B4B35OSY: Operační systémy

Lekce 5. Meziprocesní komunikace

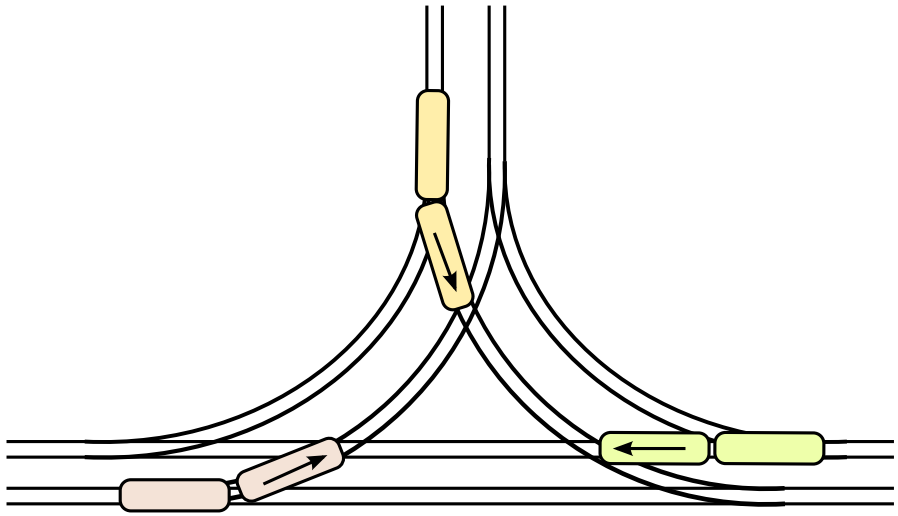
Petr Štěpán
stepan@fel.cvut.cz

September 10, 2020

Outline

Deadlock v životě

- Karlovo náměstí – tramvaje většinou jedou jen dopředu
- Zdroje jsou křížení tramvajových kolejí
- Aby tramvaj projela, musí použít dva zdroje na své cestě

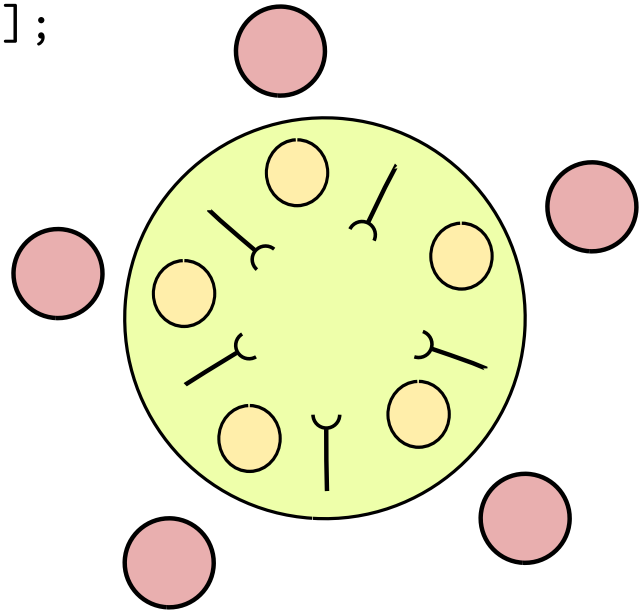


- “It takes money to make money” – anglické přísloví
- K získání kvalitního zaměstnání je potřeba kvalitní praxe, kvalitní praxi získáte pouze v kvalitním zaměstnání

Večeřící filozofové

Sdílená data

```
/* Inicializace */
  semaphore chopStick[ ] = new Semaphore[5];
  for(i=0; i<5; i++) chopStick[i] = 1;
/* Implementace filozofa i: */
do  {
    chopStick[i].wait;
    chopStick[(i+1) % 5].wait;
    eating();          // Teď jí
    chopStick[i].signal;
    chopStick[(i+1) % 5].signal;
    thinking();        //  A teď přemýšlí
} while (TRUE) ;
```



- Možné ochrany proti uváznutí
  - Zrušení symetrie úlohy
  - Jeden filozof bude levák a ostatní praváci (levák zvedá vidličky opačně)
  - Jídlo se n filozofům podává v jídelně s n+1 židlemi
  - Filozof smí uchopit vidličku jen, když jsou obě volné a uchopí obě najednou
  - Příklad obecnějšího řešení – tzv. skupinového zamykání prostředků

Časově závislé chyby

- Příklad časově závislé chyby
- Procesy P_1 a P_2 spolupracují za použití mutexů A a B

Proces P_1

```
pthread_mutex_lock(&mutex_A);
pthread_mutex_lock(&mutex_B);
a+=b;
pthread_mutex_unlock(&mutex_B);
pthread_mutex_unlock(&mutex_A);
```

Proces P_2

```
pthread_mutex_lock(&mutex_B);
pthread_mutex_lock(&mutex_A);
b+=a;
pthread_mutex_unlock(&mutex_A);
pthread_mutex_unlock(&mutex_B);
```

