Mutual Exclusion

The general mutual exclusion scenario

- N processes execute (indefinitely) instruction sequences concurrently. Each instruction belongs to either a critical or non-critical section.

- Safety property "Mutual exclusion": Instructions from critical sections of two or more processes must never be interleaved!

- Further assumptions:
  - Pre-and post-protocols can be executed before and after each critical section.
  - Processes may delay indefinitely in non-critical sections.
  - Processes do not delay indefinitely in critical sections.

Mutual exclusion: Atomic load & store operations

The general mutual exclusion scenario

- Safety property "Mutual exclusion": Instructions from critical sections of two or more processes must never be interleaved!

- Further assumptions:
  - Pre-and post-protocols can be executed before and after each critical section.
  - Processes may delay indefinitely in non-critical sections.
  - Processes do not delay indefinitely in critical sections.

Mutual exclusion: Atomic load & store operations

- No deadlock: If one or multiple processes try to enter their critical sections then exactly one of them must succeed.

- No starvation: Every process which tries to enter one of its critical sections must succeed eventually.

- Efficiency: The decision which process may enter the critical section must be made efficiently in all cases, i.e. also when there is no contention in the first place.

Problem specification

- N processes execute (indefinitely) instruction sequences concurrently. Each instruction belongs to either a critical or non-critical section.

- Safety property "Mutual exclusion": Instructions from critical sections of two or more processes must never be interleaved!

- Further assumptions:
  - Pre-and post-protocols can be executed before and after each critical section.
  - Processes may delay indefinitely in non-critical sections.
  - Processes do not delay indefinitely in critical sections.

References for this chapter

[Ben-Ari06] M. Ben-Ari
Principles of Concurrent and Distributed Programming

Mutual Exclusion

Problem specification

- N processes execute (indefinitely) instruction sequences concurrently. Each instruction belongs to either a critical or non-critical section.

- Safety property "Mutual exclusion": Instructions from critical sections of two or more processes must never be interleaved!

- Further assumptions:
  - Pre-and post-protocols can be executed before and after each critical section.
  - Processes may delay indefinitely in non-critical sections.
  - Processes do not delay indefinitely in critical sections.

References for this chapter

[Ben-Ari06] M. Ben-Ari
Principles of Concurrent and Distributed Programming

Mutual Exclusion

Problem specification

- N processes execute (indefinitely) instruction sequences concurrently. Each instruction belongs to either a critical or non-critical section.

- Safety property "Mutual exclusion": Instructions from critical sections of two or more processes must never be interleaved!

- Further assumptions:
  - Pre-and post-protocols can be executed before and after each critical section.
  - Processes may delay indefinitely in non-critical sections.
  - Processes do not delay indefinitely in critical sections.

References for this chapter

[Ben-Ari06] M. Ben-Ari
Principles of Concurrent and Distributed Programming

Mutual Exclusion

Problem specification

- N processes execute (indefinitely) instruction sequences concurrently. Each instruction belongs to either a critical or non-critical section.

- Safety property "Mutual exclusion": Instructions from critical sections of two or more processes must never be interleaved!

- Further assumptions:
  - Pre-and post-protocols can be executed before and after each critical section.
  - Processes may delay indefinitely in non-critical sections.
  - Processes do not delay indefinitely in critical sections.

References for this chapter

[Ben-Ari06] M. Ben-Ari
Principles of Concurrent and Distributed Programming

Mutual Exclusion

Problem specification

- N processes execute (indefinitely) instruction sequences concurrently. Each instruction belongs to either a critical or non-critical section.

- Safety property "Mutual exclusion": Instructions from critical sections of two or more processes must never be interleaved!

- Further assumptions:
  - Pre-and post-protocols can be executed before and after each critical section.
  - Processes may delay indefinitely in non-critical sections.
  - Processes do not delay indefinitely in critical sections.
Mutual Exclusion

Mutual exclusion: Second attempt

```plaintext
begin
   ----- non_critical_section_1:
   C1 := In_CS;
   exit when C2 = Out_CS;
   end loop;
   C2 := Out_CS;
end
```

```
begin
   ----- non_critical_section_2:
   C1 := In_CS;
end
```

```
begin
   ----- non_critical_section_1:
   C1 := In_CS;
   exit when C2 = Out_CS;
   end loop;
   C2 := Out_CS;
end
```

```
begin
   ----- non_critical_section_2:
   C2 := Out_CS;
end
```

```plaintext
C1, C2: Critical_Section_State := Out_CS;
```

```
C1, C2: Critical_Section_State := Out_CS;
```

