Operačné systémy / Medziprocesová komunikácia

Dušan Bernát
bernat@fmph.uniba.sk

Prehľad

- Súbeh, vzájomné vylučovanie, podmienky korektného riešenia.
- Programové riešenia.
- Riešenia pre viac procesov (Peterson, ticket, bakery).
- Podpora hardvéru (cmpxchg), transkačná pamäť.
- Obsadzujúce čakanie.
- Vyššie synchronizačné prostriedky, semafory, ich realizácia a použitie.
- Futex.
- Kruhový buffer, dátovody.
- Zasielanie správ.
- Problém obedujúcich filozofov, producent/konzument.

Súbeh

Interakcia procesov

- Procesy na seba vzájomne pôsobia z rôznych dôvodov:

1) Neúmyselne. Procesy si neuvedomujú iné procesy.
   - Procesy vykonávajú operácie bez ohľadu na iné procesy.
   - Súťažia o spoločné prostriedky (nevedia však s kým).
   - Prístup riadi operačný systém.
2) Nepriamo. Uvedomujú si iné procesy len sprostredkovane.
   - Vedia o existencii nejakých iných procesov, ale nevedia ich priamo pomenovať (adresovať).
   - Napríklad používajú spoločný prostriedok. Čítajú zo súboru, do ktorého musel nejaký iný proces zapisovať.
3) Úmyselne. Uvedomujú si existenciu iných procesov.
   - Vedia sa vzájomne rozlíšiť (adresovať.)
   - Spolupracujú, komunikujú na zabezpečenie spoločnej úlohy.
   - Komunikácia je explicitne vyjadrená v programe.

Komunikácia a spoločná pamäť

- Komunikácia – výmena dát.
- Požiadavka na komunikáciu procesov pri riešení úloh je častá.
  - Napríklad výstup jedného procesu môže byť vstupom druhého.
- Procesy majú spravidla izolované adresové priestory, ale pre účely vzájomnej komunikácie OS poskytuje možnosť vytvoriť spoločné úložisko.
  - Napríklad oblasť zdieľanej pamäte (možno aj v jadre).
  - Vlákna zdieľajú celý spoločný adresový priestor procesu. Problémy medziprocesovej komunikácie sa tak týkajú aj vlákien.

Synchronizácia vykonávania procesov

- Pri niektorých úlohách je potrebné dodržať predpísané poradie vykonávania procesov.
  - Je to podobné ako pri komunikácii. Dáta je možné prečítať až potom, čo sú zapísané.
  - V tomto prípade sa však dáta neprenášajú.
- Synchronizácia môže byť považovaná za špeciálny prípad komunikácie, keď sa neprenášajú žiadne dáta.
  - Veľkosť prenášaných dát je nulová.
  - Podstatné je, že sa komunikuje, respektíve kedy sa komunikuje, nie či sa pritom niečo prenáša.
- Zabezpečenie synchronizácie väčšinou znamená, že niektorý z procesov musí čakať.

Súbeh

- Procesy pristupujúce do spoločnej pamäti sa môžu vykonávať súbežne, respektíve v ľubovoľnom poradí.
- Proces môže byť operačným systémom preplánovaný v ľubovoľnom okamihu.
  - Prechod zo stavu *running* do *ready*, iniciovaný prerušením od časovača.
- Z pohľadu procesora môže nastať prerušenie len medzi dvoma inštrukciami. Vykonanie inštrukcie je atomické.
- Príklad: inkrementácia spoločnej premennej, **a++**

```
#CISC
inc a
```

✔

```
#RISC
load r1, a
inc r1
store a, r1
```

?

Súbeh

```
1:  load r1, a
2:  inc r1
3:  store a, r1
```

- Pôvodná hodnota premennej je **a=7**.
- Ak dva procesy vykonajú inkrementáciu, očakávaný výsledok je **a=9**.
- Poradie vykonania:
  - Zápis: *Px* – proces *P* vykonáva riadok *x*; (*a*, *b*) - sekvenčné vykonanie *b* po *a*.
  - (P1, P2, P3, Q1, Q2, Q3) ~ (Q1, Q2, Q3, P1, P2, P3) → **a=9** ✔
  - (P1, Q1, P2, Q2, P3, Q3) ~ (P1, P2, Q1, P3, Q2, Q3) ~ … → **a=8** ! ■
- Pri súbežnom vykonaní viacerých procesov sú možné všetky kombinácie prekrytí (sekvenčných) vykonaní jednotlivých procesov.
- Situáciu keď viacero procesov pristupuje k spoločnej premennej a výsledok závisí od poradia vykonávania nazývame **súbeh** (*race condition*).
  - Procesy súťažia (pretekajú) o prístup a výsledok závisí od toho, kto bude prvý.
  - Môže spôsobovať náhodné správanie, alebo rôzne výsledky.

Podmienka vzájomného vylučovania

- Aby sme dosiahli očakávaný výsledok, je potrebné zabezpečiť aby k spoločnému prostriedku (tu premennej) pristupoval **v každom čase najviac jeden** z procesov.
- Takejto podmienke hovoríme **vzájomné vylučovanie** (*mutual exclusion*).
  - Prístup jedného vylučuje prístup ostatných.
  - Vzájomné vylučovanie je typom synchronizácie.
- Úsek programu kde sa vykonáva prístup k spoločnému prostriedku sa nazýva **kritická oblasť** (*critical section*).
  - Kritická oblasť sa môže rozkladať na viacerých miestach programu.
  - Program môže obsahovať viacero (rôznych) kritických oblastí.
- Podmienka vzájomného vylučovania hovorí, že vykonávanie procesov sa nesmie prekrývať v kritickej oblasti.
- **Program v kritickej oblasti smie byť v každom čase vykonávaný najviac jedným procesom.**

Implementácia vzájomného vylučovania

- Možné riešenia vzájomného vylučovania môžme rozdeliť na:
  - Čisto programové
    - Vieme ich napísať len s prostriedkami jazyka, napríklad C.
    - Cykly, podmienky, prístup do (spoločnej) pamäte.
    - Bez zvláštnych systémových volaní.
  - S podporou hardvéru
    - Zakázanie prerušení.
    - Špeciálne inštrukcie.
    - Transakčná pamäť.
- Okrem implementačnej stránky sú však dôležité aj ďalšie vlastnosti riešenia, najmä spôsob čakania a jeho vplyv na zaťaženie systému.

Zakázanie prerušení

- Procesor umožňuje pomocou riadiacich inštrukcií zakázať (aj povoliť) spracovanie prichádzajúcich prerušení.
- Ak by sme zakázali prerušenia na začiatku kritickej oblasti a na konci povolili, vykonávanie by nemohlo byť prerušené.
  - Žiaden iný proces by sa do kritickej oblasti nedostal.
- Výhody
  - Jednoduchá implementácia.
- Nevýhody
  - Neobsluhujú sa prerušenia od periférnych zariadení.
  - V prípade chyby pri zakázaných prerušeniach sa “zasekne” celý systém.
  - V chránenom režime je dostupné len pre jadro (privilegované inštrukcie).
- Používa sa **v jadre**, na krátke časy, pri implementácii vyšších mechanizmov.

```
cli
load r1, a
inc r1
store a, r1
sti
```

Multiprocesorové systémy

- Pokiaľ systém obsahuje viac fyzických CPU, zakázanie prerušení na jednom z nich nestačí.
  - Nie je to riešenie. Nezaručí vzájomné vylučovanie.
- Podobne problém súbehu nevyrieši ani atomická operácia typu **`inc`** (dostupná na CISC).
  - Síce nemôže nastať preplánovanie na tomto procesore, ale súbežne môže k spoločnej premennej v pamäti pristupovať proces vykonávaný na inom procesore.
- Riešenie: **uzamknutie zbernice** signálom lock (architektúra Intel).
  - **`lock inc a;`** `lock` je tzv. prefix inštrukcie
  - Počas aktívneho signálu *lock* nemôžu iné procesory pristupovať do pamäti.
  - Do žiadnej, teda ani mimo kritickú oblasť, čím sa celková výkonnosť znižuje.

Programové riešenia

Vstupno-výstupný protokol

- Uvažujme dva procesy (bez ujmy na všeobecnosti), ktoré sa pokúšajú súbežne vstúpiť do kritickej oblasti **`UseResource()`**.
- Operácie zabezpečujúcie vyriešenie vzájomného vylučovania pri prístupe vložíme do funkcií **`EnterCS()`** a **`LeaveCS()`**.
  - **`EnterCS()`** bude zrejme obsahovať čakanie.
  - Funkcie vstupu a výstupu nemusia byť pre jednotlivé procesy identické. Spravidla však budú analogické (respektíve symetrické).
- Ilustračná situácia:

```
ProcessP() {
   while(1) {
      EnterCS_P();
      UseResource();
      LeaveCS_P();
   }
}
```

```
ProcessQ() {
   while(1) {
      EnterCS_Q();
      UseResource();
      LeaveCS_Q();
   }
}
```

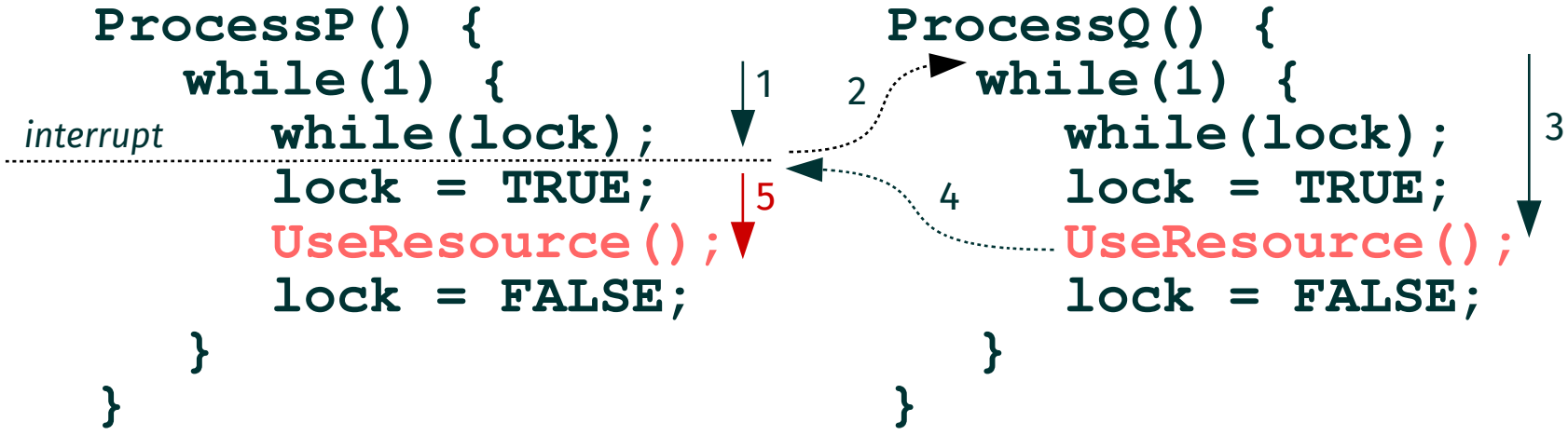
Riešenie 1: Spoločný zámok

- Spoločná premenná so sémantikou zámku. Hodnota 1 indikuje, že kritická oblasť je obsadená, 0 voľná.
- Pred vstupom sa počká, kým bude kritická oblasť voľná. Potom sa nastaví zámok na 1 a proces smie vojsť do kritickej oblasti.
- Pri výstupe sa premenná nastaví na 0, teda kritická oblasť je voľná.
- Vstupno-výstupný protokol:

```
int lock = FALSE;

                          čakanie
                        ┌──────────┐
EnterCS() { while(lock); lock = TRUE; }
LeaveCS() { lock = FALSE; }
```

Spoločný zámok nefunguje

- Problém nastane, ak bude proces preplánovaný [2] bezprostredne potom ako vyhodnotí premennú **`lock`** ako 0 a ukončí čakanie, a zároveň predtým, ako nastaví **`lock`** na 1.
- Následne aj druhý proces môže vyhodnotiť **`lock`** ako 0 a vojsť do kritickej oblasti [3], kde môže byť tiež preplánovaný [4].
- Prvý proces bude pokračovať od miesta kde bol prerušený [5] a teda tiež môže vojsť do kritickej oblasti a budú tam **súčasne oba procesy!**
- Nie je splnená ani základná podmienka pre vzájomné vylučovanie.



