Simulační model jednoduchého SHO

příklad:

SHO: jedna fronta „Fr“, jeden kanál obsluhy „Obs“

```
# include "simulation.h“              // podpora pro kvaziparalelní prostředí
class TObsluha;    CHead *Fr;    TObsluha *Obs;
class TPozadavek : public CProcess
{
 virtual void Run();    // pro popis chování požadavku
};

class TObsluha : public CProcess
{
 virtual void Run();     // pro popis chování obsluhy
};

void TPozadavek::Run()       // „proces požadavek“
{
printf (“Prichod pozadavku v  case: %f \n”, Time () ); // ladící tisky
(new TPozadavek ) -> ActivateDelay (..); //generuje další požadavek
  if (Fr->Empty())  {  Into (Fr);              // zařadí požadavek do fronty
                       Obs->ActivateAt(Time ()); } // aktivuje obsluhu
                  else  Into (Fr) ;
  Passivate();                 // čeká na dokončení obsluhy
  printf ("Odchod pozadavku v case: %f\n“ ,Time () ); //ladící tisky
 }
```

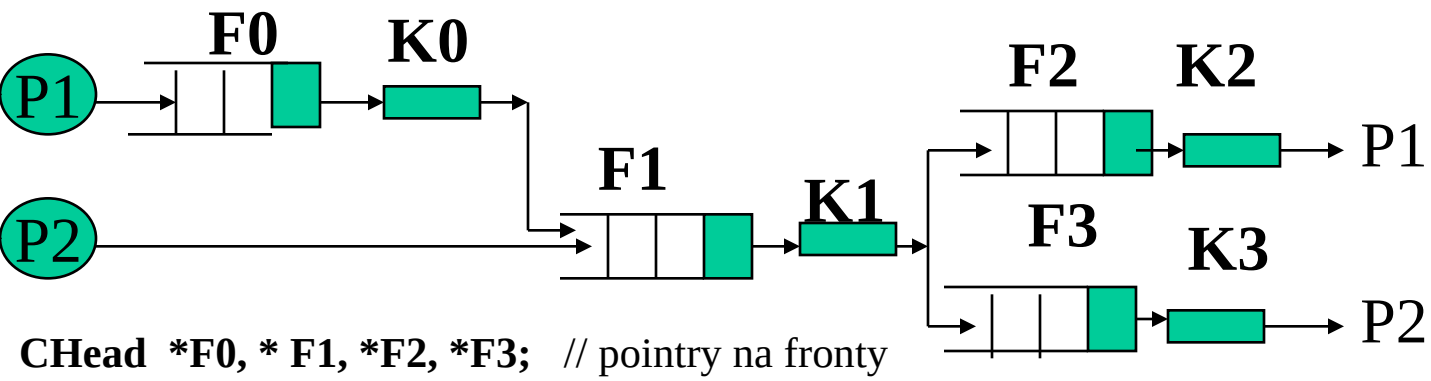
Simulační model jednoduchého SHO

```
// pokračování příkladu
void TObsluha::Run ()          // „proces obsluha“
{    CProcess* P;              // ptr na obsluhovaný požadavek
     while (true)
     {   Hold (...);           // doba obsluhy
         P=(CProcess*)Fr->First (); // odkaz na obsloužený požadavek
         P ->Out();            // vyjmutí požadavku z fronty
         P->ActivateAt (Time ());   //pro pokračování procesu požadavek
         if (Fr->Empty ())     // neni-li fronta, čeká na příchod
         Passivate();          // dalšího pozadavku
     }
}
void CSimulation::Run()              // „hlavni program“
{
 printf("---- Zacatek simulace----\n");      // indikátor začátku pro ladění
   Obs = new TObsluha ();                    //generuje objekt pro obsluhu
   Fr  =   new CHead ();                     //generuje prázdnou frontu
   (new TPozadavek)->ActivateAt (Time () );   //generuje 1. pozadavek
    Hold (1000);                             // celková doba simulace
    printf("----Konec simulace----\n\n");   // indikace konce pro ladění
};
```

poznámka: z důvodů jednoduchosti není uveden žádný statistický výpočet; jinak jde o kompletní příklad

Simulace složitějších struktur SHO

příklad struktury: následuje pouze nástin hlavních rysů:




```
CHead  *F0, * F1, *F2, *F3;   // pointry na fronty
TObsluha *K0, *K1, *K2, *K3;  // pointry na kanaly obsluhy
void  P1:: Run ( )                    // popis  chovani  pozadavku P1
 { (new P1) -> ActivateDelay (…);  generuje další požadavek
   if (F0 ->Empty{Into (F0); K0->ActivateAt (Time());}//zač. obsl. v K0
                else  Into (F0);
   Passivate ( );                     // čeká ve frontě F0
   if (F1 -> Empty  {  Into (F1);
                       K1 -> ActivateAt (Time ()); } // začátek obsl. v K1
                       else  Into (F1);
   Passivate ( );                  // čeká ve frontě F1
   if  (F2->Empty()) {…….}
   Passivate ( );       printf ( " odchod pozadavku " \n ) }
void P2 :: Run ( )    {……..}   // popis chovani pozadavku P2
TObsluha :: TObsluha  (CHead*p) : CProcess ()   { fr = p}
void TObsluha :: Run ( ) {……}   // popis chování obsluhy
void CSimulation :: Run ( )           // hlavni program
{F0 = new CHead;….. F3 = new CHead;
  K0 = new TObsluha (F0);….. K3 = new TObsluha (F3);
 (new P1 ()) -> ActivateAt (..); (new P2 ()) -> ActivateAt (..);……..
} ;
```

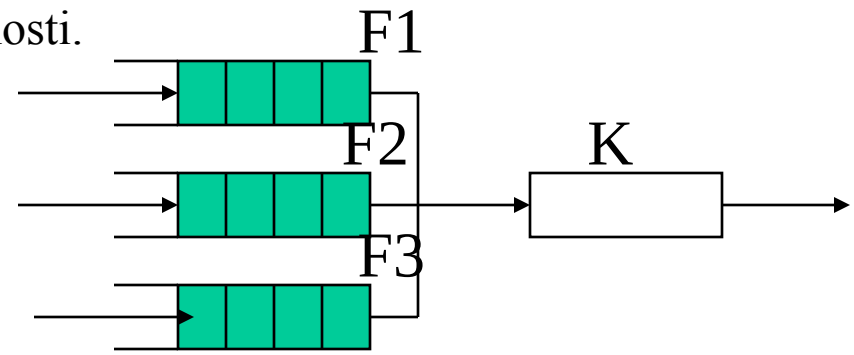
Simulace priorit

možnosti přístupu:

a) Každý požadavek disponuje atributem „priority“ => nutná změna funkcí pro zařazování do front.

b) Použití různých front pro jednotlivé priority: Proces obsluha respektuje priority výběrem příslušných front.

1) priority se slabou předností: hrají rozhodující úlohu při zahajování obsluhy; požadavek s vyšší prioritou nechá dokončit právě probíhající obsluhu. Každý proces typu požadavek může (v případě, že je prvý ve frontě) aktivovat obsluhu; toto nemá žádný vliv, pokud obsluha je právě v činnosti.




```
void  TObsluha :: Run ( )
{ while (true)                                  //  stálé  vybirani  front
   {while   ( ! (F1 -> Empty ( )))             // fronta pro nejvyšší priority
     { Hold (..);  ((CProcess*) F1  -> First ( ))  -> ActivateAt (Time ());
       F1-> First ( ) -> Out ( );  }  // opakovaná obsluha nejvyšší priority
   if ( ! (F2 -> Empty ( )))                  // fronta pro prostřední  priority
      {…….continue; }    // opět od nejvyšší priority
   if ( ! (F3 -> Empty ( )))                  // fronta pro nejnižší priority
      {…… continue; }
   Passivate ( ) ;                          // prázdné fronty
    }
}
```

Rotující slabé priority

po skončení obsluhy získává příslušná fronta nejnižší prioritu, nejvyšší priorita se posouvá na následující frontu

příklad: 2 fronty s rotující prioritou; uvedeny pouze nejdůležitější části

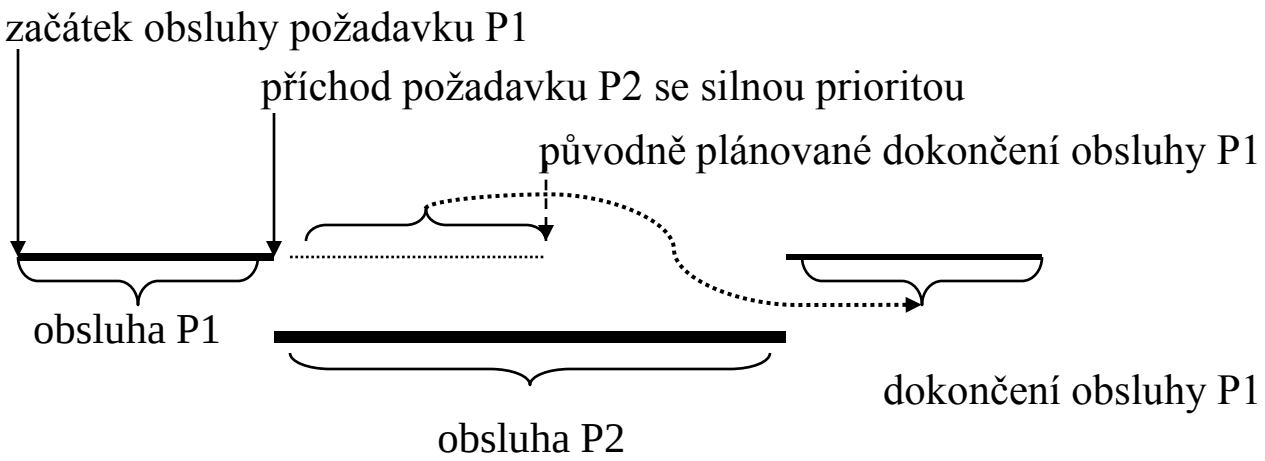
```
class TFronta: public CHead { public:TFronta *su ; };
TFronta* F1, *F2, *P;  // P..ptr na frontu s vyšší prioritou (F1 nebo F2)
TObsluha *ob;
void TObsluha :: Run ()        //  popis chování obsluhy
{while (true)
  {if (! (P->Empty ())){ Hold (...);              //obsluha vyšší priority
                         ((CProcess*) P->First ())->ActivateAt(Time());
                          P->First ()->Out (); P = P->su; //změna priority
                          continue; }
   if (!(P->su->Empty ()))){Hold (..);  //obsluha nižší priority,P je prázdná
                        ((CProcess*)P->su->First ())->ActivateAt(Time());
                         P->su->First ()->Out ();
                         P=P->su->su;  // změna priority, posuv P na další
                         continue; }
Passivate ();              // obsluha v nečinnosti
  }
}

void CSimulation :: Run ( )                    // "hlavní program"
{ob =new TObsluha ();      // následuje vytvoření „seznamu front"
 P=F1=new TFronta (); (F1->su =F2=new TFronta (); F2->su=F1;
 (new TPozadavek1 ()) -> ActivateAt (Time ())
 (new TPozadavek2 ()) -> ActivateAt (Time ()); Hold (.. );….;};
```

Priority se silnou předností

- požadavek s vyšší prioritou okamžitě přeruší právě probíhající obsluhu požadavku s nižší prioritou
- předpoklad: případné přerušení nemá vliv na kvalitu obsluhy a v přerušené obsluze lze později pokračovat

koncepce:

- proces požadavek opatříme atributem doba_ob (pro evidenci dosud vykonané obsluhy v případě přerušení)
- v případě příchodu „požadavku“ do prázdné fronty a v případě volné obsluhy „požadavek“ aktivuje obsluhu; v případě obsazenosti obsluhy volá příslušný „požadavek“ funkci Interrupt procesu obsluha; ta nastaví příznak „Interrupted“ na hodnotu true a ukončí právě probíhající obsluhu (přeplánováním na okamžitou hodnotu model. času). Obsluha pak posoudí prioritu naléhajícího požadavku (prohlédnutím nadřazených front) a zahájí obsluhu požadavku té vyšší priority. Ve zvláštním případě může jít i o pouhé pokračování obsluhy právě přerušeného požadavku.



