MPI_COMM_WORLD = komunikátor který obsahuje všechny procesy aplikace

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
MPI_Init(), MPI_Finalize();
```

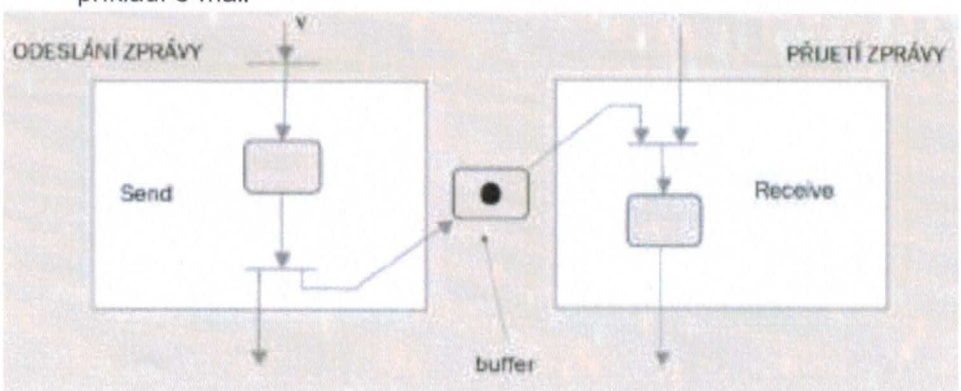
59. Distribuované a paralelní algoritmy - předávání zpráv a knihovny pro paralelní zpracování (MPI).

Existují dvě základní primitiva pro předávání zpráv:

- **Dvoubodová (Párová) komunikace**
  - **send**(channel, msg) MPI_Ssend, MPI_Send resp. MPI_Isend
  - **receive**(channel, msg) MPI_Srecv, MPI_Recv resp. MPI_Irecv
- **Kolektivní komunikace**
  - **broadcast** (vícenásobný send po více komunikačních kanálech) MPI_Bcast
  - **reduce** (vícenásobný příjem s provedením operace) MPI_Reduce

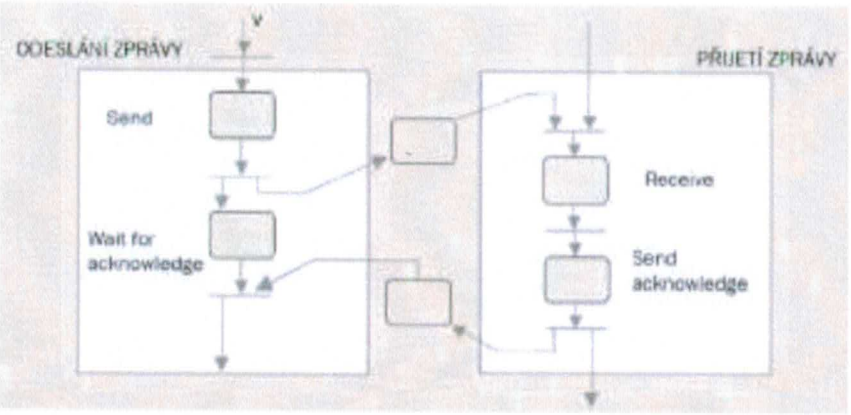
Podle **komunikačního kanálu** (použití **bufferů**) rozdělujeme operace na:

- **Asynchronní kanál**
  - neomezená kapacita
  - operace send je neblokující, tj. operace typu send nezastaví činnost procesu do přijetí zprávy adresátem
  - zamezí uváznutí z důvodu nesprávného použití blokujících komunikačních primitiv
  - složitější implementace
  - příklad: e-mail



- **Synchronní kanál**
  - omezená kapacita X nulová kapacita
  - send může být blokující
  - Podle počtu příjemců dělíme operace na:
    - jeden příjemce - kanál
    - více příjemců - mailbox

MPI_BYTE, MPI_CHAR, MPI_INT, MPI_DOUBLE



Systémy pro tvorbu paralelních programů zahrnují systémy založené na komunikaci pomocí sdílené paměti (openMP, PVM), a systémy založené na předávání zpráv (MPI).

PVM je distribuovaný operační systém. PVM běží jako démon na různých pracovních stanicích a
abstrahuje je do jednoho operačního systému. PVM umožňuje dynamickou správu procesů a prostředků a automatické zotavování se z chyb.

Message Passing Interface (MPI)

MPI je knihovna pro tvorbu paralelních programů založený na předávání zpráv. Od novější verze MPI 3 je možné komunikovat i pomocí sdílené paměti. MPI podporuje paralelismus na úrovní procesorů, později i na úrovni vláken. MPI je portabilní (spustitelný na více hostitelských OS), heterogenní (použitelná pro více architektur procesorů). Implementována primárně pro C++ a Fortran, ale i další. Odstiňuje HW od uživatele poskytováním rozhraní s knihovními funkcemi pro realizaci paralelních operací. Umožňuje komunikaci v modelech dvoubodových (P2P, Send/Receive) a kolektivních (Broadcast/Reduce).

MPI předdefinovává datové typy dat k předávaní ve zprávách (MPI_BYTE, MPI_CHAR, MPI_INT, MPI_DOUBLE, MPI_CHARACTER, MPI_COMPLEX, …), čímž odstiňuje reprezentaci datových typů v konkrétním programovém prostředí (PASCAL vs. C++ aj.). Případně je možné si vytvořit i vlastní datové typy.

Knihovní funkce

Práce s MPI musí být ohraničena funkcemi pro inicializaci a finalizaci paralelní práce.

```
MPI_Init (&argc, &argv);
        // <parallel code>.
MPI_Finalize();
```

Paralelní kód mezi těmito funkcemi už poběží na všech procesech. Omezení pro běh jen na některém z procesů pomocí zjištění vlastního ID procesu a kontroly na konkrétní ID:

Kolektivní operace (broadcast, reduce, ..) jsou blokující, dokud je neprovedou všechny procesy v uvedeném komunikátoru.

MPI_Comm_rank(MPI_COMM_WORLD, &mytid);

```
if (mytid == 3) {
   // <parallel code for process 3>.
}
```

Komunikátor

Komunikátor definuje kontext předávání zpráv. Jedná se o abstrakci skupiny procesů. Pouze procesy uvnitř komunikátoru mohou spolu komunikovat. Proces může komunikovat s jedním nebo více procesy v rámci komunikátoru.

Komunikátory lze vytvářet vlastní, pro část procesů původního komunikátoru.

Intrakomunikátor zahrnuje navzájem komunikující procesy. Interkomunikátor zahrnuje dvě skupiny procesů, které spolu mohou navzájem komunikovat (pro použití u kolektivních operací).

Konstanta MPI_COMM_WORLD zahrnuje všechny procesy aplikace.
Za běhu lze zjistit, kolik zahrnuje komunikátor procesů a který z procesů v rámci komunikátoru je daný proces (který operaci vykonává).
MPI_Comm_size(MPI_COMM_WORLD, &size) // Počet procesů.
MPI_Comm_rank(MPI_COMM_WORLD, &mytid) // Mé TID.

Komunikátor může být zdvojen (duplikován), vytvořen odstraněním některých procesů z existujícího komunikátoru, nebo vytvořen jako podmnožina skupin procesů z existujícího komunikátoru.