Riešenie 2: Striedanie

- Spoločná premenná so sémantikou určujúcou kto smie vojsť.
- Proces pred vstupom do kritickej oblasti počká, kým bude na rade.
- Pri výstupe nastaví, že na rade je druhý proces.
- Procesy sa striedajú a teda nikdy nebudú v kritickej oblasti súčasne.
  - Podmienka vzájomného vylučovanie je splnená; bez ohľadu na to, kedy nastane preplánovanie.
- Vstupno-výstupný protokol:

```
enum {P, Q} turn = P;

    EnterCS_P() { while(turn != P); }
    LeaveCS_P() { turn = Q; }

    EnterCS_Q() { while(turn != Q); }
    LeaveCS_Q() { turn = P; }
```

Striedanie je obmedzujúce

- Čo v prípade ak hodnota spoločnej premennej indikuje, že na rade je proces P, hoci OS naplánoval na vykonávanie proces Q?
  - Proces Q musí čakať, hoci kritická oblasť je voľná.
- Čo ak chce jeden z procesov pristupovať do kritickej oblasti dvakrát po sebe, respektíve častejšie, než druhý proces?
  - Proces bude zbytočne blokovaný, hoci je kritická oblasť voľná.
- Čo ak jeden z procesov nebude chcieť (dlho, alebo vôbec) pristupovať do kritickej oblasti?
- **Toto riešenie je nevyhovujúce.**
  - Spĺňa základnú podmienku, ale je obmedzujúce a neefektívne.
  - Vstup do kritickej oblasti nesmie byť blokovaný, ak je voľná.
- Pre korektné riešenie problému vzájomného vylučovania budeme preto vyžadovať aj splnenie ďalších podmienok (okrem základnej).
- **Proces ktorý sa vykonáva mimo kritickej oblasti nesmie brániť iným vstúpiť do nej.**

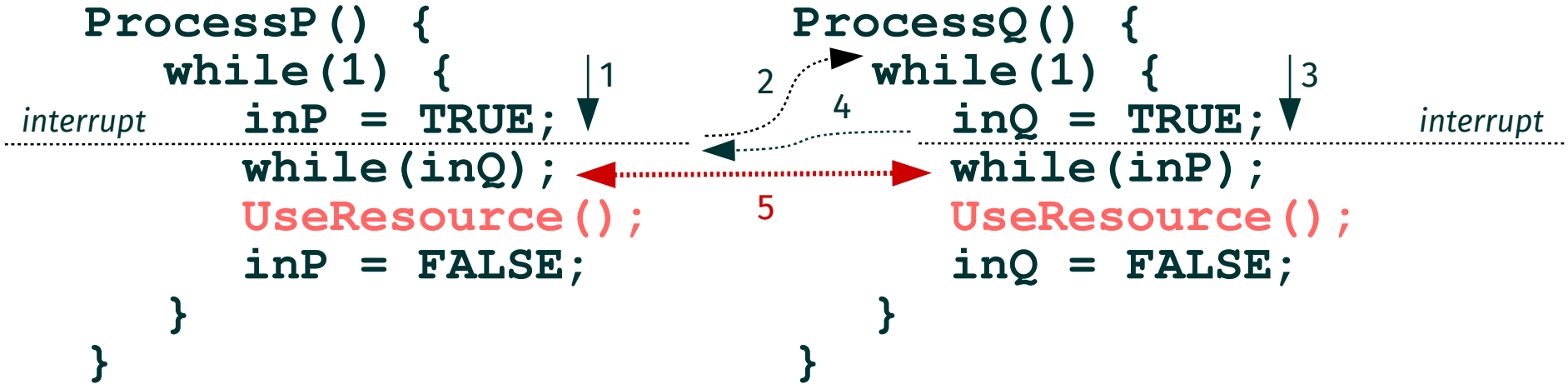
Riešenie 3: Zámok pre každý proces

- Každý proces bude mať svoju spoločnú premennú, ktorou bude vopred indikovať potrebu vstúpiť do kritickej oblasti.
- Proces pred vstupom do kritickej oblasti nastaví svoju premennú a počká, ak aj druhý proces chce vojsť.
- Obe podmienky kladené na riešenie budú splnené.
  - Nie je možné, aby do kritickej oblasti vstúpili oba procesy.
  - Proces mimo kritickej oblasti nebráni inému vstúpiť.
- Vstupno-výstupný protokol:

```
int inP = FALSE, inQ = FALSE;

    EnterCS_P() { inP = TRUE; while(inQ); }
    LeaveCS_P() { inP = FALSE; }

    EnterCS_Q() { inQ = TRUE; while(inP); }
    LeaveCS_Q() { inQ = FALSE; }
```

Nekonečné čakanie

- Problém nastane, ak budú oba procesy preplánované [2, 4] po nastavení svojej premennej na 1 [1, 3] (budú nastavené súčasne).
- Následne budú oba procesy čakať, pričom už nie je žiadna možnosť aby ktorýkoľvek z procesov zrušil podmienku čakania druhému.
- Táto situácia [5] sa nazýva **uviaznutie** (*deadlock*).
- **Rozhodnutie o vstupe musí prísť v konečnom čase.**
  - Korektné riešenie nesmie umožňovať uviaznutie.



Riešenie 4: Prerušenie čakania

- Každý proces bude mať svoju spoločnú premennú, ktorou bude vopred indikovať potrebu vstúpiť do kritickej oblasti.
- Pred vstupom proces nastaví svoju premennú na 1. Ak má však premennú nastavenú aj druhý proces, nastaví svoju naspäť na 0 a pokus o vstup zopakuje.
- Vstupno-výstupný protokol:

```
int inP = FALSE, inQ = FALSE;

EnterCS_P() {
   while(1) {
      inP = TRUE;
      if (inQ) inP = FALSE;
         else break;
   }
}
LeaveCS_P() { inP = FALSE; }
```

```
EnterCS_Q() {
   while(1) {
      inQ = TRUE;
      if (inP) inQ = FALSE;
         else break;
   }
}
LeaveCS_Q() { inQ = FALSE; }
```

Doba prístupu nie je ohraničená

- Ide o modifikáciu predchádzajúceho prístupu (3). Zaručuje splnenie troch podmienok.
  - Nie je možné, aby do kritickej oblasti vstúpili oba procesy.
  - Proces mimo kritickej oblasti nebráni inému vstúpiť.
  - Nemôže nastať uviaznutie.
- Kým by boli oba procesy preplánované vždy po nastavení premennej, oba by čakali.
  - Korektnosť tohoto riešenia je založené na predpoklade, že sa tak (dlhodobo) nestane.
  - Nejde o uviaznutie, pretože toto **čakanie** v princípe **môže skončiť**.
  - Situácia, keď sa dva procesy neustále uprednostňujú v dôsledku čoho ani jeden nevstúpi do kritickej oblasti, sa nazýva *livelock*, alebo tiež starvácia.
- Doba prístupu nie je ohraničená. Závisí od plánovača a nie len od samotného riešenia.
  - Teoreticky to môže byť aj nekonečne dlho. Prakticky je však pravdepodobnosť uviaznutia malá. Toto riešenie sa prakticky využíva (napríklad CSMA-CD).
- **Procesy nemôžu pri vstupe do kritickej oblasti predpokladať nič o vzájomnom časovaní.**

Podmienky korektného riešenia

- Na programové riešenie synchronizačného problému vzájomného vylučovania procesov teda kladieme tieto 4 podmienky:
  1) **V kritickej oblasti sa smie vykonávať v každom čase najviac jeden proces.**
  2) **Proces ktorý sa vykonáva mimo kritickej oblasti nesmie brániť iným vstúpiť do nej.**
  3) **Rozhodnutie o vstupe musí prísť v konečnom čase.**
  4) **Procesy nemôžu pri vstupe do kritickej oblasti predpokladať nič o vzájomnom časovaní (plánovaní).**
- Existuje také riešenie?

Petersonovo riešenie (1981)

- Kombinuje predchádzajúce prístupy (zámky, zapamätanie kto je na rade, upozornenie na vstup).
- Spĺňa všetky kladené podmienky bez ohľadu na plánovanie.
- Dá sa rozšíriť aj pre viac ($N>2$) procesov.
- Vstupno-výstupný protokol:

```
int inP = FALSE, inQ = FALSE;
enum {P, Q} last = P;

EnterCS_P() {
   inP = TRUE;
   last = P;
   while(inQ && last == P);
}

LeaveCS_P() { inP = FALSE; }
```

```
EnterCS_Q {
   inQ = TRUE;
   last = Q;
   while(inP && last == Q);
}

LeaveCS_Q() { inQ = FALSE; }
```

Podpora procesora

Špeciálne inštrukcie

- Väčšina procesorov poskytuje inštrukciu ktorá umožňuje vykonať **viacero operácií**, vrátane prístupu do pamäte, **atomicky** (symbolicky značíme << ... >>).
- Umožňujú jednoduchú implementáciu (nielen) vzájomného vylučovania pre N procesov.
- Inštrukcie typu:
  - TSL (*Test and Set Lock*), `TSL Rx, lock`
    - `int TSL(int *lock) { << int tmp = *lock; lock = 1; return tmp; >> }`
  - XCHG (*Exchange*), Intel: `XCHG lock, Rx` – aj s prefixom *lock*
    - `void XCHG(int a, int *lock) { << int tmp=a; a=*lock; *lock=tmp; >> }`
  - CSW (*Compare and Swap*), Intel: `CMPXCHG lock, Rx` – aj s prefixom *lock* (a=EAX, b=Rx, c=lock)

```
int CSW(int *a, int b, int *c)   { <<
   if (*a==*c) { *c=b; return 0; }
      else     { *a=*c; return 1; }  >> }
```

  - FA (*Fetch and Add*)
    - `int FA(int *var, int incr) { << int tmp=*var; *var+=incr; return tmp; >> }`

Špeciálne inštrukcie – vlastnosti

- Vstupno-výstupný protokol s použitím *Test and Set*:

```
int lock = FALSE;

EnterCS() { while (TS(&lock)); }

LeaveCS() { lock = FALSE; }
```

- Riešenia založené na špeciálnych inštrukciách:
  - majú jednoduchú implementáciu, vyžadujú podporu hardvéru (sú závislé od architektúry),
  - nezaručujú spravodlivosť,
  - čakanie je (veľmi) neefektívne.
- Zámok implementovaný pomocou čakania v tesnej slučke sa nazýva *spinlock*.
  - Nízkoúrovňový synchronizačný mechanizmus využívajúci opakované čakanie na splnenie podmienky.
- Využívajú sa v multiprocesorových systémoch.

Ticket algoritmus

- Algoritmus pre vzájomné vylučovanie *N* procesov inšpirovaný systémom čakania na poradie.
  - Jednoduchá a efektívna implementácia.
- Vstupno-výstupný protokol:

```
int number = 1, next = 1;

EnterCS() {
   << turn = number++; >>
   while(next != turn);
}

LeaveCS() { next++; }
```

Ticket algoritmus – vlastnosti

- Ticket algoritmus
  - Obsahuje tiež kritickú oblasť (inkrement – odtrhnutie lístka, aby mal každý jedinečné číslo). Vyžaduje špeciálnu inštrukciu (podpora HW), napríklad FA.
  - Zaručuje **spravodlivosť**, prístup dodržiava poradie žiadostí.
  - Pretečenie dátového typu `int` nepredstavuje problém, pokiaľ počet čakajúcich nepresiahne jeho rozsah.
  - Problém môže nastať pri zlyhaní procesu. Ostatní čakajú navždy.
  - Zložitosť *O(1)*, nezávisí od počtu procesov.
- Bakery algoritmus (Lamport, 1973)
  - Využíva princíp ticket algoritmu. Ale **nepotrebuje špeciálnu inštrukciu.**
  - Ak dva procesy dostanú rovnaké číslo, rozhodne o poradí ich jedinečné identifikačné číslo (napr. PID). Zložitosť *O(N)*.

