Mutual Exclusion

The general mutual exclusion scenario

- N processes execute different 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!

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

Atomic load & store operations

Assumption 1: every individual base memory cell load and store access is atomic.
Assumption 2: there is no atomic combined load-store access.

G := Natural := 0;
G := G := G + G;

Turn := Task_Token := 0;

P0; ------ non_critical_section_0;

P1; ------ critical_section_0;

P1; ------ non_critical_section_1;

P2; ------ non_critical_section_1;

end P1;
end P2;
end P3;

end turn := turn + 1;
end loop;

end when

after the first global initialization, G can have almost any value between 0 and 24.
after the first global initialization, G will have exactly one value between 0 and 24.
after all tasks terminated, G must have exactly one value between 0 and 24.

© 2020 Uwe R. Zimmer, The Australian National University page 209 of 758 (chapter 2: "Mutual Exclusion" up to page 253)
**Mutual Exclusion**

**Problem specification**

- Each process executes an infinite instruction sequence concurrently. Each instruction belongs to either a critical or a non-critical section.

**Instructions from critical sections of two or more processes must never be interleaved.**

- More required properties:
  - No deadlock: None of the processes ever enters a critical section.
  - No starvation: Every process that tries to enter one of its critical sections must succeed eventually.

**Bakery Algorithm**

The Bakery algorithm assigns to each process a ticket number that it must present to enter critical sections. The algorithm is designed to ensure mutual exclusion without causing deadlock or starvation.

- **Static ticket number**:
  - Each process is assigned a ticket number when it starts.
  - This number remains constant throughout the execution.

- **Dynamic ticket number**:
  - Each process is assigned a ticket number when it starts.
  - This number can change during execution.

**Memory cell reservations**:

- Each process holds a memory cell that it reserves for itself.
- When a process leaves a critical section, it resets its ticket number.
Mutual Exclusion

Mutual exclusion: atomic exchange operation

\[\text{type Flag} = \text{Natural range } 0..1; \]
\[C : \text{Flag} := 0;\]

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{end}; \]

\[\text{end}; \]

---

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

---

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

---

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

---

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

---

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

---

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

---

\[\text{task body } P_i \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_i;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]

\[\text{task body } P_j \text{ is}\]
\[L : \text{Flag}; \]
\[\text{begin}\]
\[\text{loop}\]
\[\text{Temp} := L; \]
\[L := \text{Temp}; \]
\[\text{wait when (change process and not L = Temp)}\]
\[\text{end loop}\]
\[\text{end critical section}_j;\]
\[\text{end loop}; \]
\[\text{end}; \]
\[\text{end}; \]
Mutual Exclusion

Beyond atomic hardware operations

Semaphores

- A set of processes agree on a variable $S$ operating as a flag to indicate synchronization conditions.
- An atomic operation Wait on $S$ (aka "Suspend Until True", "sem.wait",...) Process $P_j$: Wait($S$) if $S > 0$ then $S := S - 1$ else suspend $P_j$.
- An atomic operation Signal on $S$ (aka "Set True", "sem.post",...) Process $P_j$: Signal($S$) if $S = 0$ then $S := S + 1$.

A semaphore is a variable $S$ that needs to clear reservations as soon as $S > 0$ then $S := S - 1$

Von S — for ‘vrygeven’ (Dutch for ‘to release’):

$S$ is called a Semaphore when $S = 0$.

Critical section

Light weight solution – sometimes referred to as “lock-free” or “lockless”.

... as supplied by operating systems and runtime environments

Types of semaphores:

- Binary semaphores: restricted to $0$, $1$ or False, True resp.
- Multiple $V$ and $P$ calls have the same effect than a single call
- Atomic hardware operations support binary semaphores.
- Binary semaphores are sufficient to create all other semaphore forms.
- General semaphores (counting semaphores): non-negative number, range limited by the system and decrement the semaphore by one.
- Quantity semaphores: the increment (and decrement) value for the semaphore is specified as a parameter with $P$ and $V$.

Light weight solution – sometimes referred to as “lock-free” or “lockless”.

Note: $S = 0$
Mutual Exclusion

Semaphores

\[ S : \text{Semaphore} := 1; \]
\[ \text{task body } P_j \text{ is} \]
begin
loop
\[ \text{non_critical_section}_j; \]
wait (S);
end
end loop;
end P_j;
\]
\[ \text{or Works?} \]

\[ \text{S} : \quad \text{Semaphore} := 1; \]
\[ \text{task body } P_j \text{ is} \]
begin
loop
\[ \text{non_critical_section}_j; \]
wait (S);
end loop;
end P_j;
\]
\[ \text{or Works too?} \]

Summary

- Definition of mutual exclusion
- Atomic load and atomic store operations
  - Some classical errors
  - Atomic test-and-set, Atomic exchanges, Memory cell reservations
- Semaphores
  - Basic semaphore definition
  - Operating systems style semaphores
- Realistic hardware support
  - Atomic load and atomic store operations
  - Some classical errors
  - Atomic test-and-set, Atomic exchanges, Memory cell reservations
- Semaphores
  - Basic semaphore definition
  - Operating systems style semaphores