Příklad: Každý proces se před rozdělením obarví. To znamená, že si jednotlivé skupiny zvolí unikátní
hodnotu typu int, a tu přiřadí proměnné zvoleného jména (např int barva). Po provedení MPI_Comm_split(komunikator, barva, rank, &novy_komunikator); jsou vytvořeny samostatné komunikátory pro procesy stejných barev. Každý proces má uložen svůj nový komunikátor.

Je také možné použít uživatelem vytvořených značek (tag) pro jemnější řízení dosahu komunikace v rámci daného kontextu vytvořeného komunikátorem. Intrakomunikátor využívá jednu značku, interkomunikátor dvě (pro každou z obou skupin procesů jednu). Tagy není vhodné používat, pokud hrozí, že v rámci jednoho procesu může dojít k použití stejné značky pro různé komunikace. Naopak tagy jsou výhodné v případě, kdy neznáme rank příjemce (např. při duplikování komunikátorů, kdy dochází k přeuspořádání procesů a změně jejich ranků). Práce s komunikátorem zabere čas, vytvoření nového komunikátoru konzumuje více prostředků, než označení zprávy značkou.

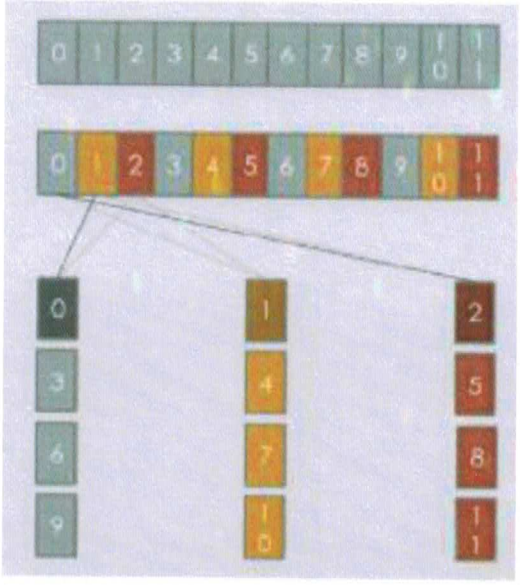
Je možné spustit nové procesy pomocí MPI_Comm_Spawn. Lze vytvořit interkomunikátor mezi skupinou již běžících procesů a skupinou procesů, které budou spuštěny. Příkaz je kolektivní a blokující, dokud všechny potomkovské procesy neprovedou inicializaci.

Pozn:
Proces může patřit do více komunikátorů současně.

Duplikace komunikátoru
MPI_Comm_dup(

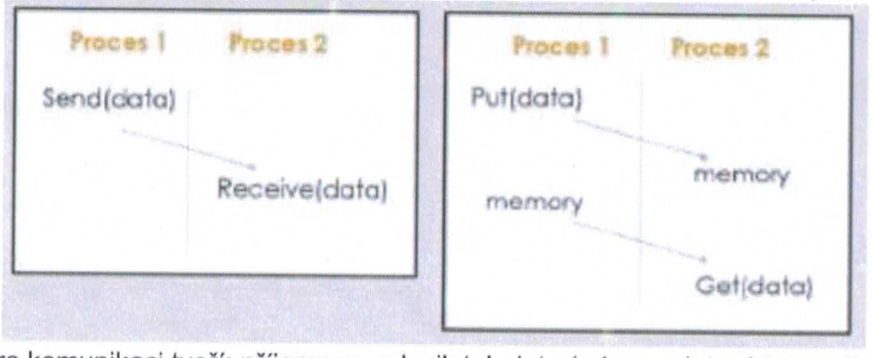
Máme skupiny procesů v komunikátorech. Tyto skupiny lze propojit Interkomunikátorem tak, že každá skupina pasuje jeden ze svých procesů za 'lídra' pro interkomunikaci a vytvoří interkomunikátor příkazem

int MPI_Intercomm_create(lokální intrakomunikátor, pověřený lídr, partnerský intrakomunikátor, lídr partnerské skupiny, tag, &výsledný interkomunikátor).



Dvoubodová komunikace

Pro dvoubodový kooperativní spoj (Point-to-point, P2P) zasílají zprávu komunikační primitiva typu Send a přijímají ji operace typu Receive.Nekooperativní komunikace probíhá uložením dat do paměti a následným výběrem dat z paměti (jednostranné operace Put a Get).



Informace pro komunikaci tvoří: příjemnce, odesilatel, data (adresa, datový typ, počet), komunikátor a značka.

Synchronní komunikace

MPI_Ssend, MPI_Srecv

- Operace jsou blokující, dokud nejsou dokončeny obě z nich.

Asynchronní komunikace

Blokující MPI_Send, MPI_Recv

- Operace je ukončena, pokud je buffer použitý ke komunikaci bezpečně k dispozici.
- int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - Přijímají se data do vyrovnávací paměti (bufferu) daného typu a daného počtu
  - A to jen od odesilatele uvedeného pozicí v komunikátoru a odpovídající značce (pomocí komunikátoru a značek oddělujeme a abstrahujeme kontext komunikace)
- int **MPI_Send**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - Zasílají se data z vyrovnávací paměti (bufferu) daného typu a daného počtu
  - Příjemce je uveden pozicí (rankem) v komunikátoru
  - Zpráva může být opatřena značkou
- Stav (v tomto případě po ukončení) komunikace lze vyčíst ze struktury status
- Výstupem je kód signalizující úspěch, nebo důvod neúspěchu operace

```
MPI_Init(&argc, &argv);

int i, j, sndrdata = 10, rcvdata = 20, sndr=1, rcvr=2, rank, data=0;
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
for(i=1; i<12; i++){
        if(rank == sndr){
                data++;
                MPI_Send(&data,1,MPI_INT,rcvr,MY_TAG,
                                        MPI_COMM_WORLD);
                printf("Sent from source %d / data %d\n",rank,data);
                  for(j=1;j<10000;j++); // delay

        }
        if(rank == rcvr){
                MPI_Recv(&data,1,MPI_INT,sndr,MY_TAG,
                                        MPI_COMM_WORLD,&status);
                printf("Received from source %d / data %d\n", data);
        }
}

MPI_Finalize();
```

Neblokující **MPI_Isend**, **MPI_Irecv**

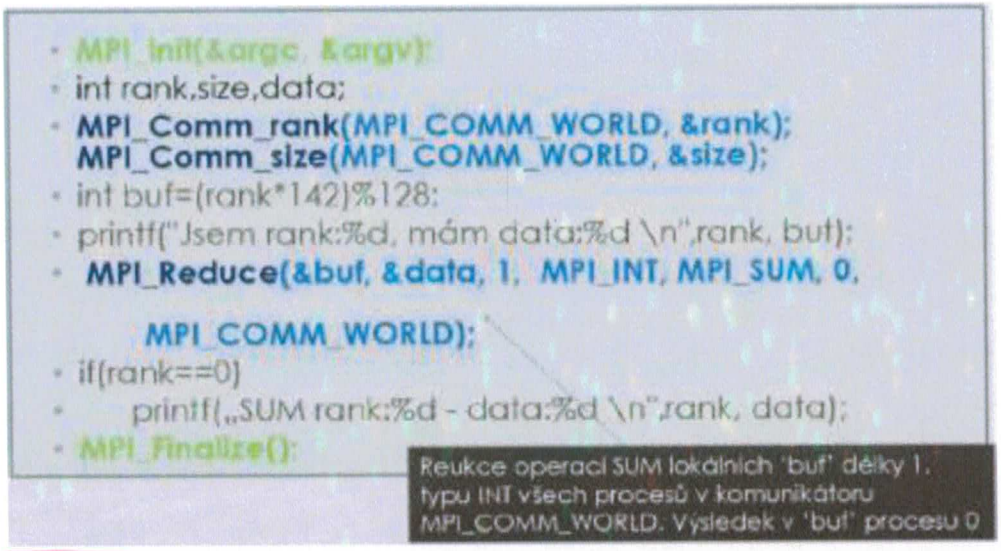
- Operace zahájí zpracování komunikace a je ukončena.