- Deadlock nastane pouze v situaci, že proces P_1 získá mutex A a proces P_2 získá mutex B
- Nebezpečnost takových chyb je v tom, že vznikají jen zřídkakdy za náhodné souhry okolností
- Jsou tudíž fakticky neodladitelné

Coffmanovi podmínky

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

1. Vzájemné vyloučení, Mutual Exclusion
   - sdílený zdroj může v jednom okamžiku používat nejvýše jeden proces
2. Postupné uplatňování požadavků, Hold and Wait
   - proces vlastnící alespoň jeden zdroj potřebuje další, ale ten je vlastněn jiným procesem, v důsledku čehož bude čekat na jeho uvolnění
3. Nepřipouští se odnímání zdrojů, No preemption
   - zdroj může uvolnit pouze proces, který ho vlastní, a to dobrovolně, když již zdroj nepotřebuje
4. Zacyklení požadavků, Circular wait
   - Existuje posloupnost čekajících procesů $P_0, P_1, \ldots, P_k, P_0$ takových, že P_0 čeká na uvolnění zdroje drženého P_1, P_1 čeká na uvolnění zdroje drženého $P_2, \ldots, P_{k-1}$ čeká na uvolnění zdroje drženého P_k, a P_k čeká na uvolnění zdroje drženého P_0.
   - V případě jednoinstančních zdrojů splnění této podmínky značí, že k uváznutí již došlo.

Co dělat?

Existují čtyři přístupy

- Zcela ignorovat hrozbu uváznutí
  - Pštrosí algoritmus – strč hlavu do písku a předstírej, že se nic neděje
  - Používá většina současných OS včetně většiny implementací UNIXů
  - Linux se snaží o prevenic deadlocku uvnitř jádra, neovlivňuje ale použití deadlocků v uživatelských programech
- Prevence uváznutí
  - Pokusit se přijmout taková opatření, aby se uváznutí stalo vysoce nepravděpodobným
  - Ale pozor! Pokud víme, že k uváznutí může dojít, ale jen s malou pravděpodobností, dojde k němu, když se to hodí nejméně
- Vyhýbání se uváznutí
  - Zajistit, že k uváznutí nikdy nedojde
  - Prostředek se nepřidělí, pokud by hrozilo uváznutí
  - hrozí stárnutí
- Detekce uváznutí a následná obnova
  - Uváznutí se připustí, detekuje se jeho vznik a zajistí se obnova stavu před uváznutím

Prevence uváznutí

- Konzervativní politikou se omezuje přidělování prostředků
- Přímá metoda – plánovat procesy tak, aby nevznikl cyklus v přidělování prostředků
  - Vzniku cyklu se brání tak, že zdroje jsou očíslovány a procesy je smějí alokovat pouze ve vzrůstajícím pořadí čísel zdrojů
  - Nerealistické – zdroje vznikají a zanikají dynamicky
  - Často ale stačí uvažovat třídy zdrojů (LOCKDEP v jádře Linuxu – podobné jako alg. vyhýbání se uváznutí dále)
- Nepřímé metody (narušení některé Coffmanovy podmínky)
  - Eliminace potřeby vzájemného vyloučení
    - Nepoužívat sdílené zdroje, virtualizace (spooling) periferií
    - Mnoho činností však sdílení nezbytně potřebuje ke své funkci
  - Eliminace postupného uplatňování požadavků
    - Proces, který požaduje nějaký zdroj, nesmí dosud žádný zdroj vlastnit
    - Všechny prostředky, které bude kdy potřebovat, musí získat naráz
    - Nízké využití zdrojů
  - Připustit násilné odnímání přidělených zdrojů (preempce zdrojů)
    - Procesu žádajícímu o další zdroj je dosud vlastněný prostředek odňat
    - To může být velmi riskantní – zdroj byl již modifikován
    - Proces je reaktivován, až když jsou všechny potřebné prostředky volné
    - Metoda inkrementálního zjišťování požadavků na zdroje – nízká průchodnost
    - Metody prevence uváznutí aplikované za běhu způsobí výrazný pokles průchodnosti systému

Vyhýbání se uváznutí

- Základní problém:
  - Systém musí mít dostatečné apriorní informace o požadavcích procesů na zdroje
- Nejčastěji se požaduje, aby každý proces udal maxima počtu prostředků každého typu, které bude za svého běhu požadovat
- Algoritmus:
  - Dynamicky se zjišťuje, zda stav subsystému přidělování zdrojů zaručuje, že se procesy v žádném případě nedostanou do cyklu
  - Stav systému přidělování zdrojů je popsán
    - Počtem dostupných a přidělených zdrojů každého typu a
    - Maximem očekávaných žádostí procesů
  - Stav může být bezpečný nebo nebezpečný

