# Téma 36

**Petr Husa husap1@fel.cvut.cz**

## Přesné zadání:

Sdílení prostředků, časově závislé chyby, kritické sekce procesu. Synchronizační nástroje, uváznutí - původ, detekce, prevence. Komponenty JOS pro podporu počítačových sítí, TCP/IP, směrování v sítích a Internetu, protokoly, porty.

## Sdílení prostředků – problém **souběhu** či **soupeření** (*race condition*)

– procesy používají a modifikují sdílená data
– operace zápisu musí být vzájemně výlučné
– operace zápisu musí být vzájemně výlučné s operacemi čtení
– operace čtení mohou být realizovány souběžně
– Pro zabezpečení integrity dat se používají kritické sekce

## Problém souběhu (race conditions problem) – příklad:

V mnohých operačních systémech mohou paralelně běžící procesy přistupovat ke ***sdíleným prostředkům***. Sdíleným prostředkem může být úsek v operační paměti, diskový soubor nebo jeho část, V/V zařízení či jiná téměř libovolná struktura.

## Kritická sekce

Odstranění problémů se souběhem při přístupu ke sdíleným prostředkům lze dosáhnout ***vzájemným vyloučením*** současného ***přístupu*** (mutual exclusion,mutex). Podstatou je nějakým způsobem zajistit, aby druhý proces nemohl začít pracovat se sdíleným prostředkem, dokud první proces svoji práci s ním nedokončí. Část programu, kdy se přistupuje ke sdílené struktuře, se nazývá ***kritická sekce*** procesu vzhledem k této sdílené struktuře. Nejsou-li žádné dva procesy současně ve svých kritických sekcích vůči uvažované sdílené struktuře, pak je problém souběhu vyřešen. Uvedený požadavek však nestačí pro uspokojivé řešení tohoto problému. **Musí** totiž **být *současně* splněny** následující **4 podmínky** (uvažujeme jen jednu sdílenou strukturu):

1. **Vzájemné vyloučení (Mutual Exclusion) – podmínka bezpečnosti** - žádné dva procesy nesmí být současně uvnitř kritické sekce.

2. Nesmí se klást žádné požadavky na vzájemnou relativní rychlost procesů.

3. **Trvalost postupu (Progress) – podmínka živosti** -proces běžící vně své kritické sekce nesmí způsobit zablokování jiného procesu.

4. **Konečné čekání – podmínka spravedlivosti-** čekání procesu na vstup do kritické sekce nesmí trvat nekonečně dlouho.

# Synchronizační nástroje:

## Vzájemné vyloučení s aktivním čekáním

Aktivní čekání samo o sobě je nevhodné, neboť mrhá strojovým časem, avšak může pomoci nalézt vhodné řešení.

### • Zamykací proměnné

kritickou sekci „ochráníme" sdílenou zamykací proměnnou (iniciálně = 0). Před vstupem do kritické sekce proces testuje tuto proměnnou a, je-li nulová, nastaví ji na 1 a vstoupí do
kritické sekce. Neměla-li proměnná hodnotu 0, proces čeká ve smyčce (aktivní čekání – busy waiting).
• while(lock != 0) ;
Při opouštění kritické sekce proces tuto proměnnou opět nuluje.
• lock = 0;
– Čeho jsme dosáhli? Problém souběhu je nyní na zamykací proměnné

### • Striktní střídání

– Zaveďme proměnnou *turn*, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce. Je-li *turn* = 0, do kritické sekce může *proces_0*, je-li = 1, pak *proces_1*.

```
Proces_0
while(TRUE) {
        while(turn!=0);   /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

```
Proces_1
while(TRUE) {
        while(turn!=1);   /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

– Nechť *proces_0* proběhne svojí kritickou sekcí velmi rychle, *turn* = 1 a oba procesy jsou v nekritických částech. *proces_0* je rychlý i v nekritické části a chce vstoupit do kritické sekce. Protože však *turn* = 1, bude čekat, přestože kritická sekce je volná.
– Je porušen požadavek 3 a navíc řešení závisí na rychlostech procesů

