Process and Thread Synchronization

Introduction and Locks

Adrian Coleșa

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

April 15, 2020

The purpose of this chapter

- Define and illustrate the context of synchronization
- Present different synchronization mechanisms: locks

The purpose of this chapter

- Define and illustrate the context of synchronization
- Present different synchronization mechanisms: locks

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, p. 100 – 132
- A. Downey, *The Little Book of Semaphores*, 2nd Edition, 2008, p. 1 – 106

Outline

Outline

Outline

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance ⇒ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance ⇒ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance ⇒ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance ⇒ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance ⇒ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance $\Rightarrow$ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance $\Rightarrow$ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance $\Rightarrow$ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance $\Rightarrow$ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance $\Rightarrow$ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Context

- more processes or threads **competing for the same resources**
  - e.g. files, memory etc.
- **concurrent (parallel) executions**
  - their scheduling to processors not controlled at the user-space level
  - their steps (instructions) could be interleaved in an unpredictable way
- **concurrence (independent parallelism)** vs **competition (dependent parallelism)**
  - parallelism: good for performance ⇒ desired
  - competition: could hurt (step on one another's toe)
- **determinism** vs **non-determinism**
  - high-level (logical) determinism of a program
  - non-deterministic occurrences of low-level events (e.g. interrupts)

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- $\Rightarrow$ **race conditions**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- $\Rightarrow$ **race conditions**

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- $\Rightarrow$ **race conditions**

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- $\Rightarrow$ **race conditions**

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- ⇒ **race conditions**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- $\Rightarrow$ **race conditions**

Problem

- uncontrolled
- concurrent executions
- competing
- on shared resources
- could lead to **unpredictable, inconsistent results**
  - i.e. state of the shared resources
- $\Rightarrow$ **race conditions**

Real-Life Example: "The Milk Buying" Problem (preview)

Time	Person A	Person B
3.00	Look in fridge. No milk.	-
3.05	Go to the store.	-
3.10	Arrive at store.	Look in fridge. No milk.
3.15	Buy milk.	Go to the store.
3.20	Arrive at home with milk.	Arrive at store.
3.25	-	Buy milk.
3.30	-	Arrive at home with milk.

- no cooperation ⇒ race conditions
- ⇒ **too much milk**
- **buying no milk** could also happen
  - though, not on the given algorithm
  - but in real life with real persons :)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

```
// Thread 1                // Thread 2
1. load Reg1, count        1.
2.                         2. load Reg2, count
3.                         3. add Reg2, 1
4.                         4. store count, Reg2
5. add Reg1, 1             5.
6. store count, Reg1       6.
```

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

```
// Thread 1                // Thread 2
1. load Reg1, count        1.
2.                         2. load Reg2, count
3.                         3. add Reg2, 1
4.                         4. store count, Reg2
5. add Reg1, 1             5.
6. store count, Reg1       6.
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

```
// Thread 1                  // Thread 2
1. load Reg1, count          1.
2.                           2. load Reg2, count
3.                           3. add Reg2, 1
4.                           4. store count, Reg2
5. add Reg1, 1               5.
6. store count, Reg1         6.
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

// Thread 1	// Thread 2
1. load Reg1, count	1.
2.	2. load Reg2, count
3.	3. add Reg2, 1
4.	4. store count, Reg2
5. add Reg1, 1	5.
6. store count, Reg1	6.

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

```
// Thread 1                 // Thread 2
1. load Reg1, count         1.
2.                          2. load Reg2, count
3.                          3. add Reg2, 1
4.                          4. store count, Reg2
5. add Reg1, 1              5.
6. store count, Reg1        6.
```

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

// Thread 1	// Thread 2
1. load Reg1, count	1.
2.	2. load Reg2, count
3.	3. add Reg2, 1
4.	4. store count, Reg2
5. add Reg1, 1	5.
6. store count, Reg1	6.

