PRL09 - MNG

Model 2022

Paralelní a distribuované algoritmy

Část 9 Vzájemné vyloučení

Pre 21/22

Souhrnné materiály

Ver 0.1

Mutual exclusion

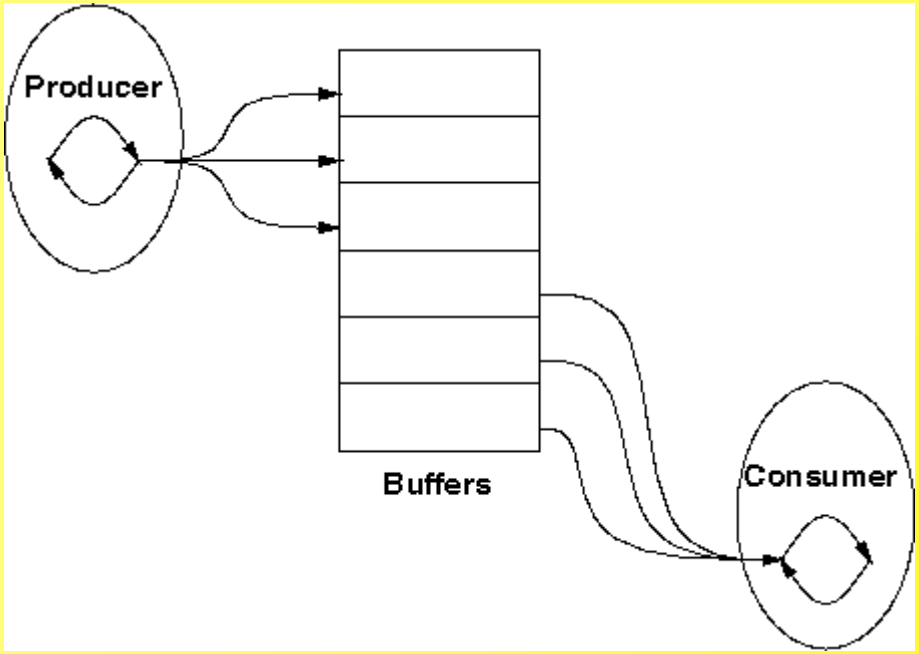
Petr Hanáček

PROCESSES INTERACTION

- Competition
  Processes compete for a resource, common competition problem is mutual exclusion
- Cooperation
  Processes cooperates or makes agreement about something

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 );
}
```

- **V OS : oba běží na jednom CPU ⇒ procesy se dostávají nepravidelně k činnosti,**
- **Zde : každý má svůj CPU**
  - **pak není přepínání kontextu**
  - **instrukce jsou prováděny nezávisle dokud spolu nezačnou interakovat (odkazují se na counter nebo buffer - sdílené proměnné)**
  - **V bufferu není problém, protože každý pracuje s jinou částí bufferu**

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

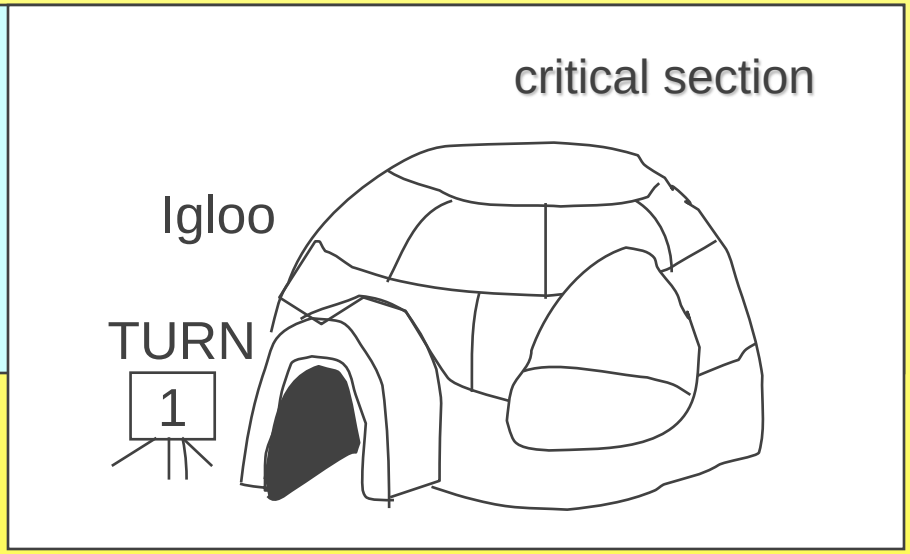
- Two Processes, P_0 and P_1.
- Use *Pi* to Mean Process of Interest.
- Use $P_{\neg i}$ to Mean the Other Process.

```
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.

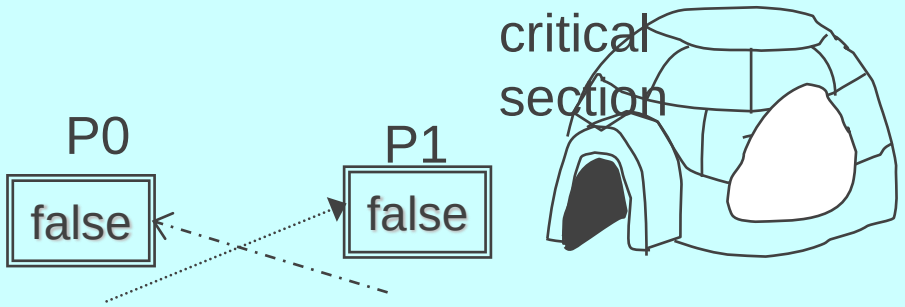
If the value on the blackboard is not the value of the process, the process leaves the igloo to wait. From time to time, the process reenters the igloo to check the blackboard.

- 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

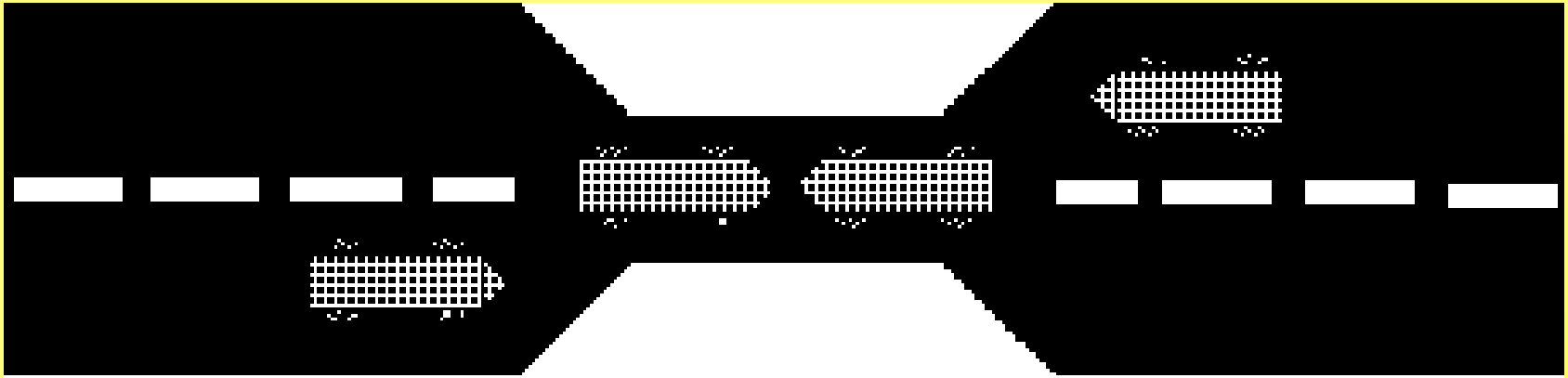
Algorithm 4

```
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;
```