### • Petersonovo řešení

Myšlenku pravidelného střídání dovedl k funkčnosti Peterson
v r. 1981.

```
int turn;                          /* kdo je na řadě     */
int interested[2];                 /* na počátku samé 0  */

enter_region(int proc_no) /*č. proc. vstup. do k.s.*/
{
  int other;                  /* č. druhého procesu    */
  other = proc_no - 1;        /* druhý proces          */
  interested[proc_no] = TRUE; /* ukaž svůj zájem       */
  turn = proc_no;             /* nastav příznak        */
  while(turn == proc_no &&
        interested[other] == TRUE);      /* čekej     */
}

leave_region(int proc_no) /*č. proc. vystup. z k.s.*/
{
  interested[proc_no] = FALSE; /*indik. svůj nezájem*/
}
```

V poli *interested* indikují procesy svůj zájem o vstup do kritické sekce. Uspěje ten, kterému se naposledy podaří zapsat do proměnné *turn*. Proměnné nejsou kriticky sdílené!

## Synchronizace bez aktivního čekání

### • Blokování a aktivace procesů (sleep a wakeup)

Vezměme na pomoc základní systémová primitiva, která umožňují procesy blokovat místo čekání. Nejjednodušší pár je dvojice služeb ***sleep*** a ***wakeup.***

**Sleep** je služba, která způsobí, že se volající proces zablokuje (pozastaví) až do probuzení jiným procesem. Služba **wakeup** má jeden parametr, jímž je *pid* (nebo jiná identifikace) buzeného procesu.
Použití těchto systémových primitiv ilustrujeme na klasickém příkladu *producent – konzument*, známém též jako *problém konečné vyrovnávací paměti* (*bounded buffer problem*).
Jedná se o model implementace meziprocesní komunikační roury. Dva procesy mají sdílet společné pole fixního rozsahu. Producent do pole vkládá záznamy, konzument je vybírá. Potíž nastane, když producent chce vložit záznam, avšak pole je už plné. Řešením je, že v takové situaci se producent zablokuje (např. voláním sleep) a bude probuzen, když konzument odebere jeden nebo několik záznamů z pole. Podobně konzument se musí zablokovat, je-li pole prázdné a být probuzen producentem. Základní kontrola obsahu pole je pomocí proměnné count. Má-li pole kapacitu N záznamů, pak count == N značí plné pole a count == 0 pole prázdné.
Pokud tedy producent zjistí, že count < N, vloží záznam do pole a inkrementuje čítač záznamů count. V opačném případě se zablokuje. Analogicky pracuje i konzument.

Poznamenejme, že sleep a wakeup nejsou standardní funkce knihovny jazyka C, avšak jsou k dispozici ve všech systémech, které implementují takové operace v JOS.

```
#define N 100          /* Kapacita vyrovnávacího pole */
int count = 0;         /* Počet záznamů v poli        */

producer()
{
  while (TRUE) {               /* Nekonečná smyčka      */
    produce_item();            /* Vytvořit záznam       */
    if (count == N) sleep();   /* Blokovat, je-li plno  */
    put_item();                /* Vlož záznam do pole   */
    count = count+1;           /* Inkrement počtu zázn. */
    if (count == 1)            /* Bylo-li pole prázdné, */
        wakeup(consumer);      /*   probudit konzumenta */
  }
}

consumer()
{
  while (TRUE) {               /* Nekonečná smyčka      */
    if (count == 0) sleep();   /* Blokovat, nic v poli  */
    remove_item();             /* Vyjmout záznam z pole */
    count = count-1;           /* Dekrement počtu zázn. */
    if (count == N-1)      /* Bylo-li pole plné,   */
        wakeup(producer);      /*   probudit producenta */
    consume_item();            /* Zpracovat záznam      */
  }
}
```

Předešlý kód však není řešením:– Je zde konkurenční souběh – *count* je opět nechráněnou sdílenou proměnnou