Bakery algoritmus

- Algoritmus pre vzájomné vylučovanie *N* procesov inšpirovaný systémom čakania na poradie.
- Vstupno-výstupný protokol:

```
int number[N] = {0}, in[N] = {0};

EnterCS(int i) {
   in[i] = TRUE;
   number[i] = 1 + max(number);  // not atomic
   in[i] = FALSE;
   for (j=0; j<N; j++) {
      while(in[j]);
      while(number[j]!=0 && (number[j]<number[i] ||
         (number[j]==number[i] && j<i)));
}

LeaveCS(int i) { number[i] = 0; }
```

Obsadzujúce čakanie

Obsadzujúce čakanie

- Všetky predchádzajúce riešenia (čisto programové, aj s použitím špeciálnych inštrukcií) majú spoločný problém.
- Čakanie v tvare: `while(lock);`
- Čakanie v slučke, v ktorej sa neustále číta hodnota z pamäti a vyhodnocuje podmienka.
  - Procesor je zaneprázdnený (aktívnym) čakaním, pričom nevykonáva žiadnu užitočnú prácu – obsadzujúce čakanie (*busy waiting*).
  - Procesorový čas je spotrebovaný len na čakanie, teda aby sa nič nerobilo.
  - Pritom v systéme môžu čakať procesy pripravené na vykonanie (*ready*).
- Ak v systéme s prioritným plánovaním takto čaká proces s vyššou prioritou, proces s nižšou prioritou nedostane pridelený procesor, aby mohol kritickú oblasť opustiť.
- Ide o veľmi neefektívny spôsob čakania.
  - Ak je však doba čakania krátka, môže to byť rýchlejšie než riešenie s podporou OS zahrňujúce prepínanie kontextu.
  - Navyše v jadre nemusia byť dostupné iné mechanizmy čakania, využívajúce plánovač.

Multiprocesorové systémy

- Neustále čítanie (spoločnej premennej) z pamäti navyše zahlcuje zbernicu.
  - Keďže pre synchronizáciu je potrebné atomické vykonanie čítania pôvodnej hodnoty a jej aktualizácie, špeciálna inštrukcia sa vykonáva s prefixom *lock*.
- Ostatné procesory, ktoré môžu vykonávať inú, nesúvisiacu činnosť, majú tiež obmedzený prístup do pamäti.
- Jeden proces vykonávajúci obsadzujúce čakanie tak má výrazne degradujúci vplyv na výkonnosť celého systému.
  - Ak je čakajúcich procesov viacero, situácia bude ešte horšia.
  - Obsadzujúce čakanie nie je dobre škálovateľné.
- Na druhej strane, v niektorých prípadoch to môže byť jediný dostupný spôsob synchronizácie v multiprocesorových systémoch.

Semafory

Semafor (Dijkstra, 1968)

- Semafor je všeobecný synchronizačný mechanizmus.
  - Nie len kritická oblasť, dá sa použiť aj na iné synchronizačné úlohy.
  - Rieši väčšinu problémov a nedostatkov, pomerne ľahko sa používa.
- Hlavná výhoda:
  - Odstraňuje obsadzujúce čakanie.
  - Čakajúci proces bude v **blokovanom stave** (*sleep*/*blocked*).
  - Nespotrebováva pri čakaní procesorový čas, neobmedzuje ostatných.
- Semafor je v podstate **celé nezáporné číslo**.
- Jeho hodnotu je možné meniť len dvomi atomickými operáciami.
  - Jeho hodnotu nie je možné priamo zistiť.
- Realizácia si vyžaduje podporu OS.

Operácie so semaforom

- Z programátorského hľadiska ide o abstraktný dátový typ.
  - Atribúty: celé nezáporné číslo, rad čakajúcich procesov.
  - Metódy: **`Init`**, **`Wait`**, **`Signal`**
- Init nastaví hodnotu semaforu na dané číslo.
- Operácie Wait a Signal najskôr vyhodnotia podmienku a potom vykonajú akciu.
- Wait, alebo Down, pôvodne P (Passeren)
  - Ak je hodnota semaforu väčšia než nula, **dekrementuje** ju.
  - Ak je to nula, volajúci proces sa **uspí** a zaradí do radu čakajúcich.
- Signal, alebo Up, pôvodne V (Vrijgeven)
  - Ak je rad čakajúcich neprázdny, vyberie sa jeden proces a **zobudí** sa.
  - Inak hodnotu semaforu **inkrementuje**.
- Obe operácie sú **atomické**. Ak sa operácia začne vykonávať, žiadny iný proces nemá prístup k semaforu kým operácia neskončí (alebo nezablokuje v prípade Wait).

Operácie so semaforom

```
struct semaphore {
   int s;
   queue q;   /* isEmpty, initQueue,
                 enqueue, dequeue */
}

Init(int i) { s = i; initQueue(q); }

Wait() {
   if (s > 0) s--;
   if (s == 0) { enqueue(q, self); sleep(); }
}

Signal() {
   if (!isEmpty(q)) { P = dequeue(q); wakeup(P); }
   else s++;
}
```

Implementácia semaforu

- Operácie semaforu môžu byť volané viacerými procesmi súčasne.
- Semafor obsahuje tiež kritickú oblasť.
  - Manipulácia s hodnotou, manipulácia s radom.
- Pre jeho implementáciu teda potrebujeme nejaký **synchronizačný mechanizmus** nižšej úrovne.
  - Napríklad špeciálne inštrukcie (aj na multiprocesorových systémoch).
  - Ak semafory implementuje OS, môže použiť zakázanie prerušení (na jednoprocesorovom systéme).
    - Procesy majú prístup cez príslušné systémové volania.
- Podstatné je, že kritická oblasť v semafore je malá (inkrementácia, dekrementácia, operácie s pointrami) v porovnaní s tým, ako dlho môže čakať proces na semafor.
- Potrebný je tiež mechanizmus OS umožňujúci **uspať a zobudiť** proces.
  - Napríklad signály, alebo systémové volania ak sú semafory implementované v jadre.

Invariant semaforu

- Operácia Wait môže volajúci proces zablokovať (uspať). Operácia Signal nikdy neblokuje.
- Hodnota semaforu musí zostať vždy nezáporná.
  - Ak by operácia Wait mala spôsobiť, že klesne pod nulu, musí byť volajúci proces zablokovaný, kým nebude iným procesom zavolaná operácia Signal.
- Invariant semaforu: **`pW − pS ≤ I`** alebo **`pW ≤ pS + I`**
  - *I*, počiatočná hodnota semaforu, nastavená operáciou Init,
  - *pS*, počet vykonaných operácií Signal,
  - *pW*, počet dokončených operácií Wait.
- Hodnota semaforu teda určuje, koľko operácií Wait sa vykoná bez toho, aby bol volajúci proces zablokovaný.

Typy semaforov

- Všeobecný semafor
  - Hodnota semaforu (teoreticky) nie je zhora obmedzená.
  - Definícia semaforu nepredpisuje, ktorý proces má byť z radu vybraný.
    - Ak sa použije rad s prístupom FIFO, semafor bude **spravodlivý**.
- Binárny semafor
  - Môže nadobudnúť len hodnotu 0 alebo 1.
  - V podstate rovnaký, ako tzv. *mutex*.
    - Nemusí sa však využívať len na vzájomné vylučovanie.
  - Ak má semafor hodnotu 1 a príde Signal:
    - ignoruje sa, vráti sa chyba, ukončí sa program, ...
- Niektoré implementácie poskytujú tiež operácie ako *trywait*, *timedwait*, ...

Použitie semaforov

Vzájomné vylučovanie

- Ak bude počiatočná hodnota semaforu 1, len jeden proces bude môcť vykonať Wait bez blokovania, kým iný proces nezavolá Signal.
- Vstupno-výstupný protokol:
  - EnterCS() = Wait(), LeaveCS() = Signal().

```
semaphore mutex;
Init(mutex, 1);

   EnterCS() {
      Wait(mutex);
   }

   LeaveCS() {
      Signal(mutex);
   }
```

```
ProcessP() {
   while(1) {
      Wait(mutex);
      UseResource();
      Signal(mutex);
   }
}
```

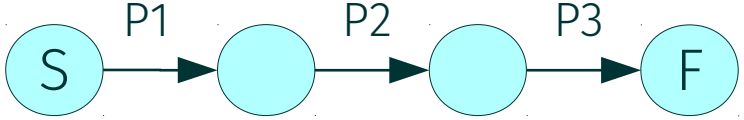
Ďalšie možnosti využitia

- Prístup k viacnásobnému prostriedku
  - Ak je k dispozícii viacero inštancií prostriedku (R), respektíve ak R procesov môže prostriedok využívať súčasne, počiatočná hodnota semaforu sa nastaví na R.
- Synchronizácia vykonávania procesov
  - Proces čaká na nejakú udalosť alebo situáciu, ktorá závisí od iného procesu.
  - Počiatočná hodnota semaforu sa nastaví na nulu.
  - Čakajúci proces zavolá Wait. Keď nastane potrebná situácia, druhý proces zavolá Signal, čím umožní prvému procesu pokračovať.

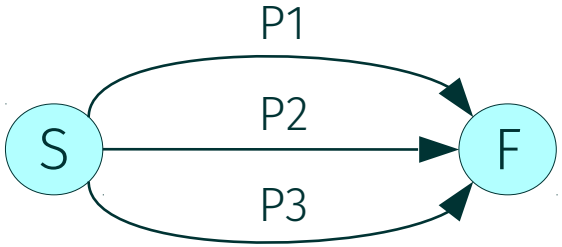
Precedenčný graf

- Precedenčný graf (*process flow graph*) vyjadruje poradie vykonania procesov.
  - Orientovaný acyklický (multi) graf, hrany vyjadrujú vykonanie procesu.
  - Každá cesta v grafe vyjadruje možné vykonanie, pričom poradie hrán vyjadruje vzájomné poradie vykonania procesov.
  - Procesy zodpovedajúce hranám vystupujúcim z vrcholu sa môžu začať vykonávať až keď sú všetky procesy zodpovedajúce vstupujúcim hranám ukončené.
- Precedenčný graf reprezentuje reláciu čiastočného usporiadania.
- Ak niektorá dvojica procesov nie je v relácii znamená to, že sa môžu vykonať v ľubovoľnom vzájomnom poradí, respektíve súbežne.

Sekvenčné
(P1; P2; P3)



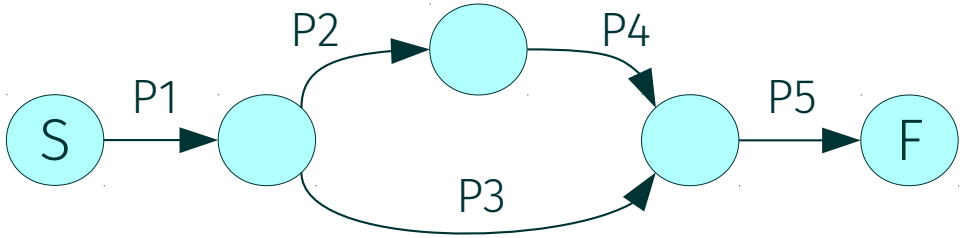
Paralelné
(P1|P2|P3)



Príklad: Synchronizácia

- Napísať kostry programov pre nasledujúce vykonanie procesov, s použitím minimálneho počtu semaforov:

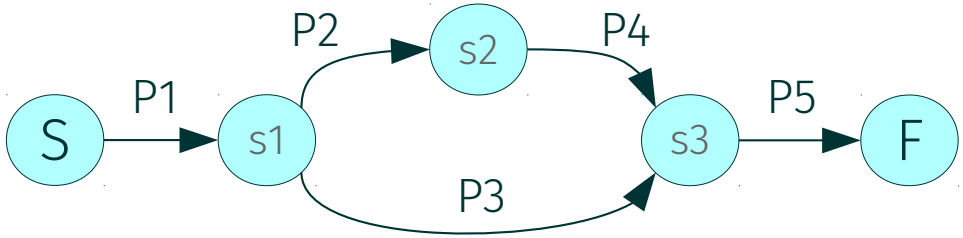
  a) všeobecných, b) binárnych.