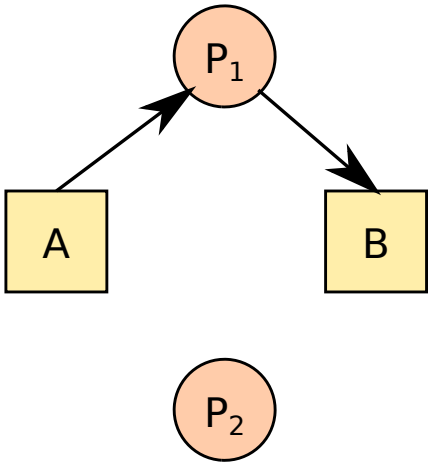
Vyhýbání se uváznutí

- Systém je v bezpečném stavu, existuje-li „bezpečná posloupnost procesů“
  - Posloupnost procesů P_0, P_1, ... , P_n je bezpečná, pokud požadavky každého P_i lze uspokojit právě volnými zdroji a zdroji vlastněnými všemi P_k , $k < i$
    - Pokud nejsou zdroje požadované procesem P_i volné, pak P_i bude čekat dokud se všechny P_k nedokončí a nevrátí přidělené zdroje
    - Když P_{i-1} skončí, jeho zdroje může získat P_i, proběhnout a jím vrácené zdroje může získat P_{i+1}, atd.
- Je-li systém v bezpečném stavu (safe state) k uváznutí nemůže dojít. Ve stavu, který není bezpečný (unsafe state), přechod do uváznutí hrozí
- Vyhýbání se uváznutí znamená:
  - Plánovat procesy tak, aby systém byl stále v bezpečném stavu
  - Nespouštět procesy, které by systém z bezpečného stavu mohly vyvést
  - Nedopustit potenciálně nebezpečné přidělení prostředku

Model uváznutí

RAG – Resource allocation graph

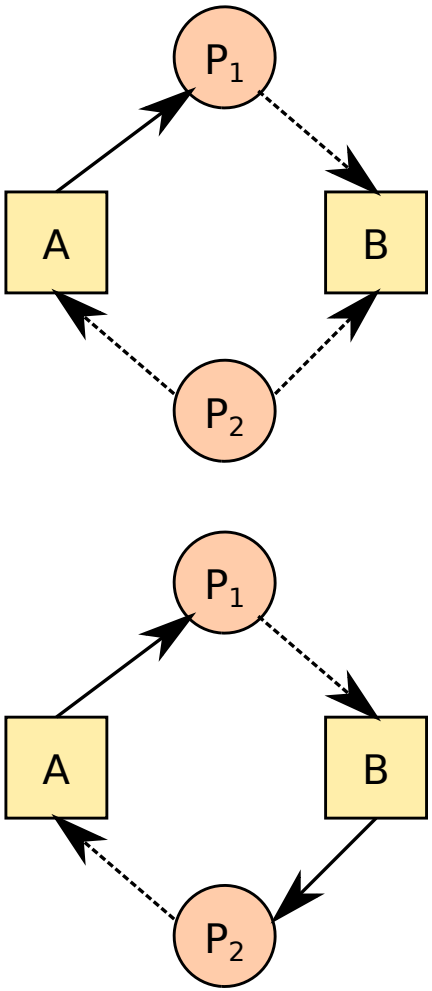
- Typy prostředků (zdrojů) R_1, R_2, ... R_m
  - např. datové struktury, V/V zařízení, ...
- Každý typ prostředku R_i má W_i instancí
  - např. máme 4 síťové karty a 2 CD mechaniky
  - často W_i = 1 tzv. jednoinstanční prostředky – např. mutex
- Každý proces používá potřebné zdroje podle schématu
  - žádost – request, wait
  - používání prostředku po konečnou dobu (kritická sekce)
  - uvolnění (navrácení) – release, signal
- žádost o zdroj značí hrana od procesu P_i ke zdroji R_j
- přidělený zdroj značí hrana od zdroje R_j k procesu P_i



Vyhýbání uváznutí – jednoinstanční zdroje

Potřebujeme znát budoucnost:

- Do RAG se zavede „nároková hrana“
  - Nároková hrana $P_i \rightarrow R_j$ značí, že někdy v budoucnu bude proces P_i požadovat zdroj R_j
  - V RAG hrana vede stejným směrem jako požadavek na přidělení, avšak kreslí se čárkovaně
- Nároková hrana se v okamžiku vzniku žádosti o přidělení převede na požadavkovou hranu
  - Když proces zdroj získá, požadavková hrana se změní na hranu přidělení
  - Když proces zdroj vrátí, hrana přidělení se změní na požadavkovou hranu
- Převod požadavkové hrany v hranu přidělení nesmí v RAG vytvořit cyklus (včetně uvažování nárokových hran)
  - LOCKDEP v Linuxu (systém běží cca $10\times$ pomaleji)



Bankéřský algoritmus

- Chování odpovědného bankéře:
  - Klienti žádají o půjčky do určitého limitu
  - Bankéř ví, že ne všichni klienti budou svůj limit čerpat současně a že bude půjčovat klientům prostředky postupně
  - Všichni klienti v jistém okamžiku svého limitu dosáhnou, avšak nikoliv současně
  - Po dosažení přislíbeného limitu klient svůj dluh v konečném čase vrátí
  - Příklad:
    - Ačkoliv bankéř ví, že všichni klienti budou dohromady potřebovat 22 jednotek a na celou transakci má jen 10 jednotek, je možné uspokojit postupně všechny klienty