• Konzument přečetl count==0 a než zavolá sleep(), je mu odňat procesor
• Producent vloží do pole položku a count=1 načež se pokusí se probudit konzumenta, který ale nespí!
• Po znovuspuštění se konzument domnívá, že pole je prázdné a volá sleep()
• Po čase producent zaplní pole a rovněž zavolá sleep() – spí oba!
– Příčinou této situace je ztráta budícího signálu
• Vytvořme tedy frontu nevyřízených signálů a sleep() způsobí blokování jen pokud nečeká žádný nevyřízený signál
• Jak má být tato fronta dlouhá?
• **Lepší řešení: Semafory**

## • Semafory

**Semafor** jako proměnná má celočíselnou hodnotu určující počet čekajících budicích signálů. Je-li hodnota 0, pak žádný signál nebyl zachycen. Na semaforu jsou definovány dvě operace (v objektové terminologii metody) **down** a **up**, kterými jsou zobecněny fce sleep a wakeup.

**Operace down** kontroluje, zda hodnota semaforu je větší než nula. Pokud ano, pak ji dekrementuje (využije jeden schovaný signál) a pokračuje. Je-li hodnota semaforu 0, pak se volající proces zablokuje. Celý trik operace down je v tom, že proběhne jako jediná *nedělitelná atomická akce*. Tak je garantováno, že jakmile operace začne, žádný jiný proces nemůže pracovat se semaforem dokud první proces svoji operaci nedokončí nebo dokud se nezablokuje.

**Operace up** kontroluje, zda "před semaforem nestojí" zablokované procesy. Pokud ne, pak up prostě inkrementuje hodnotu semaforu a končí. Jsou-li zde nějaké

blokované procesy, pak operace up vybere jeden z nich a dovolí mu dokončit jeho down akci, při níž došlo k zablokování. I operace up probíhá atomicky.
Je zřejmé, že operace down i up **musí** být akcemi jádra, neboť blokují procesy a musí zajišťovat nedělitelnost semaforových operací.

## Příklad použití semaforů:

Řešení bude používat tři semafory:
**mutex** s iniciální hodnotou 1 slouží jako zámek při přístupu ke sdílenému poli.
***full*** počítá použitá místa ve vyrovnávacím poli a má počáteční hodnotu 0,
***empty*** čítá volné pozice v poli a začíná na hodnotě N

```
#define N 100          /* Kapacita vyrovnávacího pole */
semaphore mutex=1;     /* Semafor pro přístup k poli  */
semaphore full=0;      /* Semafor pro zaplnění pole   */
semaphore empty=N;     /* Semafor pro prázdné pole    */

producer()
{
  while (TRUE) {               /* Nekonečná smyčka      */
    produce_item();            /* Vytvořit záznam       */
    down(&empty);              /* Dekrem. počet volných */
    down(&mutex);              /* Vstup do krit. sekce  */
    put_item();                /* Vlož záznam do pole   */
    up(&mutex);                /* Opuštění krit. sekce  */
    up(&full);                 /* Inkrem. počet plných  */
  }
}

consumer()
{
  while (TRUE) {               /* Nekonečná smyčka      */
    down(&full);               /* Dekrem. počet plných  */
    down(&mutex);              /* Vstup do krit. sekce  */
    remove_item();             /* Vyjmout záznam z pole */
    up(&mutex);                /* Opuštění krit. sekce  */
    up(&empty);                /* Inkrem. počet volných */
    consume_item();            /* Zpracovat záznam      */
  }
}
```

Tato implementace je již bezchybná, i když možná nadbytečně používá tři semafory. Semafory s počáteční hodnotou 1 se nazývají *binární semafory* nebo též ***mutex*****y** (***mut***ual ***ex***clusion). Jejich hlavní použití je právě pro vzájemné vyloučení při přístupu do kritické sekce.