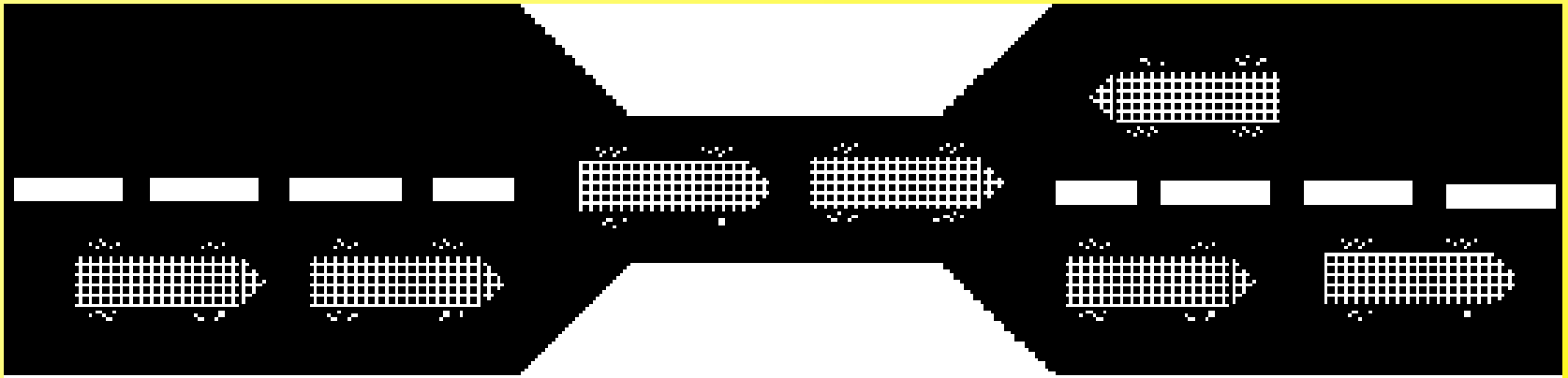
- A process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag
- Other processes are checked. If they are in the critical region, the flag is reset and later set to indicate desire to enter the critical region. This is repeated until the process can enter 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>
  - <> - atomic execution
  - B – boolean condition
  - S – statament sequence which is proved that it ends
- Meaning: its guarantied that in the beginning moment S is B = true and no inner state Sis visible for other processes.
- Example:
  - <await s>0 → s = s - 1>
  - Waits until s is positive and then decrements s.
    - » *Mutual e□clusion*<S>
    - » *Synchroni□ation on a condition* <await B>

- Properties:
  - very strong tool for synchronization, its easy to make synchronized task with this one
    ⇒ coarse-grained solution
- effectively implementable

```
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

Semaphore, Monitor, CCR,

Semaphores

Semaphores

- 1965 Dijkstra
- In design of OS, we need cooperate sequential processes
- We need efficient and reliable mechanisms for supporting cooperation

Semaphores

- Synchronization tool (provided by the OS) that do not require busy waiting
- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
  - P(S)
  - V(S)
- To avoid busy waiting: when a process has to wait, it will be put in a blocked queue of processes waiting for the same event

Semaphores

- Hence, in fact, a semaphore is a record (structure):

```
type semaphore = record
                   count: integer;
                   queue: list of process
                 end;
var S: semaphore;
```

  - When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue
  - The signal operation removes (according to a fair policy like FIFO) one process from the queue and puts it in the list of ready processes

Semaphore's operations

```
P(S):
  S.count--;
  if (S.count<0) {
    block this process
    place this process in S.queue
  }

V(S):
  S.count++;
  if (S.count<=0) {
    remove a process P from S.queue
    place this process P on ready list
  }
```

S.count must be initialized to a nonnegative value (depending on application)

Semaphores: observations

- When S.count >=0: the number of processes that can execute P(S) without being blocked = S.count
- When S.count<0: the number of processes waiting on S is = |S.count|
- Atomicity and mutual exclusion: no 2 process can be in P(S) and V(S) (on the same S) at the same time (even with multiple CPUs)
- Hence the blocks of code defining P(S) and V(S) are, in fact, critical sections

Using semaphores for solving critical section problems

- For n processes
- Initialize S.count to 1
- Then only 1 process is allowed into CS (mutual exclusion)
- To allow k processes into CS, we initialize S.count to k

```
Process Pi:
repeat
   P(S);
    CS
   V(S);
    RS
forever
```

Using semaphores to synchronize processes

- We have 2 processes: P1 and P2
- Statement S1 in P1 needs to be performed before statement S2 in P2
- Then define a semaphore synch
- Initialize synch to 0

- Proper synchronization is achieved by having in P1:
  - S1;
  - V(synch);
- And having in P2:

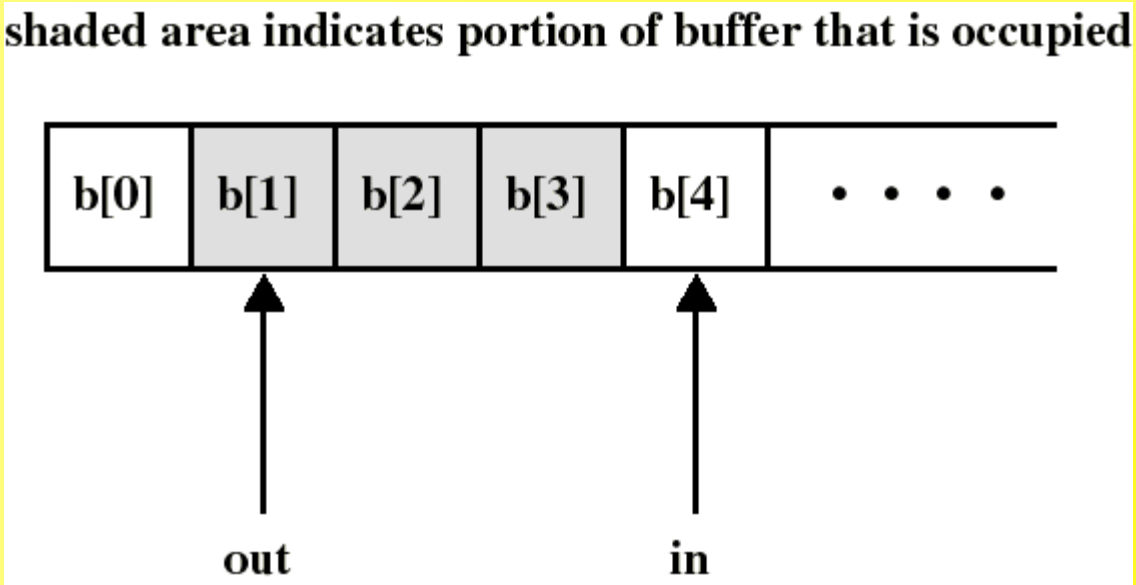
  P(synch);
  - S2;

The producer/consumer problem

- A producer process produces information that is consumed by a consumer process
  - Ex1: a print program produces characters that are consumed by a printer
  - Ex2: an assembler produces object modules that are consumed by a loader
- We need a buffer to hold items that are produced and eventually consumed
- A common paradigm for cooperating processes

P/C: unbounded buffer

- We assume first an unbounded buffer consisting of a linear array of elements
- `in` points to the next item to be produced
- `out` points to the next item to be consumed



P/C: unbounded buffer

- We need a semaphore S to perform mutual exclusion on the buffer: only 1 process at a time can access the buffer
- We need another semaphore N to synchronize producer and consumer on the number N (= in - out) of items in the buffer
  - an item can be consumed only after it has been created

Solution of P/C: unbounded buffer

```
Initialization:
  S.count:=1;
  N.count:=0;
  in:=out:=0;
```

```
append(v):
b[in]:=v;
in++;
```

```
take():
w:=b[out];
out++;
return w;
```

```
Producer:
repeat
  produce v;
  P(S);
  append(v);
  V(S);
  V(N);
forever
```

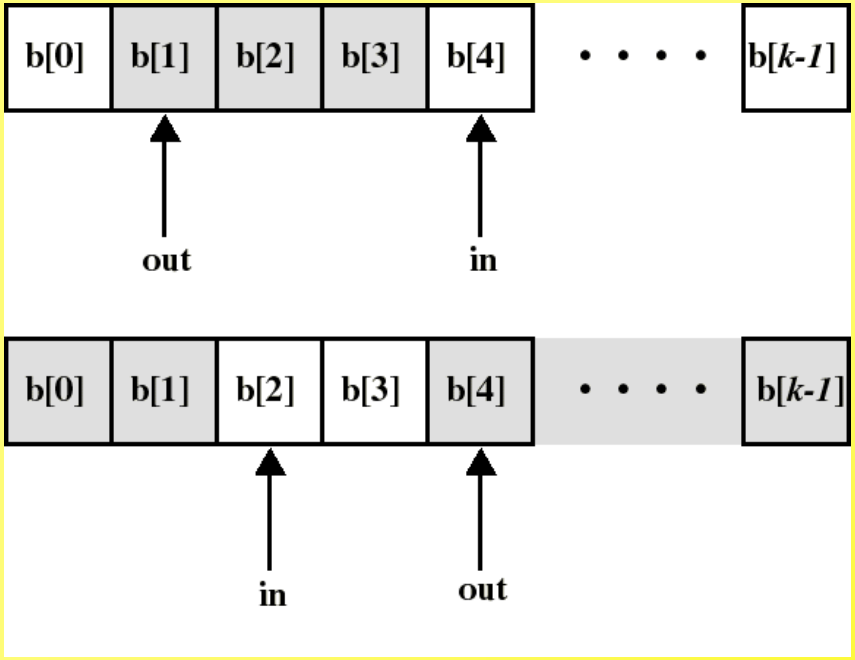
```
Consumer:
repeat
  P(N);
  P(S);
  w:=take();
  V(S);
  consume(w);
forever
```

- critical sections

P/C: unbounded buffer

- Remarks:
  - Putting V(N) inside the CS of the producer (instead of outside) has no effect since the consumer must always wait for both semaphores before proceeding
  - The consumer must perform P(N) before P(S), otherwise deadlock occurs if consumer enter CS while the buffer is empty
- *Using semaphores is a difficult art...*

P/C: finite circular buffer of size k



- can consume only when number N of (consumable) items is at least 1 (now: N!=in-out)
- can produce only when number E of empty spaces is at least 1

P/C: finite circular buffer of size k

- As before:
  - we need a semaphore S to have mutual exclusion on buffer access
  - we need a semaphore N to synchronize producer and consumer on the number of consumable items
- In addition:
  - we need a semaphore E to synchronize producer and consumer on the number of empty spaces
  - Producer cannot overflow the buffer

Solution of P/C: finite circular buffer of size k

