OS is responsible for controlling access

Concurrency Control

➜ Processes can

- compete for resources
  - Processes may not be aware of each other
  - execution must not be affected by each other
  - OS is responsible for controlling access
- cooperate by sharing a common resource
  - Programmer responsible for controlling access
  - Hardware, OS, programming language may provide support

Concurrency Control

- ➜ Processes can
  - compete for resources
    - Processes may not be aware of each other
    - execution must not be affected by each other
    - OS is responsible for controlling access
  - cooperate by sharing a common resource
    - Programmer responsible for controlling access
    - Hardware, OS, programming language may provide support
- ➜ Threads of a process usually do not compete, but cooperate.

CONCURRENCY CONTROL

- Processes can
  - compete for resources
    - Processes may not be aware of each other
    - execution must not be affected by each other
    - OS is responsible for controlling access
  - cooperate by sharing a common resource
    - Programmer responsible for controlling access
    - Hardware, OS, programming language may provide support
- Threads of a process usually do not compete, but cooperate.
- Since process access to shared resources is through OS, problems are the same (although solved on different levels)
  - e.g., kernel threads of different competing processes cooperate

We face three control problems:

① Mutual exclusion: critical resources $\Rightarrow$ critical sections

- Only one process at a time is allowed in a critical section
- Example: only one process at a time is allowed to send commands to the printer

We face three control problems:

① Mutual exclusion: critical resources $\Rightarrow$ critical sections

- Only one process at a time is allowed in a critical section
- Example: only one process at a time is allowed to send commands to the printer

② Deadlock: e.g., two processes and two resources

We face three control problems:

① Mutual exclusion: critical resources $\Rightarrow$ critical sections

- Only one process at a time is allowed in a critical section
- Example: only one process at a time is allowed to send commands to the printer

② Deadlock: e.g., two processes and two resources

③ Starvation: e.g., three processes compete for a resource

We face three control problems:

① Mutual exclusion: critical resources $\Rightarrow$ critical sections

- Only one process at a time is allowed in a critical section
- Example: only one process at a time is allowed to send commands to the printer

② Deadlock: e.g., two processes and two resources

③ Starvation: e.g., three processes compete for a resource

Let's look at these problems in turn

Mutual exclusion illustrated:

```
void proc (int i)
{
  for (;;) {
    entercritical (i);  /* blocks if other thread
                           already in critical section */
    <critical section>
    exitcritical (i);   /* allow other threads to
                           enter */
    <remainder>
  }
}
void main () {
   parbegin (proc (R_1), proc (R_2), ..., proc (R_n));
}
```

Mutual exclusion illustrated:

```
void proc (int i)
{
  for (;;) {
    entercritical (i);  /* blocks if other thread
                           already in critical section */
    <critical section>
    exitcritical (i);   /* allow other threads to
                           enter */
    <remainder>
  }
}
void main () {
   parbegin (proc (R_1), proc (R_2), ..., proc (R_n));
}
```

But how can we implement `entercritical()` and `exitcritical()`?

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource
- ➜ No deadlock or starvation

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource
- ➜ No deadlock or starvation
- ➜ A thread must not be delayed access to a critical section when there is no other thread using it

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource
- ➜ No deadlock or starvation
- ➜ A thread must not be delayed access to a critical section when there is no other thread using it
- ➜ A thread that halts in its non-critical section must do so without interfering with other thread

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource
- ➜ No deadlock or starvation
- ➜ A thread must not be delayed access to a critical section when there is no other thread using it
- ➜ A thread that halts in its non-critical section must do so without interfering with other thread
- ➜ No assumptions are made about relative thread speeds or number of processes

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource
- ➜ No deadlock or starvation
- ➜ A thread must not be delayed access to a critical section when there is no other thread using it
- ➜ A thread that halts in its non-critical section must do so without interfering with other thread
- ➜ No assumptions are made about relative thread speeds or number of processes

Usage:

- ➜ A thread remains inside its critical section for a finite time only

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource
- ➜ No deadlock or starvation
- ➜ A thread must not be delayed access to a critical section when there is no other thread using it
- ➜ A thread that halts in its non-critical section must do so without interfering with other thread
- ➜ No assumptions are made about relative thread speeds or number of processes

Usage:

- ➜ A thread remains inside its critical section for a finite time only
- ➜ No potentially blocking operations should be executed inside a critical section

Requirements for Mutual Exclusion

Implementation:

- ➜ Only one thread at a time is allowed in the critical section for a resource
- ➜ No deadlock or starvation
- ➜ A thread must not be delayed access to a critical section when there is no other thread using it
- ➜ A thread that halts in its non-critical section must do so without interfering with other thread
- ➜ No assumptions are made about relative thread speeds or number of processes

Usage:

- ➜ A thread remains inside its critical section for a finite time only
- ➜ No potentially blocking operations should be executed inside a critical section
- ➜ No deadlock or starvation

Conceptually, there are three ways to satisfy the implementation requirements:

Conceptually, there are three ways to satisfy the implementation requirements:

① Software approach: put responsibility on the processes themselves

Conceptually, there are three ways to satisfy the implementation requirements:

① Software approach: put responsibility on the processes themselves

② Systems approach: provide support within operation system or programming language

Conceptually, there are three ways to satisfy the implementation requirements:

① Software approach: put responsibility on the processes themselves

② Systems approach: provide support within operation system or programming language

③ Hardware approach: special-purpose machine instructions

Software Approaches To Mutual Exclusion

Premises:

➜ One or more processes with shared memory

SOFTWARE APPROACHES TO MUTUAL EXCLUSION

Premises:

➜ One or more processes with shared memory

➜ Elementary mutual exclusion at level of memory accesses:
  - simultaneous accesses to the same memory location are serialised

Software Approaches To Mutual Exclusion

Premises:

- ➜ One or more processes with shared memory
- ➜ Elementary mutual exclusion at level of memory accesses:
  - simultaneous accesses to the same memory location are serialised

In the following, Dijkstra's presentation of Dekker's algorithm

Software Approaches To Mutual Exclusion

Premises:

- ➜ One or more processes with shared memory
- ➜ Elementary mutual exclusion at level of memory accesses:
  - simultaneous accesses to the same memory location are serialised

In the following, Dijkstra's presentation of Dekker's algorithm (actually, we use Peterson's algorithm, which is a more elegant variant of Dekker's)

A First Attempt

➜ The Plan:
  - threads take turns in executing critical section

A FIRST ATTEMPT

- ➜ The Plan:
  - threads take turns in executing critical section
  - exploit serialisation of memory access to implement serialisation of access to critical section
  - ➨ mutual exclusion

A First Attempt

- ➜ The Plan:
  - threads take turns in executing critical section
  - exploit serialisation of memory access to implement serialisation of access to critical section
  - ➨ mutual exclusion
- ➜ We employ a variable (memory location) `turn` that indicates whose turn it is to enter the critical section:

A First Attempt

- ➜ The Plan:
  - threads take turns in executing critical section
  - exploit serialisation of memory access to implement serialisation of access to critical section
  - ➨ mutual exclusion
- ➜ We employ a variable (memory location) `turn` that indicates whose turn it is to enter the critical section:

P_0:

```
...
while (turn != 0)
  /* do nothing */;
<critical section>
turn = 1;
...
```

P_1:

```
...
while (turn != 1)
  /* do nothing */;
<critical section>
turn = 0;
...
```

Busy waiting (spin lock):

- ➜ Process is always checking to see if it can enter the critical section
- ✔ implements mutual exclusion
- ✔ simple

Busy waiting (spin lock):

- ➜ Process is always checking to see if it can enter the critical section
- ✔ implements mutual exclusion
- ✔ simple
- ✘ Process burns resources while waiting

Busy waiting (spin lock):

- ➜ Process is always checking to see if it can enter the critical section
- ✔ implements mutual exclusion
- ✔ simple
- ✘ Process burns resources while waiting

Other drawbacks of this code:

- ✘ Processes must alternate access to the critical section
- ✘ if one process fails anywhere in the program, the other is permanently blocked

THE SECOND ATTEMPT

- ➜ The Problem: `turn` stores who can enter the critical section, rather then whether anybody may enter the critical section

The Second Attempt

- The Problem: `turn` stores who can enter the critical section, rather then whether anybody may enter the critical section
- The New Plan: we store for each process whether it is in the critical section right now (in a Boolean `flag`):

  `flag[i]`: Process `i` is in the critical section

The Second Attempt

- ➜ The Problem: `turn` stores who can enter the critical section, rather then whether anybody may enter the critical section
- ➜ The New Plan: we store for each process whether it is in the critical section right now (in a Boolean `flag`):

  `flag[i]`: Process `i` is in the critical section

P_0:

```
...
while (flag[1])
  /* do nothing */;
flag[0] = true;
<critical section>
flag[0] = false;
...
```

P_1:

```
...
while (flag[0])
  /* do nothing */;
flag[1] = true;
<critical section>
flag[1] = false;
...
```

Is this a good solution?

Is this a good solution?

➜ If one thread fails
  - ✔ outside of the critical section, the other is not blocked
  - ✘ inside a critial section, other thread is blocked (however, hard to avoid)

Is this a good solution?

- ➜ If one thread fails
  - ✔ outside of the critical section, the other is not blocked
  - ✘ inside a critial section, other thread is blocked (however, hard to avoid)
- ➜ But: it does not even guarantee exclusive access!!!

Is this a good solution?