- int **MPI_Isend**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  - Položka Request identifikuje komunikaci zahájenou touto operací.
- int **MPI_IRecv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- Po zahájení asynchronní komunikace mohou procesy přizpůsobit svůj běh jejímu průběhu pomocí operací **Wait** a **Test**.
  - int **MPI_Wait**(MPI_Request *request, MPI_Status *status);

    - Operace čekání na zprávu.
    - Pozastaví činnost, než je komunikace dokončena a naplní status podle výsledku komunikace.
- int **MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status);
    - Testování stavu neblokuje činnost procesu, pouze naplní status podle výsledku komunikace.
- Operace čekání mohou být použity ve formě, kdy čekají na ukončení skupiny asynchronních operacích, daných v poli 'reqestů'. Mohou také čekat na ukončení jen jedné takové operace z uvedených
    - **MPI_Waitall**(count, array_of_requests, array_of_statuses)
    - **MPI_Waitany**(count, array_of_requests, &index, &status)
    - **MPI_Waitsome**(count, array_of_requests, array_of indices, array_of_statuses)

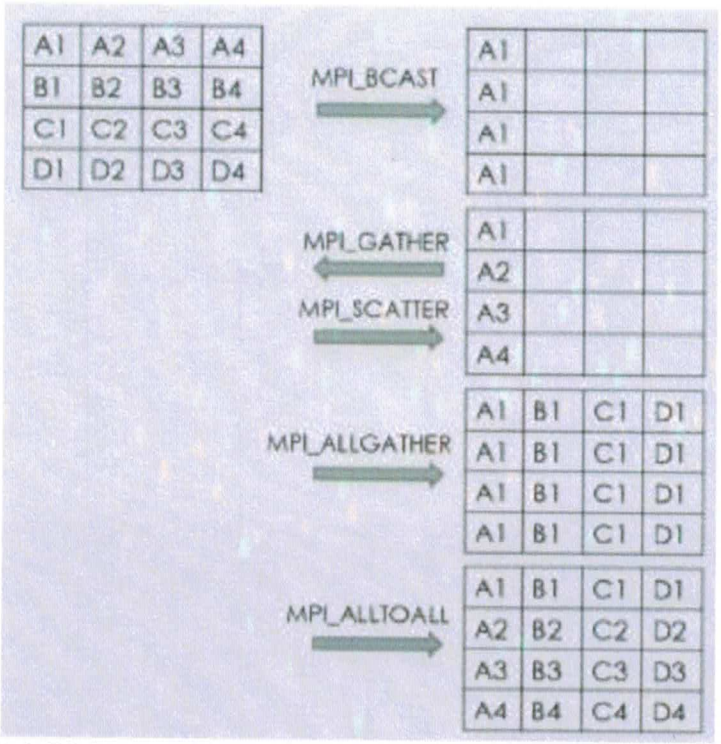
Kolektivní komunikace

Kolektivní operace jsou blokující, dokud je neprovedou všechny procesy v uvedeném komunikátoru. Stejně jako u P2P operacím je výstupem úspěch/neúspěch operace. Broadcast/Reduce model je alternativou pro P2P Send/Receive.

- int **MPI_Bcast**(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
    - Broadcast: kromě dat a obálky je třeba znát proces, který vysílá, tj. předá data všem ostatním procesům v komunikátoru.
- int **MPI_Reduce**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
    - Reduce: uvedený proces provede redukci uvedenou operací nad daty – prvky z každého procesu v komunikátoru.
    - Operace pro redukci: MPI_MAX maximum, MPI_MIN minimum, MPI_SUM sum, MPI_PROD product, MPI_LAND logical and, MPI_BAND bit-wise and, MPI_LOR logical or, MPI_BOR bit-wise or, MPI_LXOR logical xor, MPI_BXOR bit-wise xor, MPI_MAXLOC max value and location, MPI_MINLOC min value and location, …
    - Je možné definovat vlastní uživatelskou operaci pro redukci.

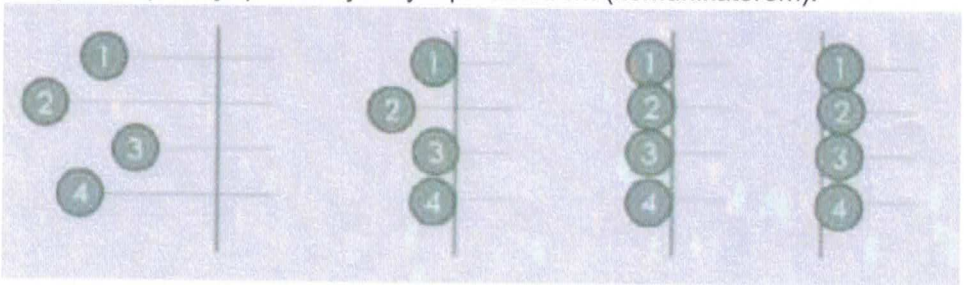


- **Gather**
  - Data jsou sesbírána a seřazena do posloupnosti ze všech procesů v komunikátoru do kořenového procesu.
- **Scatter**
  - Data z kořenového procesu jsou rozprostřena mezi ostatní procesy v komunikátoru
- **Alltoall**
  - Všechny procesy v komunikátoru rozprostřou svá data navzájem mezi sebou.
- **Scan**
  - Provede operaci Scan, každý proces dostane svoji hodnotu podle uspořádání v komunikátoru.
- Varianty většiny těchto operací **toAll** (výsledek je distribuován všem procesům), **I** (asynchronní), **V** (pro Gather, Scatter, proměnná délka distribuovaných dat u každého z procesů).



Neblokující operace, stejně jako asynchronní P2P komunikace, je provedena bez dalšího čekání.

- **MPI_IBCAST**(buffer, count, datatype, root, comm, request)
- **MPI_IREDUCE**(sendbuf, recvbuf, count, datatype, op, root, comm, request)
  - Request identifikuje započatou operaci.
  - Pro řízení běhu na základě kolektivního dokončení těchto operací se opět používají Wait a Test.
- **bariéra**
  - Bariéra je synchronizační mechanismus, který je založen na tom, že všechny procesy / část procesů musí tuto operaci provést, než mohou pokračovat.
  - Operace pracuje pouze s jediným parametrem (komunikátorem).



  - Dokonce i bariéra má svojí neblokující verzi: int **MPI_IBARRIER**(comm, request).