```
Initialization: S.count:=1; in:=0;
                N.count:=0; out:=0;
                E.count:=k;
```

```
append(v):
b[in]:=v;
in:=(in+1)
     mod k;
```

```
take():
w:=b[out];
out:=(out+1)
      mod k;
return w;
```

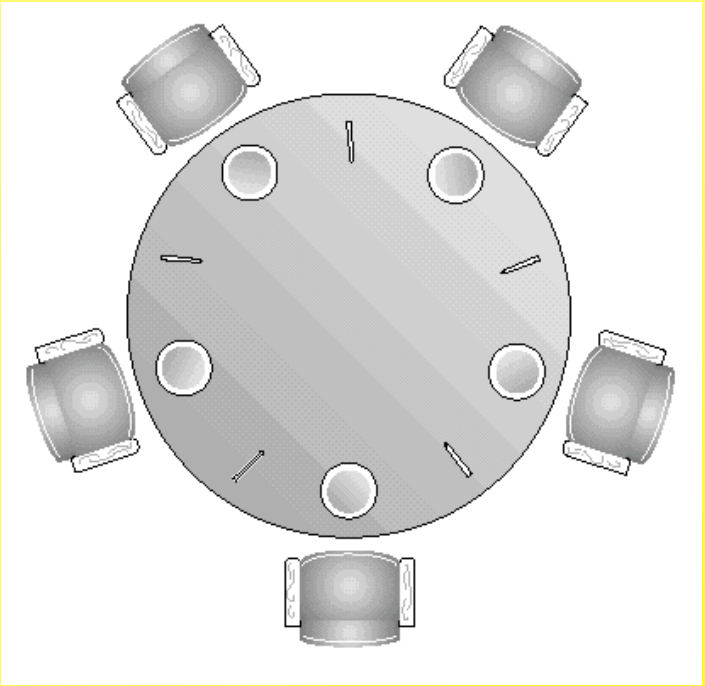
```
Producer:
repeat
  produce v;
  P(E);
  P(S);
  append(v);
  V(S);
  V(N);
forever
```

critical sections

```
Consumer:
repeat
  P(N);
  P(S);
  w:=take();
  V(S);
  V(E);
  consume(w);
forever
```

The Dining Philosophers Problem

- 5 philosophers who only eat and think
- each need to use 2 forks for eating
- we have only 5 forks
- A classical synchronization problem
- Illustrates the difficulty of allocating resources among process without deadlock and starvation



The Dining Philosophers Problem

- Each philosopher is a process
- One semaphore per fork:
  - fork: array[0..4] of semaphores
  - Initialization: fork[i].count:=1 for i:=0..4
- A first attempt:
- Deadlock if each philosopher start by picking his left fork!

```
Process Pi:
repeat
 think;
 P(fork[i]);
 P(fork[i+1 mod 5]);
 eat;
 V(fork[i+1 mod 5]);
 V(fork[i]);
forever
```

The Dining Philosophers Problem

- A solution: admit only 4 philosophers at a time that tries to eat
- Then 1 philosopher can always eat when the other 3 are holding 1 fork
- Hence, we can use another semaphore T that would limit at 4 the numb. of philosophers sitting at the table
- Initialize: T.count:=4

```
Process Pi:
repeat
 think;
 P(T);
 P(fork[i]);
 P(fork[i+1 mod 5]);
 eat;
 V(fork[i+1 mod 5]);
 V(fork[i]);
 V(T);
forever
```

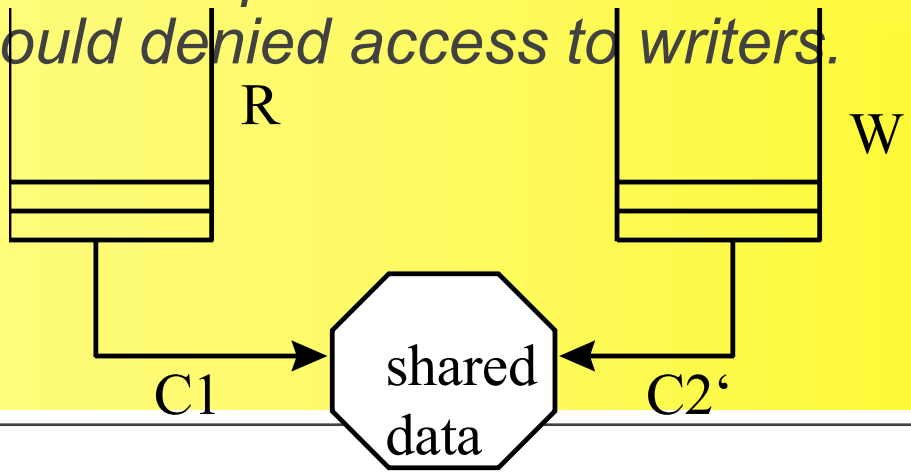
Readers/Writers Problem

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it

Readers/Writers

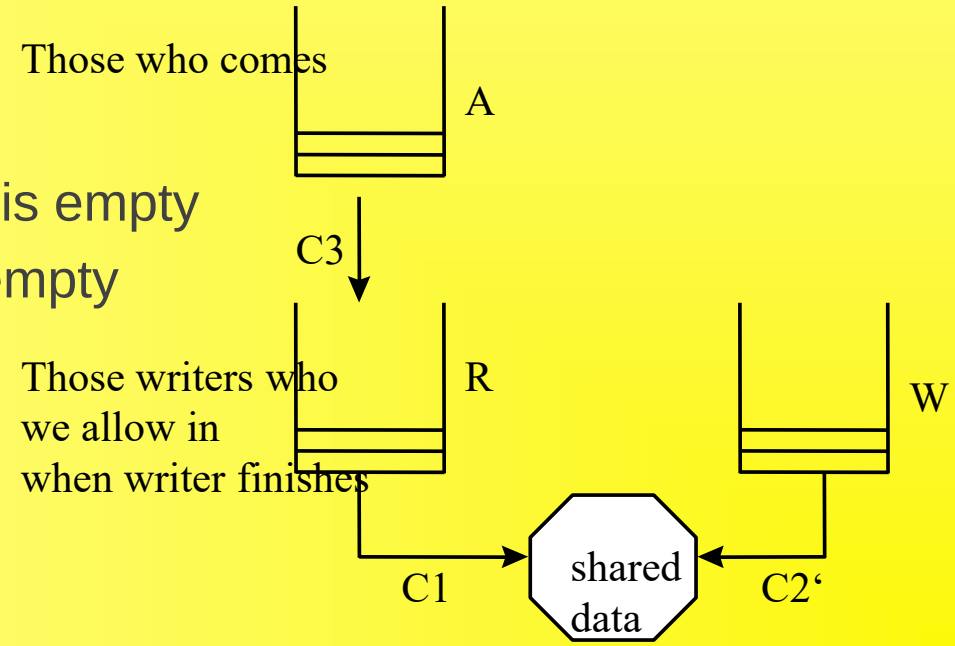
- There is shared base, which is read by readers R and modified by writers W.
- Rule R1: Several readers could read concurrently. If a writer modifies the base, nobody else could neither read nor write

  reading + writing[1] + ... + writing[n] 1
- Conditions:
  - C1: no writer writes (for readers)
  - C2: nobody reads/writes (for writers)
- *Problem: readers prefered – continuous stream of readers could denied access to writers.*



Readers/writers II

- Rule 2: If readers read and at least one writer waits then new incoming readers must wait.
- Rule 3: If writer finishes then all waiting readers start to read
- Two queues (2 for readers)
- Conditions:
  - C1: no writer writes
  - C2': nobody reads and R is empty
  - C3: writer writes or W is empty



Semaphore Solution to first Readers/Writers problem (readers priority)

- keep a count of number of current readers (readcount = 0)
- use two semaphores
  - 1. mutex: mutually exclusive access to readcount (initially 1)
  - 2. wrt: mutual exclusion for writers (initially 1)