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

```
// Thread 1                  // Thread 2
1. load Reg1, count          1.
2.                           2. load Reg2, count
3.                           3. add Reg2, 1
4.                           4. store count, Reg2
5. add Reg1, 1               5.
6. store count, Reg1         6.
```

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

// Thread 1	// Thread 2
1. load Reg1, count	1.
2.	2. load Reg2, count
3.	3. add Reg2, 1
4.	4. store count, Reg2
5. add Reg1, 1	5.
6. store count, Reg1	6.

Technical Example: Global Counter

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
```

```
// C instruction
count = count + 1;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0)
    count = count + 1;
```

```
// Machine instructions
load Reg, count
add Reg, 1
store count, Reg
```

```
// Thread 1              // Thread 2
1. load Reg1, count      1.
2.                       2. load Reg2, count
3.                       3. add Reg2, 1
4.                       4. store count, Reg2
5. add Reg1, 1           5.
6. store count, Reg1     6.
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
    - **avoiding race conditions**
    - **by cooperation**
- **synchronization** of concurrent threads / processes
    - **wait for the other threads** to reach some point in their execution
    - **establish rules** for accesses to shared resources
        - e.g. mutual exclusion: only one at a moment
- **parallelism**
    - synchronization imposed only on **critical regions**
        - i.e. code regions where race could appear
    - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. mutual exclusion: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
    - **avoiding race conditions**
    - **by cooperation**
- **synchronization** of concurrent threads / processes
    - **wait for the other threads** to reach some point in their execution
    - **establish rules** for accesses to shared resources
        - e.g. mutual exclusion: only one at one moment
- **parallelism**
    - synchronization imposed only on **critical regions**
        - i.e. code regions where race conditions appear
    - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. mutual exclusion: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions
  - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions occur
  - let the other code run in parallel (if it could)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
    - **avoiding race conditions**
    - **by cooperation**
- **synchronization** of concurrent threads / processes
    - **wait for the other threads** to reach some point in their execution
    - **establish rules** for accesses to shared resources
        - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
    - synchronization imposed only on **critical regions**
    - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could)

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Need For ...

- getting the expected results
- making concurrent executions do not step on each other's toes
  - **avoiding race conditions**
  - **by cooperation**
- **synchronization** of concurrent threads / processes
  - **wait for the other threads** to reach some point in their execution
  - **establish rules** for accesses to shared resources
    - e.g. **mutual exclusion**: only one at one moment
- **parallelism**
  - synchronization imposed only on **critical regions**
    - i.e. code regions where race conditions appear
  - let the other code run in parallel (if it could)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

General Synchronization Rules

1. no more than the safe number of threads may be simultaneous inside their critical regions
   - e.g. just one thread for mutual exclusion
2. no thread running outside its critical regions may block other threads
   - e.g. do not forget to let the "key" for the others
3. no thread should have to wait forever to enter its critical region
   - e.g. give chance to everybody
4. no assumptions may be made about speed and number of CPUs and about the system load or scheduling policy
   - e.g. do not use `sleep`-like functions

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

"The Milk Buying" Problem

Time	Person A	Person B
3.00	Look in fridge. No milk.	-
3.05	Go to the store.	-
3.10	Arrive at store.	Look in fridge. No milk.
3.15	Buy milk.	Go to the store.
3.20	Arrive at home with milk.	Arrive at store.
3.25	-	Buy milk.
3.30	-	Arrive at home with milk.