## • Monitory

-Monitory jsou ***synchronizační nástroje vysoké úrovně***
-Monitory jsou součástí např. jazyka *Concurrent Pascal*.
- Pokud v předchozím zdrojovém kódu zaměníme pořadí obou down operací dojde k uváznutí. Zde totiž je mutex dekrementován dříve než empty. Je-li pole plné, producent se zablokuje s mutex nastaveným na 0, tedy s obsazenou kritickou sekcí. Konzument, který by uvolnil místo v poli, se však k poli nedostane, neboť sekce kritická vůči vyrovnávacímu poli je obsazena, a konzument se zablokuje též.
Výsledkem je ryzí *uváznutí*. Tomu zabráníme použitím **monitoru.**
-Monitor je vlastně zapouzdřený objekt tvořený sdílenými daty a procedurami pro manipulace s nimi s vestavěnou vlastností vzájemného vyloučení přístupu k datovým položkám objektu.

-Monitor má tzv. ***podmínkové proměnné*** (condition variables), které připomínají semafory v tom, že každá má přidruženou frontu procesů čekajících na splnění podmínky. Každá podmínková proměnná má rovněž přiřazeny operace **wait** a **signal**, analogie semaforových *down* a *up*. Když monitor zjistí, že nemůže pokračovat, provede wait na nějaké podmínkové proměnné, a tak se proces zablokuje. Pro reaktivaci blokovaného procesu se používá signal. Zde je však nutno zajistit, aby nebyly dva aktivní procesy uvnitř monitoru, a proto operace signal smí být vždy jen poslední akcí monitorové procedury (překladač to hlídá!).
-V monitoru se v jednom okamžiku může nacházet nejvýše jeden proces

## Synchronizace pomocí zasílání zpráv

Tato velmi obecná metoda meziprocesní komunikace užívá dvě primitiva **send** a **receive**, která jsou (podobně jako semafory) službami JOS. **Send** zasílá libovolnou zprávu příjemci, receive přebírá zprávu od odesilatele. Není-li zpráva k dispozici receive může způsobit zablokování, dokud zpráva nedojde.

# Časově závislé chyby:

Jsou chyby, které vznikají jen občas za náhodné souhry okolností

# Uváznutí procesů *(deadlock)*

Množina procesů je v uváznutí, když každý proces v množině čeká na událost, kterou může vyvolat jen jiný proces z této množiny. Protože všechny procesy čekají, žádný z nich nemůže způsobit žádoucí událost, a tak budou čekat navždy. Např. první proces vlastní první mechaniku a potřebuje druhou, druhý proces vlastní druhou mechaniku a potřebuje první

Coffman formuloval čtyři podmínky, které musí platit ***současně***, aby uváznutí mohlo vzniknout:

**1. *Podmínka vzájemného vyloučení***. Každý prostředek je buď přiřazen nějakému procesu (je jím vlastněn) nebo je volný.

**2. *Podmínka vlastnění a čekání.*** Procesy vlastnící nějaké dříve přidělené prostředky mohou žádat o další a při tom se zablokovat.

**3. *Podmínka neodebratelnosti prostředku.*** Prostředky jednou procesu přidělené nemohou být procesu odebrány (neplatí preempce na systémových prostředcích). Vlastnící proces musí prostředek explicitně uvolnit.
**4. *Podmínka čekacího cyklu.*** Musí existovat cyklický řetězec dvou či více procesů, z nichž každý čeká na prostředek vlastněný dalším členem řetězce.

# Detekce uváznutí a zotavení

Systém zde pouze monitoruje žádosti a uvolňování prostředků, testuje potenciální vznik cyklů v čekacím grafu (= znázorňuje, jaký proces požaduje jaký

prostředek ). Jestliže je detekováno uváznutí, systém násilně ukončí jeden z procesů. Pokud to nepomůže, ukončí další.