Bankéřský algoritmus

- Zákazníci přicházející do banky pro úvěr předem deklarují maximální výši, kterou si budou kdy chtít půjčit
- Úvěry v konečném čase splácí
- Bankéř úvěr neposkytne, pokud si není jist, že bude moci uspokojit všechny zákazníky (vždy alespoň jednomu zákazníku bude moci půjčit všechny peníze a zákazník je pak vrátí)
- Analogie
  - Zákazník = proces
  - Úvěr = přidělovaný prostředek
- Vlastnosti
  - Procesy musí deklarovat své potřeby předem
  - Proces požadující přidělení může být zablokován
  - Proces vrátí všechny přidělené zdroje v konečném čase
- Nikdy nedojde k uváznutí
  - Proces bude spuštěn jen, pokud bude možno uspokojit všechny jeho požadavky
  - Sub-optimální pesimistická strategie
  - Předpokládá se nejhorší případ

Bankéřský algoritmus

Datové struktury

- n ... počet procesů
- m ... počet typů zdrojů
- Vektor $available[m]$
  - $available[j] = k$ značí, že je k instancí zdroje typu R_j je volných
- Matice $max[n, m]$
  - povinná deklarace procesů maximálních požadavků
  - $max[i, j] = k$ znamená, že proces P_i bude během své činnosti požadovat až k instancí zdroje typu R_j
- Matice $allocated[n, m]$
  - $allocated[i, j] = k$ značí, že v daném okamžiku má proces P_i přiděleno k instancí zdroje typu R_j
- Matice $needed[n, m]$ ($needed[i, j] = max[i, j] - allocated[i, j]$)
  - $needed[i, j] = k$ říká, že v daném okamžiku procesu P_i chybí ještě k instancí zdroje typu R_j

Bankéřský algoritmus

Test bezpečnosti stavu

1. Inicializace
   - $work[m]$ a $finish[n]$ jsou pracovní vektory
   - Inicializujeme $work = available; finish[i] = false; i = 1, ..., n$
2. Najdi i, pro které platí $(finish[i] = false) \wedge (needed[i] \leq work[i])$
3. Pokud takové i neexistuje, jdi na krok 6
4. Simuluj dokončení procesu i
   - $work[i] = work[i] + allocated[i]; finish[i] = true;$
5. Pokračuj krokem 2
6. Pokud platí $finish[i] = true$ pro všechna i, pak stav systému je bezpečný

Bankéřský algoritmus

- Proces P_i formuje vektor request:
- $request[j] = k$ znamená, že proces P_i žádá o k instancí zdroje typu R_j
- if($request[j] \geq needed[i,j]$) proces nárokuje víc než bylo maximum na začátku;
- if($request[j] \geq available[j]$) zatím nelze splnit, je nutné počkat na uvolnění zdrojů, proces se uspí;
- Jinak otestuj přípustnost požadavku simulováním přidělení prostředku a pak ověříme bezpečnost stavu:
  - $available[j] = available[j] - request[j]$;
  - $allocated[i,j] = allocated[i,j] + request[j]$;
  - $needed[i,j] = needed[i,j] - request[j]$;
  - Spusť test bezpečnosti stavu
  - Je-li bezpečný, přiděl požadované zdroje
  - Není-li stav bezpečný, pak vrať úpravy „Akce 3“ a zablokuj proces P_i, neboť přidělení prostředků by způsobilo nebezpečí uváznutím

Bankéřský algoritmus – příklad

Test bezpečnosti stavu:

Prostředky na začátku:

A	B	C
10	6	6

Maximum:

	A	B	C
P_1	8	4	4
P_2	2	1	4
P_3	6	3	3
P_4	5	4	3

Alokace:

	A	B	C
P_1	3	1	1
P_2	1	0	1
P_3	2	1	0
P_4	1	3	1

Požadavek = Maximum – Alokace:

	A	B	C
P_1	5	3	3
P_2	1	1	3
P_3	4	2	3
P_4	4	1	2

Dostupné prostředky

A	B	C
3	1	3

Hledáme proces, který by mohl být dokončen, požadavek $\leq$ dostupné prostředky, pouze P_2.
Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
4	1	4

Opět hledáme proces, který by mohl být dokončen, pouze P_4.
Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
5	4	5

Opět hledáme proces, který by mohl být dokončen, nyní lze dokončit P_1 a P_3. Po dokončení procesu např. P_1, proces uvolní svoje prostředky:

A	B	C
8	5	6

Nyní lze dokončit P_3, po dokončení jsou dostupné prostředky stejné jako na začátku:

A	B	C
10	6	6

Všechny procesy skončily, stav je bezpečný.