- ➜ If one thread fails
  - ✔ outside of the critical section, the other is not blocked
  - ✘ inside a critial section, other thread is blocked (however, hard to avoid)
- ➜ But: it does not even guarantee exclusive access!!!
  1. both flags are set to false
  2. T_0 enters critical section
  3. T_1 enters critical section
  4. T_1 sets `flag[0]`
  5. T_0 sets `flag[1]`
- ➜ worse if more than two threads involved

THIRD ATTEMPT

➜ The Goal: we have to get rid of the gap between toggling the two flags

THIRD ATTEMPT

➜ The Goal: we have to get rid of the gap between toggling the two flags

➜ Yet Another Plan: move setting the flag before checking whether we can enter

THIRD ATTEMPT

- ➜ The Goal: we have to get rid of the gap between toggling the two flags
- ➜ Yet Another Plan: move setting the flag before checking whether we can enter

P_0:

```
...
flag[0] = true;
while (flag[1])
  /* do nothing */;
<critical section>
flag[0] = false;
...
```

P_1:

```
...
flag[1] = true;
while (flag[0])
  /* do nothing */;
<critical section>
flag[1] = false;
...
```

THIRD ATTEMPT

- **The Goal:** we have to get rid of the gap between toggling the two flags
- **Yet Another Plan:** move setting the flag before checking whether we can enter

P_0:

```
...
flag[0] = true;
while (flag[1])
  /* do nothing */;
<critical section>
flag[0] = false;
...
```

P_1:

```
...
flag[1] = true;
while (flag[0])
  /* do nothing */;
<critical section>
flag[1] = false;
...
```

Is this working?

Third Attempt

- **The Goal:** we have to get rid of the gap between toggling the two flags
- **Yet Another Plan:** move setting the flag before checking whether we can enter

P_0:

```
...
flag[0] = true;
while (flag[1])
  /* do nothing */;
<critical section>
flag[0] = false;
...
```

P_1:

```
...
flag[1] = true;
while (flag[0])
  /* do nothing */;
<critical section>
flag[1] = false;
...
```

Is this working?

- Nice try, but we lose again! — The gap can cause a deadlock now

Fourth Attempt

➜ Previous problem: process sets its own state before knowing the other process' state and cannot back off

FOURTH ATTEMPT

- Previous problem: process sets its own state before knowing the other process' state and cannot back off
- Our plan: Process retracts its decision if it cannot enter

Fourth Attempt

- Previous problem: process sets its own state before knowing the other process' state and cannot back off
- Our plan: Process retracts its decision if it cannot enter

P_0:

```
...
flag[0] = true;
while (flag[1]) {
  flag[0] = false;
  delay ();
  flag[0] = true;
}
<critical section>
flag[0] = false;
```

P_0:

```
...
flag[1] = true;
while (flag[0]) {
  flag[1] = false;
  delay ();
  flag[1] = true;
}
<critical section>
flag[1] = false;
```

FOURTH ATTEMPT

- Previous problem: process sets its own state before knowing the other process' state and cannot back off
- Our plan: Process retracts its decision if it cannot enter

P_0:

```
...
flag[0] = true;
while (flag[1]) {
  flag[0] = false;
  delay ();
  flag[0] = true;
}
<critical section>
flag[0] = false;
```

P_0:

```
...
flag[1] = true;
while (flag[0]) {
  flag[1] = false;
  delay ();
  flag[1] = true;
}
<critical section>
flag[1] = false;
```

Did we finally make it?

Fourth Attempt

- ➜ Previous problem: process sets its own state before knowing the other process' state and cannot back off
- ➜ Our plan: Process retracts its decision if it cannot enter

P_0:

```
...
flag[0] = true;
while (flag[1]) {
  flag[0] = false;
  delay ();
  flag[0] = true;
}
<critical section>
flag[0] = false;
```

P_0:

```
...
flag[1] = true;
while (flag[0]) {
  flag[1] = false;
  delay ();
  flag[1] = true;
}
<critical section>
flag[1] = false;
```

Did we finally make it?

- ➜ Close, but we may have a livelock

Tweaking the code:

- ➜ We can solve this problem by combining the fourth with the first attempt
- ➜ In addition to the `flag`'s, we use a variable indicating whose `turn` it is to have precedence in entering the critical section

Instead of Dekker's original algorithm, let's consider Peterson's:

P_0:

```
...
flag[0] = true;
turn = 1;
while (flag[1]
        && turn == 1)
  /* do nothing */;
<critical section>
flag[0] = false;
...
```

P_1:

```
...
flag[1] = true;
turn = 0;
while (flag[0]
        && turn == 0)
  /* do nothing */;
<critical section>
flag[1] = false;
...
```

- Both processes are courteous and solve a tie in favour of the other
- Algorithm can easily be generalised to work with n processes