Na co? Je tam counter, kolik procesů mi již prošlo

60. Distribuovaný broadcast, synchronizace v distribuovaných systémech.

Distribuovaný broadcast

Máme zprávu *m* z množiny zpráv. Můžeme provést operace *b'cast(m)* a *deliver(m)*. Každá zpráva obsahuje následující položky:

- sender(m) - identita odesilatele
- seq(m) - sekvenční číslo zprávy zaslané daným procesem

Nekorektní procesy jsou takové procesy, které mohou po spuštění selhat. Rozlišujeme následující typy:

- Crash & stop - po selhání procesu se zastaví
- Crash & recovery - po nějaké době může dojít k obnovení správné činnosti procesu
- Byzantine - byzantské procesy, pro které není chování po selhání definováno

Příklady požadované vlastností v distribuovaných systémech:

- Živost (liveness): chceme, aby dříve či později nastala odpověď na požadavek
- Bezpečnost (safety): chceme, aby nenastala chyba v doručení (špatné pořadí zpráv)

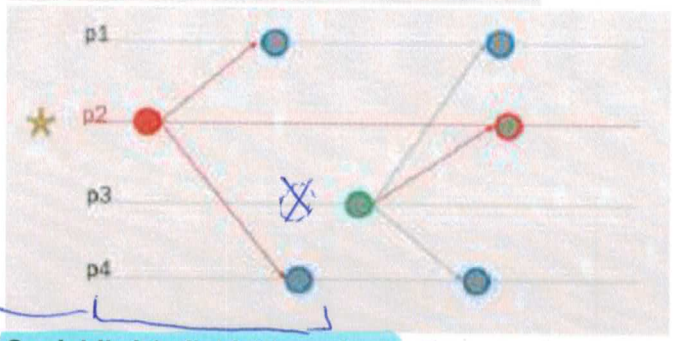
Vlastnosti spolehlivého broadcastu:

- **Platnost** (validity): pokud zpráva byla vyslána nějakým korektním procesem, pak ji dřív nebo později každý korektní proces přijme
- **Shoda** (agreement): pokud byla zpráva přijata korektním procesem, pak bude dřív nebo později přijata každým korektním procesem
- **Integrita** (no duplication): žádná zpráva není doručena více než jednou
- **Opravdovost** (no creation): pokud proces doručil zprávu m od procesu p, pak ji tento proces p opravdu odeslal

Klasifikace všesměrového vysílání:

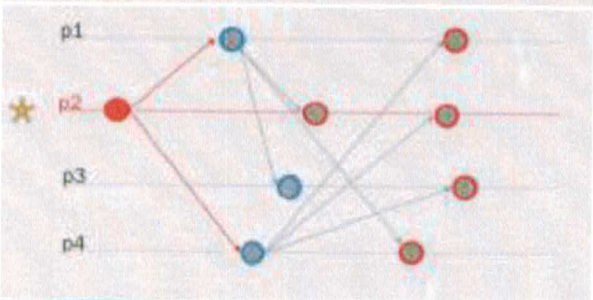
Best effort broadcast: je zde garantována platnost, ale ne shoda. Může to nastat tehdy, pokud v systému existuje nějaký nekorektní proces (na obrázku p2). Pokud je zpráva poslána nekorektním procesem a přijata nějakým korektním (ale ne všemi), je narušena shoda.

```
b'cast(m):
        Tag m with sender(m) and seq(m)
        send(m) to all neighbors including self

deliver(m):
        upon receive(m) do
                deliver(R, m)
        enddo
```



Spolehlivý (reliable) broadcast: Procesy přeposílají obdržené zprávy, je zde garantována platnost i shoda. Pokud přijímající proces vidí, že ne všichni sousedi (kromě odesílatele) zprávu m obdrželi, rozesílá zprávu broadcastem všem svým sousedům (včetně odesílatele).

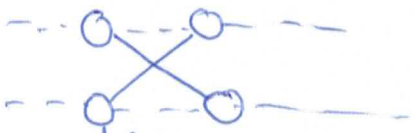
```
b'cast(R, m):
        Tag m with sender(m) and seq(m)
        send(m) to all neighbors including self

deliver(R, m):
        upon receive(m) do
                if p has not previously executed deliver(R, m) then
                        if sender(m) != p then send(m) to all neighbours
                        deliver(R, m)
                endif
        enddo
```



FIFO broadcast: Spolehlivý broadcast, pro který platí, že pokud nějaký proces vyslal zprávy v určitém pořadí, je toto pořadí dodrženo i při doručování zpráv procesům.

Kauzální broadcast: Spolehlivý broadcast, pro který platí, že pokud jeden proces obdrží zprávu *ma* a pak vyšle zprávu *mb*, tak všechny procesy musí doručit *ma* před *mb*, nezáleží ale, v jakém pořadí doručování zpráv před doručením *mb* probíhá.

Pozn. - program napsaný pro systém s asynchronní komunikací je proveditelný systémem se synchronní komunikací, pokud v něm neexistuje koruna



send(m_1) →$_2$ receive(m_2)
send(m_2) →$_2$ (receive(m_1))

Relace →$_a$ kauzálně předchází, když
- $a \rightarrow_a b$ pokud jeden process vykonal tyto události v tomto pořadí
- broadcast(ma) →$_a$ deliver (ma)
- Tranzitivita

Atomický broadcast (ABCAST): Spolehlivý broadcast, pro který platí, že jsou všechny zprávy doručeny všemi procesy ve stejném pořadí. (úplné uspořádání)

Distributed Systems 4.2: Broadcast ordering

Synchronizace v distribuovaných systémech

Synchronizace zaručuje (částečné) uspořádání mezi událostmi. U systémů se sdílenou pamětí můžeme použít synchronizační mechanismy jako semafory nebo monitory. Synchronizace distribuovaných systémů, které nekomunikují sdílenou pamětí, ale pouze předáváním zpráv, je obtížnější. V distribuovaných systémech nemáme k dispozici globální hodiny. Pro synchronizaci máme dva požadavky:

- **kauzalita**: uspořádání událostí dle logických hodin nebo časových razítek (korektní chování z pohledu uživatele), a
- všechny procesy uspořádávají události v tom samém pořadí.

Budeme se zabývat následujícími typy synchronizace: synchronizace globálním (reálným) časem, synchronizace řízená master uzlem a synchronizace založená na shodě mezi procesy.

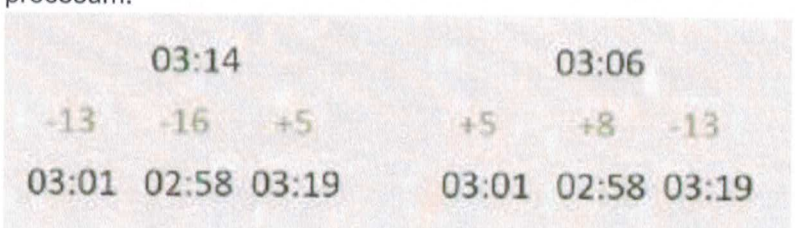
Synchronizace globálním (reálným) časem

předpokládá, že komunikace je bez zpoždění

Berkley algoritmus – asi nejprimitivnější, úplně se nepoužívá

Jedná se o jednoduchý algoritmus pro synchronizaci fyzického času. Algoritmus předpokládá, že zpoždění v komunikaci mezi procesy je dostatečně krátké a lze jej zanedbat. Hlavní uzel si vyžádá od všech ostatních uzlů hodnotu posunu vůči svému aktuálnímu času. Následně vypočte průměrnou hodnotu posunu a o tuto hodnotu posune svoje hodiny. Z průměrné hodnoty posunu také vypočte posuny pro jednotlivé uzly.

V příkladu níže získá hlavní uzel hodnoty posunů od všech ostatních uzlů. Průměrný posun je (-13-16+5)/3=-8. Posune tedy svoje vlastní hodiny o -8 a upraví i hodnoty posunů vůči ostatním procesům.