Bankéřský algoritmus – příklad

Proces P_3 žádá o 2 zdroje A:

Prostředky na začátku:

A	B	C
10	6	6

Maximum:

	A	B	C
P_1	8	4	4
P_2	2	1	4
P_3	6	3	3
P_4	5	4	3

Alokace po simulovaném přidělení prostředků:

	A	B	C
P_1	3	1	1
P_2	1	0	1
P_3	4	1	0
P_4	1	3	1

Požadavek = Maximum − Alokace

	A	B	C
P_1	5	3	3
P_2	1	1	3
P_3	2	2	3
P_4	4	1	2

Dostupné prostředky:

A	B	C
1	1	3

Hledáme proces, který by mohl být dokončen, požadavek $\leq$ dostupné prostředky, pouze P_2.
Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
2	1	4

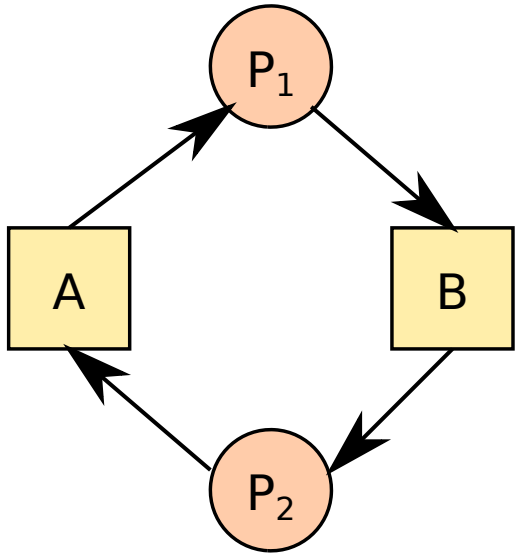
- Nyní nelze dokončit žádný proces, neboť dostupné prostředky nestačí pro dokončení žádného procesu (procesu P_1 chybí 3 zdroje A, 2 zdroje B; procesu P_3 chybí 1 zdroj B, procesu P_4 chybí 2 zdroje A).
- Stav není bezpečný – pokud by všechny procesy chtěli své maximum a teprve potom uvolnili zdroje, pak nastane deadlock.
  - O chování procesů nic nevíme, ale abychom se vyhnuli deadlocku musíme předpokládat nejhorší případ

Detekce uváznutí

- Strategie připouští vznik uváznutí:
- Uváznutí je třeba detekovat
- Vznikne-li uváznutí, aplikuje se plán obnovy systému
- Aplikuje se zejména v databázových systémech, kde je obnova dotazu běžná

Detekce uváznutí s RAG

- Případ jednoinstančního zdroje daného typu
  - Udržuje se čekací graf – uzly jsou procesy
  - Periodicky se provádí algoritmus hledající cykly
  - Algoritmus pro detekci cyklu v grafu má složitost $O(n^2)$, kde n je počet hran v grafu



Detekce uváznutí

- Případ více instancí zdrojů daného typu
  - n ... počet procesů
  - m ... počet typů zdrojů
  - Vektor $available[m]$
    - $available[j] = k$ značí, že je k instancí zdroje typu R_j je volných
  - Matice $allocated[n, m]$
    - $allocated[i, j] = k$ značí, že v daném okamžiku má proces P_i přiděleno k instancí zdroje typu R_j
  - Matice $request[n, m]$
    - Indikuje okamžité požadavky každého procesu:
    - $request[i, j] = k$ znamená, že proces P_i požaduje dalších k instancí zdroje typu R_j

Detekce uváznutí pro více-instanční problémy

Obdoba bankéřského algoritmu (nevíme budoucnost – neznáme maximum):

- $work[m]$ a $finish[n]$ jsou pracovní vektory
  1. Inicializujeme $work = available; finish[i] = false; i = 1, ..., n$
  2. Najdi i, pro které platí $(finish[i] = false) and (request[i] \leq work[i])$
  3. Pokud takové i neexistuje, jdi na krok 6
  4. Simuluj dokončení procesu i
     - $work[i] += allocated[i];$
     - $finish[i] = true;$
  5. Pokračuj krokem 2
  6. Pokud platí
     - $finish[i] = false$ pro některé i, pak v systému došlo k uváznutí.
     - Součástí cyklů ve WG jsou procesy P_i, kde $finish[i] == false$
- Algoritmus má složitost $O(m \cdot n^2)$, m a n mohou být veliká a algoritmus časově značně náročný

Detekce uváznutí

Kdy a jak často algoritmus vyvolávat? (Detekce je drahá)

- Jak často bude uváznutí vznikat?
- Kterých procesů se uváznutí týká a kolik jich „likvidovat“?
  - Minimálně jeden v každém disjunktním cyklu v RAG
  - Násilné ukončení všech uvázlých procesů – velmi tvrdé a nákladné
  - Násilně se ukončují dotčené procesy dokud cyklus nezmizí
    - Jak volit pořadí ukončování
    - Jak dlouho už proces běžel a kolik mu zbývá do dokončení
    - Je to proces interaktivní nebo dávkový (dávku lze snáze restartovat)
    - Cena zdrojů, které proces použil
    - Výběr oběti podle minimalizace ceny
    - Nebezpečí stárnutí
    - některý proces bude stále vybírán jako oběť