Príklad: Synchronizácia

- Napísať kostry programov pre nasledujúce vykonanie procesov, s použitím minimálneho počtu semaforov:

  a) všeobecných, b) binárnych.




```
P1() {
   Work1();
   Signal(s1);
   Signal(s1);
}

P2() {
   Wait(s1);
   Work2();
   Signal(s2);
}
```

```
P3() {
   Wait(s1);
   Work3();
   Signal(s3);
}

P4() {
   Wait(s2);
   Work4();
   Signal(s3);
}
```

```
s1=s2=s3=0;

P5() {
   Wait(s3);
   Wait(s3);
   Work5();
}
```

■

Buffer pevnej dĺžky

- Buffer – prostriedok pre asynchrónnu komunikáciu procesov. Vyrovnáva rôzne rýchlosti zápisu a čítania.
- Môže byť implementovaný ako abstraktný dátový typ.
  - Pamäť, rad pevnej dĺžky typu FIFO + operácie **`Send`** a **`Receive`**
- Rad je spoločný. Operácie môžu byť volané **súbežne**, rôznymi procesmi.
  - Obsahujú kritickú oblasť.
- Čo sa stane, ak chce proces zapísať a nie je voľné miesto?
- Čo sa stane, ak chce proces čítať, ale rad je prázdny?
- Pri implementácii je potrebná **synchronizácia**.
  - Operácia **`Send`** sa môže bez blokovania vykonať toľkokrát, koľko je voľného miesta.
  - Operácia **`Receive`** sa môže bez blokovania vykonať toľkokrát, koľko je v rade položiek.

Implementácia buffera (semafory)

```
struct buffer {
   semaphore full, empty, mutex;
   queue q;
}

CreateBuffer(int N) {
   Init(full, 0);
   Init(empty, N);
   Init(mutex, 1);
   initQueue(q, N);
}

Send(message msg) {
   Wait(empty);
   Wait(mutex);
   enqueue(q, msg);
   Signal(mutex);
   Signal(full);
}
```

```
Receive(message *msg) {
   Wait(full);
   Wait(mutex);
   *msg = dequeue(q);
   Signal(mutex);
   Signal(empty);
}
```

Producent – konzument

- Implementácia komunikácie procesov pomocou radu pevnej dĺžky rieši problém typu producent – konzument.
- Jeden proces dáta vytvára a zapisuje, druhý ich číta a spracováva.
  - Producentov aj konzumentov môže byť aj viac.
- Efektívne znovupoužitie uvoľneného miesta → kruhový buffer.
- OS typu Unix implementujú kruhový buffer, pričom vloženie a odobranie dát je realizované súborovými operáciami (read/write).
  - Dátovod (*pipe*) slúži na komunikáciu dvoch procesov.

Problém obedujúcich filozofov (Dijkstra, 1965)

- Príklad: Na stole je v kruhu 5 tanierov a 5 vidličiek. Za stolom sedí 5 filozofov, ktorí striedavo premýšľajú a jedia.
- Filozof potrebuje na jedenie vždy dve vidličky. Po premýšľaní, keď vyhladne, pokúsi sa vziať ľavú a pravú vidličku. Potom môže jesť.
- Keď doje, obe vidličky položí späť na stôl.
- Analógia: Máme 5 procesov ktoré súťažia o obmedzené zdroje.
- Návrh 1: Filozof počká na ľavú vidličku, potom na pravú vidličku.
  - Problém: ...
- Návrh 2: Filozof počká na ľavú vidličku, ak je pravá obsadená, ľavú položí a skúsi znova.
  - Problém: ...

Problém obedujúcich filozofov – riešenia

- Návrh 3: Jeden z filozofov môže brať vidličky v opačnom poradí.
- Návrh 4: Nechať súťažiť o vidličky vždy len 4.
- Návrh 5: Filozof môže jesť len ak ani jeden jeho sused práve neje.
  - Test predstavuje kritickú oblasť → použijeme *mutex*.
  - Spoločná premenná pre každého filozofa: stav (thinking, hungry, eating).
  - Pred začatím jedenia sa počká na semafor ktorý indikuje, či sú obe vidličky voľné (ani jeden sused neje).
    - Ak nie, hladný filozof čaká.
    - Každý kto skončí skontroluje, či jeho susedia sú hladní. A ak áno, či aj ich druhý sused je.
  - Ani toto riešenie (Tanenbaum) nie je bez problémov.

Riešenie 4

```
semaphore chair = N-1;
semaphore fork[N] = {1};

Philosopher (int i) {
   think();
   Wait(chair);           /* take a seat */
   Wait(fork[i]);         /* take left   */
   Wait(fork[(i+1)%N]);   /* take right  */
   eat();
   Signal(chair);
   Signal(fork[i]);
   Signal(fork[(i+1)%N]);
}
```

Problém: Readers / Writers

- K spoločnému prostriedku (pamäti, alebo databáze) smie mať prístup alebo jeden zapisujúci proces, alebo viacero čítajúcich.
  - Prítomnosť zapisujúceho procesu vylučuje všetky ostatné procesy.
  - Prítomnosť čítajúceho procesu vylučuje prítomnosť zapisujúceho, ale nie ďalších čítajúcich.
- Výlučné je len zapisovanie.
- Je potrebné zabezpečiť vzájomné vylučovanie nie medzi všetkymi procesmi, ale medzi dvomi skupinami procesov.

Riešenie (semafory)

```
Reader() {
    Wait(mutex);
    n_readers++;
    if (n_readers==1)
        Wait(wr);
    Signal(mutex);
    do_read();
    Wait(mutex);
    n_readers--;
    if (n_readers==0)
        Signal(wr);
    Signal(mutex);
}
```

```
semaphore mutex = 1;
semaphore wr = 1;
int n_readers = 0;

Writer() {
    Wait(wr);

    do_write();

    Signal(wr);
}
```

- **mutex** chráni počítadlo **n_readers**, počet čítajúcich (čaká sa krátko).
- **wr** (v podstate tiež mutex), zamkne a odomkne každý zapisujúci, ale len prvý čítajúci proces (čakanie môže trvať dlho).
- Nezaručuje spravodlivosť, čítania sú uprednostňované.

Podmienené premenné

Podmienené premenné

- Vzájomné vylučovanie neumožňuje situáciu, kedy by mal proces čakať (v kritickej oblasti) na splnenie nejakej podmienky.
  - Zablokovanie v kritickej oblasti prakticky znamená uviaznutie.
- Podmienené premenné umožňujú procesu ktorý zamkol zámok čakať, pričom tento zámok bude počas čakania uvoľnený a keď sa proces zobudí, zámok bude opäť zamknutý.
  - Operácia `CondWait(condition, lock);`
  - Umožňuje (atomicky) uvoľniť zámok a zablokovať volajúci proces.
- Keď iný proces zmení podmienku, zobudí niektorý z procesov čakajúcich na túto podmienku
  - Ak nikto nečaká, nič sa nestane. Signál sa stratí.
  - Volajúci musí mať vopred zamknutý ten istý zámok, ktorý predtým zamkol čakajúci proces.
  - `CondSignal(condition);` `CondBroadcast(condition);`

Konečný rad (podmienené premenné)

```
semaphore mutex = 0;  queue q;
condition cond;       int count = 0;

Send(message msg) {
   Wait(mutex);
   while(count >= N)
      CondWait(cond, mutex);
   enqueue(q, msg);
   count++;
   CondBroadcast(cond);
   Signal(mutex);
}
```

```
Receive(message *msg) {
   Wait(mutex);
   while(count == 0)
      CondWait(cond, mutex);
   msg* = dequeue(q);
   count--;
   CondBroadcast(cond);
   Signal(mutex);
}
```

- Keď je proces **zablokovaný** v `CondWait`, mutex je **uvoľnený**.
- Po prebudení z `CondWait` **nie je zaručené**, že podmienka je splnená.
- Ak by bolo producentov a konzumentov viac, dochádzalo by k zbytočnému prebudeniu
  - Riešením by boli dve podmienené premenné, zvlášť pre obsadené a voľné miesto.

Synchronizácia v knižnici pthread

- Knižnica poskytuje mutex:
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- a podmienené premenné:
  - `int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);`
  - `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - `int pthread_cond_signal(pthread_cond_t *cond);`

Implementácia rýchlych zámkov

FUTEX (Fast Userspace Mutex)

- Systémové volanie umožňujúce procesu čakať na zmenu hodnoty na danej adrese.
  - Využíva sa na implementáciu zámkov a semaforov (napríklad *pthread_mutex*).
- `int futex(int *uaddr, int op, int val, const struct timespec *timeout, int *uaddr2, int val3);`
  - FUTEX_WAIT, ak sa hodnota na adrese *uaddr* rovná *val*, volajúci proces sa uspí.
  - FUTEX_WAKE, zobudí najviac *val* procesov čakajúcich na adresu *uaddr*.
- FUTEX je identifikovaný adresou v (spoločnej) pamäti.
  - Pre toto volanie sú rovnaké adresy v rôznych procesoch zhodné.
- V prípade, že je zámok voľný, systémové volanie nie je potrebné. Podobne v prípade uvoľnenia, ak ešte nikto nečaká. V týchto prípadoch je to rýchle.
  - Na atomické prečítanie pôvodnej hodnoty a zamknutie alebo uvoľnenie sa môžu využiť špeciálne inštrukcie, napríklad CMPXCHG, FA.
  - Tento prípad je štatisticky pomerne častý, úspory môžu byť veľké (napr. *semop* syscall).

Príklad: Zámok (FUTEX)

```
int lock = 0;    /* 0 - free, 1 - locked by only 1,
                    2 - locked and waiting */

Lock() {
   int old = 0; CSW(&old, 1, &lock);
   if (old) {    /* already locked */
      do {
         *lock = 2;    /* atomic */
         futex(&lock, FUTEX_WAIT, 2);
         old = 0; CSW(&old, 2, &lock);
      } while(old);
   }
}

Unlock() {
   if (FA(&lock, -1) != 1) {
      *lock = 0;
      futex(&lock, FUTEX_WAKE, 1);
   }
}
```

```
int CSW(int *a, int b, int *c)      { <<
    if (*a==*c) { *c=b; return 0; }
        else    { *a=*c; return 1; } >> }
```

IPC v prostredí OS typu Unix

IPC štandardy

- Existujú odlišné štandardy – a aj implementácie – podobných mechanizmov
  - správy, semafory, zdieľaná pamäť.
- **System V** – staršia (pôvodná) špecifikácia, veľa rozšírených implementácií, dostupné (takmer) na každom systéme.
  - zložitejšie rozhranie.
- **POSIX** – novší štandard (nemá úplnú podporu všade)
  - *thread-safe*,
  - jednoduchšie rozhranie,
  - nové možnosti (notifikácia).

Semafory – System V

- `#include <sys/sem.h>`, príkazy: ipcs, ipcrm
- `int semget(key_t key, int nsems, int semflg)`
  - vráti identifikátor na pole obsahujúce *nsems* semaforov pre daný kľúč (`ftok()`)
- `int semop(int semid, struct sembuf *sops, unsigned nsops)`
  - urobí danú operáciu na *nsops* semaforov naraz (alebo vôbec),
- `struct sembuf`
  `{ unsigned short sem_num; short sem_op; short sem_flg; }`
  - kladné: pripočíta sa k hodnote semaforu, nikdy neblokuje,
  - nula: ak hodnota semaforu nie je 0, blokuje kým nebude 0,
  - záporné: odpočíta ak sa dá, inak blokuje.
- Sú to systémové volania (semget, semop, semctl).

