# Mutual exclusion

Petr Hanáček

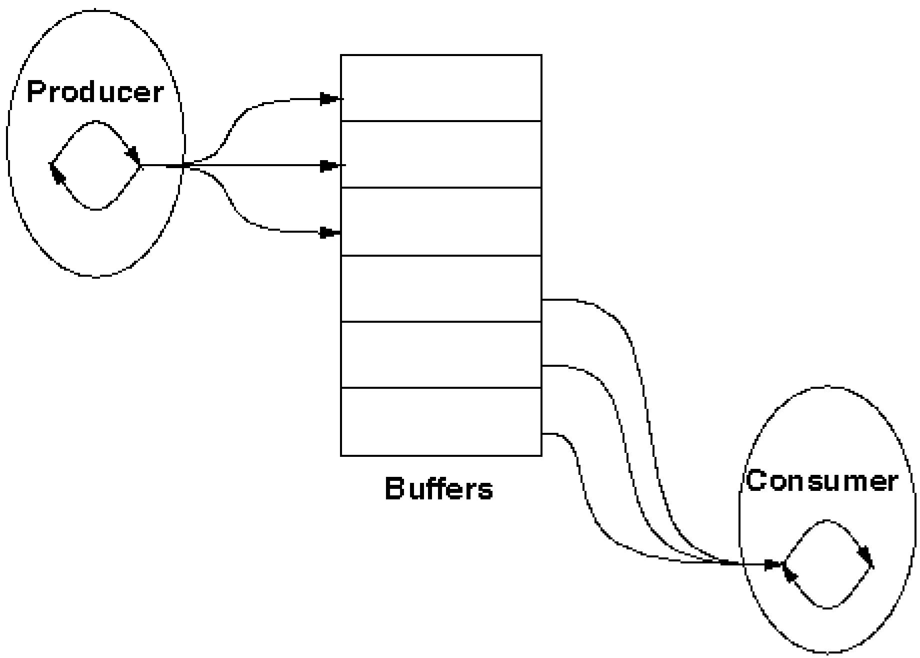
Corr pre 2007/8, pre 2009/10

# INTERAKCE MEZI PROCESY

- Competition (soupeření)
  - For use of resources
  - Because we are multiplexing resources
  - <u>Mutual exclusion problem</u>
- Cooperation (kooperace)
  - Each process solves part of problem

# Example - buffer

- Unbounded-Buffer: unlimited buffers, producer can always produce, consumer may have to wait
- Bounded-Buffer: limited buffers, both producer and consumer may have to wait



# Solution

```
Circular Array of Buffers
typedef ... item;
item buffer[n-1];
int in = 0;
int out = 0;
```

```
Producent
while( 1 ) {
      item nextp;

      produce( nextp );
      while( (in+1 % n) == out) ;
      buffer[in] = nextp;
      in = in+1 % n;
}
```

```
Konzument
while( 1 ) {
      item nextc;

      while(in == out) ;
      nextc = buffer[out];
      out = out+1 % n;
      consume( nextc );
}
```

- **Monoprocessor architecture:**
  - **Context switching**
- **Multiprocessor architecture:**
  - **No context switching**
  - **Instructions are performed in arbitrary order (interleaving)**

# No Waste Solution

Circular Array of Buffers

```
typedef ... item;
item buffer[n-1];
int in = 0;
int out = 0;
int counter = 0;
```

Producent

```
while( 1 ) {
      item nextp;
      produce( nextp );
      while( counter == n ) ;
      buffer[in] = nextp;
      in = in+1 % n;
      counter = counter + 1;
}
```

Konzument

```
while( 1 ) {
      item nextc;

      while( counter == 0 ) ;
      nextc = buffer[out];
      out = out+1 % n;
      counter = counter - 1;
      consume( nextc );
}
```

# One Big Problem

- The counter variable is shared and may be simultaneously updated by both the producer and consumer

```
counter = counter + 1;

        load   A,counter
        add    1,A
        store  A,counter

counter = counter - 1;

        load   A,counter
        sub    1,A
        store  A,counter
```

```
.Producer : "load A,counter"
.Consumer : "load A,counter"
.Producer : "add 1,A"
.Consumer : "sub 1,A"
.Producer : "store A,counter"
.Consumer : "store A,counter"
```