Uváznutí – shrnutí

- Metody popsané jako „prevence uváznutí“ jsou velmi restriktivní
  - ne vzájemnému vyloučení, ne postupnému uplatňování požadavků, preempce prostředků
- Metody „vyhýbání se uváznutí“ nemají dost apriorních informací
  - zdroje dynamicky vznikají a zanikají (např. úseky souborů)
- Detekce uváznutí a následná obnova
  - jsou vesměs velmi drahé – vyžadují restartování aplikací
- Smutný závěr
  - Problém uváznutí je v obecném případě efektivně neřešitelný
  - Existuje však řada algoritmů pro speciální situace – zejména používané v databázových systémech
    - Transakce vědí, jaké tabulky budou používat
  - Praktickým řešením jsou distribuované systémy
  - Minimalizuje se počet sdílených prostředků
  - Nutnost zabývat se uváznutím v uživatelských paralelních a distribuovaných aplikacích

Meziprocesní komunikace

Přehled meziprocesní komunikace

Název	**Anglicky**	**Standard**
Signál	Signal	POSIX
Roura	Pipe	POSIX
Pojmenovaná roura	Named pipe	POSIX
Soubor mapovaný do paměti	Memory-mapped file	POSIX
Sdílená paměť	Shared memory	System V IPC
Semafor	Semaphore	System V IPC
Zasílání zpráv	Message passing	System V IPC
Soket	Socket	Networking

Signály

- seznámili jste se již na cvičeních
- zaslání jednoduché zprávy (nastavení 1 bitu), která je definována číslem signálu
- příjemcem signálu je pouze proces, odesílatel je buď proces, nebo jádro OS
- obsluha signálů:
  - struct sigaction – sa_handler, či sa_sigaction
  - funkce sigaction – připojení funkce k obsluze signálu
- signál se zpracovává asynchroně (nezávisle) na přijímajícím procesu
  - dojde k přepnutí kontextu a spustí se připojená funkce
  - POZOR na kritické sekce se sdílenými proměnnými

Signály

Použití signálů:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int volatile zalohuj = 0;

void handler(int num) {
  zalohuj = 1;
}

int main() {
  pid_t child = fork();
  if (child == 0) {
    int prace = 30;
    struct sigaction action;
    memset(&action, 0, sizeof(action));
    action.sa_handler = handler;
    if (sigaction(SIGUSR1, &action, NULL)!=0)
      { return 2; }
    while (prace-->0) {
      printf("PRACUJI\n");
      sleep(1);
      if (zalohuj) {
        printf("Ukladam mezivysledek\n");
        zalohuj = 0;
      }
    }
  } else {
    int status;
    while (!waitpid(child, &status, WNOHANG)) {
      sleep(5);
      kill(child, SIGUSR1);
    }
  }
  return 0;
}
```

Roury

- seznámili jste se již na cvičeních
- zaslání dat mezi procesy systémem FIFO
- vlastně simulovaný neexistující soubor
- může být více příjemců i více zapisujících procesů – vznikají ale problémy stejné jako při psaní a čtení více procesů ze souborů
- použití roury:
  - `int pipe(int [2])` – rodič vytvoří rouru
  - všichni potomci i rodič může do roury zapisovat i číst
  - standardně rouru zavřou všichni, kdo ji nepoužívají a slouží k přesunu dat mezi dvěmi procesy

Roury

Použití roury:

```c
int main() {
  int pipef[2];
  if (pipe(pipef)!=0) {return 2;}

  pid_t child = fork();
  if (child ==  0) {
    dup2(pipef[1],1);
    close(pipef[0]);
    close(pipef[1]);
    for (int i=0; i<10000; i++) {
       printf("Data %d\n", i);
       fflush(stdout);
    }
  } else { // rodič
    char line[256];
    dup2(pipef[0], 0);
    close(pipef[0]);
    close(pipef[1]);
    while (fgets(line, sizeof(line), stdin))
    {
      printf("Prijata data: %s\n", line);
    }
    wait(&status1);
  }
  return 0;
}
```

Pojmenované roury

- stejný princip – zaslání data mezi procesy systémem FIFO
- může být více příjemců i více zapisujících procesů – vznikají ale problémy stejné jako při psaní a čtení více procesů ze souborů
- oproti normální rouře ji mohou používat libovolné procesy
- použití pojmenované roury:
  - `mkfifo` – vytvoření roury z příkazové řádky
  - `int mknod(const char *, mode_t, dev_t)` – vytvoří pojmenovanou rouru programem pokud mode využije S_IFIFO
  - všechny procesy pak mohou rouru využít jako ”standardní”soubor

Pojmenované roury

Použití roury: Producent

```
int main() {
  char line[1000];
  mknod("/tmp/myfifo", S_IFIFO | 0660, 0);
  int fd = open("/tmp/myfifo", O_WRONLY, 0);
  for (int i=0; i<100000; i++) {
    sprintf(line, "Data %i\n", i);
    write(fd, line, strlen(line));
  }
  return 0;
}
```

Konzument