Semafory – POSIX

- **`#include <semaphore.h>`**
- pomenované: **`sem_open()`**, **`sem_close()`**, **`sem_unlink()`**,
- nepomenované: **`sem_init()`**, **`sem_destroy()`**,
  - musia byť umiestnené v spoločnej pamäti,
- **`sem_getvalue()`**, **`sem_wait()`**, **`sem_trywait()`**, **`sem_post()`**
- efektívnejšie, operácie s jedným semaforom,
- zvyčajne dostupné cez súborový systém **`/dev/shm`**

Zdieľaná pamäť

- **System V**
  - `int shmget(key_t key, size_t size, int shmflg)`
    - vráti ID nového alebo existujúceho segmentu zdieľanej pamäte s daným kľúčom,
  - `void * shmat(int shmid, const void *shmaddr, int shmflg)`
    - namapuje zdieľaný segment *shmid* na adresu *shmaddr*,
  - ďalšie operácie: `shmdt()`, `shmctl()`.
- **POSIX**
  - `shm_open()`, `shm_unlink()`,
  - implementácia založená na `mmap()`.

Správy (message queues)

- **System V**
  - `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`, `msgop()`
  - typ správy – `struct msgbuf {long mtype; char mtext[1]; };`
- **POSIX**
  - `mq_open()`, `mq_close()`, `mq_unlink()`, `mq_send()`, `mq_receive()`, `mq_notify()`, `mq_setattr()`, `mq_getattr()`
  - notifikácia (NONE, SIGNAL, THREAD),
  - priority; správy sú doručované v poradí priorít.

Zasielanie správ

Vlastnosti zasielania správ

- Komunikácia medzi procesmi prostredníctvom spoločnej pamäte môže byť zložitá.
- Korektné použitie semaforov pre synchronizáciu nie je priamočiare.
  - Operácie so sémantikou *wait* a *signal* nie sú pre komunikáciu prirodzené.
- Monitory tieto nedostatky odstraňujú, používajú sa jednoduchšie. Nie sú však podporované v mnohých jazykoch.
- Zasielanie správ predstavuje mechanizmus komunikácie pomocou funkcií `send(destination, &message)` a `receive(source, &message)`.
  - Sémantika je jasná. Operácie explicitne vyjadrujú komunikáciu.
  - Zdroj a cieľ predstavujú adresy pričom nemusia zodpovedať jedinému uzlu.
  - Zdroj nemusí byť určený (ANY), cieľ môže zodpovedať viacerým (BROADCAST).
- Zasielanie správ môže byť na rozdiel od spoločnej pamäte implementované aj medzi procesmi na rôznych uzloch (rôznych OS).
  - Aj formou knižnice, napríklad MPI (Message Passing Interface), PVM (Parallel Virtual Machine).

Adresovanie

- Komunikácia môže byť priama, alebo nepriama.
- **Priama** – adresa špecifikuje konkrétny komunikujúci proces.
- **Nepriama** – adresa špecifikuje miesto (schránku) na ktoré budú správy ukladané a z ktorého budú čítané.
  - Komunikuje sa cez tzv. schránky (*mailbox*).
  - Schránky môžu mať (konečnú) kapacitu pre viac správ (*buffer*).
  - Príkladom môžu byť aj dátovody.
- Správy, ktoré boli odoslané ale neboli ešte prijaté uchováva operačný systém.
  - Z praktického hľadiska môže byť implementácia správneho uvoľňovania správ problematická. Vysielajúci proces nemusí mať istotu, že dáta už boli doručené.
  - Kto a kedy má uvoľniť správu? Vysielajúci môže, ale nevie kedy. Prijímajúci vie kedy, ale nemá priamo prístup k dátam vo vysielajúcom procese.

Príklad: Producent – konzument (zasielanie správ)

- Producent – konzument, priama komunikácia:
  - Konzument pošle producentovi toľko prázdnych správ, koľko správ s dátami môže prijať .
  - Keď má producent pripravenú správu pre konzumenta, prijme (spracuje) prázdnu správu a pošle správu s dátami.
- Producent – konzument, komunikácia prostredníctvom schránok:
  - Pri použití schránok je ukladanie správ priamočiare.
  - Producent aj konzument si vytvoria schránky pre *N* správ.
  - Producent posiela správy s dátami do schránky konzumenta. Ten posiela prázdne správy do schránky producenta.
  - Správy ktoré ešte neboli prijaté zostávajú v schránke.

Synchronizácia prenosu

- Operácie vysielania a prijímania môžu byť blokujúce, alebo neblokujúce.
- **Blokujúce prijímanie** – ak operácia prijímania predchádza vyslanie správy ktorá má byť prijatá, zostane zablokovaná (bude čakať).
- **Blokujúce vysielanie** – operácia vysielania správy zablokuje volajúci proces, až kým prijímajúci proces nezavolá zodpovedajúcu operáciu *receive*.
- Neblokujúca operácia sa vráti hneď, bez ohľadu na druhú stranu.
  - Návratová hodnota môže indikovať stav operácie.
- Typicky sa využíva najmä:
  - Blokujúci *receive* a neblokujúci *send*.
  - Blokujúci *receive* a blokujúci *send* → tzv. **rendezvous** (vzájomné čakanie).
  - Neblokujúci *receive* a neblokujúci *send*.

Ďalšie problémy zasielania správ

- Veľkosť správ.
  - Pevná vs. premenlivá.
- Konverzia formátu prenášaných dát.
  - Napr. *little endian* vs. *big endian*.
- Poradie doručovania a prioritizácia správ.
- Spoľahlivosť prenosu.
  - Preusporiadanie správ pri prenose sieťou.
  - Potvrdzovanie doručenia. Znovuposielanie.
- Autentifikácia.
  - Kto smie prijímať správy.

Sokety

Soket

- Koncový bod komunikácie.
  - soket = (adresa, port)
  - spojenie = (protokol, soket_zdroj, soket_ciel)
- Doména soketu: Unix (IPC) **`AF_UNIX`**, Internet (network) **`AF_INET`**, **`AF_INET6`**, ...
- Typ soketu: **`SOCK_DGRAM`**, **`SOCK_STREAM`**, **`SOCK_RAW`**.
- Protokol: **`IPPROTO_TCP`**, **`IPPROTO_UDP`**, väčšinou stačí 0.
- Soket je z programátorského hľadiska číslo (typ int), podobne ako deskriptor súboru, alebo dátovod.
- Podobne sa aj používa: **`read()`**, **`write()`**, **`close()`**, ...

Operácie – spoločné

- Vytvorenie soketu, vráti deskriptor (číslo)
  - `int socket(int family, int type, int protocol);`
- Zviazanie soketu s lokálnou adresou a portom
  - `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - Argumenty udávajú lokálnu (zdrojovú) adresu.
  - Klientská aplikácia túto operáciu nemusí použiť.
  - Zdrojový port v tom prípade nastaví systém.

Operácie – server

- Nastaviť soket na čakanie na prichádzajúce spojenia (*listening*)
  - `int listen(int socket_file_descriptor, int backlog);`
- Prijatie prichádzajúceho spojenia (*connected*)
  - `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
  - Čaká (zablokuje volajúci proces), kým nepríde spojenie. Soket môže byť nastavený ako neblokujúci. Je možné použiť aj volanie `select()`.
  - Vráti deskriptor na nový soket v stave *connected*, z ktorého je možné čítať a zapisovať doň.
  - Pôvodný soket zostane nezmenený, v stave *listening*, a naďalej čaká na nové spojenia. Bežne sa po `accept()` robí `fork()`.

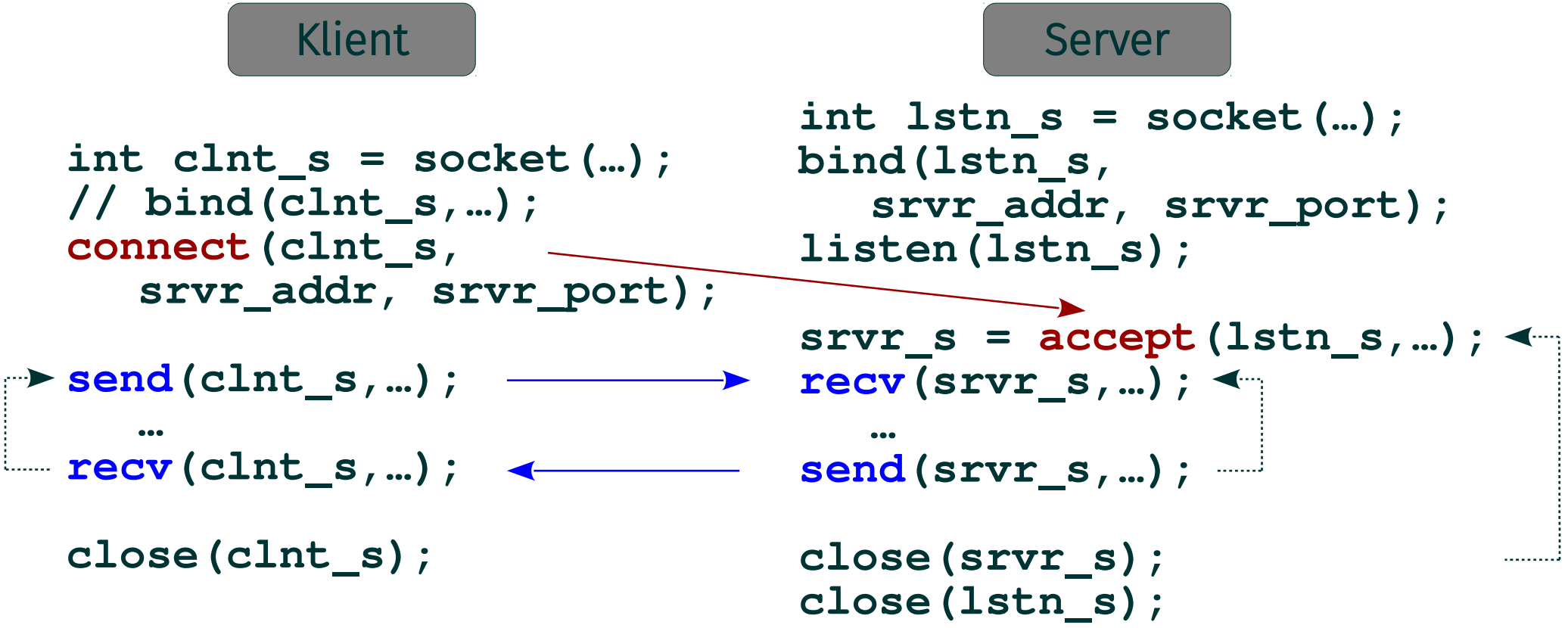
Operácie – klient

- Nadviazanie spojenia
  - `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
- Argumenty udávajú cieľovú adresu (prípadne port), teda adresu servera.
- Po tom, ako sa na strane servera ukončí operácia **`accept()`** a na strane klienta **`connect()`**, môžu byť prenášané dáta.
- Soket je možné použiť na čítanie a zápis.

Operácie – prenos dát a iné

- So soketom je možné použiť bežné operácie **`read()`** a **`write()`**.
- Pre datagramy:
  - **`send()`**, **`sendto()`**, **`sendmsg()`**, **`recv()`**, **`recvfrom()`**, **`recvmsg()`**
- Ukončenie spojenia
  - **`shutdown()`** - čítanie, zápis, oboje; **`close()`** - zrušenie soketu.
- Čítanie a nastavenie vlastností soketu:
  - **`getsockopt()`**, **`setsockopt()`**.
- Pomocné funkcie: **`htonl()`**, **`ntohl()`**, ...

Príklad: Komunikácie cez sokety



- **clnt_s** a **srvr_s** tvoria spolu spojenie na prenos dát (*connected*).
- **lstn_s** – soket, na ktorom server čaká na nové spojenia (*listening*).
- **srvr_addr**, **srvr_port** – adresa servera na ktorú sa klient pripája.
- Operácia **accept** čaká na príchod spojenia, potom vráti nový soket.