# Prevence uváznutí

**Přímá metoda** – plánovat procesy tak, aby nevznikl cyklus v čekacím grafu. Vzniku cyklu se brání tak, že zdroje jsou očíslovány a procesy je smějí alokovat pouze ve vzrůstajícím pořadí čísel zdrojů

**Nepřímé metody** - eliminace některé z Coffmanových podmínek:

- Eliminace potřeby **vzájemného vyloučení**
  - Nepoužívat sdílené zdroje, virtualizace (spooling) periferií
  - Bez sdílení se mnoho aktivit neobejde

– Eliminace **postupného uplatňování požadavků**
  - Proces, který požaduje nějaký zdroj, nesmí dosud žádný vlastnit
  - Všechny potřebné prostředky musí získat naráz
  - Nízké využití zdrojů

– Připustit **násilné odnímání přidělených zdrojů** (preempce zdrojů)
  - Procesu žádajícímu o další zdroj je dosud vlastněný prostředek odňat - to může být velmi riskantní – zdroj byl již zmodifikován
  - Proces je reaktivován, až když jsou všechny potřebné prostředky volné

## Závěrečné úvahy o uváznutí

• Metody popsané jako **„prevence uváznutí“** jsou velmi restriktivní
  – **ne** vzájemnému vyloučení, **ne** postupnému uplatňování požadavků, **preempce prostředků**

• Metody **„vyhýbání se uváznutí“** nemají dost apriorních informací
  – zdroje dynamicky vznikají a zanikají (např. úseky souborů)

• **Detekce uváznutí a následná obnova**
  – jsou vesměs velmi drahé – vyžadují restartování aplikací

• **Obecný závěr – problém uváznutí je efektivně neřešitelný**

• Existuje však řada algoritmů pro speciální situace

# TCP/IP Internet

**Protokoly** – určují formáty a pravidla pro zasílání zpráv po síti.
**Datagramy - i**nternet vytváří virtuální síť, po níž se přenáší tzv. **IP datagramy.**

## Směrování datagramů

Směrování (routing) je proces rozhodování o cestě, kudy poslat datagram (nebo jeho fragment).

Za směrovač se považuje libovolný stroj přijímající takové rozhodnutí.
Směrování může být **přímé** nebo **nepřímé**.
**Přímé směrování** nastává, když je cílový stroj součástí lokální sítě přímo spojené se směrovačem;
**jinak** je směrování vždy **nepřímé.**

Směrovače v Internetu tvoří kooperativní propojenou strukturu. Datagramy putují od jednoho směrovače k druhému dokud nedosáhnou směrovače, který umí zaslat datagram přímo cílovému stroji.

## 1)Nepřímé směrování

### Tabulkou řízené směrování

Každý směrovač obvykle obsahuje tzv. směrovací tabulku. Ta je tvořena dvojicemi (N, G), kde N je IP adresa cílové sítě a G je IP adresa "příštího" směrovače podél cesty k cílové síti N. **Tento směrovač musí být dosažitelný přímo.**

### I. Implicitní směry (default routes)

Velmi často jsou lokální sítě propojeny se "zbytkem Internetu" prostřednictvím jediného směrovače. Pak tento směrovač představuje pro tzv. default gateway.

### II. Specializované směry ke strojům (Host-Specific Routes)

Někdy je výhodné přiřadit jednomu nebo několika strojům speciální směrovací informaci. Důvody mohou být např. bezpečnostní

### Směrovací algoritmus:

1. Vyjmi z datagramu cílovou IP adresu *ID* a s použitím síťové masky urči *netid* cílové sítě
2. Pokud *ID* odpovídá některému spec. směru (*host-specific route*), pak pošli datagram přímo tomuto stroji
3. Pokud *netid* se shoduje s některou přímo připojenou sítí, směruj přímo
4. Pokud *netid* se nachází ve směrovací tabulce, pošli datagram odpovídajícímu směrovači
5. Pokud bylo specifikováno implicitní směrování (*default route*), pošli datagram na "*default gateway*"
6. Jinak oznam chybu směrování zasláním ICMP zprávy odesilateli

## 2) Lokální doručení datagramu (přímé směrování)

- Přímé směrování musí doručit datagram lokálně.
- Totéž se děje při předání datagramu přímo dostupnému směrovači připojenému přes LAN (nikoliv při *point-to-point* spoji) – Datagram obsahuje IP adresu, avšak doručit nutno na fyzickou adresu LAN