```
int main() {
  char line[1024];
  int fd = open("/tmp/myfifo", O_RDONLY, 0), rd;
  while ((rd=read(fd, line, 1000))>0) {
^^Iline[rd]=0;
    printf("Prijata data: %i %s\n", rd, line);
  }
  return 0;
}
```

Sdílený soubor mapovaný do paměti

- data lze přenášet mezi procesy pomocí souborů
  - jeden proces zapisuje do souboru, druhý čte ze souboru
- použití normálních souborů je pomalejší, i když je zápis bufferován v paměti
- rychlejší varianta je soubor mapovaný do paměti
- vybraný úsek souboru je mapován přímo do paměti procesu:
  - `mmap(void *addr, size_t delka, int proto, int typ, int fd, off_t posunuti)`
  - vrací adresu, kterou lze použít pro zápis/čtení přímo do/z paměti
  - soubor musí mít alespoň délku, kterou mapujeme do paměti
  - synchronizace je složitější, nejlépe za použití jiného mechanismu
  - Mapování souboru */dev/zero* = alokace paměti

Soubor mapovaný do paměti

```c
int main() {
  char *shared_mem, buf[256];
  int fd = open("/tmp/mapedfile",
                O_RDWR | O_CREAT, 0660);
  printf("Open file %i\n", fd);
  for (int i=0; i<1000; i++) {
    sprintf(buf, "Data %03i\n", i);
    buf[9]=0;
    write(fd, buf, 10);
  }
  shared_mem = mmap(NULL, 10000,
     PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
  printf("mmap %p\n", shared_mem);
  close(fd);
  return 0;
}
```

```c
int main() {
  char *shared_mem;
  int fd = open("/tmp/mapedfile",
                O_RDWR | O_CREAT, 0660);
  shared_mem = mmap(NULL, 10000,
     PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
  for (int i=0; i<1000; i++) {
    printf("Ulozeno: %s\n", shared_mem);
    shared_mem+=10;
  }
  close(fd);
  return 0;
}
```

Sdílená paměť

- velmi podobné, jako soubory mapované do paměti
- soubor je pouze virtuální a nezapisuje se na disk
- po vypnutí počítače se data ve sdílené paměti ztratí
- použití je velmi podobné:
  - shm_open – vytvoří virtuální soubor, nebo připojí k existujícímu podle jména
  - mmap(void *addr, size_t délka, int proto, int typ, int fd, off_t posunutí)
    - vrací adresu, kterou lze použít pro zápis/čtení přímo z paměti
  - ftruncate nebo write, kvůli vytvoření místa ve virtuálním souboru
  - synchronizace je složitější, nejlépe za použití jiného mechanismu

Sdílená paměť – příklad

```c
int main() {
  char *shared_mem;
  int fd = shm_open("pamet",
     O_RDWR | O_CREAT | O_TRUNC, 0660);
  ftruncate(fd, 10000);
  shared_mem = mmap(NULL, 10000, PROT_READ
     | PROT_WRITE, MAP_SHARED, fd, 0);
  for (int i=0; i<1000; i++) {
    sprintf(shared_mem, "Data %03i\n", i);
    shared_mem[9]=0;
    shared_mem += 10;
  }
  close(fd);
  shm_unlink("pamet");
  return 0;
}
```

```c
int main() {
  char *shared_mem;
  int fd = shm_open("pamet",
      O_RDWR | O_CREAT, 0660);
  shared_mem = mmap(NULL, 10000, PROT_READ
      | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);
  for (int i=0; i<1000; i++) {
    printf("Ulozeno: %s\n", shared_mem);
    shared_mem+=10;
  }
  return 0;
}
```

Pojmenovaný semafor

- podobně jako pojmenovaná roura je možné k němu přistoupit z nového procesu
- semafor se připojuje k již existujícímu souboru
  - pouze identifikace, nic do souboru neukládá
- podobně jako semafory pro vlákna, umožňuje implementovat kritickou sekci, nebo počítat
- použití semaforu:
  - ftok – vytvoří identifikátor (klíč) podle jména souboru
  - semget – připojí/vytvoří semafor ke klíči
  - semctl – nastaví hodnotu semaforu
  - semop – provede operaci (odečtení, nebo přičtení)

Semaphore System V a sdílená paměť

```c
int main() {
  key_t s_key;
  union semun {
    int val;
    struct semid_ds *buf;
    ushort array [1];
  } sem_attr;
  struct sembuf asem;
  int buffer_count_sem, spool_signal_sem;
  char *shared_mem;

  /* key identifikuje semafor  */
  if ((s_key = ftok ("/tmp/free", 'a')) == -1)
  { perror ("ftok"); exit (1); }
  if ((buffer_count_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1)
   { perror ("semget"); exit (1); }
  sem_attr.val = 10; // nastav na velikost bufferu
  if (semctl (buffer_count_sem, 0, SETVAL, sem_attr)
    == -1) { perror (" semctl SETVAL "); exit (1); }
  /* key druheho semaforu  */
  if ((s_key = ftok ("/tmp/data", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((spool_signal_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1) {
    perror ("semget"); exit (1);
  }
  sem_attr.val = 0;  // inicializace na 0
  if (semctl (spool_signal_sem, 0,
    SETVAL, sem_attr) == -1)
    { perror (" semctl SETVAL "); exit (1); }
  int fd = shm_open("pamet", O_RDWR
   | O_CREAT | O_TRUNC, 0660);
  ftruncate(fd, 1000);
  shared_mem = mmap(NULL, 1000, PROT_READ
   | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);
  asem.sem_num = 0;
  asem.sem_flg = 0;
  for (int i=0; i<50; i++) {
    asem.sem_op = -1;
    if (semop (buffer_count_sem, &asem, 1) == -1)
    { perror ("semop: buffer_count_sem"); exit (1); }
    sprintf(shared_mem, "Data %03i\n", i);
    printf( "Data %03i %p\n", i, shared_mem);
    shared_mem[9]=0;
    shared_mem += 10;
    if (i%10==9) {
      shared_mem-=100;
^^I}
    asem.sem_op = 1;
    if (semop (spool_signal_sem, &asem, 1) == -1)
    { perror ("semop: spool_signal_sem"); exit (1); }
  }
  return 0;
}
```

Semaphore System V a sdílená paměť