Signály

Signály

- Signál je asynchrónne doručená informácia o tom, že nastala nejaká udalosť.
- Podobajú sa prerušeniam. Generujú sa programovo.
  - Po doručení signálu procesu sa jeho vykonávanie preruší, kým sa signál neobslúži. Potom pokračuje tam, kde bol prerušený.
- Signál typicky posiela jeden proces inému.
  - V niektorých prípadoch môže signál procesu zaslať aj OS.
- Signálov je viacero typov, rozlišujú sa číslom.
- Každý typ signálu má priradenú akciu, ktorá sa má vykonať po jeho doručení.
  - Napríklad ukončenie procesu, alebo vykonanie zvolenej funkcie (*handler*).
- So signálom môže byť spojené aj malé množstvo prenášaných dát.

Zaslanie signálu

- Systémové volanie pre zaslanie signálu (komu, aký):
  - `int kill(pid_t pid, int sig);`
- Proces posielajúci signál na to musí mať práva
  - len vlastník, alebo *root*.
- Proces alebo vlákno môže poslať signál sám sebe:
  - `int raise(int sig);`
- Na signál je možné čakať (v zablokovanom/uspanom stave):
  - `pause(), sigsuspend(), sigtimedwait()`

Blokovanie signálu

- Každý proces má definovanú masku (*signal mask*) a jednotlivé signály môžu byť (dočasne) blokované.
  - SIGKILL a SIGSTOP nemôžu byť blokované (ignorujú masku).
- Ak príde signál ktorý má byť blokovaný, nestratí sa. Je pridaný do radu signálov čakajúcich na obslúženie (*pending signal*).
- Signál ktorý sa práve obsluhuje je automaticky blokovaný.
  - Uľahčuje to písanie funkcií obsluhujúcich signály.
  - Ak však jedna funkcia (*handler*) obsluhuje rôzne signály, môže sa stať že bude počas svojho vykonávania zavolaná znova.
- Signál môže prísť aj keď proces vykonáva systémové volanie.
  - Toto môže byť prerušené a skončí sa s kódom (chybou) EINTR,
  - alebo môže byť reštartované po dokončení obsluhy signálu, ak bol *handler* nastavený s príznakom SA_RESTART. Toto však nie je možné pre všetky systémové volania.

Signály generované jadrom

- Signál môže v istých prípadoch vygenerovať aj jadro.
  - Väčšinou je to reakcia OS na výnimku, čiže chybový stav HW, ktorý spôsobil proces svojím vykonávaním,
  - alebo na situáciu keď proces nemôže pokračovať ďalej.
- **SIGSEGV** – *segmentation violation*
  - väčšinou pri prístupe do pamäti na zlú (neplatnú, neexistujúcu, nealokovanú) adresu.
- **SIGILL** – *illegal instruction*
  - z pamäte inštrukcií procesor prečítal hodnotu ktorá nezodpovedá žiadnej inštrukcii.
- **SIGBUS** – *bus error*,
- **SIGFPE** – *floating point exception*,
- **SIGPIPE** – proces zapisuje do zatvoreného dátovodu,
- **SIGSTOP** – pri zápise do plného *buffer*-a dátovodu,
- **SIGCHLD** – rodičovský proces dostane tento signál, ak jeho potomok zmení stav.
  - Ak skončí alebo je zablokovaný; ak dostane SIGSTOP, SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU.

Akcie signálu

- Proces môže na jednotlivé signály reagovať rôzne.
- Bežné akcie sú:
  - Term – *terminate*, ukončenie procesu,
  - Ign – *ignore*, ignorovanie signálu, bude zahodený,
    - Ignorovanie signálov SIGFPE, SIGILL, SIGSEGV, SIGBUS vedie k nedefinovanému správaniu procesu.
  - Core – ukončenie procesu a vytvorenie súboru s obrazom jeho pamäte (*core dump*),
  - Stop – zastavenie volajúceho procesu (prejde do stavu T),
  - Cont – pokračovanie procesu, ak bol zastavený.

Obsluha signálu

- Proces môže na jednotlivé signály nastaviť vlastnú obslužnú rutinu, ktorá sa vykoná po príchode daného signálu.
- Systémové volanie:
  - `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
- Štruktúra *sigaction* obsahuje aj pointer na novú obslužnú funkciu.
- Obslužná funkcia (*signal handler*) musí byť **reentrantná**. Ak nie je nastavené inak, vykonáva sa so zásobníkom procesu prijímajúceho signál.
- Signály SIGKILL, SIGSTOP nemôžu byť procesom zachytené, blokované alebo ignorované a nie je možné zmeniť ich štandardnú akciu.

Sigaction

```
int struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

- Tretí argument pre `sa_sigaction()` je na linux-e typu `ucontext_t`.
- Umožňuje napríklad zistiť hodnoty registrov v čase príchodu signálov, ktoré sú dôsledkom výnimiek.
  - Napríklad, ak štruktúru nazveme *context*, register PC (resp. IP) je dostupný ako
  - `((ucontext_t*)context)->uc_mcontext.gregs[REG_PC]`
- Prečítať man 2 sigaction, man 7 signal.

Siginfo

```
siginfo_t {
    int si_signo;       /* Signal number */
    int si_errno;       /* An errno value */
    int si_code;        /* Signal code */
    int si_trapno;      /* Trap number that caused
                           hardware-generated signal
                           (unused on most architectures) */
    pid_t si_pid;       /* Sending process ID */
    uid_t si_uid;       /* Real user ID of sending process */
    int si_status;      /* Exit value or signal */
    clock_t si_utime;   /* User time consumed */
    clock_t si_stime;   /* System time consumed */
    sigval_t si_value;  /* Signal value */
    int si_int;         /* POSIX.1b signal */
    void *si_ptr;       /* POSIX.1b signal */
    int si_overrun;     /* Timer overrun count; POSIX.1b timers */
    int si_timerid;     /* Timer ID; POSIX.1b timers */
    void *si_addr;      /* Memory location which caused fault */
    long si_band;       /* Band event (was int in
                           glibc 2.3.2 and earlier) */
    int si_fd;          /* File descriptor */
    short si_addr_lsb;  /* Least significant bit of address
                           (since Linux 2.6.32) */
}
```

Zoznam signálov

```
$kill -l

 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Použitie signálov

- Používajú sa hlavne na synchronizáciu procesov.
- Funkcia **`alarm()`** zabezpečí, že volajúci proces dostane signál SIGALRM po uplynutí zadaného času.
  - **`unsigned int alarm(unsigned int seconds);`**
  - Funkcia **`sleep()`** môže byť takto implementovaná.
- Signál USR1 pre príkaz **`dd`** spôsobí, že vypíše štatistiky (na *stderr*) a pokračuje.
- Signál HUP pre démon **`sshd`** spôsobí, že znovu načíta svoj konfiguračný súbor.

Súbežné operácie

Blokujúce a neblokujúce operácie

- **Blokujúca operácia** sa po zavolaní nevráti, pokiaľ nie je ukončená.
  - I/O operácie; čakanie na dáta z disku/na disk, zasielanie správ cez sieť; alokácia pamäte (), etc ...
- Ak požadované dáta nie sú pripravené, operácia (read/recv) blokuje vykonávanie procesu, pokiaľ nebudú.
- Pri čakaní nespotrebováva čas, proces však stojí (nevie ako dlho) a nemôže vykonávať nič iné.
- Akonáhle sú dáta k dispozícii a operácia môže byť dokončená, OS opäť naplánuje vykonávanie tohoto procesu, ktorý operáciu vyvolal.
- **Neblokujúca operácia** - ak požadovanú operáciu nie je možné vykonať, vráti chybový kód, vykonávanie však pokračuje ďalej,
  - operáciu je možné opakovať neskôr znova = polling,
  - opakované vykonanie operácie keď dáta nie sú pripravené je zbytočné (neužitočná réžia),
  - umožňuje však vykonávať počas čakania inú činnosť.
- Riešenie – viacvláknové procesy
  - komunikujúce vlákno môže byť blokované pokiaľ neprídu dáta, ostatné môžu zatiaľ pracovať.

Asynchrónne operácie

- Ich vykonanie nie je závislé (resp. odvodené) od vykonávania iných operácií,
  - napríklad obslužné rutiny (*handler*, *callback*) prerušení, výnimiek, signálov.
- Operácia (vykonávaná v kontexte nejakého procesu) je asynchrónna (vzhľadom k tomuto procesu) ak jej vykonanie (spustenie) od tohoto procesu nezávisí.
- Na asynchrónne operácie však môžu byť kladené zvláštne implementačné nároky – **reentrantnosť**.
- To, či je operácia blokujúca alebo nie, závisí len od jej implementácie.
  - Pohľadom na implementáciu vieme vždy rozhodnúť, či je operácia (potenciálne) blokujúca, alebo nie.
  - V praktických prípadoch je možné konkrétne správanie nastaviť príznakom.
- To, či je operácia asynchrónna je otázkou spôsobu jej použitia (nie jej vnútornej implementácie)
  - ide o to kedy, resp. ako, dochádza k spusteniu operácie.

Reentrantnosť vs. thread-safety

- **Reentrantnosť** (*reentrancy, signal-safety*)
  - funkcia môže byť počas vykonávania zavolaná znova, bez ovplyvnenia korektnosti výsledkov,
  - funkcia teda musí byť bez následkov spustiteľná viackrát, **v rámci toho istého kontextu**.
- Bezpečnosť vykonávania vláknami (*thread-safety*)
  - funkcia môže byť vykonávaná súbežne viacerými vláknami, bez ovplyvnenia správnosti výsledkov,
  - musí byť spustiteľná viackrát súbežne, **v rôznych kontextoch**.
- Vlastnosť "*thread-safety*" sa dá zabezpečiť doplnením synchronizačných mechanizmov pri prístupe k spoločným dátam (zmenou implementácie).
  - Použitie synchronizačných funkcií v reentrantných funkciách môže viesť k uviaznutiu.
- Pre zabezpečenie reentrantnosti môže byť potrebné zmeniť nielen implementáciu, ale aj rozhranie funkcie.
- Reentrantnosť je silnejšia požiadavka, implikuje *thread-safety*, nie naopak.

Požiadavky reentrantnosti

- Nutné podmienky kladené na reentrantnú funkciu:
  1) neobsahuje globálne (nekonštantné) dáta,
  2) nevracia pointer na statické nekonštantné dáta,
  3) pracuje len s dátami od volajúceho,
  4) neblokuje prístup k jedinečnému zdroju (singleton),
  5) nevolá iné nereentrantné funkcie.
- Reentrantné funkcie je možné používať v obsluhe signálov.
  - Napríklad write() áno, printf() nie.
  - Nie je možné použiť nič čo volá malloc()/free().

Dátovody (nepomenované)

Dátovody

- Dátovod (*pipe*) predstavuje jednosmerný kanál pre komunikáciu medzi dvoma procesmi.
- Má dva konce reprezentované deskriptormi súborov.
- Volanie **`pipe()`** alokuje *buffer* v jadre a vráti dva deskriptory súborov, jeden pre zápis, druhý pre čítanie.
  - Oba však vráti tomu istému, hoci každý koniec by mal byť použitý iným procesom (jeden zapisuje, druhý číta).
  - Ako dostať deskriptor dátovodu do iného procesu?
- Dáta ktoré sa do dátovodu zapíšu zostanú uložené v jadre až kým nie sú z druhého konca prečítané.
- Pre zápis a čítanie sa používajú volania **`write()`** a **`read()`**, ako na súbory.
- Dátovod má konečnú kapacitu. Operácie čítania a zápisu sa môžu zablokovať.