```
Reader
P(mutex);
   readcount = readcount + 1;
   if (readcount == 1) then P(wrt);
V(mutext);
readTheFile();
P(mutex);
   readcount = readcount - 1;
   if (readcount == 0) then V(wrt);
V(mutex);
```

```
Writer
P(wrt);
writeTheFile();
V(wrt);
```

Binary semaphores

- The semaphores we have studied are called counting (or integer) semaphores
- We have also binary semaphores
  - similar to counting semaphores except that count is Boolean valued
  - counting semaphores can be implemented by binary semaphores...
  - generally more difficult to use than counting semaphores (eg: they cannot be initialized to an integer k > 1)

Binary semaphores

```
Pb(S):
   if (S.value = 1) {
     S.value := 0;
   } else {
    block this process
    place this process in S.queue
  }

Vb(S):
  if (S.queue is empty) {
    S.value := 1;
  } else {
    remove a process P from S.queue
    place this process P on ready list
  }
```

Problems with semaphores

- semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes
- But P(S) and V(S) are scattered among several processes. Hence, difficult to understand their effects
- Usage must be correct in all the processes
- One bad (or malicious) process can fail the entire collection of processes

CR, CCR

Critical Regions

- Declare a shared variable v of type T as
  - **var v: shared T;**
    - The variable v can *only* be accessed inside a `region` statement.
  - **region v do S;**
    - While S is being executed, no other process can access v
- Using a Critical Region For Mutual Exclusion

```
var count: shared integer;

producer {
      .
      .
      region count do
          count = count + 1;
      .
      .
}
```

```
consumer {
      .
      .
      region count do
          count = count - 1;
      .
      .
}
```

Implementation of a Critical Region

- Assign each shared variable, x, a semaphore, xmutex, initialized to 1
- P &V semaphore around any *region* referencing the variable
- *The Code:*
  - `var x: shared T;`
    `region x do S;`
  - *Becomes:*

```
var x: T;
var xmutex: semaphore;

P(xmutex);
S;
V(xmutex);
```

- Nebe peč :
  - 1. Proces performs : **"region x do region y do S1;"**
  - 2. Proces performs : **"region y do region x do S2;"**

Becomes:

```
Process1:

   "P(x.mutex); P(y.mutex);"
   "S1;"
   "V(y.mutex); V(x.mutex);"
```

```
Process2:

   "P(y.mutex); P(x.mutex);"
   "S2;"
   "V(x.mutex); V(y.mutex);"
```

Solution: conditional critical section(CCR)

- **region v when B do S;**
  - » B is a boolean expression which must be true before S is executed

Example: Bounded Buffer

```
var buffer: shared record
               pool: array[0..n-1] of item;
               count, in, out: integer;
            end;
```

Producer:

```
region buffer when count < n
     do begin
         pool[in] := nextp;
         in := in + 1 mod n;
         count := count + 1;
        end;
```

Consumer:

```
region buffer when count > 0
     do begin
         nextc := pool[out];
         out := out + 1 mod n;
         count := count - 1;
        end;
```

Implementation of the CCR

```
region v when B do S
var     e: semaphore; {entry semaphore}
        b: array[1..m] of sema; {delay semaphore}
                musí jich být tolik, kolik je podmínek v programu
        d: array[1..m] of int; {delay counters}
 { for every shared variable }

CCR:    P(e);
        if not Bi then
                d[i]++;
                V(e);
                P(b[i]);
        end if
        S;      ke splnění podmínky nesmí dojít jinde, než v S, což je zajištěno
        if ((B1) and (d[1] > 0)) then { d[1]--; V(b[1]); }
                elseif ((B2) and (d[2] > 0)) then {d[2]--; V(b[2]); }
                        ...
                        else V(e);
```

- Problems
  - large number of semaphores
  - re-evaluation of all conditions
  - every process must evaluate conditions of all other processes (locals!)

Monitors

Monitors

- Are high-level language constructs that provide equivalent functionality to that of semaphores but are easier to control
- Found in many concurrent programming languages
  - Concurrent Pascal, Modula-3, C++, Java...
- Can be implemented by semaphores...

Monitor

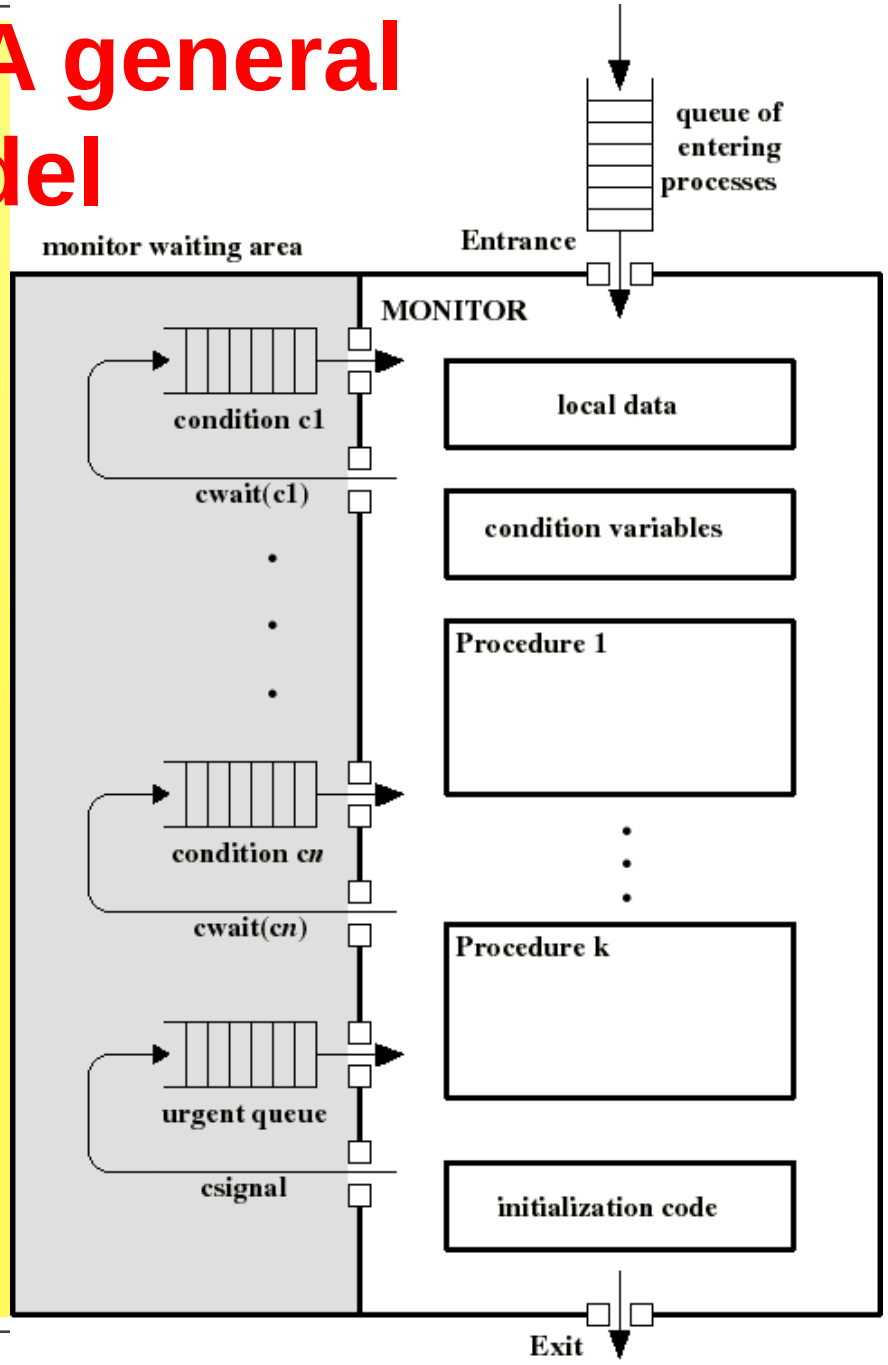
- Is a software module containing:
  - one or more procedures
  - an initialization sequence
  - local data variables
- Characteristics:
  - local variables accessible only by monitor's procedures
  - a process enters the monitor by invoking one of it's procedures
  - only one process can be in the monitor at any one time

Monitor

- The monitor ensures mutual exclusion: no need to program this constraint explicitly
- Hence, shared data are protected by placing them in the monitor
  - The monitor locks the shared data on process entry
- Process synchronization is done by the programmer by using condition variables that represent conditions a process may need to wait for before executing in the monitor

Monitor – A general model

- Awaiting processes are either in the entrance queue or in a condition queue
- A process puts itself into condition queue cn by issuing wait(cn)
- signal(cn) brings into the monitor 1 process in condition cn queue
- Hence signal(cn) blocks the calling process and puts it in the urgent queue (unless signal is the last operation of the monitor procedure)



Condition variables

- are local to the monitor (accessible only within the monitor)
- can be access and changed only by two functions:
  - wait(a): blocks execution of the calling process on condition (variable) a
    - the process can resume execution only if another process executes csignal(a)
  - signal(a): resume execution of some process blocked on condition (variable) a.
    - If several such process exists: choose any one
    - If no such process exists: do nothing
  - Optional:
    - bool empty (c)
      - Returns true if delay queue is empty
    - signal_all (c)
      - while not empty (c) do signal (c);

Implementing monitor using semaphore