# Requirements for Mutual Exclusion

- Only one process at a time is allowed into its critical section
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation
- When no processes in critical section, any process that requests entry to its critical section must be permitted to enter without delay
- No assumptions are made about relative process speeds or number of processors
- A process remains inside its critical section for a finite time only

# Critical section approaches

- Software approaches:
  - Without support from programming language or OS
- Hardware support:
  - Special purpose machine instructions
- OS or programming language provides some level of support
  - (semaphore, monitor, …)

# A Generic 2 Process Software Solution

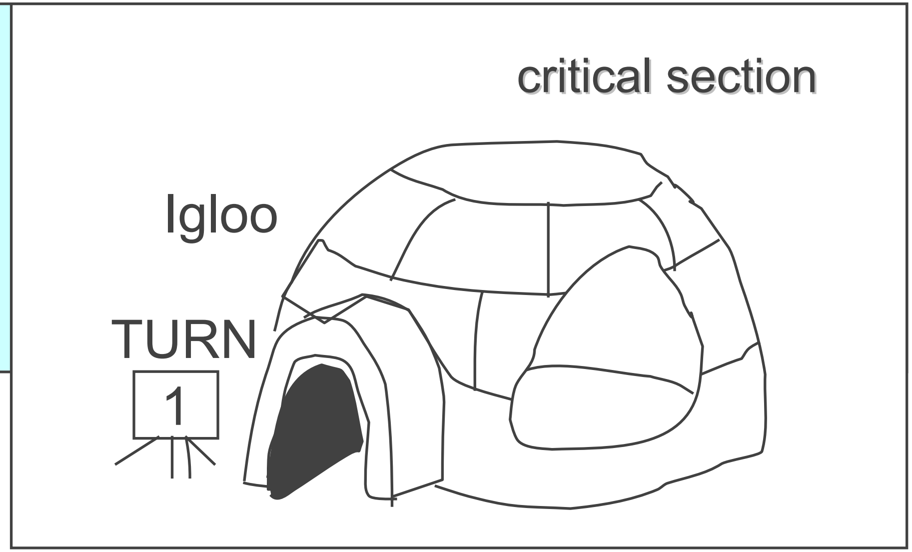
- Two Processes, *P1* and *P2* .
- Use $P_i$ to Mean Process of Interest.
- Use $P_j$ to Mean the Other Process.

```
repeat
 entry section;
 critical section
 exit section;
 remainder section
until false;
```

- Assumptions made about computer architecture:
  - A single read of memory is atomic. (!! pagging)
  - A single write of memory is atomic.
  - Simultaneous reads/writes will be interleaved in some order.

# Algorithm 1

```
shared var turn = 1;
repeat
  while turn<>i do skip;
  critical section
  turn := j;
  remainder section
until false;
```




Example:

Igloo has small entrance so only one process at a time may enter to check a value written on the blackboard. If the value on the blackboard is the same as the process, the process may proceed to the critical section.

If the value on the blackboard is not the value of the process, the process leaves the igloo to wait. From time to time, the process reenters the igloo to check the blackboard.

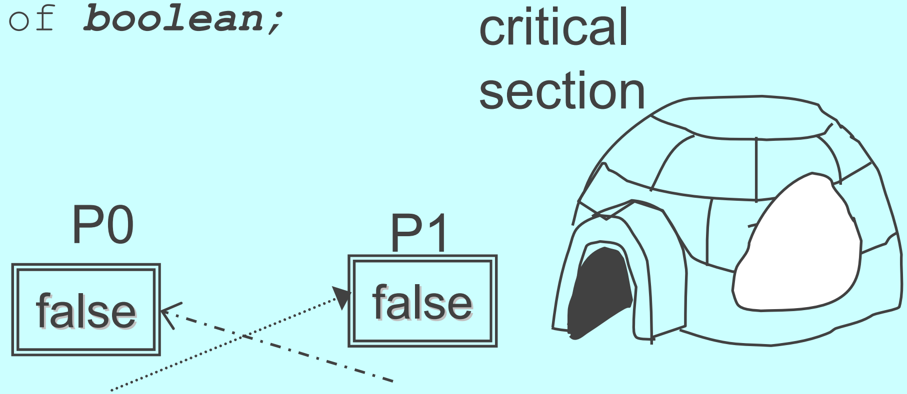
- Analysis of Algorithm 1
  - Only one process at a time in its critical section (GOOD).
  - Strict alternation in the execution of P0, P1 in the critical section (BAD).
  - Does not satisfy progress requirement