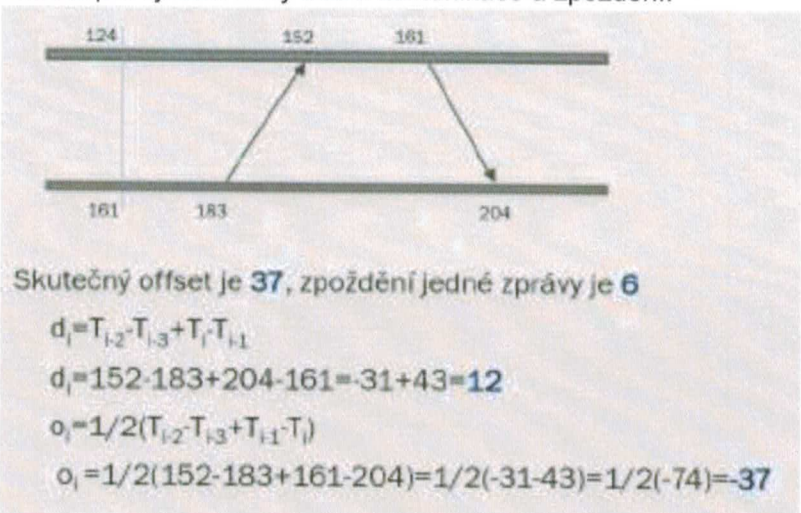
- zprůměrují se hodiny všech ostatních uzlů, to se prohlásí za aktual. čas a ten se nastaví jak na hlavním uzlu tak i na všech ostatních

Network Time Protocol (NTP)

Jde o protokol, který zajišťuje synchronizaci zařízení v síti na různých úrovních. Komunikace probíhá přes UDP. Má různé módy synchronizace:

- **Multicast**: opakované vysílání aktuálního času v rámci skupiny. Tento přístup je vhodný pro malé sítě s vysokou rychlostí přenosu dat
- **Klientský přístup**: volání procedury na serveru klientem, použitelné v případech, kdy není možné použít multicast
- **Párový přístup**: synchronizace s velkou přesností

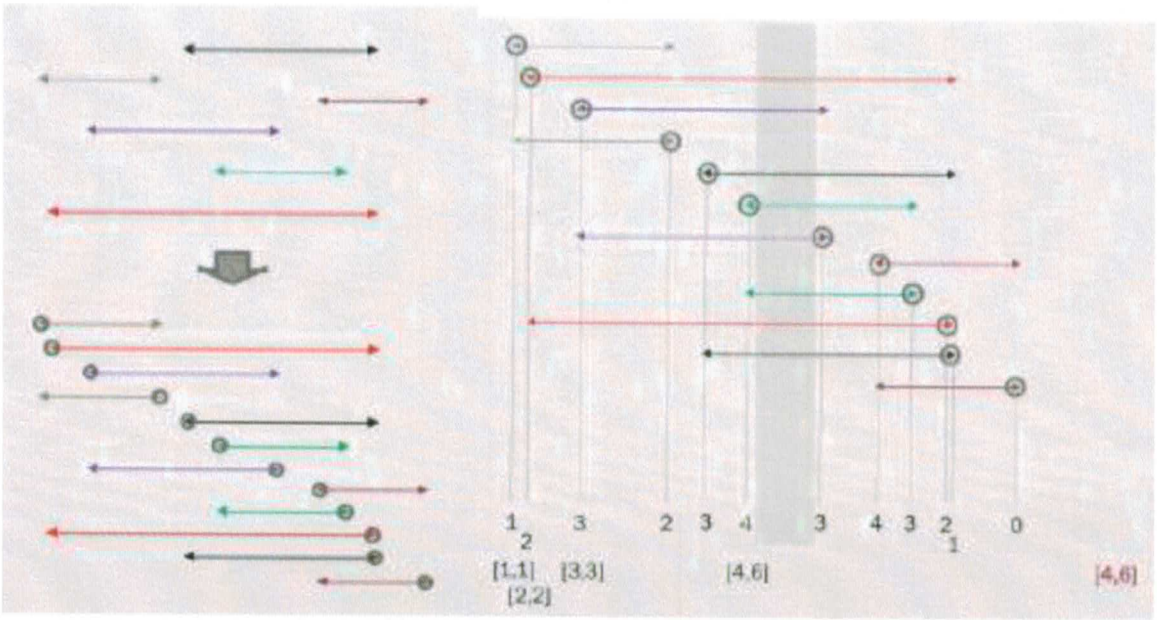
Příklad pro zjištění doby trvání komunikace a zpoždění:



Marzullo-ův algoritmus

Výše uvedeným způsobem může jeden proces upravit svůj čas vzhledem k některému z NTP serverů. Pokud máme ale několik NTP serverů, které udávají různé časy, používá se Marzullo-ův algoritmus. Předpokládaný sdělený čas od serveru je dán nějakým intervalem. Máme množinu takovýchto intervalů. Účelem algoritmu je z množiny intervalů vybrat takový časový úsek, který spadá do co nejvíce intervalů.
Všechny intervaly se uspořádají podle toho, kdy v nich nastala počáteční nebo koncová událost. Každý interval bude v uspořádání dvakrát (jednou pro počáteční událost a podruhé pro koncovou událost). Následně se spočítají průniky intervalů a interval, kde je nejvíc průniků, je výsledek. Pokud je více intervalů se stejným množstvím průniků, vybírám ten nejkratší.

Uspořádání a výběr intervalu. Čísla dole znázorňují počet průniků v následujícím intervalu.



Synchronizace řízená master uzlem

Algoritmy pro volbu hlavního uzlu nejsou užitečné jen pro synchronizaci, ale obecně pro řešení konfliktů nebo spolupráci procesů, což zahrnuje obecné problémy paralelismu.

Volba master uzlu pro obecnou topologii

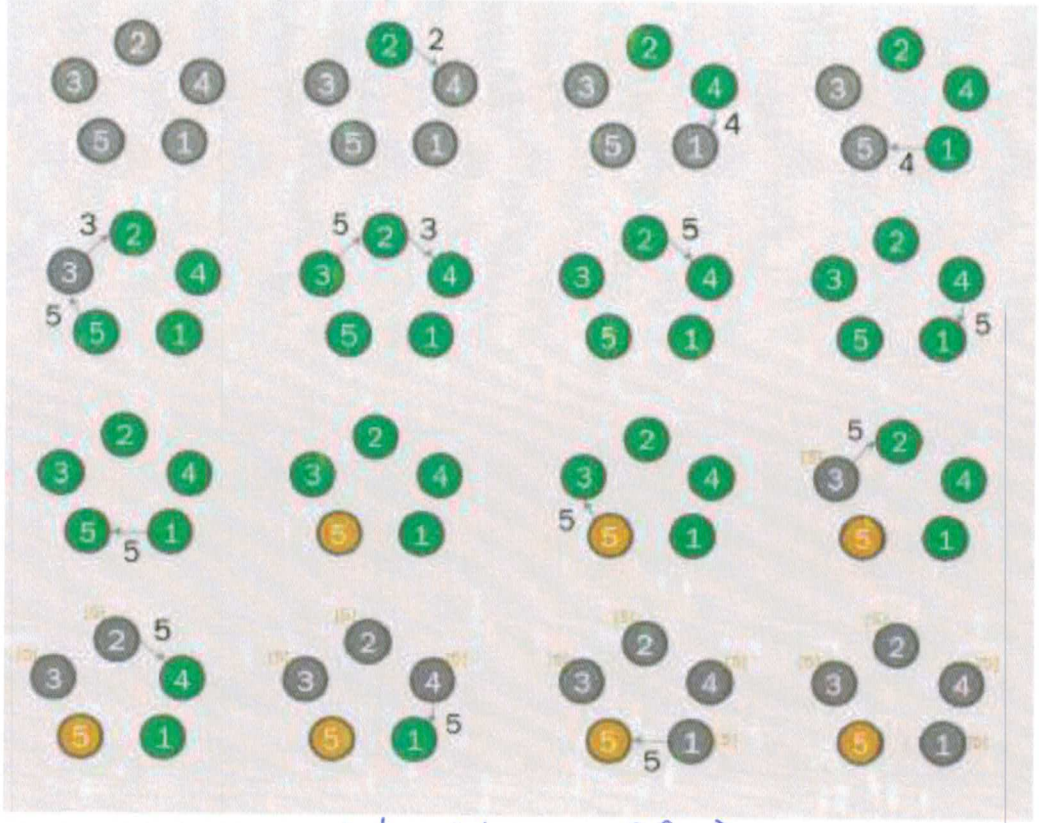
Ustaví se graf, například metodou BFS. Od zvoleného uzlu se hledají sousedé jako uzly další úrovně, pokud již nejsou součástí stromu. Extrahujeme největší UID ze stromu (například algoritmum MinExtractionSort). Toto UID propagujeme všem uzlům do stromu. Složitost je O(m + nlogn), kde m je počet hran a n je počet uzlů.