Priority se silnou předností

příklad: 1 kanál obsluhy, 2 fronty , 2 třídy požadavků; hlavní rysy:

```
class TPozadavek1 : public CProcess    // pro silnou přednost
      {virtual void Run ( );  public: TIME doba_ob; };
class TPozadavek2 : public CProces     // pro slabou přednost
      {virtual void Run ( );  public: TIME doba_ob; };
class TObsluha : public CProcess
     { TIME  zac_ob;              bool  Interrupted;
        virtual void Run ( );     public:   void Interrupt ( ); }
TObsluha *ob;  CHead* F1, *F2;   // pointry na obsluhu a 2 fronty

void TPozadavek1 :: Run ( )        //  popis chování P1
  {(new TPozadavek1 ) ->ActivateDelay (…);   // zavlečení dalšího
    doba_ob = …;
    printf "prichod pozadavku v case %f  \n, Time() );
    if (F1 -> Empty ())
                { Into (F1);
                  if  (ob->Idle ( )) ob->ActivateAt (Time()); // zač. obsluhy
                                  else  ob ->Interrupt();  // přerušení obsluhy
                }
          else Into (F1);     // čeká ve frontě stejných priorit
    Passivate();  // je-li prvý, čeká na ukončení, jinak na nové zahájení ob
    printf "odchod pozadavku v case %f  \n, Time() );      }
void TPozadavek2 :: Run ( ) {…….}   // popis chování P2
 // analogické P1, pouze vstupují  do fronty  F2
```

Priority se silnou předností

```
// pokračování příkladu:
void TObsluha :: Interrupt  ( )
     {
       Interrupted = true;                  // nastaví příznak přerušení
       this -> ReactivateAt (Time());  // přeplánování obsluhy
     }

void TObsluha :: Run()                  // popis chování obsluhy
{ while (true)
   { while (! (F1->Empty ()))                    // fronta vyšších priorit
       {  zac_ob = Time();                  // paměť okamžiku zahájeni
          Hold (((TPozadavek1*) F1->First())->doba_ob );
                                          // předpokládaná doba obsluhy
         if (Interrupted )   // test  příznaku přerušení
          {((TPozadavek1*) F1->First() )->doba_ob- =(Time()-zac_ob);
              Interrupted = false;       // bylo přerušení  obsluhy
              continue;        //   bude pokračovat obsluhou z F1
          }
          else {  ((TProcess*)F1->First ())->ActivateAt (Time ());
                    F1->First ()->Out ();
                    continue;   // řádné ukončení;  bude pokračovat
                                // obsluhou z F1
                }
       };                  // fronta F1 je prázdná
```

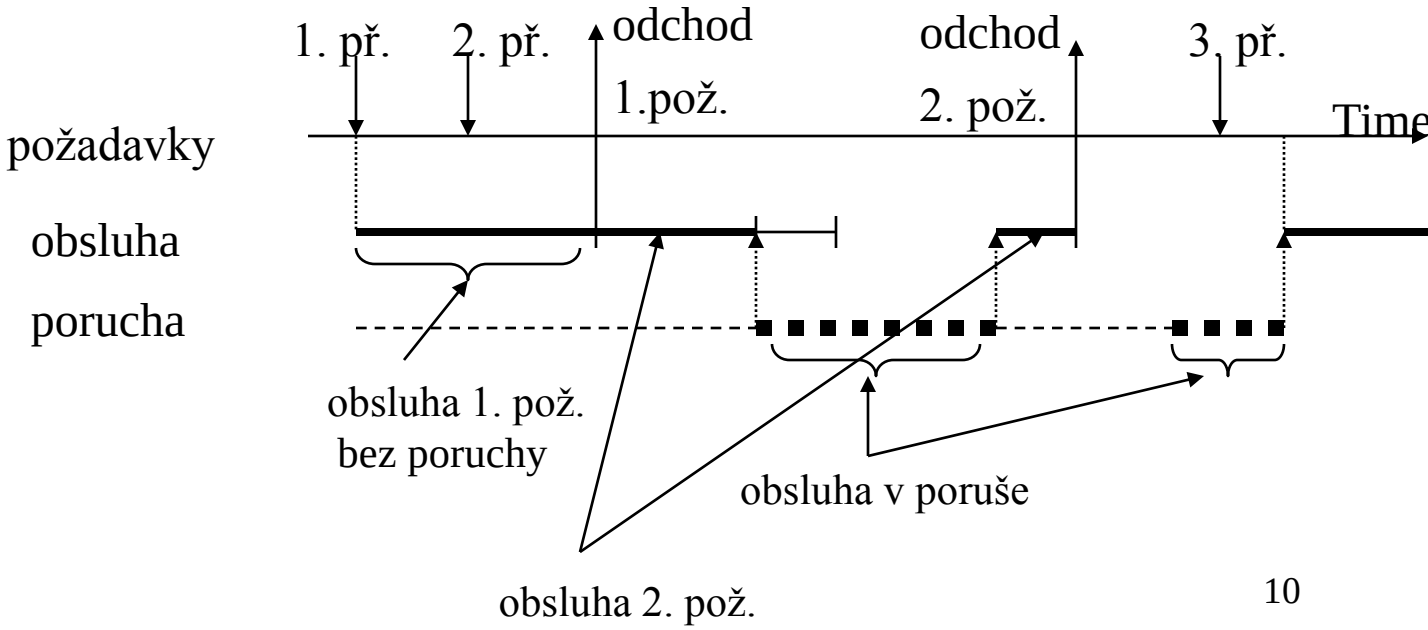
Priority se silnou předností

```
    if ((!F2->Empty()))            // fronta nižších priorit
       { zac_ob = Time();  // paměť okamžiku zahájeni
         Hold (((TPozadavek2*) F2->First())->doba_ob);
                                   // předpokládaná doba obsluhy
        if (Interrupted )   // test  příznaku přerušení
         { ((TPozadavek2*) F2->First() )->doba_ob-=(Time()-zac_ob);
            Interrupted = false;       // bylo přerušení
             continue;       //   bude pokračovat obsluhou z F1
         }
         else {  ((TProcess*)F2->First ())->ActivateAt (Time ());
                  F2->First ()->Out ();
                  continue;   // řádné ukončení;  bude pokračovat
                              // obsluhou z F1
                }
       };             // fronta F2 je prázdná
  Passivate();       // není co obsluhovat: F1 i F2 jsou prázdné
};

void CSimulation :: Run ()     // „hlav. prog.“ : generování  aktérů
    {
       ob = new TObsluha ();
        F1=new CHead  ();
        F2=new CHead ();
      (new TPozadavek1 ()) -> ActivateAt (Time ());
      (new TPozadavek2 ()) -> ActivateAt (Time ());
      Hold (...);
    };
```

Simulace poruch:

a) nezávislé poruchy - mohou nastat kdykoliv; možná koncepce:

- proces požadavek – standardní chování
- proces porucha -
  - běží nezávisle na procesu obsluhy (dle střední doby bezporuchového provozu a dle střední doby trvání poruchy)
  - v případě poruchy volá funkci Interrupt a po ukončení poruchy naplánuje obsluhu pro daný čas
- proces obsluha
  - přerušení obsluhy - podobně jako u silných priorit pomocí funkce Interrupt; proces však nepokračuje obsluhou požadavku z jiné fronty. ale čeká na aktivaci od procesu porucha
  - reakce procesu obsluha na přerušení: předpokládáme, že v obsluze lze pokračovat (jiná možnost: je-li obsluha nepřerušitelná vyřadit obsluhovaný požadavek jako zmetek)



Nezávislé poruchy

```
class TPorucha;
class TObsluha : public CProcess
 { TIME  Zac_ob,  Doba_ob;     // paměť začátku a doby obsluhy
    virtual void Run ();  bool porucha;
    public:    void Interrupt ();      TObsluha ();
               bool Interrupted ();       friend class TPorucha;          };
class TPorucha : public CProcess
 {TObsluha* ptr_ob;              //ptr na přidruženou obsluhu
  virtual void Run ();           // popis průběhu poruchy
  TIME doba_por,  do_por,  integr;  //doba por., doba bezpor. provozu
  public:TPorucha  (TObsluha*o, DWORD dobam, DWORD dobap); };
CHead* F;  TObsluha* ob;
void TObsluha :: Run ()
{  while (true)
     { if (! (F->Empty()) )           // je co obsluhovat
          {  Doba_ob =...;            // určení doby obsluhy
             while (true)             // může jít o přerušovanou obsluhu
              { Zac_ob =Time();       // paměť začátku obsluhy
                Hold (Doba_ob);       // předpokládané trvání obsluhy
                if (! Interrupted () )          // řádné ukončení obsluhy
                        { ((TProcess*)F->First())->ActivateAt(Time());
                          F -> First () -> Out ();  // odchod požadavku
                          break;       // řádně končí  obsluha
                        }
```

Nezávislé poruchy

```
// pokračování příkladu
                else { Doba_ob -= (Time() - Zac_ob); //kolik obsluhy zbývá
                       do Passivate (); while (Interrupted());   // v poruše
                                                  // bude se  pokračovat v obsluze
                     }     // konec  if
            }          //konec cyklu pro obsluhu jednoho pozadavku
      continue;   // další iterace vnějšiho cyklu: obsluha byla ukoncena
    }               // konec if (! F -> Empty()), tedy prazdna fronta
    do Passivate();  while (Interrupted ()); // prázdná fronta, muze nastat
                                                  // samovolný vznik poruchy
    }               // konec věčného cyklu
}                   // konec Run ()

void TObsluha :: Interrupt ()
{
  porucha = true;                                          // příznak poruchy
  if ( !(this -> Idle())  this -> ReactivateAt (Time ()); // přerušení obsluhy
}

bool TObsluha :: Interrupted ()   { return porucha; }

TObsluha :: TObsluha() : CProcess ()
      {
         porucha = false;
         new TPorucha (this,12, 3 ) ->ActivateAt(Time ());
      } // generuje objekt poruchy s příslušnými parametry
```