---

Mutual Exclusion: Third attempt

```plaintext
begin
   ------ non_critical_section_1;
   C1 := In_CS;
   exit when C2 = Out_CS;
   end loop;
   C2 := Out_CS;
end
```

```
begin
   ------ non_critical_section_2:
   C2 := In_CS;
end
```

```
begin
   ------ non_critical_section_1:
   C1 := In_CS;
   exit when C2 = Out_CS;
   end loop;
   C2 := Out_CS;
end
```

```
begin
   ------ non_critical_section_2:
   C2 := Out_CS;
end
```

```plaintext
C1, C2: Critical_Section_State := Out_CS;
```

```
C1, C2: Critical_Section_State := Out_CS;
```

---

Mutual Exclusion: Forth attempt

```plaintext
begin
   ------ non_critical_section_1;
   C1 := In_CS;
   exit when C2 = Out_CS;
   end loop;
   C2 := Out_CS;
end
```

```
begin
   ------ non_critical_section_2:
   C2 := In_CS;
end
```

```
begin
   ------ non_critical_section_1:
   C1 := In_CS;
   exit when C2 = Out_CS;
   end loop;
   C2 := Out_CS;
end
```

```
begin
   ------ non_critical_section_2:
   C2 := Out_CS;
end
```

```plaintext
C1, C2: Critical_Section_State := Out_CS;
```

```
C1, C2: Critical_Section_State := Out_CS;
```

---

Mutual exclusion: Decker's Algorithm

```plaintext
begin
   -- Mutual exclusion?
end
```

```
begin
   -- Potential deadlock?
end
```

---

Mutual Exclusion: Peterson's Algorithm

```plaintext
begin
   -- Two tasks only?
end
```

```
begin
   -- No starvation?
end
```

```
begin
   -- No deadlock?
end
```

```
begin
   -- No livelock?
end
```
Mutual Exclusion

Problem specification

Mutual exclusion: Peterson's Algorithm

- N processes execute (infinite) instruction sequences concurrently. Each instruction belongs to either a critical or a non-critical section.
- Safety property: Mutual exclusion:
  - Instructions from critical sections of two or more processes must never be interleaved!

Issues:

- More required properties:
- No livelock!
- No deadlock!
- No starvation!
- No mutual exclusion!
- No starvation of the critical sections

Beyond atomic memory access

Realistic hardware support

Atomic test-and-set operations:
- \( R = C \) or \( R = C' \)

Memory cell reservations:
- \( i \in [0, N-1] \), \( C \rightarrow \) read by a special instruction, which puts a "reservation" on \( C \)
- Calculates new values for \( C \) and \( C' \)
- \( C \) and \( C' \) not manipulated by other processes or devices once the reservation is active

Mutual exclusion: Bakery Algorithm

The idea of the Bakery Algorithm

A set of \( N \) Processes \( P_1, P_2, \ldots, P_N \) competing for mutually exclusive execution of their critical regions. Every process \( P_i \) with \( i < N \) supplies a globally readable number \( \lambda_i \) (ticket) initialized to \( 0 \).

- Before a process \( P_i \) enters a critical section:
  - \( P_i \) draws a number \( \lambda_i \), \( \lambda_i \neq i \)
  - If \( i \) is allowed to enter the critical section iff \( \lambda_i \neq i \) and \( \lambda_i = 0 \)
- After a process \( P_i \) leaves a critical section:
  - \( P_i \) sets its \( \lambda_i \) to \( 0 \)

Can you ensure that processes won’t read each others ticket numbers while still calculating?
Can you ensure that no two processes draw the same number?
Mutual Exclusion

Mutual exclusion: atomic exchange operation

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag := 1;
begin loop
loop
[Temp := L; L := C; C := Temp];
exit when L = 0; change process
end loop;
critical_section_i;
L := 1; C := 0;
end loop;
end Pi;

wx Mutual exclusion! No deadlock! No global livelock!
wx Works for any dynamic number of processes.
wx Individual starvation possible! Busy waiting loops!

Mutual exclusion: memory cell reservation

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
L : Flag := 1;
begin loop
loop
[Temp := L; L := C; C := Temp];
exit when L = 0; change process
end loop;
critical_section_i;
L := 1; C := 0;
end loop;
end Pi;

wx Mutual exclusion! No deadlock! No global livelock!
wx Works for any dynamic number of processes.
wx Individual starvation possible! Busy waiting loops!