- no cooperation ⇒ **too much milk**
- correctness requirements
  1. never more than one person buys milk
  2. someone buys, if needed

Solution 1: "Lock Variable"

- **idea: let a note *when* leaving buying milk**
  - acts as a lock variable
  - "no note" (lock = 0) ⇒ OK to act (go for it)
  - "there is a note" (lock = 1) ⇒ wait (let the other does his job)
- **generality** (symmetry): every person (thread) executes the same function

```
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

Solution 1: "Lock Variable"

- **idea: let a note *when* leaving buying milk**
  - acts as a lock variable
  - "no note" (lock = 0) ⇒ OK to act (go for it)
  - "there is a note" (lock = 1) ⇒ wait (let the other does his job)
- **generality** (symmetry): every person (thread) executes the same function

```
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 1: "Lock Variable"

- **idea: let a note *when* leaving buying milk**
  - acts as a lock variable
  - "no note" (lock = 0) ⇒ OK to act (go for it)
  - "there is a note" (lock = 1) ⇒ wait (let the other does his job)
- **generality** (symmetry): every person (thread) executes the same function

```
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 1: "Lock Variable"

- **idea: let a note *when* leaving buying milk**
  - acts as a lock variable
  - "no note" (lock $= 0$) $\Rightarrow$ OK to act (go for it)
  - "there is a note" (lock $= 1$) $\Rightarrow$ wait (let the other does his job)
- **generality** (symmetry): every person (thread) executes the same function

```
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

Solution 1: "Lock Variable"

- **idea: let a note *when* leaving buying milk**
  - acts as a lock variable
  - "no note" (lock = 0) ⇒ OK to act (go for it)
  - "there is a note" (lock = 1) ⇒ wait (let the other does his job)
- **generality** (symmetry): every person (thread) executes the same function

```
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
    - still has race conditions
    - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note;
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 1: "Lock Variable" (cont.)

- **problem: the solution fails occasionally**
  - still has race conditions
  - both threads can be simultaneously in their own critical region

```
// Thread 1
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

```
// Thread 2
person()
{
    if (no milk)
        if (no note) {
            leaves a note;
            goes and buys milk;
            when back, removes the note
        }
}
```

Solution 2: "Pre-Locking"

- **idea: announce the intention (let the note) *before* checking the context**
- **particularity** (asymmetry): each person executes its own function
  - though could be generalized

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking"

- **idea: announce the intention (let the note) *before* checking the context**
- **particularity** (asymmetry): each person executes its own function
  - though could be generalized

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking"

- **idea: announce the intention (let the note) *before* checking the context**
- **particularity** (asymmetry): each person executes its own function
  - though could be generalized

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking"

- **idea: announce the intention (let the note) *before* checking the context**
- **particularity** (asymmetry): each person executes its own function
  - though could be generalized

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 2: "Pre-Locking" (cont.)

- **solution works**
  - mutual exclusion provided
  - i.e. only one thread can be in its critical region
- **problem: possible starvation**
  - it is possible that no thread enters in its critical region
  - a thread could wait indefinitely to enter its critical region
  - though no other thread is inside its critical region

```
person_A()
{
    leave note_A;

    if (no note_B)
        if (no milk)
            goes and buys milk;

    removes note_A;
}
```

```
person_B()
{
    leave note_B;

    if (no note_A)
        if (no milk)
            goes and buys milk;

    removes note_B;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, ... , Person N, Person 1, Person 2, ... , Person N, ...
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, ... , Person N, Person 1, Person 2, ... , Person N, ...
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, ... , Person N, Person 1, Person 2, ... , Person N, ...
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, . . . , Person N, Person 1, Person 2, . . . , Person N, . . .
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation"

- **idea**: impose a **fixed order** on persons' accesses to the shared resource
  - e.g. Person 1, Person 2, ... , Person N, Person 1, Person 2, ... , Person N, ...
- **symmetry**: each person execute the same function, though with a different argument

```
int turn = 0, noPers = N;

person(int id) // IDs go from 0 to N-1
{
    // wait until my turn
    while (turn != id) {
      // BUSY WAITING, i.e. does nothing useful, but
      // just loops, waiting for the condition to be fulfilled
    }

    // it is my turn
    if (no milk)
        buy milk;