Nezávislé poruchy

```
// pokračování příkladu
TPorucha :: TPorucha ( TObsluha* o, TIME dobam, dobap )
        { ptr_ob = o;  do_por = dobam;      //  konstruktor poruch
          doba_por = dobap;                 }     // nastavení parametrů poruchy

void TPorucha :: Run ()      // proces  porucha
    {while (true)
        { Hold (do_por);                        // bezporuchový provoz
          ptr_ob -> Interrupt ();     // zacatek poruchy -vyřazení obsluhy
          Hold ( doba_por);                     // doba trvání poruchy
          ptr_ob-> porucha= false;      // uschopnění obsluhy
          integr += doba_por;                 //součet všech dob mimo provoz
          ptr_ob -> ActivateAt (Time ());  // aktivace obsluhy
        }
    }
void CSimulation :: Run ()          // „hlavní program“
    {
     ob = new  TObsluha;
     F  = new  CHead;
     printf ("zacatek simulace\n");             //  kontrolní tisk
     (new TPozadavek) -> ActivateAt (Time ());
     Hold (100);                              // celková doba simulace
     printf ("konec   simulace\n");             // kontrolní tisk
    };
```

Poruchy způsobené opotřebením

- mohou nastat pouze během obsluhy; možná koncepce:

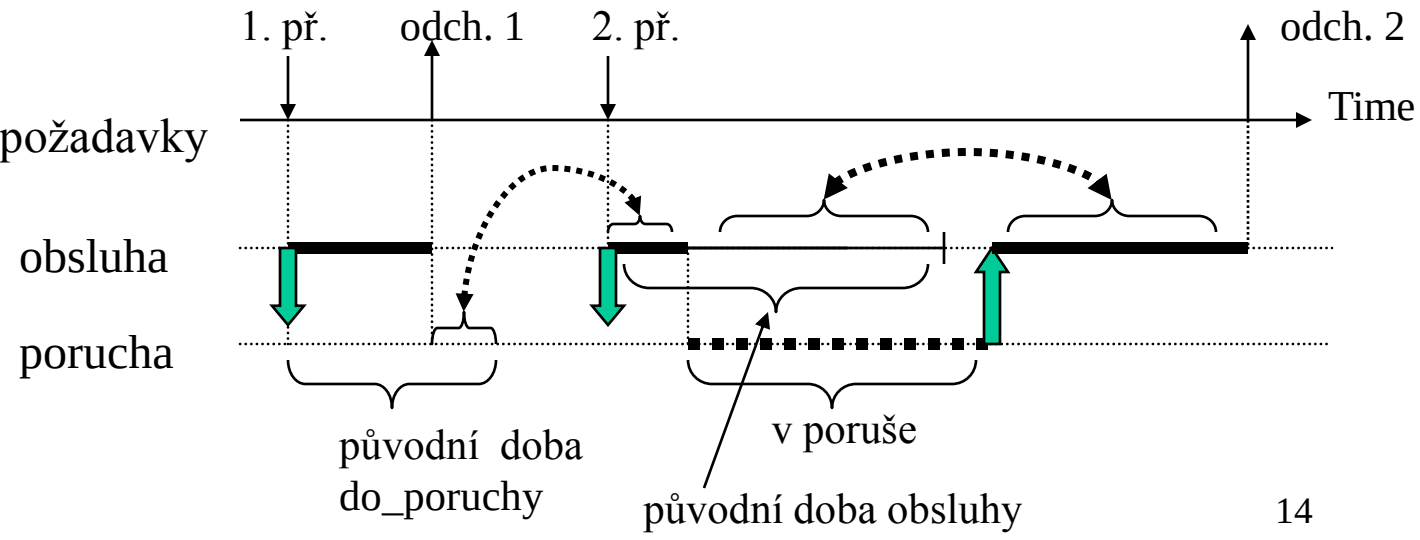
proces požadavek : beze změn

proces obsluha :

- při každém začátku nějaké obsluhy aktivuje proces porucha
- při každém ukončení obsluhy volá funkci Interrupted() (zda jde o řádné ukončení obsluhy či nikoliv):
  - při řádném ukončení pokračuje obsluhou dalšího požadavku
  - při poruše se proces pasivuje a čeká na aktivaci od poruchy; pak pokračuje obsluhou přerušeného požadavku

process porucha:

- generuje dobu bezporuchového provozu
- testuje, zda porucha nastane při právě probíhající obsluze;
  - pokud ne tak provede modifikaci atributu „do_poruchy“ a pasivaci.
  - pokud ano přeruší obsluhu pomocí funkce Interrupt()
- čeká na odstranění poruchy, pak aktivuje obsluhu (s prioritou)



Poruchy způsobené opotřebením

příklad:

```
class TPorucha;
class TObsluha : public CProcess
     { TPorucha* por;  TIME zac_ob, doba_ob;
       virtual void Run ();
       public: TObsluha ();  bool  porucha; void Interrupt();
               bool Interrupted (); friend class TPorucha;  };

class TPorucha : public CProcess
{ TObsluha*ptr_ob;TIME doba_mezi_por, doba_por, do_poruchy;
   virtual void Run ();
   public: TPorucha (TObsluha* o, TIME dobam,  dobap); };

CHead* F;  TObsluha* ob;   //ptr na frontu a obsluhu
class TPozadavek : public CProcess {……..};
TPozadavek :: Run ()  {                          } //chování požadavku

 void TObsluha :: Run ()              // chování obsluhy
 {  por = new TPorucha (this, 12, 3); //generuje sdružený proces porucha
    while (true)                     // cyklus periodického chování obsluhy
     { if (!(F->Empty ())             //  začátek obsluhy
       { por->ActivateAt (Time ());   // aktivace procesu porucha
         doba_ob=...;                 // generuje dobu obsluhy
```

Poruchy způsobené opotřebením

```
// pokračování příkladu
            while (true)              // cyklus obsluhy jednoho požadavku
               { zac_ob =Time ();       //začátek obsluhy
                  Hold (doba_ob);      // předpokládaná doba obsluhy
                  if ( ! Interrupted() )
                     { ((TPozadavek*)F->First())->ActivateAt (Time ());
                       F -> First () -> Out ();    //odchod  požadavku
                       break;  //výstup z cyklu obsluhy jednoho požadavku
                     }                 // nastalo řádné ukončení obsluhy
                  else { doba_ob - = (Time () - zac_ob);    // v poruse
                         do Passivate (); while ( Interrupted()) ; // trvání por.
                         continue;  // pokračuje  v obsluze
                       }
                }                     // konec cyklu obsluhy jednoho požadavku
             continue;                // začne obsluha dalšího požadavku
          }                           // konec if  (!F -> Empty ())
     Passivate ();                    // fronta prázdná
   }                                  // konec cyklu periodického chování obsluhy
 }                                    // konec Run ()
```

TObsluha :: TObsluha() : CProcess ()

```
            {  porucha = false; }         //konstruktor obsluhy
```

Poruchy způsobené opotřebením

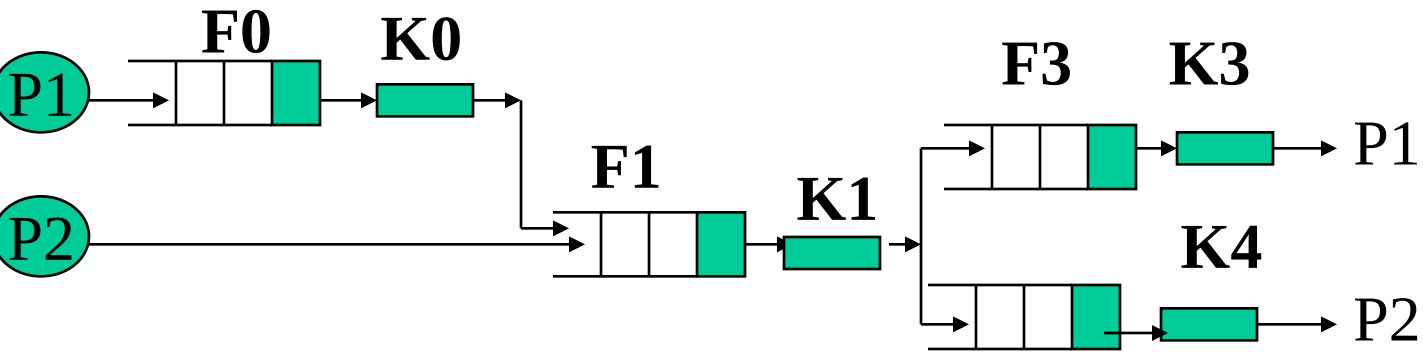
```
// pokračování příkladu
TPorucha :: TPorucha ( TObsluha* o, TIME dobam,  dobap)
     { ptr_ob = o; doba_mezi_por = dobam; doba_por = dobap;  }

void TPorucha :: Run ()
{  while   (true)                                  // periodické chování poruchy
    { do_poruchy = doba_mezi_por;       // co zbývá do poruchy
      while ( do_poruchy  >=  (( ptr_ob -> EvTime()) -Time ()))
            {                                      // porucha nebude
              do_poruchy - = ( ptr_ob -> EvTime()) - Time ();
              Passivate ();       // čeká na další aktivaci  od obsluhy
             }                     // začala další obsluha
            Hold (do_por); // porucha nastane během probíhající obsluhy
            ptr-ob->Interrupt();   // přeplánuje obsluhu a nastaví příznak
                                   // porucha = true
            Hold ( doba_por);     // doba trvání poruchy
            ptr_ob ->porucha = false;
            ptr_ob->ActivateAtPrior (Time ());  // bezprostřed. aktivace!!
// nyní nastane potlačení „poruchy" a „bezprostřední aktivace obsluhy"
// (aby nastal výpočet  jejího trvání a následné potlačení) což umožní
// poruše přečíst čas ukončení obsluhy
    }              // konec cyklu periodického chování poruchy
 }

void CSimulation :: Run () {…….};
```

Pozn.: tuto úlohu lze zjednodušit tím, že proces „obsluha" bude simulovat případnou poruchu prodloužením vlastní doby obsluhy

Semafory

- binární , vícehodnotové
- implementace: třída TRes
  - atributy: Q (fronta FIFO pro uložení pozdržených procesů)
  - metody: acquire (n), release (n)
- samostatný proces „obsluha“ není třeba (zde jako pasivní zdroj)

příklad použití:




```
TRes *K0, *K1, *K2, *K3, *K4;
void  TPozadavek 1 :: Run ( )          // chování  pozadavku
   {
        K0 -> acquire (1);             // žádost o přidělení zdroje K0
        Hold (..);                     // doba obsluhy v  K0
        K0 -> release (1);             // uvolnění zdroje K0
        K1 -> acquire (1);             // žádost o přidělení K1
        Hold (..);                      // doba obsluhy v  K1;
        K1 -> release (1);             // uvolnění K1
        K3 -> acquire (1);
        Hold (..);                      // doba obsluhy v  K3
        K3 -> release (1);
   }
```