### Mapování IP adres na fyzické adresy

– **ARP (= *Address Resolution Protocol*)** – dynamické mapování
– Řešení Ethernet – zaslání datagramu strojem *A* stroji *B*

• Tazatel s IP adresou *Ia* a fyzickou adresou *Fa*, který potřebuje zjistit fyzickou adresu *Fb* k jemu známé IP adrese *Ib*, vyšle „ARP ethernet broadcast“ rámec, v jehož datové části bude vedle *Ia* i *Ib*. Tento rámec přijmou všechny stroje v LAN.
• Stroj, který rozpozná svoji adresu *Ib*, na tuto „všeobecnou výzvu“ odpoví a sdělí tak tazateli svoji fyzickou adresu *Fb*.
• Ethernet „broadcast“ však zatěžuje LAN, proto si tazatel získanou *Fb* jistou dobu (standardně 5 minut) pamatuje.
• Vzhledem k tomu, že se dá očekávat brzká odpověď *B* → *A*, stroj *B* získá a zapamatuje si z ARP rámce i adresy *Ia* a *Fa*.

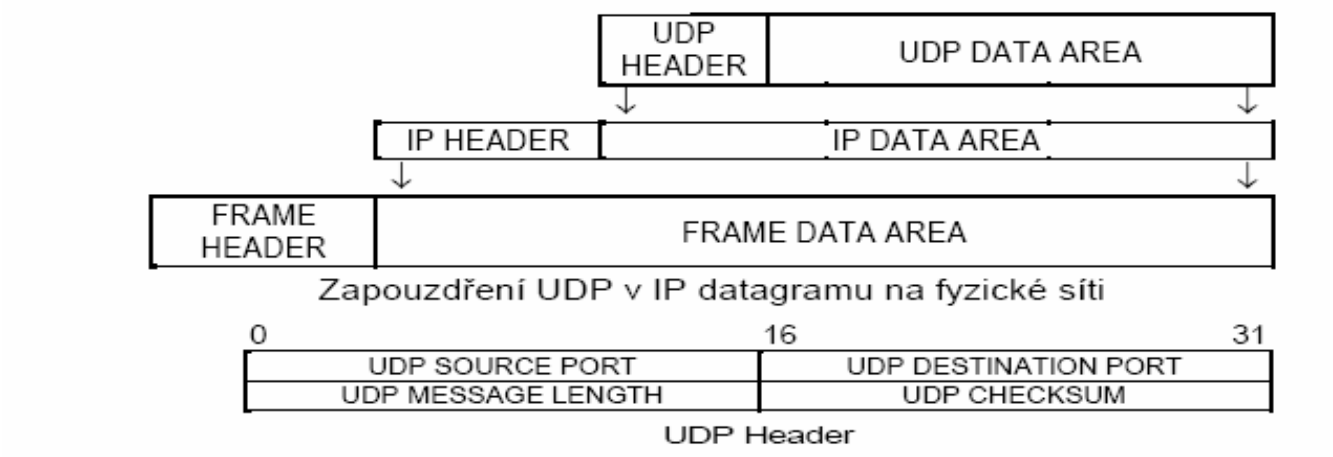
# Protokoly:

## ICMP (= *Internet Control Message Protocol*)

– Nejjednodušší protokol pro řízení a předávání chybových hlášení
– Hlavička ICMP datagramu nemá (kromě prvních 4 bytů) pevnou strukturu

## UDP (= *User Datagram Protocol*)

– jeden z jednodušších transportních protokolů.
– Poskytuje tzv. **nespojovanou** a **nezabezpečenou** službu doručování uživatelských datagramů
– Oproti ryzím IP datagramům má schopnost rozlišit mezi různými cílovými procesy na adresovaném počítači pomocí položky **port**




– Položky **PORT** se používají k rozlišení výpočetních procesů čekajících na cílovém stroji na UDP datagramy. Položka SOURCE PORT je nepovinná; není-li použita, musí být 0. Jinak označuje číslo *portu*, na nějž má být zaslána případná odpověď.