    // transfer the turn to next
    turn = (turn + 1) % noPers;
}
```

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis, if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Solution 3: "Strict Alternation" (cont.)

- **solution works**
  - mutual exclusion provided (`turn` cannot have but only one value at one moment)
  - no starvation: each thread gets its chance (turn)
- **limitations**: not appropriate for practical situations
  - impose a strict order on the execution
  - the execution speed of all threads is slowed down to the speed of the slowest one
    - in extremis: if one thread dies, all the others are blocked forever
  - generally, **busy waiting is not efficient**

Lessons Learned From The “Milk Buying” Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - ⇒ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - ⇒ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - ⇒ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - ⇒ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - ⇒ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - ⇒ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The “Milk Buying” Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - $\Rightarrow$ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - $\Rightarrow$ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - ⇒ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - ⇒ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Lessons Learned From The "Milk Buying" Problem

- **synchronization is a (really) complex** problem
  - race conditions or starvation
    - not always obvious
    - do not occur all the time
  - ⇒ difficult to debug
    - bug effects visible occasionally under particular scheduling conditions
    - bugs are not necessarily detectable during debug process
  - we need
    - general (scalable) solutions: symmetry regarding concurrent thread' functions
    - efficient solutions: no busy waiting, allow flexibility, parallelism
- **synchronization is difficult** (even impossible) to be solved using only user space mechanisms
  - no control on the nondeterministic hardware events
  - ⇒ **application-level synchronization needs OS support**
    - while the OS controls and manages the hardware

Practice (1)

- Identify on the code below the critical regions.

```
int n = 0;

th1()                          th2()
{                              {
    int c = 0;                     int c = 0;

    n++;                           c++;
    c++;                           n++;

    if (c)                         if (c)
      n--;                           c--;