Semafory

příklad: jednoduchý SHO: nástin použití semaforů a dvou procesů (požadavek, obsluha)

```
TObluha *Obs ;
TRes * zac_obs, * kon_obs;   // jeden semafor pro zahájení a jeden
                             // semafor pro ukončení obsluhy
void TPozadavek :: Run ( )          // popis chování požadavku
{ …….zac_obs -> acquire (1);         //čekání na zahájení obsluhy
        Obs -> ActivateAt (Time ()); // zahájení obsluhy
        kon_obs -> acquire (1);      //čekání na konec obsluhy
         ……..}
void Obsluha  :: Run ( )            // popis chování obsluhy
{   while (true)
     { Hold (..);                          // doba trvání obsluhy
       kon_obs -> release (1); // uvolní již obsloužený požadavek
       zac_obs ->  release (1); // uvolní další požadavek ve frontě
                                // a umožňuje jeho vstup do obsluhy
       Passivate ( );       // čeká na aktivaci dalším požadavkem
     }
}
void CSimulation :: Run ( )
{ zac_obs  = new TRes (1);     //  jeden zdroj pro začátek obsluhy
  kon_obs  = new TRes (0);     // žádný zdroj pro konec obsluhy
  Obs = new TObsluha;……....};
```

Semafory

příklad :synchronizace dvou procesů producent - konsument

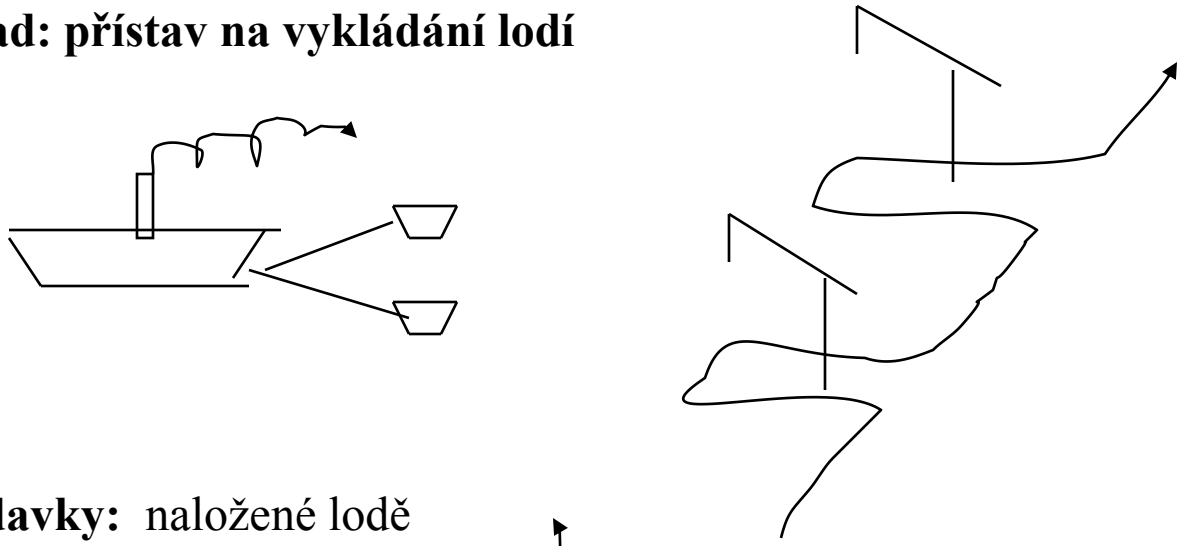
```
class TRes
     { int  avail;    //počet dostupných zdrojů
       public: CHead* Q; void acquire (int n); void release (int n);
      TRes (int n); };
int  buf ;                  //globální proměnná sloužící jako buffer
TRes* zapis, *cteni;   // pointry na zdroje
TRes :: TRes (int n)    //konstruktor třídy TRes
      { avail = n;
        Q =new CHead;  }
void TRes :: acquire ( int n) // nejsou-li zdroje pozdrží aktiv.  proces
      { while  (avail < n)   CProcess  ::  Current () -> Wait (Q);
        avail = avail  -  n;            // zdroje jsou k disposici
       }
void TRes :: release ( int n ) //uvolní  zdroje a aktiv. čekající procesy
  {   CProcess * pr, *po;        // pomocné pointry
      avail= avail + n;            // vrací zdroje
      po = CProcess :: Current ();            // odkaz na 1. proces ze SQS
      while (!Q -> Empty ())         // dá šanci všem čekajícím procesům
                { pr = (CProcess*)  Q -> First (); // 1. čekající proces  v Q
                  pr -> ActivateAfter (po);
                  po = po -> NextEv ();     //odkaz na 2. proces v SQS
                  pr ->Out ();
                }
   }
```

Semafory

```
//pokračování   příkladu     producent - konsument:
class TProducent :   public CProcess  {……..};
class TKonsument : public CProcess  {……..};
void TProducent :: Run()
   {   while (true)  {  i =…….;  // výpočet hodnot dat
                        Hold (...);   // simulace doby  výpočtu dat
                        zapis -> acquire (1);        //  žádost o „zápis“
                        Hold (...);      // simulace doby plneni bufferu
                        buf = i;         // skutečné naplnění  bufferu
                        cteni -> release (1);    //  “uvolnění bufferu“      }
   }
void TKonsument :: Run ()
  {  while (true)  {cteni -> acquire (1);  // žádost o „čtení“
                     Hold (...);                // simulace doby cteni bufferu
                     i = buf;                   //  čtení bufferu
                     zapis -> release (1);  //“uvolnění bufferu pro zápis“
                     Hold (..);         // simulace doby  zpracovani dat     }
   }
void CSimulation :: Run ()
 {
   zapis = new TRes (1);    cteni = new TRes (0);      //generování zdrojů
   new TProducent -> ActivateAt (Time ());
   new TKonsument -> ActivateAt (Time ());
   Hold (100);            // trvání simulace
 };
```

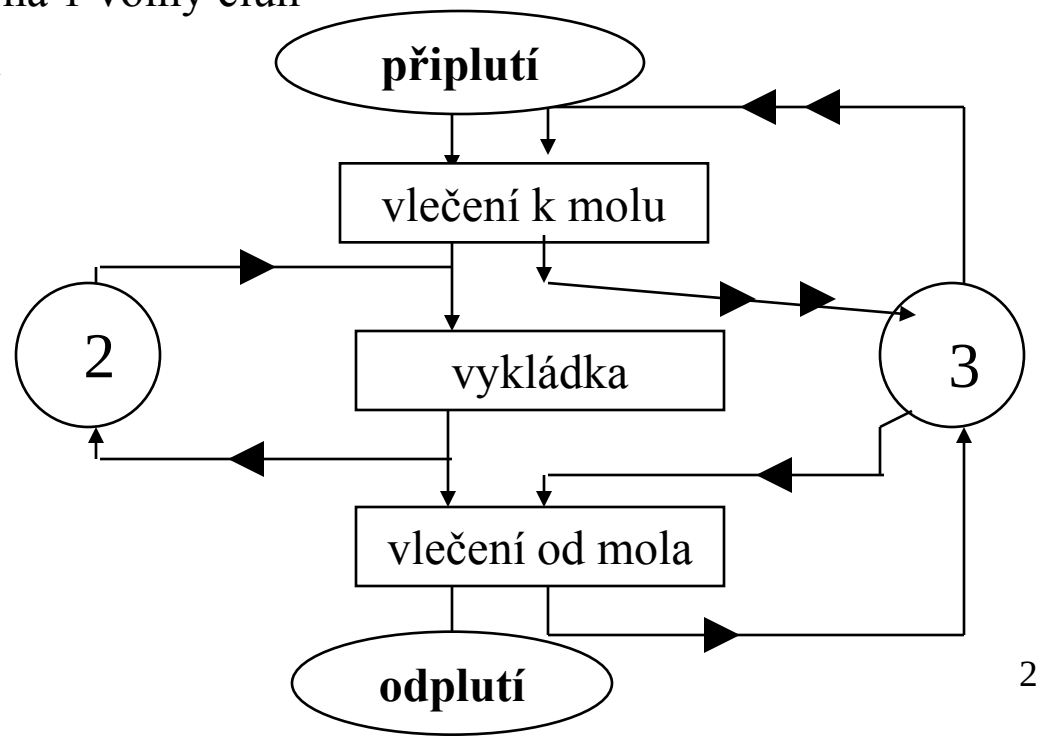
Semafory

příklad: přístav na vykládání lodí



požadavky: naložené lodě
kanály obsluhy: 2 jeřáby
3 vlečné čluny
} zdroje

provoz v přístavu:
čekání plné lodi na 2 volné vlečné čluny a 1 jeřáb
vykládání lodi u jeřábu
čekání lodi na 1 volný člun
odplutí lodi



Semafory

přístav - nástin řešení:

```
TRes * vlec_cluny, * jeraby;   // zdroje v přístavu
 void TLod :: Run ( )      // popis chování lodi v přístavu
        {…
         vlec_cluny -> acquire (2);  // žádost o 2 vleč. čluny
         Hold (..);                  // vlečení k molu
         jeraby -> acquire (1);      // žádost o 1 jeřáb
         vlec_cluny ->release (2);   // uvolnění  dvou vleč. člunů
          Hold ( ..);                // vykládka
          vlec_cluny -> acquire (1);
          Hold (..);                 // uvolnění místa u mola
          jeraby -> release (1);     // uvolnění jeřábu
          Hold (..);                 // vlečení na rejdu
         vlec_cluny -> release (1);
         ……};

void CSimulation :: Run ( )
   { vlec_cluny = new TRes  (3);
     jeraby = new TRes (2);
     while (true) { (new TLod) -> ActivateAt (Time ());
                    Hold (..);    // intervaly mezi příjezdy lodí
                    if (…..)  break;  // generování lodí do určitého počtu
                  }
  };
```

Semafory

předcházející příklad - diskuse:

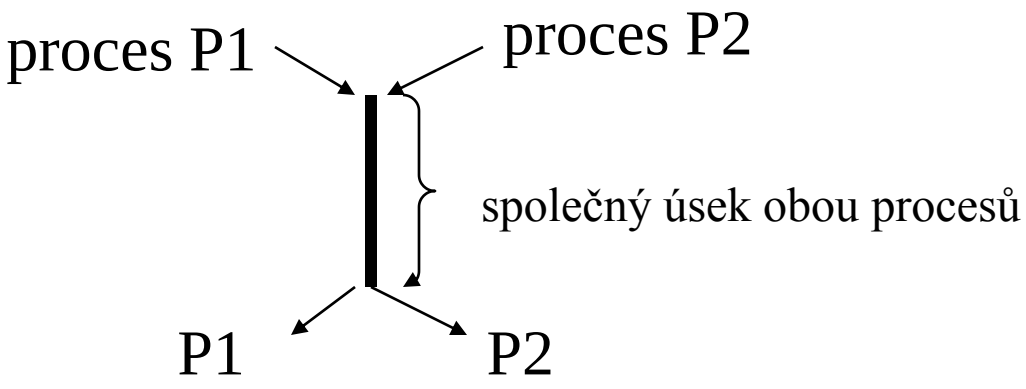
možnost smrtelného objetí (dead embrance)