```
int main() {
  key_t s_key;
  union semun {
    int val;
    struct semid_ds *buf;
    ushort array [1];
  } sem_attr;
  struct sembuf asem;
  int buffer_count_sem, spool_signal_sem;

  /* pouzij stejny semafor jako producent  */
  if ((s_key = ftok ("/tmp/free", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((buffer_count_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1) {
    perror ("semget"); exit (1); }

  /* pouzij stejny semafor jako producent  */
  if ((s_key = ftok ("/tmp/data", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((spool_signal_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1) {
    perror ("semget"); exit (1); }
```

```
  char *shared_mem;
  int fd = shm_open("pamet", O_RDWR |
     O_CREAT, 0660);
  ftruncate(fd, 1000);
  shared_mem = mmap(NULL, 1000, PROT_READ
   | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);
  asem.sem_num = 0;
  asem.sem_flg = 0;
  for (int i=0; i<50; i++) {
    asem.sem_op = -1;
    if (semop (spool_signal_sem, &asem, 1) == -1) {
      perror ("semop: buffer_count_sem"); exit (1);
    }
    printf("Ulozeno: %s, %p\n", shared_mem, shared_mem);
    shared_mem += 10;
    if (i%10==9) {
      shared_mem-=100;
      sleep(1);
^^I}
    asem.sem_op = 1;
    if (semop (buffer_count_sem, &asem, 1) == -1) {
      perror ("semop: spool_signal_sem"); exit (1);
    }
  }
  close(fd);
  return 0;
}
```

Fronta zpráv

- zprávy jsou zasílány a vyzvedávány do/z fronty zpráv identifikovaných libovolným souborem
- podobně jako pojmenovaná roura a semafor je možné k němu přistoupit z nového procesu
- zprávy mají povinně typ, podle kterého je možné vybírat z fronty zpráv pouze zprávy zadaného typu
- použití fronty zpráv:
  - msgget – vytvoří virtuální frontu zpráv, nebo připojí k existující frontě podle jména souboru zadaného jeho klíčem
    - nutné vytvořit si vlastní strukturu zpráv, která jako první obsahuje long – typ zprávy
  - msgsnd – zaslání zprávy, pozor délka zprávy je délka struktury zmenšená o velikost long – typ zprávy
  - msgrcv – přijmutí zprávy zadaného typu
  - msgctl – odstranění fronty zpráv

Fronty zpráv

```c
struct my_msg {
  long mtype;
  int len;
  char txt[10];
};
int main() {
  key_t s_key;
  int msg_id;
  struct my_msg msg;
  if ((s_key = ftok ("/tmp", 'a')) == -1)
  { perror ("ftok"); exit (1); }
  if ((msg_id = msgget(s_key, 0660 | IPC_CREAT))
    == -1) { perror ("msgget"); exit (1); }
  for (int i=0; i<50; i++) {
    msg.mtype=1;
    msg.len = 10;
    sprintf(msg.txt, "Data %03i\n", i);
    msg.txt[9]=0;
    if (msgsnd(msg_id, &msg, sizeof(msg)-sizeof(long),0)
      == -1) { perror ("msgsnd"); exit (1); }
  }
  if (msgrcv(msg_id, &msg, sizeof(msg)-sizeof(long), 2, 0)
     == -1) { perror ("msgrcv"); exit (1); }
  printf("Prijato: %s\n", msg.txt);
  if (msgctl(msg_id, IPC_RMID, 0) == -1)
  { perror ("msgctl"); exit (1); }
  return 0;
}
```

```c
int main() {
  key_t s_key;
  int msg_id;
  struct my_msg msg;

  if ((s_key = ftok ("/tmp", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((msg_id = msgget(s_key, 0660 | IPC_CREAT))
    == -1) { perror ("msgget"); exit (1); }
  for (int i=0; i<50; i++) {
    if (msgrcv(msg_id, &msg, sizeof(msg)-sizeof(long), 1, 0)
     == -1) { perror ("msgrcv"); exit (1); }
    printf("Prijato: %s\n", msg.txt);
  }
  msg.mtype=2;
  msg.len = 10;
  sprintf(msg.txt, "Koncime\n");
  msg.txt[9]=0;
  if (msgsnd(msg_id, &msg, sizeof(msg)-sizeof(long), 0)
   == -1) { perror ("msgsnd"); exit (1); }
  return 0;
}
```