## Protokol zabezpečeného datové toku TCP

- TCP je nejdůležitější a obecná zabezpečená služba s přímým spojením mezi dvěma počítači. TCP/IP je Internetová implementace této služby.

### Vlastnosti zabezpečené dodací služby

· **Datový tok-** Aplikace komunikující po TCP/IP spoji považují komunikační kanál za tok bytů (oktetů) podobně jako UNIXový soubor.

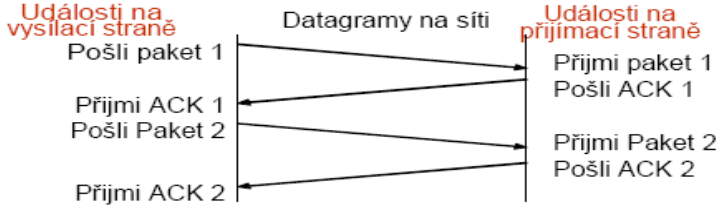
· **Virtuální spoj -** Dříve než může začít přenos dat, komunikující aplikace si musí dohodnout spojení prostřednictvím síťových komponent svých operačních systémů. -
- To se dohodne zasíláním zpráv po síti, ověří se, že spojení lze spolehlivě navázat a že oba koncové systémy jsou připraveny ke komunikaci.
- Poté jsou aplikace nformovány o ustaveném spojení a datová komunikace může být zahájena.
- Pokud se spojení během komunikace přeruší, obě strany jsou o tom informovány.
Termín
***virtuální spoj*** je požíván k vyjádření iluze, že aplikace jsou propojeny dedikovaným spojem. Spolehlivosti je dosaženo plně vázanou komunikací po spoji.

· **Přenos s vyrovnávací pamětí**
Pro zlepšení efektivity přenosu skládá protokolový modul v JOS data tak, aby se po síti posílaly pakety rozumné velikosti. Pokud to není žádoucí (např. TELNET), je TCP/IP vybaveno mechanismem, který vynutí přenos i velmi krátkého datagramu.

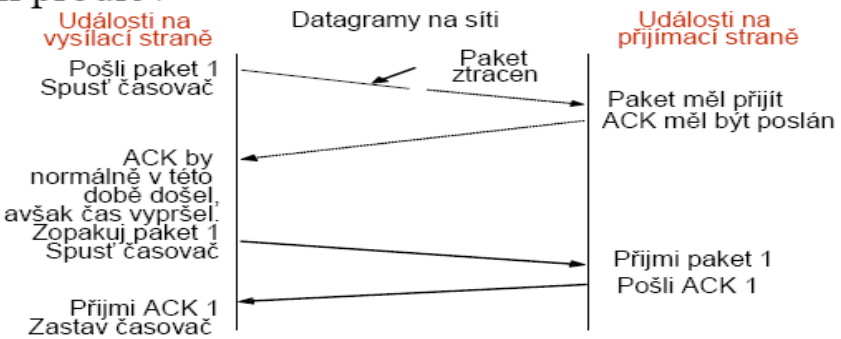
· **Plně duplexní spojení**
Aplikační procesy vidí TCP/IP spojení jako dva nezávislé datové toky běžící v opačných směrech bez zjevné interakce. Protokolový software potvrzuje data běžící v jednom směru v paketech spolu s daty posílanými ve směru opačném.

· **Porty**
Podobně jako UDP i TCP používá čísla portů k rozlišení cílové aplikace na spojených počítačích. Čísla portů pro TCP mohou být stejná jako pro UDP, neboť tyto dva protokoly jsou rozlišeny na obou koncích spoje automaticky.

- Zajištění spolehlivého přenosu
  - pozitivní potvrzování došlých dat spolu s opakováním přenosu




  - Ztracené pakety mohou být zopakovány na základě vhodných časových prodlev




## LITERATURA:

Přednášky na webu **http://labe.felk.cvut.cz/ftp/vyuka/X33OSA/**
A jejich starší verze **http://labe.felk.cvut.cz/ftp/vyuka/33OSA**