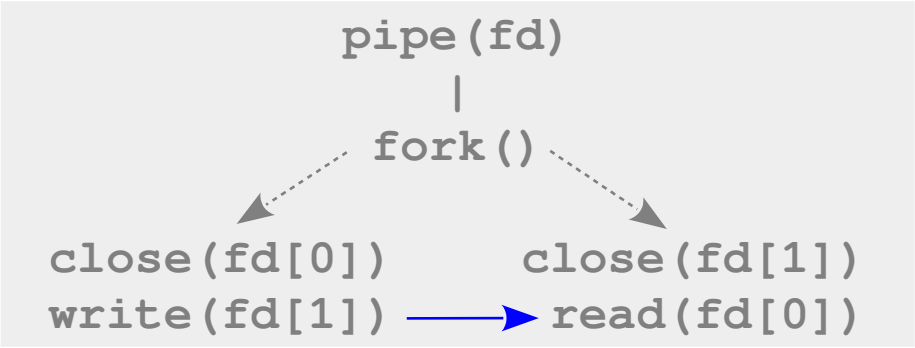
Vytvorenie dátovodu

- Ak proces vytvorí dátovod, má otvorené oba jeho konce.
  - Volanie `pipe()` vytvorí záznamy priamo v tabuľke otvorených súborov a vráti ich deskriptory (indexy).
  - Neexistujú však názvy týchto súborov, ktoré by bolo možné použiť na ich otvorenie (získanie deskriptora) v inom procese.
- Dátovody vytvorené volaním `pipe()` sú **nepomenované** (anonymné) a preto je možné ich použiť len medzi rodičom a potomkom.
  - Poznámka: Existujú aj pomenované dátovody reprezentované špeciálnym súborom, `mkfifo()`.
- Pri vytvorení nového procesu sa skopíruje tabuľka otvorených súborov a teda aj nový proces bude mať otvorené oba tieto konce.
- Každý z dvojice procesov zavrie nepotrebný koniec.
  - Proces, ktorý bude zapisovať zavrie čítací koniec a naopak.
  - Bez toho by nebolo možné zistiť či už proces na druhej strane skončil a mohlo by dôjsť k čakaniu na dáta, ktoré neprídu.

Zatvorenie nepotrebného konca dátovodu

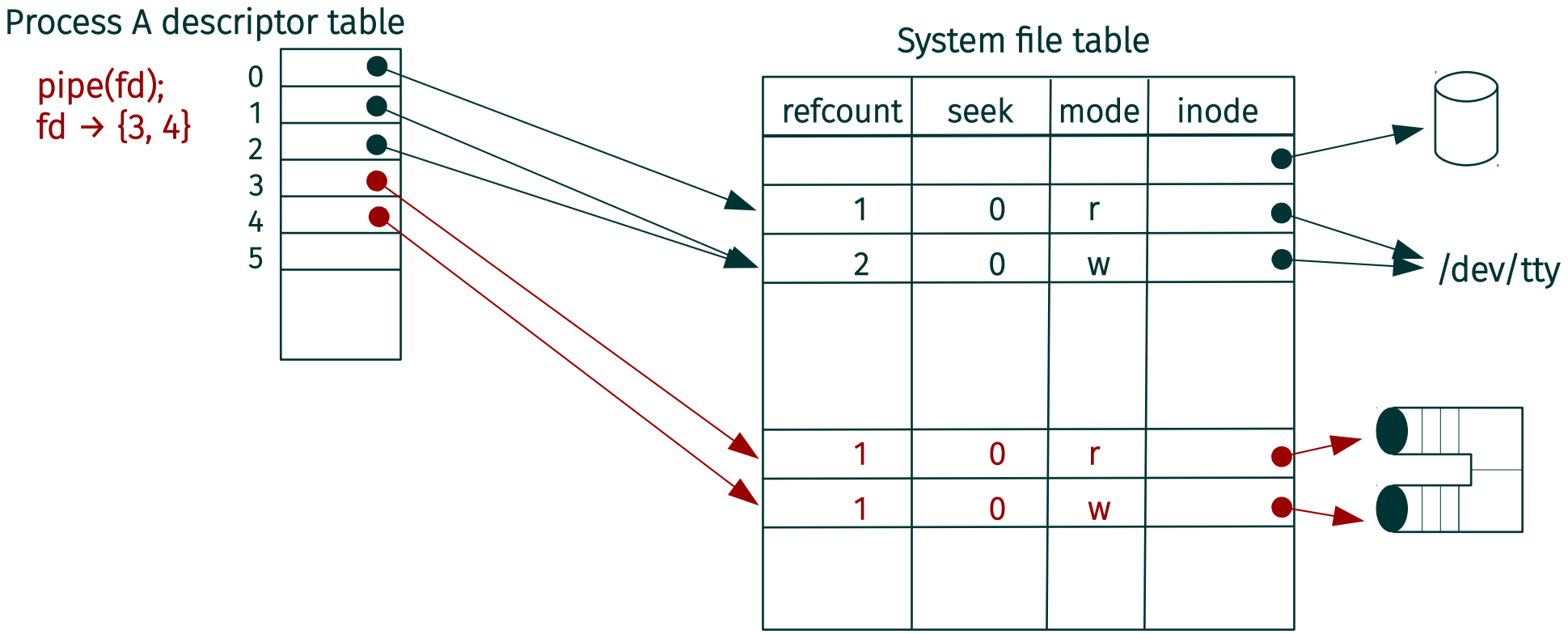
- Každý z dvojice procesov zavrie nepotrebný koniec.
  - Proces, ktorý bude zapisovať zavrie čítací koniec a naopak.
- Ak proces zapíše do dátovodu ktorého čítací koniec nie je otvorený, napríklad čítajúci proces sa už ukončil, zapisujúci proces dostane signál (SIGPIPE).
- Ak by zapisujúci proces nechal otvorený aj čítací koniec dátovodu, v prípade ukončenia čítacieho procesu by OS nevedel, že má pri zápise poslať SIGPIPE.
  - Po naplnení buffera by zostal zapisujúci proces zablokovaný. A keďže by nemohol zároveň čítať a tak uvoľniť miesto, nastalo by **uviaznutie**.
- Operácia čítania z dátovodu so zatvoreným zapisovacím koncom vráti EOF.
  - Čítajúci proces môže zistiť, že čítanie môže ukončiť.
- Ak by zapisovací koniec zostal naďalej otvorený v čítajúcom procese, čítanie by zostalo zablokované, hoci žiadne dáta už nebudú dostupné.
  - Keďže čítajúci proces je zablokovaný, nemôže zároveň zapísať → **uviaznutie**.

Príklad: Použitie dátovodu

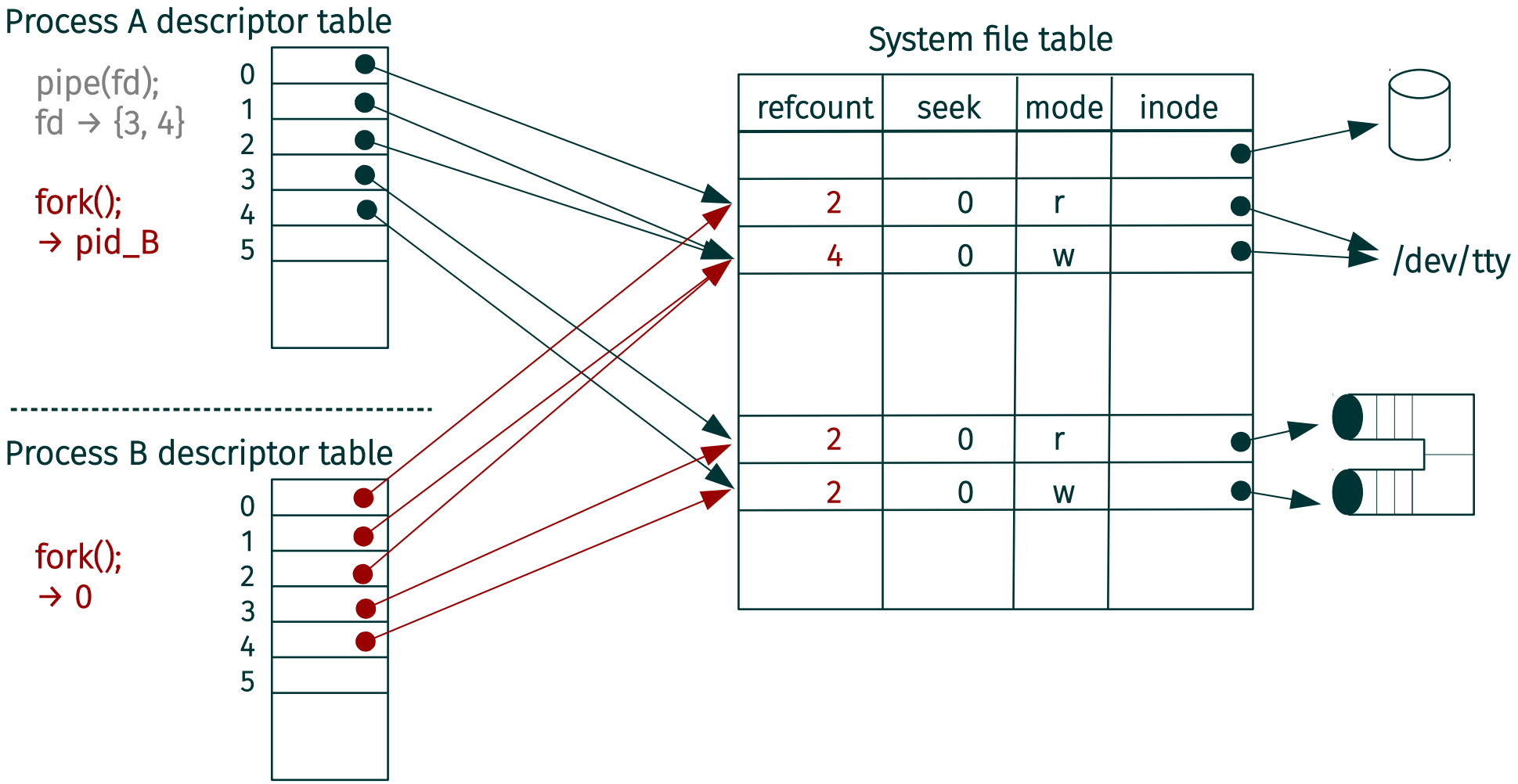



```
int fd[2], child_pid, n;

pipe(fd);

child_pid = fork();
if (child_pid == 0)
{                                        /* child process   */
   char buf[SIZE];
   close(fd[1]);                         /* close write end */
   n = read(fd[0], &buf, SIZE);          /* read from pipe  */
   printf("%s", buf);
}
else
{                                        /* parent process  */
   char msg[] = "Hello from parent!\n";
   close(fd[0]);                         /* close read end  */
   write(fd[1], &msg, sizeof(msg));      /* to pipe         */
   wait();                               /* wait for child  */
}
```