- zde není žádná prevence
- nastane-li, pak simulace obvykle končí (všechny procesy kromě hlavního programu jsou pasivovány)

příklad: 2 typy lodí: velká loď - potřebuje 2 jeřáby,
malá loď - stačí 1 jeřáb

```
void TVelka_lod  :: Run ( )      // popis chování
    {
      Hold ( ..);
      vlec_cluny -> acquire (2); // získání vleč. člunů
      Hold (…);   // pomalé vlečení k molu; není zajištěn  jeřáb
      jeraby -> acquire (2);                  // není volný jeřáb  !!!!!
      ……}

void TMala_lod  :: Run ()        // popis chování
    {
      Hold ( ..);
      jeraby -> acquire (1);   // zamluvi telefonem jeřáb
      Hold (…);                // chvíli váhá s objednáním vleč. člunů
      vlec_cluny -> acquire (2);    // nejsou volné 2 vleč. čluny !!!!!
      ……}
```

Slučování a rozlučování procesů



řešení: 2 typy procesů:

- dominantní - určuje společný děj
- doprovodný - ve společném úseku bez vlastního děje

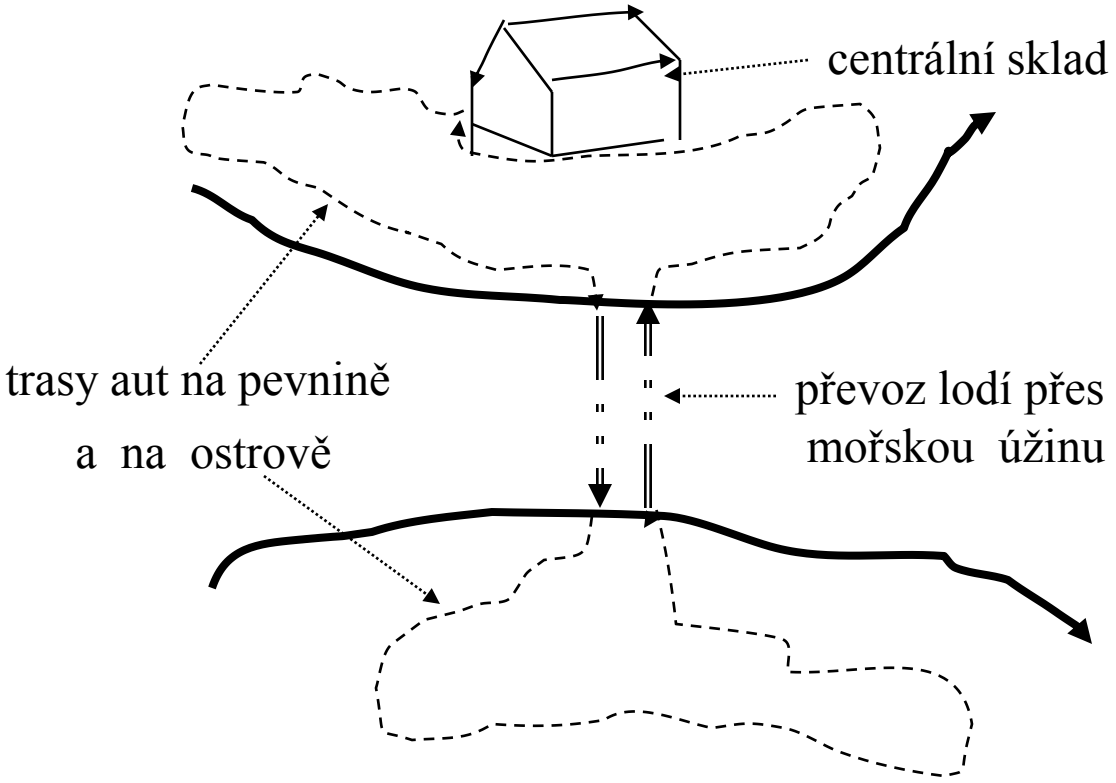
implementace: pomocí objektů třídy TCoopt - vlastnosti:

- master_q……fronta dominantních procesů
- slave_q……...fronta doprovodných procesů
- coopt ( )……..metoda pro sloučení dominantního procesu s procesem doprovodným
- waitq ( )……..metoda pro sloučení doprovodného procesu s procesem dominantním
- find ( )…….metoda pro sloučení dominantního procesu s procesem doprovodným, který navíc vykazuje určité vlastnosti (definované pomocí parametru této funkce)

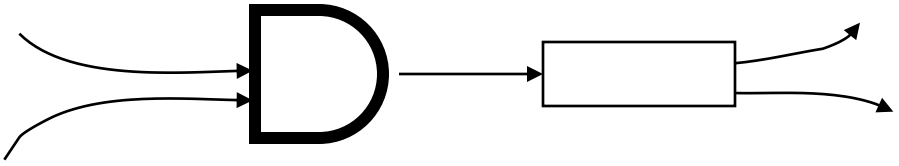
Slučování a rozlučování procesů

příklady použití:

1) převoz aut přes mořskou úžinu



2) „připasování součástek“ v obslužném místě



3) „seznamka“: nezávislé životy před sňatkem, společný život po sňatku

Slučování a rozlučování procesů

příklad: převoz aut přes mořskou úžinu

```
class TCoopt                    // třída umožňující slučování procesů
  {public: CHead  * masterq, *slaveq;
             void waitq ();
              void find ( CProcess *& m, bool  podm (CProcess*) );
             CProcess*  coopt () ;  TCoopt (); };
class TLod : public CProcess  {….};      // atributy třídy TLod
class TAuto : public CProcess  {…..};   // atributy třídy TAuto
TAuto* A;   TCoopt* FR[2];   //ptr na objekty třídy TAuto TCoopt

 void TCoopt :: waitq ()  //  pro chování doprovodného procesu
{ while (! (masterq -> Empty ()))  // probudí dominantní  proces
          {  ((CProcess*) masterq->First () ->ActivateAfter
                                                (CProcess::Current ());
              masterq->First()->Out();
          }
   CProcess ::  Current ()-> Wait (slaveq);    //čeká na dominantní proces
   return;  }

CProcess* TCoopt :: coopt ()  // pro  chování dominantního procesu
{
   CProcess* proc;    // ptr pro uchopení doprovodného procesu
   while (slaveq ->Empty ())  CProcess :: Current() -> Wait (masterq);
     // čeká na   doprovodný proces
```

Slučování a rozlučování procesů

```
//pokračování minulého příkladu:
   proc = (CProcess*) slaveq->First ();
   proc -> Out ();       // vyjmutí doprovodného procesu
   return  ( proc )     // vrátí ptr na doprovodný proces
}    // konec funkce coopt ()

void TCoopt :: find (CProcess* &m, bool  podm (CProcess*))
{ while (true)
   { if  (! (slaveq ->Empty ())           // je z čeho vybírat
             { m = (CProcess*) slaveq -> First ();   //odkaz na 1. oběť
               while (m )                          // je-li co  (tj. není NULL)
                        {  if (podm (m))         // požadovaná vlastnost
                                 { m-> Out ();  return; } // oběť vyhovuje
                             else  m = (CProcess*)  m -> Suc ();   // další
                        }
             }  //  vhodná oběť nebyla nalezena
    CProcess :: Current ()-> Wait (masterq);      // čeká na oběť
   }
}
TCoopt :: TCoopt ()   // konstruktor třídy TCoopt
     {        masterq = new CHead;
              slaveq    = new CHead;
     }
```

Slučování a rozlučování procesů

```
void TAuto :: Run ()              // doprovodný proces
  {   Hold (…);                   // jízda po pevnině k moři
      // následuje synchronizace na pevnině pomocí FR[0]
       FR[0] -> waitq ();         // čeká na převoz na ostrov
       Hold (...);                // simulace trvání cesty po ostrove
      // následuje synchronizace na ostrově pomocí FR[1]
       FR[1] -> waitq ();         //čeká na převoz zpět na pevninu
       Hold (…);   }              // návrat po pevnině
void TLod :: Run ()               // dominantní proces
 { Z:  for ( int i = 0; i < 2 ; i++)
            { A= (TAuto*) FR[i] -> coopt ();  //naložení čekajícího auta
              Hold (3);   // doba nakladani
              Hold (6);  // doba preplavby
              Hold (2);  // doba vykladani
             A -> ActivateAt (Time ()); } // pro pokračování jízdy auta
             goto Z;   }                  // periodická činnost lodi
void CSimulation:: Run ()
       {  FR[0] = new TCoopt;   FR[1] = new TCoopt;
          new TLod -> ActivateAt (Time ());
          for (int i=1; i<5; i++)
                        {new TAuto -> ActivateAt (Time ());
                         Hold (...);  } // interval mezi prijezdy aut
           Hold ( ...);    // pro  dokonceni prevozu vsech aut
       };
```

Slučování a rozlučování procesů

příklad: seznamka s „dámskou volenkou“ - nástin řešení

```
 TMuz : public CProcess
    { void Run();                  // popis chování muže
      public: int  vek, plat;     //důležité atributy muže
     TMuz (int v, int pl);  }; //parametry konstruktoru pro nast. atibutů

class TZena : public CProcess
          {
           void Run ();           //popis chování ženy
           public:
           };
TCoopt* seznamka;       // ptr jako „adresa seznamovací agentury“

TMuz :: TMuz  ( int v, int pl ) : CProcess () //konstruktor třídy TMuz
            { vek = v; plat = pl; }

 void TMuz :: Run ()       //popis chování muže - dobráka
          { ……;           // zatím v pohodě a bez úmyslu na ženitbu
            seznamka -> Waitq ();   // podá záznam v seznamce a
                                    // čeká na projevení zájmu
            ……..; }      // popis chování  po případném rozchodu
       // v opačném případě je stále ve vleku své ženy
```

Slučování a rozlučování procesů

```
//pokračování  příkladu
 bool  podm1 (TProcess* m)  // 1. kriterium mladé ženy
{ return (bool) (((TMuz*)m)->vek >20 &&((TMuz*)m)->vek < 30) ; }
 bool  podm2 (TProcess* m)   // 2. kriterium pro výběr manžela
{   return (bool) (((TMuz*)m)->plat > 100 000 );  }
 void TZena :: Run ()     // popis chování ženy
       {  Hold (20);            // je jí už dvacet - ráda by se vdala
          CProcess* znamost; //  pointer (pro nabídku z agentury)
          seznamka -> find (znamost, podm1);   // nyní  hledá v agentuře
                                // mladého muže ve věku od 20 do 30 let
           // začátek známosti - začal společný život
          Hold (1);                          // doba trvání známosti
          ((TMuz*)znamost) ->ActivateAt (Time ());  // není to ono
          // nemá prachy, poslala jsem ho k vodě
          Hold (1);             // počká, oklepe se  a zase myslí na vdavky
          seznamka ->find (znamost,podm2);  // tentokrát hledá bohatého
          // a už ho má -život s druhým mužem
          …….; }
void CSimulation:: Run ( )
{   seznamka = new TCoopt();
   (new TMuz (25, 10000))  ->ActivateAt (Time ());  // mladý muž
   (new TMuz ( 60, 125000))->ActivateAt (Time ());   //prachatý muž
   ( new TZena (); -> ActivateAt (Time ());   Hold(...);…... ;
};
```