    c = n;                         c = n;
    c++;                           c++;
    n = c;                         n = c;
}                              }
```

Outline

Outline

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
  - any time a **thread exits its critical region, it must inform the mediator**

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

Synchronization Pattern

- **synchronization imposes rules** on accessing shared resources
  - one thread allowed entering its critical region only if not conflicting other thread inside their own critical regions
  - e.g. mutual exclusion: if a thread already inside its critical region, no other thread allowed in its own one
- synchronization rules imposed by a **synchronization mediator (mechanism)**
  - must be called in relation to any event regarding critical sections
  - any time a **thread wants entering its critical region, it must ask the mediator for permission**
    - if permission not allowed, the thread is blocked
    - until safe conditions are fulfilled
  - any time a **thread exits its critical region, it must inform the mediator**
    - must know about resource availability
    - to let (some of) other blocked (waiting) threads enter their critical region

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a thread could be blocked for a while
  - **announce exit** from critical region
    - a thread is never blocked when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a thread could be blocked for a while
  - **announce exit** from critical region
    - a thread is never blocked when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a thread could be blocked for a while
  - **announce exit** from critical region
    - a thread is never blocked when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - **a thread could be blocked** for a while
  - **announce exit** from critical region
    - **a thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a **thread could be blocked** for a while
  - **announce exit** from critical region
    - a **thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a **thread could be blocked** for a while
  - **announce exit** from critical region
    - a **thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
    - entrances
    - exits
- **provided services**
    - **ask for permission to enter** critical region
        - a **thread could be blocked** for a while
    - **announce exit** from critical region
        - a **thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
    - system calls provided for both types of services

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
    - entrances
    - exits
- **provided services**
    - **ask for permission to enter** critical region
        - a **thread could be blocked** for a while
    - **announce exit** from critical region
        - a **thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
    - system calls provided for both types of services

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a **thread could be blocked** for a while
  - **announce exit** from critical region
    - a **thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Mechanism's Functionality

- acts as a **checkpoint** for both critical region's
  - entrances
  - exits
- **provided services**
  - **ask for permission to enter** critical region
    - a **thread could be blocked** for a while
  - **announce exit** from critical region
    - a **thread is never blocked** when exiting
- provided by OS, i.e. implemented in OS
  - system calls provided for both types of services

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
    - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
    - executed by a thread without interleaving other thread's execution
    - logically, executed like a single, indivisible instruction
    - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
    - **needs hardware support to get atomicity** on normally non-atomic operations

Synchronization Mechanism's Problem

- becomes itself a concurrently accessed resource
  - **race conditions could occur inside the mediator itself**
- **synchronization mechanism's functions need atomicity**
  - executed by a thread without interleaving other thread's execution
  - logically, executed like a single, indivisible instruction
  - named **primitive functions (primitives)**
- **OS** must control non-deterministic hardware events
  - **needs hardware support to get atomicity** on normally non-atomic operations

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Disable/Enable Interrupts

- the mechanism
  - **disable interrupts** when entering the synchronization mechanism's functions
  - **(re)enable interrupts** before returning from the synchronization mechanism's functions
- result: **functions' atomicity**
  - code between interrupt disable and interrupt disable is executed with no interruption
  - i.e. atomically
- **limitations**
  - it works only on uni-processor systems
  - not accessible from user space

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

UNIVERSITATEA TEHNICĂ

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic “Read-Modify-Write”/“Test-And-Set-Lock” Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register’s value and overwrite it with another register’s value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the [illegible] memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the involved memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the involved memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the involved memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Atomic "Read-Modify-Write"/"Test-And-Set-Lock" Instructions

- instructions consisting in **more steps executed atomically by the hardware**
- examples
  - atomically interchange a memory location and a register
  - atomically increment a memory location
  - atomically read a memory location, compare it with some register's value and overwrite it with another register's value in case of equality
- synchronization mechanism functions built on such small-size atomic operations
- advantages
  - **work on multi-processor systems**
    - the hardware assures their atomicity by blocking access to the involved memory location for all other processors
  - accessible even from user space as so called **inter-locked instructions**

Intel Atomic Instructions

intel® INSTRUCTION SET REFERENCE

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Description
F0	LOCK	Asserts LOCK# signal for duration of the accompanying instruction

Description

This instruction causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that in later Intel Architecture processors (such as the Pentium® Pro processor), locking may occur without the LOCK# signal being asserted. Refer to Intel Architecture Compatibility below.

The LOCK prefix can be prepended only to the following instructions and to those forms of the instructions that use a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. An undefined opcode exception will be generated if the LOCK prefix is used with any other instruction. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Intel Architecture Compatibility

Beginning with the Pentium® Pro processor, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism insures that the operation is carried out atomically with regards to memory. Refer to Section 7.1.4., *Effects of a LOCK Operation on Internal Processor Caches* in Chapter 7, *Multiple-Processor Management* of the *Intel Architecture Software Developer's Manual, Volume 3*, the for more information on locking of caches.

Operation

AssertLOCK#(DurationOfAccompaningInstruction)

Flags Affected

None.

Taken from Intel Instructions Manual

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock (acquire)**
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock (release)**
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock (acquire)**
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock (release)**
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock** (**acquire**)
  - **unlock** (**release**)

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock (acquire)**
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock (release)**
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock** (**acquire**)
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock** (**release**)
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock (acquire)**
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock (release)**
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock (acquire)**
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock (release)**
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock** (**acquire**)
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock** (**release**)
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock** (**acquire**)
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock** (**release**)
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock** (**acquire**)
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock** (**release**)
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Characteristics

- **provides mutual exclusion** (called **mutex**)
- has two distinct states (like a flag)
  - **free** (unlocked, released): allow access to critical region
  - **busy** (locked, acquired): does not allow access to critical region
- provides two primitives
  - **lock** (**acquire**)
    - called by a thread just before entering in its critical region
    - only one thread can acquire a mutex even if more try that simultaneously
  - **unlock** (**release**)
    - called by a thread just after leaving its critical region
    - allowed only to the lock's holder

Possible Implementations (uni-processor systems)