- Conclusion
  - If one process fails, the other process is permanently blocked
  - Each process should have its own key to the critical section so if one process is eliminated, the other can still access its critical section

# Algorithm 2

```
shared var flag: array [0..1] of boolean;

repeat
 while flag[j] do skip;
 flag[i] := true;
 critical section
 flag[i] := false;
 remainder section
until false;
```




- Each process can examine the other's status but cannot alter it
- When a process wants to enter the critical section is checks the other processes first
- If no other process is in the critical section, it sets its status for the critical section

- Analysis of Algorithm 2
  - Does not even ensure mutual exclusion.
    - » A) P0 enters the while statement {flag[1] = false}.
    - » B) P1 enters the while statement {flag[0] = false}.
    - » C) P1 sets flag[1] = true and enters the critical section.
    - » D) P0 sets flag[0] = true and enters the critical section.
  - The problem with the algorithm is that process Pi made a decision concerning the state of Pj before Pj changed the state of flag[j].

- This method does not guarantee mutual exclusion
- Each process can check the flags and then proceed to enter the critical section at the same time

# Algorithm 3

```
shared var flag: array [0..1] of boolean;

repeat
 flag[i] := true;
 while flag[j] do skip;
 critical section
 flag[i] := false;
 remainder section
until false;
```

- Set flag to enter critical section before check other processes
- If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section

- Analysis of Algorithm 3
  - Does guarantee mutual exclusion
  - Does not guarantee bounded wait
    - » A. P0 sets flag[0] = true
    - » B. P1 sets flag[1] = true
    - » C. Both process loop forever.
  - The Problem with this algorithm is due to the fact that process Pi sets its flag[i] = true without knowing the precise state of the other process.
- Deadlock is possible when two process set their flags to enter the critical section. Now each process must wait for the other process to release the critical section

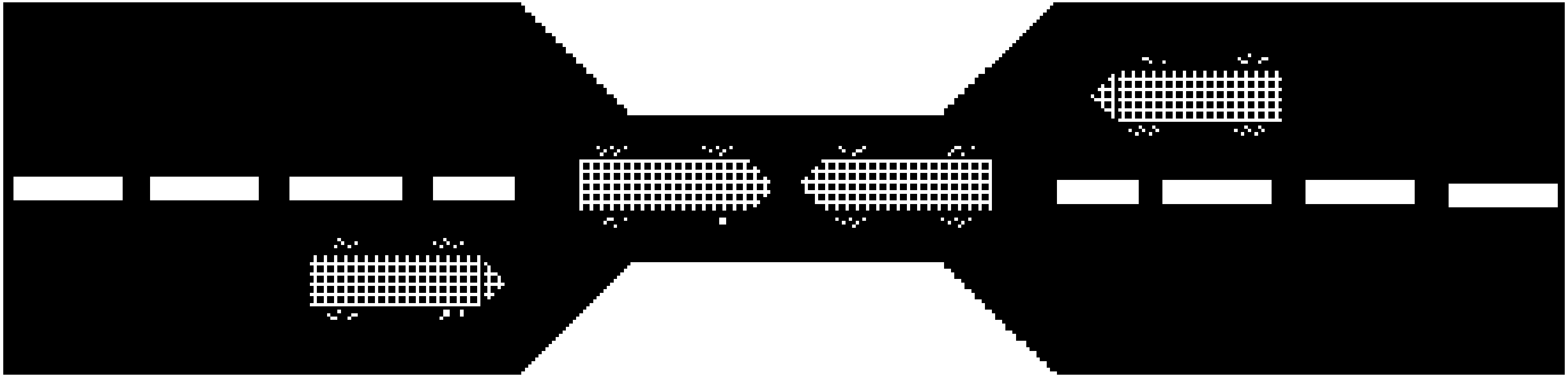
# Algorithm 4

```
shared var flag: array[0..1] of boolean;
repeat
      flag[i] = true;
      while flag[j] do
            flag[i] = false;
            wait;
            flag[i] = true;
      endwhile;
      critical section;
      flag[i] = false
      remainder section;
until false;
```