```
shared var
        e: semaphore = 1;
        c: semaphore = 0;                   // for each condition
        nc: integer = 0;                    // for each condition

monitor entry
        P(e)
wait(cond)
        nc = nc + 1;
        V(e); P(c); P(e);
signal(cond)
        if (nc > 0) then nc = nc - 1; V(c);
        end if
monitor exit
        V(e)
```

Implementing semaphore using monitor

```
monitor Semaphore
        var     s: integer = 1;
                pos: condition;
        procedure P();
        begin
                while (s=0) do wait (pos);
                s = s-1;
        end;
        procedure V();
        begin
                s = s+1;
                signal(pos);
        end;
end;
```

Monitor for the bounded P/C problem

- Monitor needs to hold the buffer:
  - buffer: array[0..k-1] of items;
- needs two condition variables:
  - notfull: signal(notfull) indicates that the buffer is not full
  - notemty: signal(notempty) indicates that the buffer is not empty
- needs buffer pointers and counts:
  - nextin: points to next item to be appended
  - nextout: points to next item to be taken
  - count: holds the number of items in buffer

Monitor for the bounded P/C problem - the general model

```
Monitor boundedbuffer:
  buffer: array[0..k-1] of items;
  nextin:=0, nextout:=0,
          count:=0: integer;
  notfull, notempty: condition;

  Procedure Append(v);
    begin
    if (count=k) wait(notfull);
    buffer[nextin]:= v;
    nextin:= nextin+1 mod k;
    count++;
    signal(notempty);
    end;
```

```
Procedure Take(v);
    begin
    if (count=0) wait(notempty);
    v:= buffer[nextout];
    nextout:= nextout+1 mod k;
    count--;
    signal(notfull);
    end;
```

Deadlock Free Dining Philosophers

```
Monitor dining-philosophers

   var state: array[0..4] of (thinking,hungry,eating);
   var philosopher: array[0..4] of condition;

   procedure entry pickup( i: 0..4 );
       begin
          state[i] := hungry;
          test(i);
          if state[i] != eating
             then philosopher[i].wait;
       end;

   procedure entry putdown( i: 0..4 );
       begin
          state[i] := thinking;
          test( i-1 mod 5 );
          test( i+1 mod 5 );
       end;
```

Deadlock Free Dining Philosophers

```
procedure test( k: 0..4 );
    begin
       if state[k-1 mod 5] != eating
          and state[k] = hungry
          and state[k+1 mod 5] != eating
          then begin
                 state[k] := eating;
                 philosopher[k].signal;
               end;
    end;

begin
   for i := 0 to 4 do state[i] := thinking;
end.
```

The END

Podpůrný text k přednášce “Vzájemné vyloučení”, Paralelní a distribuované systémy, 2020

FZjr

Tento text obsahuje komentáře k přednášce kurzu PRL na téma obvyklých problémů paralelismu se zaměřením na implementaci mechanismů pro vzájemné vzloučení procesů.

Text vznikl jako provizorní řešení nahrazení přednášek ve stavu nouze a zavření škol v ČR v roce 2020. Text v této neprošel korektůrou a proto může obsahovat jazykové, doufáme však že nikoli věcné, chyby.

Součástí tohoto textu budou odkazy nebo výňatky volně přístupných textů, ze kterých bylo čerpáno. Studenty prosíme, aby tyto texty, pokud jsou uvedeny, považovali za hlavní zdroj studia. Tento text pak má jednotlivé části propojovat, dovysvětlovat, případně demonstrovat na příkladech.

Klasické problémy paralelismu

Mezi klasické problémy paralelismu patří problém **soupeření** o zdroje. Obecně ke zdroji v daný okamžik se může snažit přistupovat jeden, několik (omezený počet), nebo libovolný počet procesů. Jinou výzvou pro řešení u paralelních systémů je to, jak zajistit **spolupráci** mezi procesy. V tomto případě se jedná o problém rozdělení úloh, ať již počáteční, nebo během výpočtu, detekci správného nebo i nesprávného ukončení výpočtu, synchronizaci apod. V této přednášce se budeme většinou věnovat soupeření mezi procesy. Spolupráci mezi nimi se budou dotýkat metody, které budou obsaženy v navazujících přednáškách.

Ve našem případě budeme předpokládat, že objektem soupeření je místo v paměti, do kterého mohou procesy zapisovat a ze kterého mohou číst. Problém je pak ilustrován na slidu 7. Jelikož čtení a zápis nemusí být atomické na strojové úrovni, může dojít ke zmatení, pokud tyto probíhají souběžně, resp. pokud ze souběžných instrukcí je alespoň jedna instrukcí zápisu. Je zřejmé, že například dvě instrukce zapisující rozdílné hodnoty na stejné místo v paměti spolu soupeří a výsledná hodnota u jednoho z nich pak nemusí odpovídat skutečnosti. V paměti se nachází hodnota o jedna menší, než byla původně, zatím co první z procesů má uloženou v proměnné *count* nesprávnou hodnotu o jedna větší než původní.

Tedy je třeba ošetřit to, že pokud procesy přistupují ke sdíleným zdrojům, v tomto případě k místu v paměti, je třeba, aby se vzájemně vylučovali v tomto přístupu. Pokud jeden proces se nachází v části kódu, ve kterém se pracuje se sdíleným prostředkem, a takové části kódu budeme dále nazývat **kritickými sekcemi**, jiný proces musí čekat, než vstoupí do této kritické sekce. Musí tedy dojít k **vzájemnému vyloučení** procesů v kritické sekci.

Pro první demonstraci typických problémů paralelismu, se kterými se programátor může běžně setkat, uvažujme vyrovnávací paměť. Jelikož se již vžil a do našeho jazyka přibyl výraz buffer, budu jej nadále v tomto textu používat. Buffer může být nekonečný, což je ovšem jen teoretický předpoklad, nebo konečný, což jsou naopak jejich praktické reaizace. Pro manipulaci s nim mohou procesy vykonávat dvě operace, jednu pro **čtení** a jednu pro **zápis**. Po provedení těchto operaci se posune ukazatel, který udává, ze které pozice má být čteno, případně na kterou pozici má být zapsáno příště.

(viz slide 3)

Úloha, která demonstruje problém zápisu a čtení položek z bufferu je známá jako **problém producent / konzument**. Jeden proces, producent, používá operaci zápisu a předpokládáme, že tuto operaci vykonává v nějakých časových okamžicích. Konzument vykonává operaci čtení, opět v okamžicích s jistým časovým odstupem.

Základní problém je ten, že může docházet ke konfliktu při souběžném působení konzumenta i producenta v systému. U neomezeného i omezeného bufferu pak nastane problém také, pokud konzument nemá co konzumovat. V případě omezeného bufferu nastává navíc další problém, když v případě úplného jeho naplnění nemůže producent vkládat nové položky.

Na slidech 4 a 5 jsou kódy programu, který simuluje chování těchto dvou aktérů s tím, že oba dva řeší výše uvedený problém pro případ omezeného bufferu. Vidíme ale, že oba přistupují, krémě samotného bufferu, také ke sdíleným proměnným *in*, *out*, nebo *count*. Souběh instrukcí kde jeden proces proměnnou modifikuje a druhý čte (např in=in+1 / while(in==out)) může způsobit nežádoucí chování celého systému.

Proto potřebujeme umět zabezpečit kritickou sekci tak, aby bylo zajištěno vzájemné vyloučení procesů. Pro mechanismus přijatelný pro vzájemné vyloučení budeme předpokládat následující (slide 7)

- Pokud neuvedeme jinak, budeme uvažovat o pouze jednom možném procesu, který se může nacházet v jeden okamžik kritické sekci
- Procesy, které mimo kritickou sekci ukončí svoji činnost, neintereagují s ostatními procesy
- Nedochází k uváznutí ani vyhladovění procesů
- Pokud proces chce vstoupit do kritické sekce a není důvod mu nevyhovět, tedy jiný proces se zde nenachází, je tomuto procesu bezodkladně umožněn vstup
- Procesy tráví v kritické sekci časově omezenou dobu

Uváznutí je situace, kdy procesy čekají na událost, která nemůže nastat. Pokud jeden proces drží prostředek a požaduje prostředek držený jiným procesem, může dojít k uváznutí, pokud druhý porces neuvolní prostředek držený prvním procesem. Obecněji může existovat uzavřený cyklus takovýchto procesů, kdy P1 drží R1 a požaduje R2, P2 drží R2 a požaduje R3 atd, až Pi držá Ri a požaduje R1. Pro to, aby vznikl deadlok, jsou splněny nyní tři podmínky možnosti existence uváznutí, také známé jako Coffmanovy podmínky. A to 1, prostředek má výlučné použití 2, proces žádá jiný prostředek, i když nějaký již drží 3, existuje cyklus žádostí, jak jsme ukázali. Poslední podmínka, kterou jsme nezmínili, ale uvažovali implicitně, je ta, že procesu nemůže být prostředek odejmut jiným procesem, je také platná a uváznutí je na světě.

Vyhladovění si představíme stručněji pouze tak, že proces hladoví, pokud se pokouší získat prostředek, ale ten získávají i opakovaně jiné procesy a na hladovějící proces se nedostává.

Ve zbývajícím textu probereme některé přístupy k vytváření mechanismů vzájemného vyloučení. Budou to přístupy jednak čistě algoritmické, dále přístupy s využitím hardwarové podpory a na závěr přístupy s použitím mechanismů, které jsou dnes běžně součástí operačních systémů a vývojových prostředí.

Algoritmická řešení

Problém vzájemného vyloučení bývá na přednáčkách vysvětlován pomocí ‘eskymáčků’ (vizte slidy 9 – 17). Zde jsou představeny algoritmické přístupy k řešení problému vzájemného vyloučení, ale tato řešení nesplňují všechny vytyčené podmínky. Na slidu 19 je ale jedno funkční řešení uvedeno.

K nastudování problematiky, kterou tyto slidy návodně pokrývají, lze uvést volně dostupný článek od Edsgera Dijkstry níže.

http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF, sekce 2.1, strany 11 - 18

Petersonův algoritmus je dalším možným algoritmickým řešením vzájemného vyloučení a tímto navazujeme na odkazovaný text, konkrétně na Dekkerův algoritmus.