```
mutex_lock()
{
    disable_interrupts();
    while (value != FREE) {
        insert(crt_thread, w_queue);
        block_current_thread();;
    }
    value = BUSY;
    enable_interrupts();
}
```

```
mutex_unlock()
{
    disable_interrupts();
    value = FREE;
    if (! empty(waiting_queue)) {
        th = remove_first(w_queue);
        insert(ready_queue, th);
    }
    enable_interrupts();
}
```

- **interrupts are disabled**, to provide **code atomicity**
  - after critical code is executed, interrupts are enabled back
- "**sleep / wake-up technique**" is used instead of the "*busy waiting*"
  - "**sending to sleep**" (**blocking**) a thread
    - append it to a waiting queue and
    - take the CPU from it
  - "**waking up**" (**unblocking**) a thread
    - remove it from waiting queue
    - append it to the ready queue
    - eventually being given the CPU again

Possible Implementations (multi-processor systems)

```
mutex_lock:
    mov eax, 1                  ; prepare the value for acquired lock
    xchg eax, [lock_value]      ; atomic instruction
    cmp eax, 0                  ; check if lock was free
    jz lock_taken               ; if it was 0, now it belongs to the calling thread
    jmp mutex_lock              ; otherwise, was taken by other thread and must wait
lock_taken:
    ret

mutex_unlock:
    mov [lock_value], 0
    ret
```

- the atomic "**xchg**" instruction is used to provide **lock's test and set atomicity**
  - disabling interrupts does not work on multi-processor systems
- **busy-waiting** is used in this example
  - it correspond to "spin locks", useful in multi-processor systems
  - sleep / wakeup technique is more difficult to implement and requires the use of a spin lock

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Usage Example

```
int v[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
count = 0;
Lock l;
```

```
// Thread 1
for (i=0; i<5; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

```
// Thread 2
for (i=5; i<10; i++)
  if (v[i] != 0) {
    l.acquire();
    count = count + 1;
    l.release();
}
```

Practice (2)

- Protect the critical regions below with locks.

```
int n = 0, m = 0;

th1()                     th2()
{                         {
    int c = 0;                int c = 0;

    n++;                      c++;
    c++;                      n++;

    if (c)                    if (c)
      m--;                      m--;

    c = n;                    c = n;
    c++;                      c++;
    n = c;                    n = c;
}                         }
```

Outline

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "acquire()" and "release()" primitives

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

What we talked about

- race conditions
  - could lead to unpredictable, bad results
- need for synchronization
  - yet do not affect too much the possible parallelism
  - as an OS support
- locks
  - provide mutual exclusion
  - "`acquire()`" and "`release()`" primitives

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

1. race conditions could lead to problems (bugs) difficult to find and investigate
2. synchronization is difficult
3. we need specialized OS provided synchronization mechanisms
4. locks could be too restrictive
5. synchronization could reduce the possible parallelism, though both are needed

Lessons Learned

1. race conditions could lead to problems (bugs) difficult to find and investigate
2. synchronization is difficult
3. we need specialized OS provided synchronization mechanisms
4. locks could be too restrictive
5. synchronization could reduce the possible parallelism, though both are needed

Lessons Learned

1. race conditions could lead to problems (bugs) difficult to find and investigate
2. synchronization is difficult
3. we need specialized OS provided synchronization mechanisms
4. locks could be too restrictive
5. synchronization could reduce the possible parallelism, though both are needed

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

1. race conditions could lead to problems (bugs) difficult to find and investigate
2. synchronization is difficult
3. we need specialized OS provided synchronization mechanisms
4. locks could be too restrictive
5. synchronization could reduce the possible parallelism, though both are needed

Lessons Learned

1. race conditions could lead to problems (bugs) difficult to find and investigate
2. synchronization is difficult
3. we need specialized OS provided synchronization mechanisms
4. locks could be too restrictive
5. synchronization could reduce the possible parallelism, though both are needed