Mutual Exclusion

Mutual exclusion: ... or the lack thereof

Count : Integer := 0;

task body Enter is
begin
for i := 1 .. 100 loop
Count := Count + 1;
end loop;
end Enter;

wx What is the value of Count after both programs complete?

Count : word 0 hexadecimal

Word lock, word 0 hexadecimal: All means unlocked

Word lock, word 0 hexadecimal: All means unlocked

Word lock, word 0 hexadecimal: All means unlocked

Any context switch needs to clear reservations
a set of processes P₁, P₂ agree on a variable by operating a flag to indicate synchronization conditions.

An atomic operation on S: (aka 'Suspend_Until_True', 'sem_wait', …) Multiple V (Signal) calls have the same effect as a single call.

Binary semaphores are sufficient to create all other semaphore forms.

General semaphores (counting semaphores): non-negative number (range limited by the system P and increment and decrement the semaphore by one.

Critical sections The increment (and decrement) value for the semaphore is specified as a parameter with P and V.

All types of semaphores must be initialized: often the number of processes which are allowed inside a critical section, i.e.: T

A set of processes P₁, P₂ agree on a variable by operating a flag to indicate synchronization conditions.

An atomic operation on S: (aka 'Suspend_Until_True', 'sem_wait', …) Multiple V (Signal) calls have the same effect as a single call.

Binary semaphores are sufficient to create all other semaphore forms.

General semaphores (counting semaphores): non-negative number (range limited by the system P and increment and decrement the semaphore by one.

Critical sections The increment (and decrement) value for the semaphore is specified as a parameter with P and V.

All types of semaphores must be initialized: often the number of processes which are allowed inside a critical section, i.e.: T

A set of processes P₁, P₂ agree on a variable by operating a flag to indicate synchronization conditions.

An atomic operation on S: (aka 'Suspend_Until_True', 'sem_wait', …) Multiple V (Signal) calls have the same effect as a single call.

Binary semaphores are sufficient to create all other semaphore forms.

General semaphores (counting semaphores): non-negative number (range limited by the system P and increment and decrement the semaphore by one.

Critical sections The increment (and decrement) value for the semaphore is specified as a parameter with P and V.

All types of semaphores must be initialized: often the number of processes which are allowed inside a critical section, i.e.: T

A set of processes P₁, P₂ agree on a variable by operating a flag to indicate synchronization conditions.

An atomic operation on S: (aka 'Suspend_Until_True', 'sem_wait', …) Multiple V (Signal) calls have the same effect as a single call.

Binary semaphores are sufficient to create all other semaphore forms.

General semaphores (counting semaphores): non-negative number (range limited by the system P and increment and decrement the semaphore by one.

Critical sections The increment (and decrement) value for the semaphore is specified as a parameter with P and V.

All types of semaphores must be initialized: often the number of processes which are allowed inside a critical section, i.e.: T

A set of processes P₁, P₂ agree on a variable by operating a flag to indicate synchronization conditions.

An atomic operation on S: (aka 'Suspend_Until_True', 'sem_wait', …) Multiple V (Signal) calls have the same effect as a single call.

Binary semaphores are sufficient to create all other semaphore forms.

General semaphores (counting semaphores): non-negative number (range limited by the system P and increment and decrement the semaphore by one.

Critical sections The increment (and decrement) value for the semaphore is specified as a parameter with P and V.

All types of semaphores must be initialized: often the number of processes which are allowed inside a critical section, i.e.: T
Semaphores

\[
\begin{align*}
\text{S1, S2 : Semaphore } &= 1; \\
\text{task body } P_i \text{ is} & \begin{align*}
\text{begin} & \\
\text{loop} & \begin{align*}
\text{--- non_critical_section_i;} & \\
\text{wait (S1);} & \\
\text{wait (S2);} & \\
\text{signal (S1);} & \\
\text{signal (S2);} & \\
\text{end loop;} & \\
\text{end } P_i;
\end{align*}
\end{align*}
\end{align*}
\]

\(\text{Works too!}\)

Mutual Exclusion

\[
\begin{align*}
\text{S1, S2 : Semaphore } &= 1; \\
\text{task body } P_j \text{ is} & \begin{align*}
\text{begin} & \\
\text{loop} & \begin{align*}
\text{--- non_critical_section_j;} & \\
\text{wait (S2);} & \\
\text{wait (S1);} & \\
\text{signal (S2);} & \\
\text{signal (S1);} & \\
\text{end loop;} & \\
\text{end } P_i;
\end{align*}
\end{align*}
\]

\(\text{Works too!}\)