Čekání procesů na podmínku

formy podmíněného čekání (tj. na dobu neurčitou) - dosud:

- čekání na spojení s jiným procesem
- čekání na volný zdroj , ale problém: potřebujeme např. více zdrojů, které postupně obsazujeme => můžeme zbytečně blokovat již obsazené zdroje při čekání na další zdroje. Řešení: potřeba mechanismu, který při čekání neblokuje zdroje

příklad: přístav s vykládkou lodí a s přílivem

vlečení naložených lodí - pouze při přílivu

příliv - trvá 4 hod. a nastává každých 12 hod.)

vlečení prázdných lodí - kdykoliv

1. přístup: - přírodní úkaz příliv modelujeme dostupností zdroje

```
void  TPriliv :: Run ( )  //proces na „uvolňování a obsazování přílivu“
{while(true) {vys_stav -> release (2);    // povolen vjezd  pro 2 lodě
              Hold ( “4 hod.”);            //  trvání přílivu
              vys_stav -> acquire (2);   // zablokování vjezdu do přístavu
              Hold ( “ 8 hod. “);  }          // trvání odlivu
 }
```

procesy lodí - možné situace:

a) …..vlec_cluny ->acquire (2);
vys_stav -> acquire (1);

← při čekání na vys_stav- blokování člunů

b)……..vys_stav -> acquire (1);
vlec_cluny -> acquire (2);

← nejsou-li vleč. čluny, nemůže nastat odliv (zdroj vys_stav je blokován)

Čekání procesů na podmínku

2. přístup: implementace třídy TCondq

- atribut:
  - Q……..fronta procesů čekajících na určitou podmínku
- metody:
  - waituntil (P);……metoda pro testování podmínky
    - v případě nesplněné podmínky pozdrží se Current() proces ve frontě Q
    - v případě splněné podmínky jde o prázdný příkaz
  - signal ( ); …metoda pro aktivaci procesů z fronty Q

poznámka: V případě jednoho objektu třídy TCondq mohou ve frontě Q čekat všechny procesy na tutéž podmínku nebo na různé podmínky. Pro čekání na různé podmínky je možné:

- modifikovat metodu signal tak, aby aktivovala všechny procesy Q (časově náročné)
- použít různé objekty třídy condq a tudíž různé fronty Q
- **příklad:** nástin řešení přístavu s přílivem

```
class  TCondq
{ public: CHead* Q;
          void waituntil ( bool  podm () );
          void signal ();   TCondq ();          };
TRes * vlec_cluny, * jeraby;  bool  vysoky_stav;  TCondq * fr;
 TCondq  :: TCondq ( )   { Q = new CHead; }
```

Čekání procesů na podmínku

přístav s přílivem - pokračování:

```
void TCondq : : waituntil ( bool  podm () )
{ if (! (podm() ))         //není splněná uvedená podmínka
    { CProcess :: Current () -> Wait (Q);     //čeká na splnění podmínky
      while (1)                    // zde pokračuje po aktivaci funkcí signal
        { if (podm () )     // otestování podmínky po aktivaci
              {
                Q->First ()->Out (); //konec čekání, proces opouští frontu
                if ( ! ( Q->empty ())) //dává šanci dalšímu procesu
                  ((CProcess*)Q->First ())->ActivateAfter
                                                ( CProcess :: Current ());
                break;              // výstup z  věčného cyklu
              }
           Cprocess :: Current () -> Passivate (); //čeká dál
        }
    }   //  podmínka splněna - jako prázdný příkaz
 }
void TCondq:: signal ( );  //signalizuje možnost splnění podmínky
  {if (!(Q->Empty ())) ((CProcess*)Q->First ())->ActivateAt
                                          ( CProcess ::Time () );  }

// následující funkce specifikuje podmínku čekání lodí v přístavu
bool  podmf ()  { return ( (bool )(((jeraby->avail ()) >= 1 &&
              ((vlec_cluny -> avail () )>= 2)  && (vysoky_stav)));  }
```

Čekání procesů na podmínku

```
// pokračování příkladu
void TLod :: Run ()            //popis chování lodě
{fr->waituntil (podmf);        // test na dostupnost zdrojů
  jeraby -> acquire(1);     vlec_cluny -> acquire(2);
  Hold (...);                    //vlečení lodi k jeřábu
  vlec_cluny -> release(2);
  fr -> signal ();    //případná aktivace lodí čekajících na čluny
  Hold ( ...);
  jeraby -> release(1); //uvolnění jeřábu po vykládce
  fr -> signal ();    //případná aktivace lodí čekajících na jeřáb
  vlec_cluny -> acquire(1);
  Hold (..);    // vleceni lodi z pristavu
  vlec_cluny -> release(1);
  fr -> signal ();        //případná aktivace lodí čekajících na člun
  }        // konec popisu chování lodi

void TPriliv :: Run ()
{ while (1)
     {  vysoky_stav = true;
        fr -> signal ();  //případná aktivace lodí čekajících na příliv
         Hold („ 4 hod“);                          // trvani  prilivu
        vysoky_stav =  false;
        Hold („8 hod“);                           // trvani odlivu
     }
}
```

Čekání procesů na podmínku

// pokračování příkladu

void CSimulation:: Run ()

```
{
    fr = new  TCondq;   // generuje objekt pro testování podmínek
    jeraby = new TRes (2);
    vlec_cluny  = new TRes (3);
    (new TPriliv ())  -> ActivateAt (Time ());
    for (int i=0; i <„max_počet“; i++)   // cyklus pro zavlékání lodí
        {
            (new TLod()) -> ActivateAt (Time ());
            Hold (...);   // interval mezi prijezdy lodí
         }
     Hold ( ...);   // pro dokonceni vykladky vsech lodi
};
```

Simulace logických obvodů

1) přímé použití modulu SIMULATION:

- chybí podpora pro simulaci struktur
- principiálně možný přístup:
  - buzení: procesy ve stavu aktivní či potlačený
  - signály: proměnné
  - součástky: procesy plánované po změně hodnot budičů vstupních signálů
- nevýhoda: změna struktury vyžaduje zásahy do modelů dílčích součástek

2) návrh podpory jako další vrstvy nad SIMULATION:

koncepce: **vytvoření vnitřní reprezentace struktury** (umožní automatické plánování procesů dílčích komponent)

dílčí objekty struktury a jejich hlavní požadované vlastnosti:

- prototyp signálů (třída TSignal):
  - ukazatele na připojené komponenty
  - resolve -resoluční funkce
  - schedule - funkce pro plánování připojených komponent
- prototyp součástek (třída TComponent):
  - input, output, inpout - umožňují z popisu zapojení vytvořit vnitřní representaci simulované struktury
  - architecture - umožní popsat vnitřní funkci komponenty
  - delayed_change (…) - umožní modelovat setrvačné zpoždění na výstupech jednotlivých komponent
- prototyp procesu pro výstup hodnot:
  - outputs - funkce pro specifikaci sledovaných signálů

Simulace logických obvodů;

```
/*----------následuje stručný popis jednoduché podpory pro simulaci ---
----------logických obvodů----------*/
class TSignal; class TComponent;
class TDelay: public CProcess   // pomocný proces realizující zpoždění
{TSignal *psig;                 // ptr na buzený signál
  TComponent* pdriver;      // ptr na přidruženou součástku
  char  oldval, newval;    // stávající hodnota a budoucí hodnota výstupu
  void  Run ();               // chování procesu
  public: TDelay  (TSignal* s,  TComponent *comp );
 char value ();  void setnewval (char); // vrací oldval,  nastaví newval
 TComponent* driver (); };    // vrací ptr na přidruženou součástku
class TSignal                 // pro reprezentaci vodičů
{ int branch_num;          // počet připojených součástek
  char  val;                  // skutečná (efektivní) hodnota signálu
  TComponent* P[5];     //pointry na připojené součástky
  CHead* list_of_dr;      //ptr na seznam budičů tohoto signálu
   void resolve (); void schedule (); //resoluč. f-ce;  plánování následníků
   public: TSignal ();
   void connect (TComponent* elem ); //připojení signálu na součástku
   void set1();  void set0 ();                 // set a reset signálu
   char value ();                              // vrací hodnotu val signálu
   CHead* drivers ();              // vrací odkaz na seznam budičů signálu
    friend class TDelay; };
```

Simulace logických obvodů

```
class TComponent : public CProcess   // prototyp obecné součástky
{void Run ();                        //popisuje chování komponenty
 protected:
        void delayed_change (TSignal* s, char val, float del); // funkce
        // pro uložení hodnoty  val do signálu s se zpožděním del
        void input  (TSignal *s );    //připojí signál s jako vstup
        void output (TSignal *s);     //připojí signál s jako výstup
        void inpout (TSignal *s);     //připojí s jako vstup i výstup
        virtual void architecture () = 0;  };     // pro funkční popis
class TResults: public CProcess  // proces zajišťující výstupní tisky
{  void Run ();
   virtual void outputs(); };          // funkce pro specifikaci  výstupů
   TResults PRINT;                     // proces pro výstupy
/* ---------------------- definice funkcí ---------------------------------*/
 TDelay  ::TDelay ( TSignal * s, TComponent * comp ) : CProcess ()
        {  pdriver = comp;   // nastavení ptr na přidruženou součástku
           oldval = 'Z';        //  počáteční nastavení
           psig = s; }          //  nastavení ukazatele na buzený signál
void TDelay  :: Run()   //popis pomocného procesu pro impl. zpoždění
      { while (true)
           {  oldval = newval;            // změna hodnot po překlopení
              psig -> resolve();          // volá resol. funkci signálu
              Passivate();        // čeká na příští naplánování
           }
      }
```

Simulace logických obvodů

```
char TDelay :: value ()                 // vrací stávající hodnotu budiče
        { return (oldval); }
void TDelay  :: setnewval (char s)   //nastaví novou hodnotu budiče
        { newval = s; }
TComponent*  TDelay :: driver ()
        { return  (pdriver);  }  // vrací ptr na přidruženou součástku
 TSignal :: TSignal()
        {
          val = 'X';                //počáteční nastavení hodnoty signálu
          list_of_dr = new  CHead();  // ptr na prázdný seznam budičů
        }
void TSignal :: schedule()    // pro plánování připojených komponent
{ int i = 0;                          // inicializace indexu na připojené součástky
  while( P[i] )           //  plánování všech připojených součástek
    {  P[i]  ->  ActivateAt ( CProcess :: Time () )  ; i++;  };
       PRINT->  ReactivateAt (CProcess :: Time ());
       // plánování či přeplánování  výstupních  tisků
}
void TSignal :: connect (TComponent *elem)
{
 P [ branch_num ] = elem;    // připojení  vstupu součástky elem
 branch_num ++;
}
```