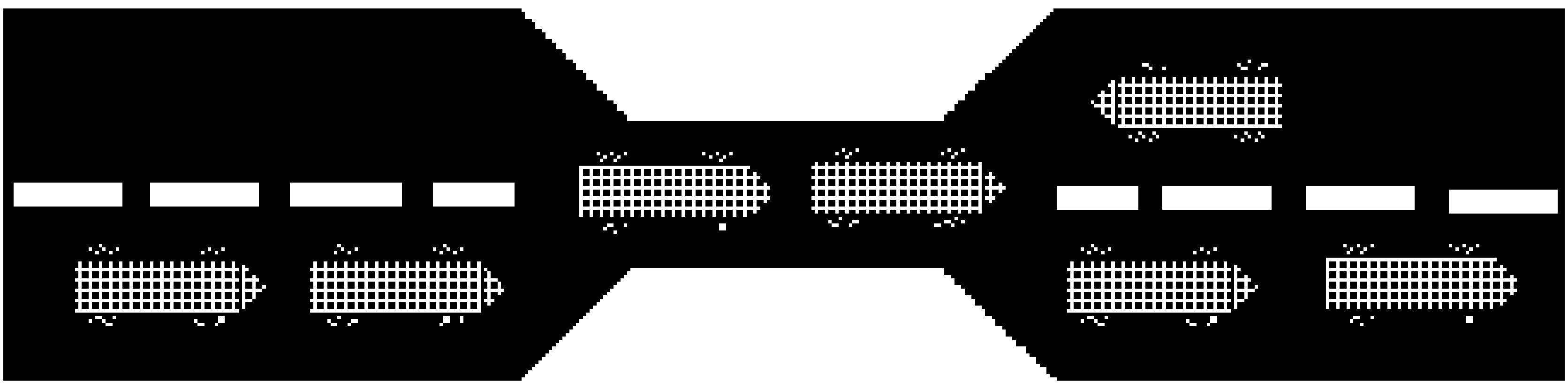
- A process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag
- Other processes are checked.  If they are in the critical region, the flag is reset and later set to indicate desire to enter the critical region.  This is repeated until the process can enter the critical region.

- Analysis 4
  - Does guarantee mutual exclusion
  - Does not guarantee bounded wait, e.g. if both processes execute line-by-line synchronously, no one can enter the critical section
- Indefinite postponement (starvation)
- It is possible for each process to set their flag, check other processes, and reset their flags. This scenario will not last very long so it is not deadlock. But it is undesirable.

# Deadlock and Starvation



Deadlock



Starvation

# Dekker's Algorithm (A Working Solution)

```
shared var flag: array [0..1] of boolean;
    turn: 0..1;
repeat
 flag[i] := true;
  while flag[j] do
    if turn=j then             // Only one process releases flag
       flag[i] := false;
       while turn=j do skip;   // wait
       flag[i] := true;
    endif
  endwhile
 critical section
 turn := j;
 flag[i] := false;
 remainder section
until false;
```

# Peterson's Algorithm (A Working Solution)

```
shared var flag: array [0..1] of boolean;
    turn: 0..1;
repeat
 flag[i] := true;
 turn := j;
  while (flag[j] and turn=j) do skip;
 critical section
 flag[i] := false;
 remainder section
until false;
```

- Each process gets a turn at the critical section
- If a process wants the critical section, it sets its flag and may have to wait for its turn