Algoritmus Chang and Roberts

Procesy jsou propojeny v kruhové topologii a každý má přiděleno unikátní číslo UID. Algoritmus hledá maximální hodnotu ze všech hodnot těchto uzlů. Zprávy jsou posílány po směru hodinových ručiček. Algoritmus probíhá následujícím způsobem:

1. Uzel zahájí komunikaci, označí se za účastníka a pošle zprávu se svým UID následujícímu uzlu
2. Pokud uzel přeposílá zprávu, označí se za účastníka
3. Každý uzel po obdržení zprávy provede následující:
   Pokud UID ve zprávě je větší než jeho UID, je zpráva přeposlána dále tak, jak je
   - Pokud je číslo ve zprávě menší než jeho UID, potom:
     i. Pokud je již účastníkem, tak zprávu zahodí
     ii. Pokud není účastníkem, nahradí původní hodnotu za svoje UID a přepošle zprávu dál
   - Pokud je číslo ve zprávě stejné jako jeho UID, potom uzel volbu vyhrál a zahájí druhou část algoritmu (rozeslání UID vítěze)
4. Vítěz volby se odznačí jako účastník a pošle svoje číslo dále

5. Každý, kdo obdrží zprávu a je stále účastníkem, si číslo poznačí, odznačí se jako účastník a pošle zprávu dále
6. Pokud zprávu obdrží vítěz volby, zprávu zahodí

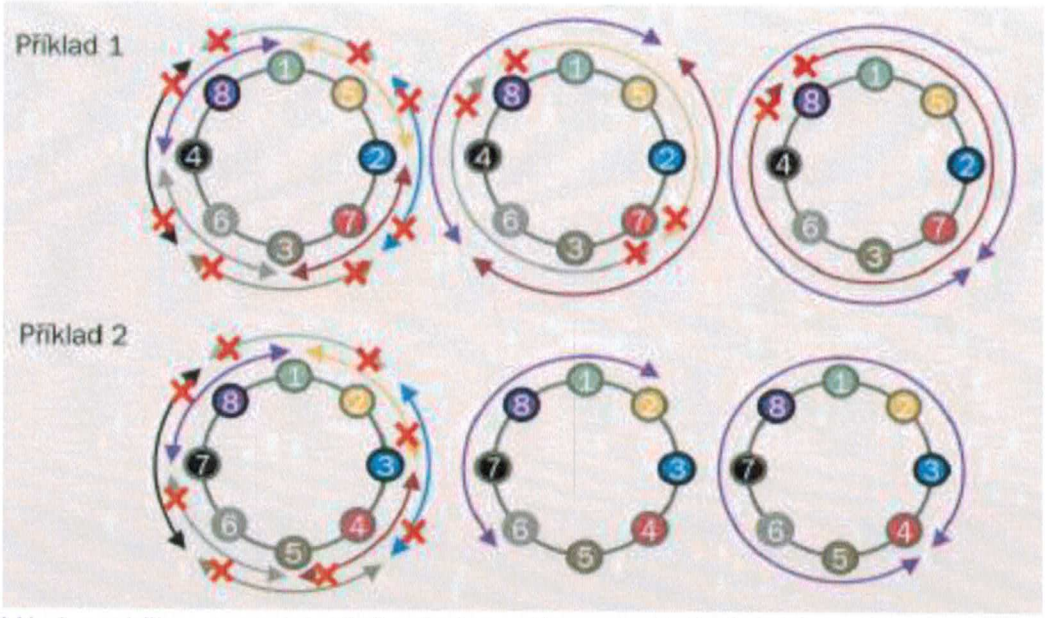


Analýza algoritmu: - počet zpráv až $O(n^2)$

- Nejlepší případ: hlasování zahájí pouze uzel s nejvyšším UID, pak proběhne 2n zpráv (n pro zjištění vítěze a n pro rozhlášení výsledku)
- Nejhorší případ: uzel s maximálním indexem je první za iniciátorem hlasování a každý uzel zahájí hlasování před odesláním první zprávy některým z uzlů - až 3(n-1) zpráv pro jeden uzel, celkově je možné pracovat až s $O(n^2)$ zprávami (uzly od největšího po nejmenší ve směru posílání zpráv, každý uzel inicializuje volbu)

Algoritmus Hirschberg-Sinclair - zlepšení, počet zpráv $O(n \log n)$

Procesy jsou propojeny v kruhové topologii a každý má přiděleno unikátní číslo UID. Algoritmus hledá maximální hodnotu. Uzel pošle zprávu levému a pravému sousedovi. Pokud jeho UID není největší, je zpráva zahozena. Pokud je největší, posílá se o jednoho souseda dál oběma směry.



V jednom běhu usne minimálně polovina uzlů, maximálně všichni kromě jednoho. Jediný uzel zůstane zaručeně po $\log_2 n$ bězích. V posledním běhu bude zasláno 4n zpráv. Počet zpráv i časová složitost je O(n*log n).