```
1. shared var flag: array [0..1] of boolean;
2.                turn: 0..1;
3. repeat flag[i] := true;
4.    turn := j;
5.    while (flag[j] and turn=j) do skip;
6.    critical section
7.    flag[i] := false;
8.    remainder section
9. until false;
```

Funkčnosti tohoto algoritmu vidíme, pokud si uvědomíme, kdy proces vstupuje do kritické sekce. Takový proces má zajištěno, že buďto druhý proces neavizoval úmysl vstupu do kritické sekce, nebo má přednost. V prvním případě je jisté, že druhý proces pokud by žádal o vstup do kritické sekce by čekal, jelikož první proces jistě avizoval úmysl a druhý proces mu sám dá přednost (4) nastavením proměnné turn, tedy musí čekat. Pokud by nastal druhý případ, že povolení vstupu do kritické sekce je na základě nastavení 'přednosti' turn na tento process. Pak toto nastavení muselo proběhnout až po té, co první proces avizoval svůj zájem o kritickou sekci a nastavil turn na druhý proces. Tedy je jisté, že druhý proces avizuje zájem, stejně jako první, a jelikož přednost má proces první, druhý proces je zablokován.

Druhou otázkou je, zdali může docházet u tohoto řešení k vyhladovění. Co když jeden proces je rychlejší než druhý a vítězil by (téměř) vždy v souboji o kritickou sekci? Čekající druhý proces je v situaci, že je zablokován na podmínce (5). Pokud první proces ukončí svoji činnost, zvedne šraňky (tedy nastaví avízo na nezájem o kritickou sekci) a I kdyby dříve, než druhý process stihne zareagovat, znou šraňky spustil, následně dá přednost druhému, čekajícímu (4), tím je zablokován (5) a druhý process si pohodlně vstoupí do kritické sekce. Tedy ani tento problem u Petersonova algoritmu nehrozí.

Teoretickým přístupem bez rozšířenější implementace je zavedení instrukce, která by atomicky prováděla složenou operaci, konkrétně vyhodnocení podmínky a na základě její pravdivosti provedení nebo neprovedení akce dané příkazem. Takovou instrukci vidíte na slidech 21 a 22. Pokud je v příkazu AWAIT<B -> S> splněna podmínka B, je proveden příkaz S a to atomicky. Jak toto zjednoduší implementaci vidíme na druhém slidu, kde se v testuje flag druhého procesu a v případě, že není nastaven, nastaví první proces svůj flag, to vše atomicky, tedy nemůže dojít k problémům sdílené proměnné, jak jsme je zmínili výše.

Hardwarové řešení

Jednou z možností, jak řešit vzájemné vyloučení u paralelismu založeném na přepínání kontextu, je zakázat přerušení. Některé nevýhody tohoto přístupu jsou vypsány na slide 23. Hlavní důvod nevhodnosti tohoto přístupu je ten, že v případě multiprocesorů, které jsou dnes již naprosto běžné, vypínání kontextů u jednotlivých procesorů není řešením.

Cesta ke snažší a bezpečnější realizaci mechanism vzájemného vyloučení vede přes instrukce, které atomicky vykonávají několik operací, konkrétně čtení a zápis. Dvě takové instrukce jsou k dispozici

Instrukce “test and set”, případně též známá jako atomická výměna, byly implementovány již v prvních multiprocesorech, které vznikaly od počátku šedesátých let, prvně implementována v Burroughs B5000 []. Později, od roku 1970, byla v systému IBM System/370 součástí instrukční sady instrukce compare-and-swap. Instrukci test-and-set a instrukci podobnou compare-and-swap, konkrétně jen její swap verzi, použijeme nyní pro demonstraci hardwarové podpory pro řešení problému paralelismu.

Algoritmicky je funkčnost obou instrukcí ukázána na slidu 24

Použití těchto instrukcí pro implementaci vzájemného vyloučení je na slidu 25. Jak test-and-set, tak I instrukce swap, umožňuje velice jednoduchou implementaci vzájemného vyloučení. V prním případě proces, který vyhrál souboj o kritickou sekci, jako první provede tuto atomickou instrukci a nastaví hodnotu na adrese &lock na jedničku. Žádný jiný proces poté nemůže projít přes *while* podmínku, dokud tento vítězný proces nastaví &lock zpět na nulu poté, co opustí kritickou sekci.

Vzájemné vyloučení více procesů

Majíc tyto instrukce k dispozici, uvedeme nyní dva algoritmy, které řeší problém vzájemného vyloučení více procesů.

Prvním z nich je **Bounded Wait Test-and-Set**. Algoritmicky jej představuje slide 26. Pro pochopení principu tohoto algoritmu je třeba si uvědomit, že algoritmus funguje tak, že vstoupí-li jednou nějaký proces do kritické sekce, zavře ji, a ta zůstává zavřená, dokud tento proces neskončí a žádný jiný proces o kritickou sekci neusiluje. Jinak řečeno, pokud proces, který dobyl kritickou sekci, má tuto sekci opustit, neotvírá ji, pokud existuje jiný proces, který avizval o vstup zájem. To zjistí tak, že ‘v kruhu‘ otestuje, zdali je nastaven flag některého z ostatních procesů (while (j<>i) and (not waiting[j]) do j := j+1 mod n;). Pokud tomu tak není, otevře kritickou sekci (if j=i then lock := 0). Pokud je takový proces nalezen, tedy ukazatel *j* nedoběhl na původní index procesu, pak pustí tento žádající proces do kritické sekce (else waiting[j] := false;). Timto nastavením padne podmínka (while waiting[i] and key do), proces ukončí čekání a vstoupí do kritické secke. Do té doby byl zablokován s využitím instukce test-and-set.

Druhý mechanismus vzájemného vyloučení více procesů, nazvaný **Ticket algorithm**, dobře známe, pokud chodíme na poštu, do banky a podobně. Princip je ten, že příchozí (proces) čekající na obsloužení (v kritické sekci) si z přístroje odebere své číslo a sleduje na panelu, zdali se toto číslo objeví. Pokud tomu tak je, je proces do kritické sekce vpuštěn. Algoritmus je uveden na slidu 27, v tomto případě je pro řešení kritických částí přístupu k proměnným zvolena instukce Await a atomického provádění více instrukcí znázorněného přes lomené závorky.

Všechny, jak funkční, tak nefunkční řešení, používali pro realizaci mechanismů pro vzájemné vyloučení **aktivního čekání**. Viděli jsme, že procesy, pokud nemohou vstoupit do kritické sekce, probíhají, čekají ve smyčce, a spotřebovávají systémový čas. Takové řešení nemá praktické uplatnění a řešení, které si uvedeme v následující části, budou bez takového aktivního čekání.

Semafory

Semafory, které jsou obvyklým řešením pro vzájemné vyloučení, jsou dostatečně popsány v literatuře. Opět doporučujeme prostudovat text Dijkstry, na který jsme odkazovali a to i z historického důvodu, jelikož semafory jsou zde jako synchronizační nástroj poprvé uvedeny.

http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF

Semafor je zde představen jako hodnota, která je přístupná přes operace V (z nizozemštiny Verhoog, zvyš, inkrementuj) a P (Probeer, pokus se), které mají jako argument právě semafor. Tyto operace jsou vykonávány atomicky a tím zabraňují konfliktům při přístupu k čitači semaforu.

Všimněte si, že na konci strany 29EWD je operace P uvedena tak, že pokud je hodnota čitače nulová, tato operace není ukončena, dokud nenastane operace V. Ve formě, která se nyní běžně prezentuje, predpokládáme jako součást semaforu také nějakou frontu procesů. V této frontě jsou umístěny procesy, které čekají na dokončení operace P.