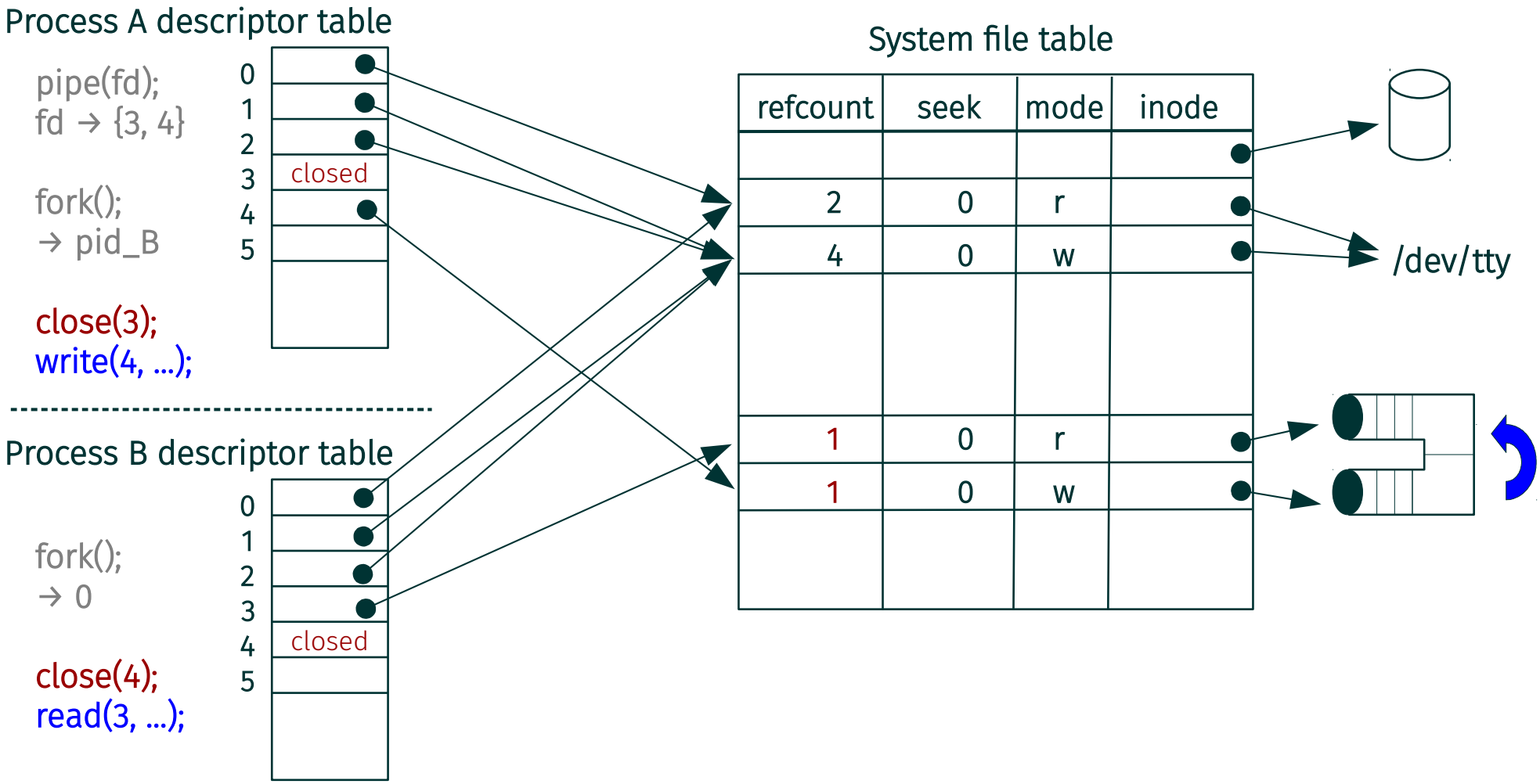
Príklad: Použitie dátovodu, tabuľka súborov (pipe)



Príklad: Použitie dátovodu, tabuľka súborov (fork)



Príklad: Použitie dátovodu, tabuľka súborov (close)



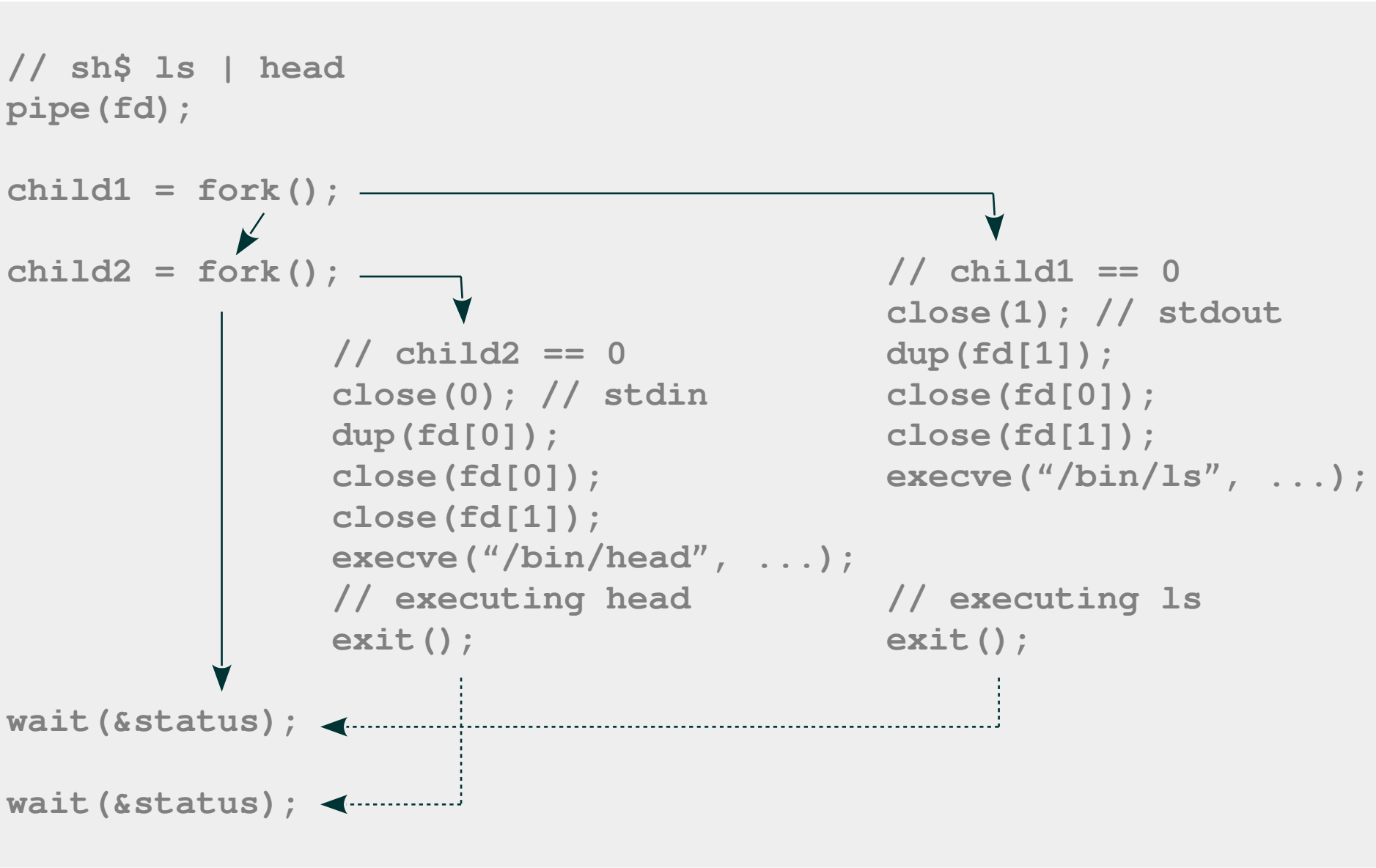
Štandardný vstup a výstup procesu

- Typické použitie dátovodu predstavuje napríklad spojenie štandardného výstupu jedného procesu so štandardným vstupom iného.
  - Ako sa toto presmerovanie vytvorí?
  - Ako sa vytvorí dátovod medzi dvoma novými procesmi?
  - Napríklad, čo sa stane, ak *shell* vykoná `ls | head`?
- Štandardne má každý proces otvorené prvé tri deskriptory súborov pre terminál (je tiež reprezentovaný súborom).
  - *0*, štandardný vstup, pre čítanie vstupných dát na spracovanie,
  - *1*, štandardný výstup, pre výpis výsledkov,
  - *2*, chybový výstup, pre výpis pomocných a chybových hlásení.
- Pri prihlásení používateľa ich otvorí proces inicializujúci terminál a všetky nasledujúce procesy ich už zdedia. Je to dôsledok hierarchie procesov.

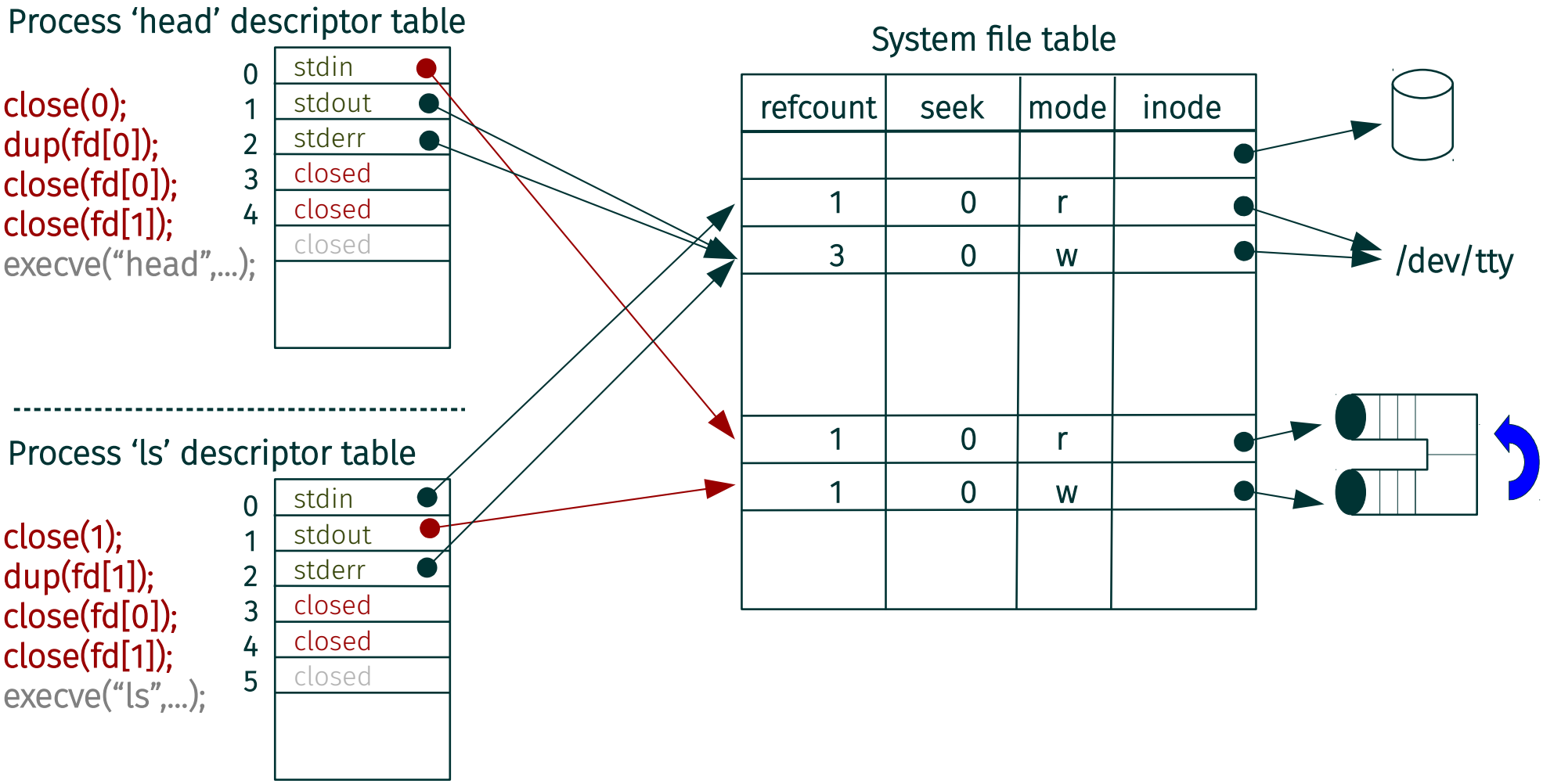
Duplikácia deskriptora

- Súčasťou PCB každého procesu je aj tabuľka otvorených súborov.
  - Linux: položka **`files`** v **`task_struct`**.
- Deskriptor je indexom do tejto tabuľky.
- Každé úspešné volanie **`open()`** pridá jeden nový riadok (a vráti deskriptor).
- Volanie **`pipe()`** pridá dva nové riadky (a vráti dva deskriptory).
- Pri vytváraní nového záznamu operačný systém spravidla použije prvý voľný deskriptor.
- Volanie **`dup(int old)`** skopíruje daný deskriptor na prvý voľný a vráti jeho číslo.
  - Oba deskriptory zodpovedajú tomu istému otvorenému súboru.
- Operácia **`dup2(int old, int new)`** skopíruje deskriptor **`old`** na pozíciu **`new`**. Ak **`new`** nebol voľný, najskôr ho zatvorí.
- Duplikácia sa využíva pri implementácii presmerovania.

Presmerovanie štandardného vstupu a výstupu



Presmerovanie štandardného vstupu a výstupu



To je zatiaľ všetko