Synchronizace logickým časem

Následující algoritmy zabezpečují synchronizaci procesů při snaze o přístup do kritické sekce, jedná se tedy o algoritmy pro vzájemné vyloučení procesů. Obecně existují dva druhy algoritmů pro zajištění vzájemného vyloučení:

- Algoritmy založené na **časových razítcích** (Lamportův algoritmus, algoritmus Ricard-Agrawala, Meakawův algoritmus)
- Algoritmy založené na **tokenech** (Suzuki-Kasami vysílací algoritmus, Raymondův stromový algoritmus)

Lamportův algoritmus

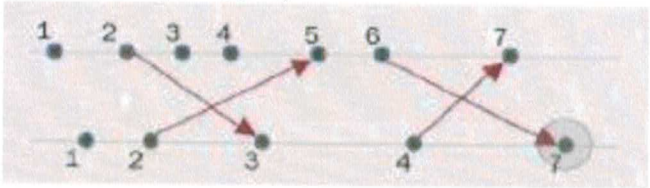
Logické hodiny jsou funkce, které pro nějakou událost v procesu zobrazí logický čas odpovídající této události. Každý proces *i* má jedny logické hodiny *Ci*. Logické hodiny před každou událostí *e* zvyšují čítač (monotónně, např. o 1). *C(e)* zobrazuje pro každou událost odpovídající logický čas. Důležitou součástí algoritmu je relace *happened before*. Při uspořádávání událostí bereme v potaz jednak pořadí událostí, které se odehrály v rámci jednoho procesu, a taky pořadí událostí mezi procesy, které je dáno tím, že odeslání zprávy musí předcházet jejímu přijetí jiným procesem. Formálně je relace *happened-before* definována následovně:

R(e1,e2) iff e1 předchází e2 v rámci jednoho procesu

e1 je send(p2,m,ix) v procesu p1 a e2 je receive(p1,m,ix) v procesu p2

R(e1,e3) a R(e3,e2) // tranzitivita

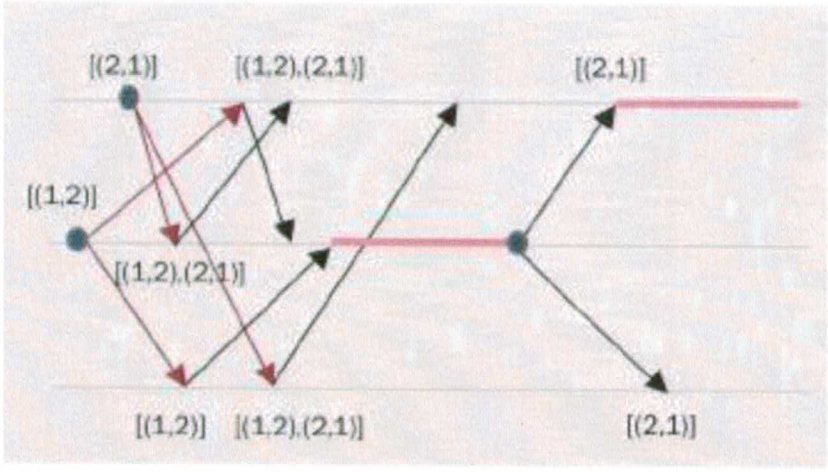
Implementace logických hodin: spolu se zprávou se posílá i časové razítko dle logického času vysílacího procesu, při přijetí zprávy *rec(m,p,tp)* příjemce *i* nastaví/aktualizuje svůj logický čas na *Ci := max(Ci+1,tp+1)*.



Happens-before je nereflexivní relace částečného uspořádání. Abychom dosáhli úplného uspořádání, je třeba dodat další informaci, zde lze použít ID procesů. Pak procesy s nižším číslem mají přednost a jejich události, pro které nemůžeme uspořádání původně určit, předcházejí událostem procesů s vyšším číslem.

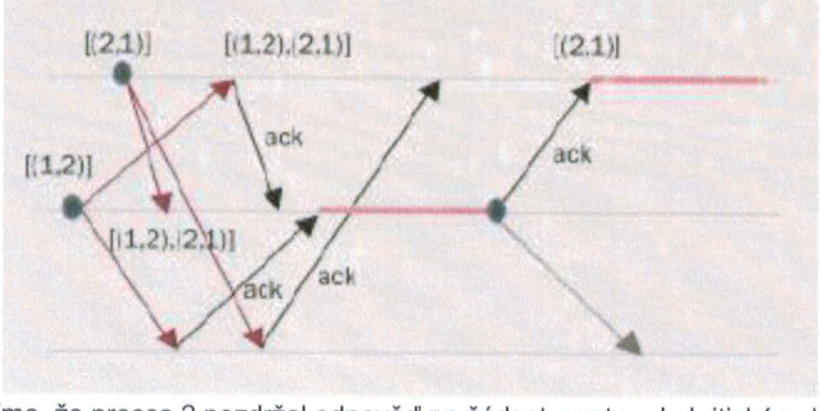
Vzájemné vyloučení v distribuovaných systémech s předáváním zpráv je podle Lamportova algoritmu realizováno následovně:

1. Pokud chce proces vstoupit do kritické sekce, odešle žádost opatřenou svým aktuálním logickým časem (časovým razítkem) všem procesům a čeká na reakce také všech procesů. Zapamatuje si, že takovou žádost poslal, včetně časového razítka. *(červené šipky na obrázku)*
2. Pokud proces dostane žádost o vstup od jiného procesu, zapamatuje si tuto žádost včetně časového razítka, zařadí ji do seznamu žádostí uspořádaného podle razítek, nebo v případě shodných časových razítek podle identifikátorů žadatelů. Pokud doposud žádný nemá, vytvoří jej s touto žádostí. Pak odešle tento seznam jako odpověď žadateli.
3. Pokud žadatel obdrží odpověď od všech procesů, zjistí, zda-li se ve všech odpovědích nachází na prvním místě seznamu. Pokud ano, vstoupí do kritické sekce. Pokud ne, čeká. *(nejprve vstupuje do kritické sekce proces 2, znázorněno růžovou čárou)*
4. V okamžiku, kdy proces vystupuje z kritické sekce, odešle o tom zprávu ostatním procesům.
5. Pokud proces obdrží zprávu o tom, že nějaký proces opustil kritickou sekci, odstraní si tento proces ze svého seznamu žadajících procesů, resp. ze všech seznamů žadatelů, pokud je sám žádající a obdržel tyto seznamy od ostatních procesů. Pokud je takový proces žadatelem a po odstranění takového procesu nyní ví, že je ve všech těchto seznamech na prvních místech, vstoupí do kritické sekce.



Algoritmus Ricart-Agrawala

Jedná se o optimalizaci Lamportova algoritmu. Lamportův algoritmus vyžaduje 3(n-1) zpráv: (n-1) požadavků, (n-1) odpovědí a (n-1) uvolnění. Algoritmus Ricart-Agrawala naproti tomu vyžaduje pouze 2(n-1) zpráv, protože slučuje dohromady zprávy odpovědi a uvolnění. Myšlenka spočívá v tom, že pokud proces dostal žádost o vstup do kritické sekce a sám je žadatelem, počká s odpovědí, pokud sezná, že jeho žádosti dají ostatní procesy přednost. Tedy že časové razítko, které v žádosti uvedl, je menší, nebo v případě shody jeho identifikátor má přednost, a tak žádající proces musí být vpuštěn do kritické sekce až po něm. Pamatuje si však příchozí žádosti a ty ve formě uspořádaného seznamu odešle ostatním procesům včetně žadatelů až v okamžiku, kdy sám opustil kritickou sekci a již se v tomto seznamu nenachází.

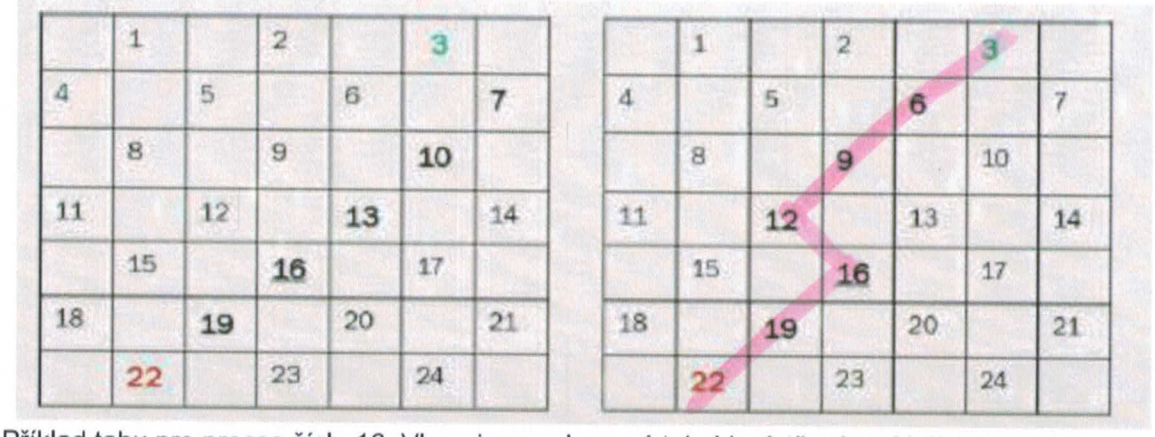


V příkladu vidíme, že proces 2 pozdržel odpověď na žádost o vstup do kritické sekce procesu 1, protože má menší časové razítko a tedy si může být jistý, že proces 1 bude vstupovat do kritické sekce až po něm (a proces 1 tedy musí čekat na doručení odpovědi na jeho žádost od procesu 2). Po výstupu z kritické sekce rozešle odpověď společně s uvolněním.