Algoritmicky je činnost obou operací pro obecný semafor ukázána na slidu 6. Vidíme zde, že v případě, že operace P nemůže být dokončena, protože snížená hodnota čitače je záporná, je proces, který operaci spustil, umístěn do fronty semaforu. V případě provádění operace V je hodnota čitače zvýšena a dále pokud se ve frontě semaforu nachází nějaký proces, je ten, který je na čele fronty, znovu spuštěn. Při této implementaci si můžeme všimnout dvou věcí. Pokud je hodnota čitače kladná, udává počet procesů, které mohou 'projít' přes semafor, aniž by byly blokováni. Záporná hodnota čitače udává počet procesů blokovaných ve frontě semaforu.

Na slidu 8 vidíme použití semaforů pro ošetření kritické sekce. Pokud bychom požadovali vzájemné vyloučení procesů, tedy že pouze jeden proces se může nacházet v kritické sekci, pak by počáteční hodnota čitače semaforu S měla být nastavena na jedničku. Je pak dobře vidět, že kritická sekce je vymezena použitím operací P(S) při vstupu do kritické sekce a použitím operace V(S) při jejím opuštění. Pokud by počáteční hodnota čitače byla větší než jedna, pak by se v kritické sekci mohlo nacházet až tolik procesů, kolik je tato hodnota.

Jiné použití je uvedeno na slidu 9. Semafor zde slouží pro synchronizaci procesů. Synchronizací rozumíme zajištění určitého pořadí dvou nebo víc událostí. V tomto případě chceme, aby událost vykonání příkazu S1 zaručeně předcházela události vykonání příkazu S2. Semafor nám toto elegantně umožní. Nejprve nastavíme počáteční hodnotu semaforu 'synch' na nulu. Semafor je tedy zavřen a operace P není dokončena před provedením opeace V. Tím také máme zabezpečeno, že pokud proces před provedením příkazu S2 zažádá o semafor a ten dostane jen pokud byla provedena operace V, tedy tím pádem i příkaz S1.

Producent / konzument řešený pomocí semaforů

Nyní se dostáváme k některým typickým problémům paralelismu a jejich řešení pomocí semaforů. Budeme předpokládat jak buffer s neomezenou kapacitou, tak i později buffer s omezenou kapacitou. A to proto, abychom postupně ukázali řešení pro oba problémy, které mohou nastat, jak jsme uvedli na začátku této lekce.

Tedy nejprve mějme buffer s neomezenou kapacitou (viďte slidy 11-14). Co musíme ošetřit je situace, kdy konzument chce konzumovat z prázdného bufferu. Dále potřebujeme ošetřit samotný přístup k bufferu. Budeme potřebovat dva semafory. Onen, který zajistí vzájemné vyloučení procesů v přístupu k bufferu, nazveme **S**, a inicializujeme jej na jedničku. Druhý semafor, nazvěme jej **N**, je nastaven na nulu, což udává, že v bufferu není žádná položka. Konzument, chce-li konzumovat, musí zažádat o oba tyto semafory, tedy jak o přístup k bufferu, tak o položku, existuje - li. Existenci jedné nebo více položek zajišťuje producent. Ten také musí zažádat o semafor **S**, ale semafor **N** zvyšuje s každou položkou, kterou vyprodukuje použitím operace V. Slide 13 ukazuje korektní implementaci řešení tohoto problému semafory. Také si zde ale můžeme uvědomit, jak ošemetné může být používání semaforů. Pokud bychom u konzumenta prohodili pořadí operací P u obou semaforů, tedy pokud by P(S) předcházelo P(N), dočkali bychom se uváznutí velmi brzy. Stačilo by, aby konzument zažádal o položku při prázdném bufferu. Pak by nejprve zavřel semafor **S** a žádal by neúspěšně o semafor **N**. Nidky by se tohoto semaforu nedočkal, protože pro jeho zvednutí by musel producent položku vyprodukovat a umístit ji do bufferu. Jelikož je ale semafor **S** zavřený, nedočkal by se přístupu k bufferu a systém by uváznul.

Nyní se podívejme na problém bufferu s omezenou kapacitou. Původní problém prázdného bufferu zůstává, navíc se musíme vyrovnat se situací, kdy je naopak buffer plný. Potřebujeme další semafor, nazveme jej **E** a jeho počáteční nastavení je na hodnotu kapacity bufferu. Je zřejmé, že nyní je možné provést úspěšně operaci P nad tímto semaforem tolikrát, kolik je kapacita. Tuto operaci nad E vykonává producent a je úspěšný, dokud je v bufferu místo. Naopak konzument hodnotu semaforu E zvyšuje, jelikož konzumací položky z bufferu uvolňuje místo. Pečlivá a na deadlock opatrná realizace řešení je uvedena na slidu 17. Jistě snadno naleznete takové pořadí užití semaforů, které by opět umožnilo nežádoucí uváznutí systému.

Problém pěti filosofů

Pokud jedna z Coffmanových podmínek zněla, že deadlock umožní požadování prostředků procesy v uzavřeném cyklu, je dobré si tento případ demonstrovat. U problému nazvaném "Problém pěti filosofů" máme zadání takové, že několik procesů, tedy pět, ale toto číslo nemusí být nezbytně takové, může požadovat až dva prostředky. Tedy prozaičtěji, jak zní úloha oficiálně, pět filosofů (procesů) v některých časových úsecích přemýšlí a v některých chtějí jíst, případně jí. Pokud jedí, jedí těstoviny dvěma vidličkami, které musí získat. Jak je vidět na obrázku na slidu 18, každý filosof sdíli vidličky se svými sousedy po levici a po pravici. Pokud jeden ze sousedů jí, nemůže tento prostředek získat a musí počkat, až soused dojí. Pokud by mělo dojít k uváznutí, musí nastat situace, kdy každý filosof drží jednu vidličku a požaduje druhou. Pak všechny prostředky / vidličky jsou používány a přitom žádný proces / filosof nemůže pokračovat.

Pomocí semaforů by takové řešení umožňující deadlock mohlo být implementováno podle algoritmu uvedeného na slide 19. Vzájemné vyloučení prostředků i=1 .. 5 je zajištěno semafory fork[i]. Uváznutí si snad není obtížné představit v okamžiku, když všech pět procesů provede první operaci P. Řešení tohoto problému není o Složité. Myšlenka, že úplně postačí, když pouze čtyři procesy mohou bojovat o prostředky, tedy že vždy aspoň jeden z filosofů je mimo oblast boje o semafory fork[i], nám zapezpečí, že Coffinova podmínka o cyklickém uzavření žádostí o prostředky nemůže být splněna a tím také nemůže dojít k uváznutí. Implementace tohoto opatření s jedním semaforem inicializovaným na hodnotu o jedna menší, než je počet filosofů, je ukázána na slidu 20.

Problém čtenáři / písaři

Třetím z typických problémů paralelismu, který si v této lekci uvedeme (po producentech/konzumentech a filosofech) je problém čtenářů a písařů. Na začátku této lekce jsme již o procesech které zapisují či čtou do/z jednoho místa hovořili. Nyní se podíváme na možná řešení tohoto problému, který je znám jako problém čtenářů a písařů. Víme, že pokud by v kritické sekci bylo více čtenářů, ale žádný písař, je situace v pořádku. Ten, kdo vylučuje přítomnost jiných procesů v kritické sekci, ať již čtenářů nebo písařů, je písař.