# AWAIT (operator)

- <await B → S>
  - <> - atomicity
  - B – Boolean expression
  - S – sequence of statements, that eventually terminates
- S is executed only when B = true
  - No intermediate status of S is visible for other processes
- Example:
  - <await s>0 → s = s - 1>
  - Waits until s is positive and then decremets s
    - *Mutual exclusion* <S>
    - *Conditional synchronization* <await B>

- Properties:
  - Very powerful mechanism
    $\Rightarrow$ coarse-grained solution
- The await statement is generally too expensive to implement
  - There are special-case uses of the await statement that are efficient to implement

```
shared var flag: array [0..1] of boolean;
repeat
  <await not flag[j] → flag[i] = true>
    critical section
  flag[i] = false;
until false;
```

# Hardware Support for Mutual Exclusion - Interrupt Disabling

- A process runs until it invokes an operating-system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion
- Processor is limited in its ability to interleave programs
- Efficiency of execution could be noticeably degraded
- Multiprocessing
  - Disabling interrupts on one processor will not guarantee mutual exclusion

# Mutual Exclusion - Machine Instructions

- Assume you have either:

```
Test-and-set
        int testAndSet (target)
        int *target;
        {       int value = *target;
                *target = 1;
                return (value);
        }
Swap
        void Swap(a, b)
        int *a;
        int *b;
        {       int temp = *a;
                *a = *b;
                *b = temp;
        }
```

- Implemented in hardware as a single atomic instruction
- 80x86

**Test-and-set solution**

```
shared var lock = 0;
repeat
        while testAndSet(&lock) do skip;
        critical section
        lock = 0;
        remainder section
until false;
```

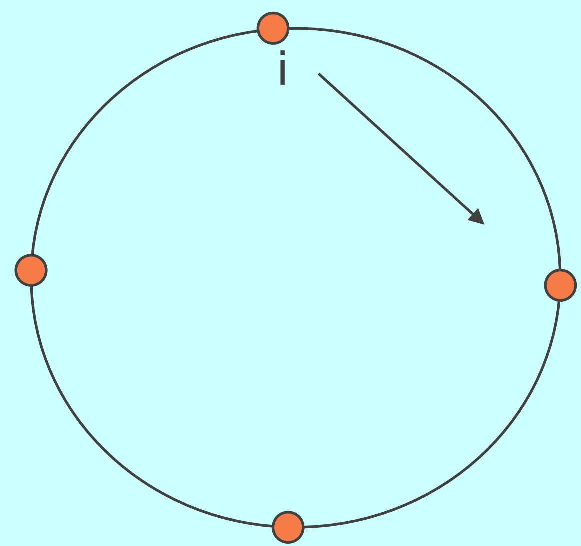
**Atomic Swap Solution**

```
shared var lock = 0;
repeat
        key := 1;
        repeat
                swap (&lock, &key);
        until key=0;
        critical section
        lock := 0;
        remainder section
until false;
```

- The problem with the previous solutions is that they do not give a bounded wait to processes wishing to enter their critical sections. In practice, however, this has not shown to be a problem.

# Bounded Wait Test-and-Set Solution

```
shared var
   waiting: array [0..n-1] of boolean; // Want to enter
   lock: boolean
var j: 0..n-1;
     key: boolean;
repeat
 waiting[i] := true;                    // Wants to enter
 key := true;
 while waiting[i] and key do
       key := testAndSet (&lock);
 waiting[i] := false;
 critical section
 j := i+1 mod n;
 while (j<>i) and (not waiting[j])
     do j := j+1 mod n;
 if j=i then lock := 0
        else waiting[j] := false;
 remainder section
until false;
```




- For any number od processes

# Ticket algorithm (serializer)

- Simple solution for pro n processes

```
shared var number = 1, next = 1, turn: array [1..n];
repeat
        <turn[i] = number; number = number + 1>
        <await turn[i] = next>
        critical section
        <next = next + 1>
until false;
```

Implementation using fetch and add

```
FA(var, incr) = <t = var; var = var + incr; return t>

shared var ...
repeat
        turn[i] = FA(number, 1);
        while turn[i] <> next do skip;
        critical section
        next = next + 1; {need not be atomic}
until false;
```

# The End