Meakawův algoritmus

Povolení ke vstupu do kritické sekce stačí získat jen od nějaké podmnožiny procesů. Zavedou se tzv. kvóra. Každý proces má svoje kvórum a každá dvě kvóra sdílejí alespoň jeden prvek (tedy žádná dvě nejsou disjunktní). Pro vytvoření kvór se používá billiardová metoda. Máme

čtvercovou matici procesů a pro daný proces vytvoříme kvórum tak, že vykonáme tah billiardu a posbíráme procesy v tomto tahu. Abychom zamezili tomu, aby všechna čísla na trase měla tu samou posloupnost, používáme zalomenou variantu. Tah provádíme vždy v tom samém směru, například směrem doprava a nahoru.



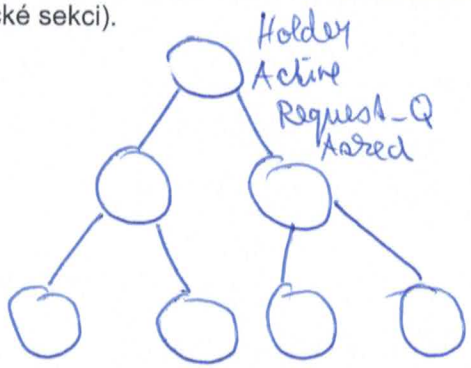
Příklad tahu pro proces číslo 16. Vlevo je nezalomený tah. V místě odpovídajícímu číslu procesu, pro který hledáme kvórum, zalomíme na opačnou stranu. Potom zalomíme zpět tak, aby cílové pole bylo stejné jako v nezalomeném případě.

Vstup do kritické sekce (základní verze s možností deadlocku):
Každý proces *i* má svou množinu *Ri*. Pokud chce proces *i* vstoupit do kritické sekce, pošle žádost o povolení vstupu všem procesům ze své množiny *Ri*. Do kritické sekce vstoupí, až když dostane garanci od každého z těchto obeslaných procesů. Pokud proces dostane žádost o vstup do kritické sekce, potvrdí ji v případě, že nevydal svolení ke vstupu jinému procesu a ten dosud neoznámil, že již kritickou sekci absolvoval. Pokud takové svolení vydal, oznámí žadateli neúspěch a zařadí si jej do fronty žádajících procesů do té doby, než tento proces nebude na jejím čele. Jakmile proces vystoupí z kritické sekce, informuje o tom všechny procesy ze své množiny *Ri*. Ty si odstraní daný proces ze své fronty a vydají povolení ke vstupu do kritické sekce procesu, který je aktuálně na čele této fronty. V této variantě je možné, že dojde k uváznutí.

Vstup do kritické sekce (varianta bez uváznutí):
Algoritmus je upraven tak, aby se procesy, které dostaly jen část potvrzení od své skupiny procesů, v případě žádosti o pozdržení vzdaly snahy vstoupit do kritické sekce, dokud nedostanou dostatečný grant.

Raymondův algoritmus

Algoritmus založený na tokenech. Jeden token reprezentuje v systému poukázku na kritickou sekci. Proces může vstoupit do kritické sekce, pokud obdrží token. Důkaz o vzájemném vyloučení procesu je triviální (token může držet jen jeden proces a ten jej předává, pokud není v kritické sekci).



- stromová struktura
- předáváme si token, jen držitel tokenu může vstoupit do kritické sekce

Algoritmus předpokládá propojení procesů ve stromové topologii. Pokud chce proces vstoupit do kritické sekce, vytvoří agenta a ten se snaží získat token a dopravit jej k žádajícímu procesu. Aby toho byl schopen, využívá následujících znalostí:

1. Původně je token umístěn v procesu, který je kořenem této topologie
2. Pokud nějaký agent nesl token (vždy směrem k procesu, který jej vytvořil), zanechá v uzlech, kterými prošel, stopu, kterým směrem se z tohoto uzlu pohyboval. Každý uzel/proces si uchovává informaci pouze o posledním z agentů, kteří jím s tokenem prošli.

Hledání tokenu agentem je pak založeno na následujících pravidlech:

1. Pokud je agent v uzlu, ve kterém není uchována informace o směru pohybu jiného agenta s tokenem, pak tímto tokenem takový agent nikdy neprošel a aktuální agent se pohne směrem ke kořenovému uzlu.
2. Pokud v uzlu je taková informace, pohne se tímto směrem.
3. Pokud je token přímo na tomto uzlu a je volný, vezme jej a vydá se směrem ke svému domovskému uzlu. Předpokládáme, že si svoji cestu od domovského uzlu zapamatoval a dokáže se dostat zpět.
4. Pokud je token na uzlu držen jiným agentem, musí počkat (a v dalším okamžiku opakovat krok 2 nebo 3).
5. Pokud agent doručí token do domovského uzlu/procesu, tento proces může vstoupit do kritické sekce a po jejím opuštění token uvolní, ale i tak token zůstává na tomto uzlu, dokud jej nesebere jiný agent.

V algoritmu může dojít k vyhladovění.

Suzuki-Kasami vysílací algoritmus

- dále pracuje s kořeny
- plně propojená síť

Tento algoritmus pracuje s vektorem hodnot, jehož rozměr odpovídá počtu procesů. Takový vektor je držen jak tokenen, tak každým z procesů. U tokenu jednotlivé hodnoty udávají, kolikrát byl tento token kterému procesu přidělen. U procesů hodnoty udávají, kolikrát o token který proces žádal. Celý systém lze popsat v následujících bodech:

1. Pokud proces drží token a nechce již vstoupit do kritické sekce, resp. svoji činnost v této sekci ukončil, nebo jej drží od samého počátku algoritmu, kontroluje, zda hodnoty ve vektoru tokenu odpovídají hodnotám jeho vektoru. Pokud se liší a v jeho vektoru je některá z hodnot vyšší, odešle token prvnímu, u kterého je tato hodnota vyšší. Takový rozdíl mezi hodnotami znamená, že proces ví o jiném procesu, který token požaduje, ale doposud mu pro takovou žádost nebyl token přidělen.
2. Pokud proces chce vstoupit do kritické sekce, zašle zprávu všem ostatním procesům s číslem, které udává, pokolikáté o kritickou sekci žádá. Ti po obdržení této zprávy zvýší hodnotu ve svém vektoru pro tento proces, pokud je nižší než aktuální.
3. Pokud proces obdržel token, zvýší svoji hodnotu ve vektoru tokenu o jedničku a provede kritickou sekci.

Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele barborasmahlikova.

11/11