Na slidech 22 a 23 jsou uvedena dvě řešení, přičemž to první může vést k hladovění písaru. Začněme tedy tím, že budeme mít nastaveny dvě podmínky, které umožňují vstup jednak čtenářům a jednak písařum. Pro čtenáře (nikoli pro písaře, jak je uvedeno na slidu) stačí když platí, že žádný písař aktuálně v kritické sekci nic nezapisuje. Pro písaře pak podmínka umožnění vstupu je širší a zní, že v kritické sekci se nemůže nacházet žádný proces. Vyhladovění je ale snad dobře vidět. V případě většího množství čtenářů může dojít k tomu, že po celou dobu chodu systému bude vždy aspoň jeden proces číst a žádný písař nemá šanci se do kritické sekce dostat. Proto je dobré podmínky upravit. Férovější způsob je ten, kdy čtenáři jsou rozděleni do dvou front a ta , ze které se vstupuje do kritické sekce obsahuje čtenáře, kteří se do ní dostali v situaci, kdy žádný písař nežádal o vstup do kritické sekce. Potom žádost písaře o vstup zablokuje frontu čtenářů (na slidu A) a po vyprázdnění fronty R a ukončení činnosti čtenářů v kritické sekci se může písař dostat na řadu.

Realizace řešení pro první. hladovějící způsob je uveden na slidu 24. Vedle semaforu zabezpečujicího exkluzivní přístup k proměnné, která uchovává aktuální počet čtenářů v kritické sekci, máme ještě semafor, který zabezpečuje, že v kritické sekci je písař vždy sám. Tedy tento semafor zamyká vstupující písař, nebo první ze vstupujících čtenářů a ti jej také odemykají.

Binární semafory

Binární semafory jsou specializací obecných semaforů. Čitač v tomto případě může obsahovat jen dva stavy a proto může být nahrazen booleovskou hodnotou. Jejich realizace, zejména algoritmy provádění operací P a V jsou uvedeny na slidech 25 a 26

Kritická sekce a podmíněná kritická sekce

O něco bohatší nástroj řešení vzájemného vyloučení představují kritické sekce. U kritických sekcí můžeme uvést nějakou ze sdílených proměnných a pro tuto proměnnou je poté zaručen exkluzivní přístup. Jak je tato konstrukce prováděna v programovacím jazyce je uvedeno na slidu 28

Implementace pomocí semaforů je vidět na slidu 30. Každá chráněná proměnná v kritické regionu má svůj vlastní semafor, který je zamknut v okamžiku, když jeden proces vstoupí do tohoto regionu. Po opuštění regionu procesem musí být i tento semafor opět zvednut. Použití semaforu, jak již víme, přináší hrozbu uváznutí, pokud tyto nejsou použity správně. A platí to i v případě kritických regionů. Víme, že k uváznutí může dojít, pokud proces drží prostředek a žádá o jiný. To nastane, pokud budou dvě vnořené kritické sekce, resp. pokud budou v některé části kódu chránit více než jednu sdílenou proměnnou. Pak na slidu 31 vidíme ukázku jak snadno a rychle vyrobit v tomto mechanusmu deadlock. Jeden proces zamkne proměnnou x a žádá o semafor pro proměnnou y a druhý toto udělá naopak. Takže všechny Coffmanovy podmínky jsou splněny a systém může uváznout.

Rozšíření tohoto mechanismu představují podmíněných kritických regionů. Jednostlivé části kódu s regiony jsou proveditelné na základě podmínek. Přístup k proměnné může být takto možné za různým účelem v různých situacích daných právě těmito podmínkami. Producent nevstupuje do kritického regionu, pokud nemá co produkovat a konzument sem nevstoupí, pokud nemá co konzumovat. To je řešením problému uváznutí těchto dvou procesů, které nastávalo při špatném použití semaforů, jak jsme viděli dříve. Na slidu 33 je kód s programem implementujícím podmíněné kritické regiony pro více podmínek pomocí semaforů. Každý podmíněný kritický region má svůj semafor a dále máme čitače pro tyto regiony udávající, kolik procesů žádá o vstup do nich. Proces, který chce vsoupit do kritického regionu, nejpve zažádá o semafor pro danou proměnnou a pokud je úspěšný, testuje platnost podmínky. Pokud by s podmínkou neuspěl, uvolní semafor proměnné a je uzamknut na semaforu dané podmínky. Úspěšný proces před opuštěním kritického regionu a před uvolněním semaforu pro proměnnou nejprve zkontroluje, jestli neexistuje platná podmínka, na kterou čeká nějaký jinný proces a pokud ano, pak tento proces pustí do regionu a nechá semafor zavřený.

Tuto část jsme uvedli jako mezičlánek spojující semafory s monitory. Viděli jsme, že použití semaforů je krásné, leč nebezpečné a lze se snadno dobrat uváznutí. Monitory coby synchronizační nástroj, který je dnes běžným prostředkem pro vzájemné vyloušení, toto řeší.

Monitory

V současnoti obvyklým synchronizačním mechanismem jsou monitory. Tyto jsou součástí řady programovacích jazyků, za včechny uveďme Java-u. Monitor představíme v jeho klasickém tvaru. Zahrnuje procedury, které přistupují k prostředku s výlučným přístupem, které chápeme jako nějaké

pro monitor lokální proměnné. V této části se opět odkážeme na historicky cenný a pro pochopení původních monitorů přínostný text C.A.Hoareho níže.

https://www.inf.ed.ac.uk/teaching/courses/ppls/hoare.pdf

Již na druhé straně tohoto textu se setkáváme s podmínkovými proměnnými. Jak je psáno, tyto proměnné jsou reprezentovány frontou procesů, které čekají na splnění podmínky. Nejedná se o proměnnou nesoucí nějakou hodnotu, jak jsme běžně zvyklí. Nad těmito podmínkovými proměnnými můžeme provádět dvě operace – **signal** a **wait.** Hoare ukazuje, jak semafory mohou být implementovány pomocí monitorů a naopak. Můžete se opřít i o slidy 40 a 41. Všimněte si rozdílu mezi originálním textem a tím, jak tyto operace prezentujeme na přednáškách a zvažte, jestli se tyto dva způsoby liší, případně jak, nebo neliší. Nyní si jistě prohlédnete a pochopíte strukturu monitoru na slidu 38 a význam jednotlivých částí. Urgentní fronta, jak se píše i v článku, slouží pro uchování procesů, které ještě nevybojovaly vstup do monitoru a nejsou uzavřeny na některé z podmínek. Hoare dale ukazuje, jak semafory mohou být implementovány pomocí monitorů a naopak. Můžete se opřít i o slidy 40 a 41. Všimněte si rozdílu mezi originálním textem a tím, jak tyto operace prezentujeme na přednáškách a zvažte, jestli se tyto dva způsoby liší, případně jak, nebo neliší. Stejně tak si projděte řešení problému producent / konzument pro omezený buffer, na slidech 43 a v textu kap. 4, zájemci si mohou projít i další kapitoly (a vlastně celý text), včetně kapitoly řešící monitorem problém čtenářů a písařů.

My v souladu s předloženými podkladovými slidy uzavřeme tuto lekci řešením problému filosofů pomocí monitorů. Nezmate nás dvojí použití procedury ***test***, která je na posledním ze slidů. Tato procedura v případě, že pokud filosof, pro kterého je tento test zavolán, signalizoval, že chce jíst a v okamžiku provádění této procedury jeho sousedé nejí, uvolní tohoto filosofa do monitoru, tedy nechá ho jíst, a příslušně nastaví jeho stav. Nyní tedy k hlavnímu algoritmu. Pokud se filosof rozhodne jíst, je zavolána procedura ***pickup***. Pro filosofa je nastaveno, že chce jíst a je proveden tento ***test***. A výsledek testu je buď takový, že filosof jí, a vše je v pořádku, nebo má filosof momentálně smůlu a je umístěn do fronty čekajících. Z té jej může uvolnit jen některý ze sousedů, který právě dojedl. Pokud se tak totiž stane, zavolá takový soused opět proceduru ***test*** pro oba své sousedy a pokud ti jsou hladoví a mají nyní obě vidličky k dispozici, vstupují do monitoru.