Simulace logických obvodů

```
void TSignal:: resolve()        // resoluční funkce
{TDelay * de;                   // pomocný  ukazatel
 char res;              // pro uložení výsledné hodnoty resoluční funkce
 de= (TDelay*)list_of_dr -> First();     // ptr na 1. budič seznamu
 res= de -> value();                     // hodnota 1. budiče
 while ( de -> Suc () )  //v seznamu budičů signálu existují další budiče
      { de= (TDelay*) de -> Suc ();          // ptr na další budič
        if (res == de -> value() )  continue;    // stejné hodnoty
        if(res == ´Z´) {res = de -> value(); continue }  // res = další budič
        if (de -> value() = =’Z’) continue;     //další budič je odpojen
        res = ´X´; break;
      }
if ( res != val) { val = res; schedule ();}  // uložení nové hodnoty signálu
                              // a naplánování všech připojených komponent
}
void TSignal :: set1()        // nastaví signál na 0
          { val = '1';  schedule (); }
void TSignal :: set0()          // nastaví signál na 1
          { val = '0'; schedule (); }
char TSignal  :: value ()         // vrací hod notu  val daného signálu
          { return (val)}
CHead* TSignal :: drivers ()    // vrací ptr na seznam budičů
          { return (list_of_dr ); }
```

Simulace logických obvodů

```
void TComponent :: input (TSignal *s) // začlení danou  komponentu
    {  s->connect (this); }              // seznamu následovníků signálu s

void TComponent :: output  ( TSignal *s)
              { TDelay *pdel;     // pomoc. ptr
                pdel = new TDelay (s,this); // generuje pom. zpožď. proces
                pdel -> Into( s-> drivers()); } // začlení  mezi budiče sign. s

void TComponent :: inpout (TSignal *s)   // pro obě předchozí funkce
               { input (s);  output (s);   }

void TComponent :: delayed_change (TSignal *s, char val, float del)
{    TDelay *pdel;                                    // pomocný  ukazatel
     pdel = (TDelay*) s->drivers() -> First();
     while ((pdel -> driver () != this) && pdel -> Suc () != NULL))
           {  pdel = (TDelay*) pdel -> Suc ();   } // v seznamu budičů
  // signálu s se hledá přidružený  proces pro realizaci zpoždění výstupu
  // dané komponenty
     if (pdel ->value() != val)
               { pdel -> setnewval (val);     // předání nové hodnoty
                 pdel -> ReactivateDelay (del);  // naplánování či
                 // případné přeplánování pomoc. zpožďovacího procesu
               }
}
```

Simulace logických obvodů

```
void TComponent :: Run ()  // pro periodické chování komponenty
{ while (true)
    { architecture ();              // definuje uživatel
      Passivate ();       }
 }

void TResults :: Run ()      // proces pro zajištění výstupů
            {
             while (true) { outputs ();   Passivate ();  }
             }
/*-----následuje  funkční specifikace součástek („knihovna“)----------*/
class TTimer : public CProcess      // objekt pro realizaci časovače
    { TSignal * po;        // ptr na výstupní signál
      float dw0, dw1;      // parametry
      void Run();             // popis chování
      public: TTimer  (TSignal &o,  float w0,  float w1);  //konstruktor
    };
class TNand2 : public TComponent   //  pro realizaci nand
    { TSignal *pi1,*pi2,*po;   // ukazatelé na 2 vstupy a 1 výstup
        void architecture ();  // pro popis chování
        public: TNand2 ( TSignal &i1, TSignal &i2, TSignal &o);
        // konstruktor pro zapojení dané součástky na  vodiče
    };
```

Simulace logických obvodů

```
void TTimer :: Run()
    {  while (true)
            { po->set0 ();           // nastaví 0
              Hold (dw0);            // trvání nuly
              po->set1();            // nastaví 1
              Hold (dw1);   }     // trvání  jedničky
    }
TTimer :: TTimer ( TSignal &o, float w0, float w1) :CProcess ()
                    { po=&o; dw0=w0; dw1=w1; }

void TNand2 :: architecture ()   // funkční popis nand
            { char aux;
              if(pi1-> value() == ´X' || pi2 -> value() == ´X')  aux = ´X';
              if(pi1-> value() ==  '0'  || pi2 -> value() == '0')    aux = '1';
              if(pi1-> value() == '1' && pi2 -> value() == '1')  aux = '0';
              delayed_change (po, aux, 1);   // uloží hodnotu „aux“ do
                                             // signálu „po“ se zpožděním „1“
            }
TNand2::
TNand2 (TSignal &i1, TSignal &i2, TSignal &o) : TComponent ()
  {     pi1=&i1; pi2=&i2; po=&o;  // pro zapojení na signály
        input (pi1); input (pi2); output (po);       }
```

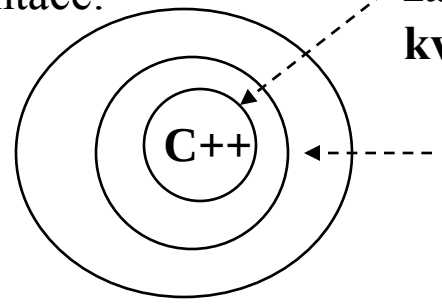
Simulace logických obvodů

```
/*------------ následuje  popis struktury  (zapojení součástek) ---------*/
TSignal  I1, I2, I3, I4, O1, O2;                    // deklarace signálů
void TResults :: outputs ()       // specifikace výstupních hodnot
{
     printf("  %f:  %c %c %c %c%c%c%c \n",  Time(), I1.value(),
    I2.value(),  I3.value(), I4.value(), O1.value(), O2.value()  );
}
TTimer T1 ( I1, 3, 3);       // deklarace a zapojení T1: buzení I1
TTimer T2 ( I2, 6, 6);       // deklarace a zapojení T2: buzeni I2
TTimer T3 ( I3, 1, 1);       // deklarace a zapojení T3: buzení I3
TTimer T4 ( I4, 2, 2);       // deklarace a zapojení T4: buzení I4
TNand2 E1 ( I1, I2, O1);  // deklarace a zapojení E1
TNand2 E2 ( I3, I4, O1);  // deklarace a zapojení E2

void CSimulation :: Run()
   {   printf("  Time    I1    I2    I3   I4    O1 \n");   // nadpis hlavičky
       T1.ActivateAt(0);
       T2.ActivateAt(0);
       T3.ActivateAt(0);
       T4.ActivateAt(0);                // spuštění časovačů
        Hold (20);
        printf ("konec simulace\n");
   };
```

Výkonnostní simulace lokálních sítí

cíl: zjištění zátěžových charakteristik sítě, délky front neodeslaných paketů, ověřování různých strategií (pro přidělování komunikačního kanálu, pro chování mostů apod.)

možná implementace:



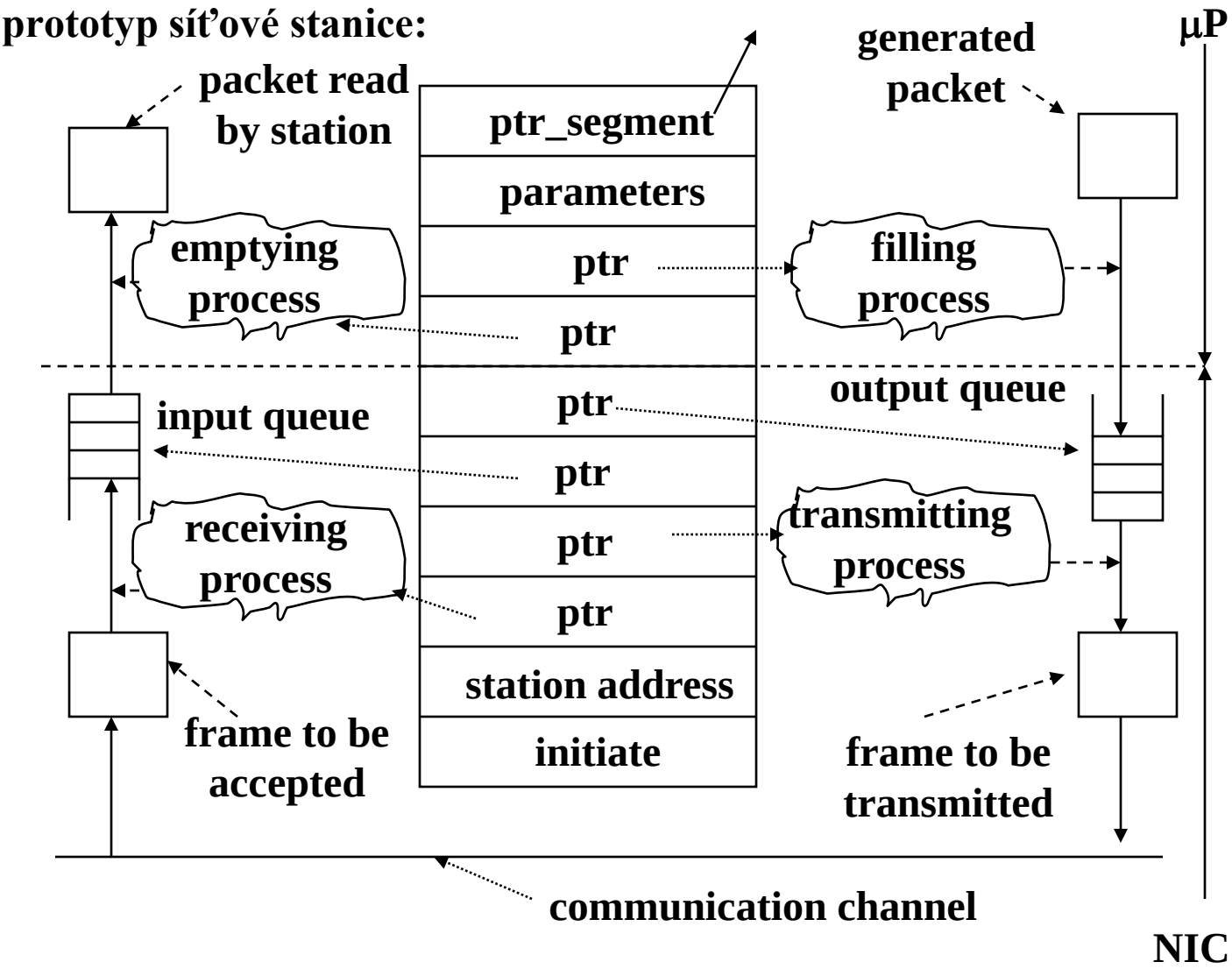
základní vrstva kvaziparalelního prostředí

vrstva prototypů pro LAN:
prototyp rámce,
prototyp stanice
prototyp síťového segmentu,
prototyp mostu

skutečný rámec:	**prototyp modelu rámce:**
- pasivní objekt :bez vlastního chování	
adresa zdroje	ano
adresa cíle	ano
datové pole	ne
délka rámce	ano
CRC	ne

stanice:	**prototyp stanice**
aktivní objekt: vykazuje vlastní chování	
• základní mikroprocesor	proces pro generování paketů proces pro zpracování paketů
• síťová karta (NIC: Network Interface Controller)	proces pro přijímání rámců proces pro vysílání rámců

Výkonnostní simulace lokálních sítí



poznámky:

ptr_segment….pointer na objekt reprezentující segment jako celek,

parameters……parametry stanice (aktivita :četnost generování paketů, charakteristika délky paketů , charakteristika provozu stanice, atd)

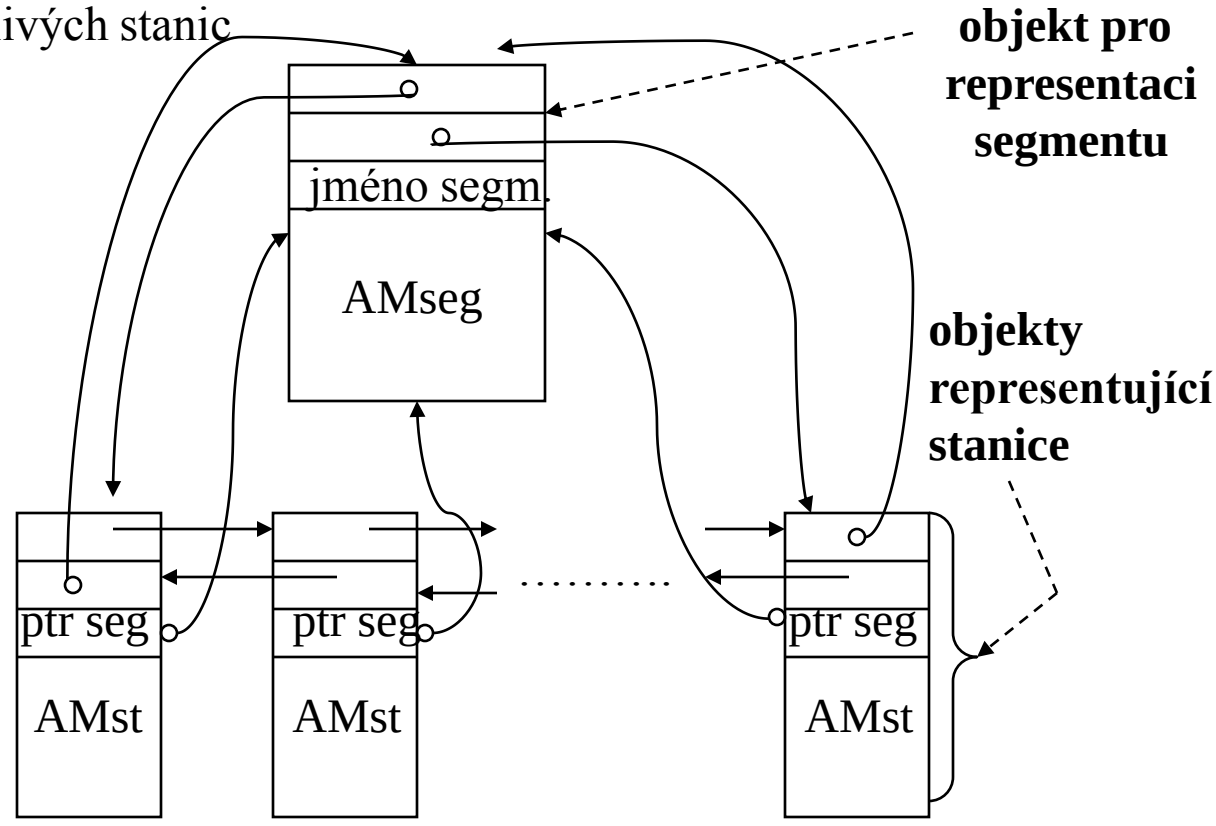
station address……adresa stanice v síti (pro směrování v mostu)

initiate……..inicializační metoda pro generování a aktivaci všech přidružených procesů a generování přidružených front

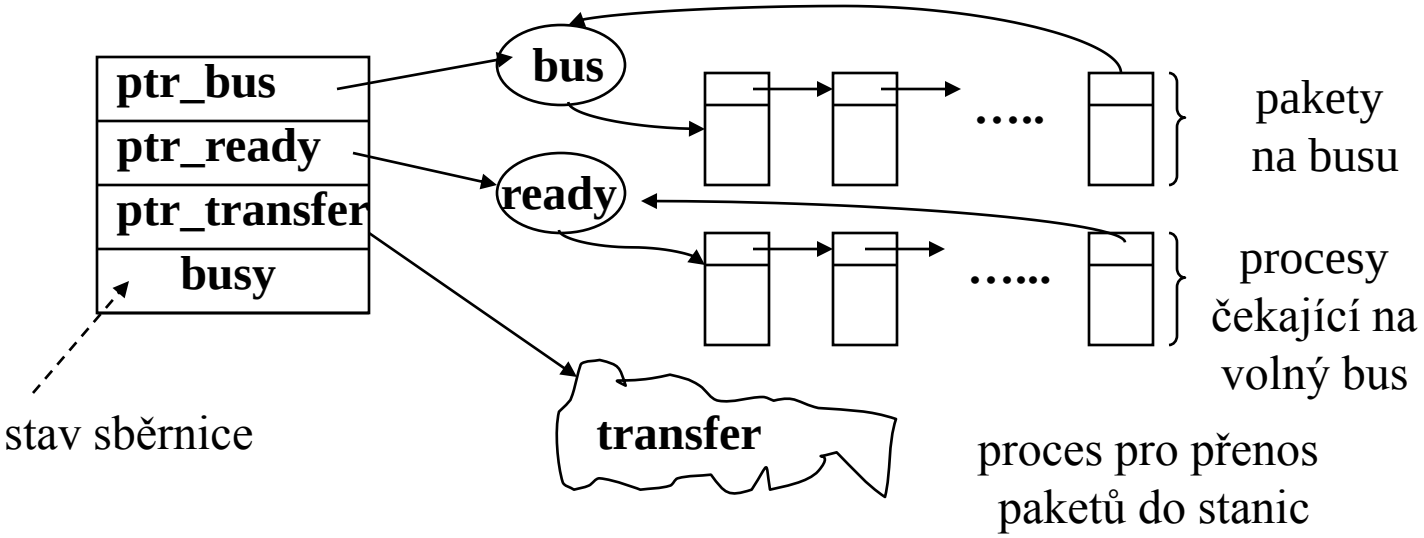
Výkonnostní simulace lokálních sítí

prototyp síťového segmentu:

cíl: umožnit šíření paketů v celém segmentu, odpojování a připojování jednotlivých stanic



AMseg ...atributy a metody segmentu sítě typu Ethernet:



Výkonnostní simulace lokálních sítí

poznámky:

bus……...seznam současně vysílaných rámců,

ready…....seznam procesů „transmitting“ těch stanic, které čekají na uvolnění sběrnice,

transfer…proces rozesílající přenášené rámce jednotlivým stanicím

Algoritmus procesu „filling“:

1) Generuj nový paket a zařaď jej do seznamu „output queue.
2) Aktivuj „transmitting process“ pro danou hodnotu model. času.
3) Čekej dokud nebude připraven nový paket.
4) Pokračuj od bodu 1)

Algoritmus procesu „receiving “:

1) Je-li přijatý rámec adresován pro danou stanici, pak jej zařaď do seznamu „input queue“ a aktivuj process “emptying”; v opačném případě jej zruš.
2) Pasivuj proces.
3) Pokračuj od bodu 1.

Algoritmus procesu „emptying“:

1) Vyjmi paket ze seznamu „ input queue“.
2) Čekej po dobu zpracování paketu procesorem.
3) Není-li seznam „input queue“ prázdný, pak pokračuj od bodu 1); v opačném případě pasivuj tento proces.
4) Pokračuj od bodu 1)

Algoritmus procesu „transfer“:

1) Předej přijatý rámec všem stanicím segmentu a aktivuj jejich procesy „receiving“ pro okamžitou hodnotu modelového času.
2) Nastav busy = false;
3) Není – li seznam „ready“ prázdný tak vyjmi všechny procesy z tohoto seznamu a aktivuj je pro okamžitou hodnotu modelového času.
4) Pasivuj tento proces.
5) Pokračuj od bodu 1).

Výkonnostní simulace lokálních sítí

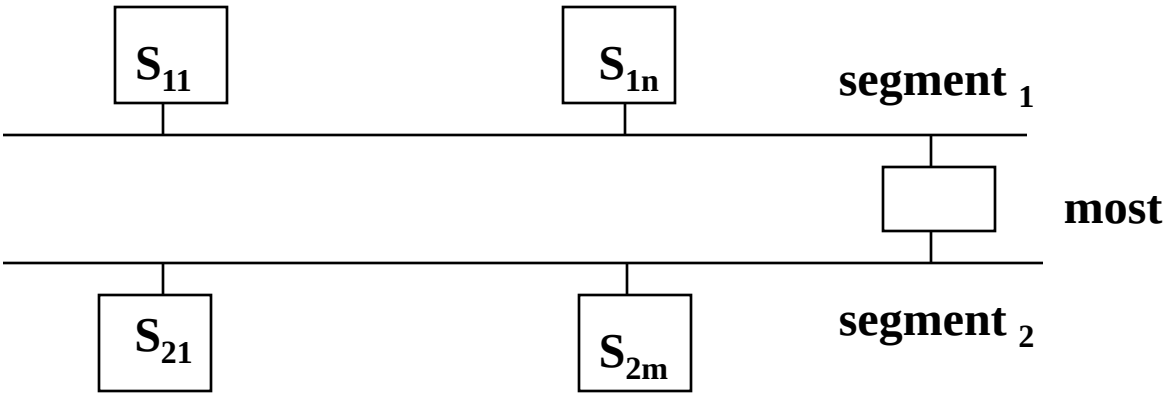
Algoritmus procesu „transmitting“:

(strategie CSMA/CD pro přístup na sběrnici)

1) Pokud je sběrnice obsazena (busy = true), zařaď právě aktivní proces do seznamu „ready“ a pasivuj tento proces.
2) Generuj kopii prvního rámce ze seznamu „output queue“ právě vysílací stanice a zařaď ji do seznamu „bus“.
3) Čekej po dobu nezbytnou pro šíření elektrického signálu podél celého segmentu.
4) Pokud je počet rámců v seznamu „bus“ roven 1, pak přejdi do bodu 8, jinak pokračuj bodem 5)
5) Čekej po dobu trvání kolizního slotu.
6) Vyjmi objekt reprezentující právě přenášený rámec ze seznamu „bus“.
7) Čekej po náhodnou dobu potřebnou pro pozdržení dalšího pokusu o získání sběrnice . Pokračuj od bodu 1).
8) Nastav atribut „busy“ = true a čekej dokud není dokončen přenos právě vysílaného paketu (závisí na délce paketu).
9) Vyjmi první rámec ze seznamu „output queue“ a předej ho procesu „transfer“.
10) Aktivuj proces „transfer“ pro okamžitou hodnotu simulovaného času a s prioritou.
11) Je-li seznam „output queue“ prázdný, pasivuj tento proces.
12) Pokračuj od bodu 1)

Výkonnostní simulace lokálních sítí

použití mostu: isolace sousedních segmentů



prototyp mostu: