**BI-SPOL-16 Procesy a vlákna, jejich implementace. Synchronizační nástroje. Klasické synchronizační úlohy. Plánování vláken. Přidělování prostředků, Coffmanovy podmínky, způsoby řešení uváznutí**

BI-OSY

#### Základní pojmy

Privilegované módy běhu CPU, SW běžící v daném módu CPU může:

* používat pouze instrukce povolené v tomto módu
* modifikovat určité registry
* modifikovat určité oblasti paměti, …

**Kernel mód**: vše je povoleno, typicky v něm běží jádro OS

**User mód**: omezený mód (nelze přímo manipulovat s periferními zařízeními, …), typicky v něm běží uživatelské procesy

**Přerušení (interrupts)**

* asynchronní reakce na nějakou událost
* normální zpracování instrukčního proudu jádra je přerušeno a začne se provádět obslužná rutina přerušení
* mechanismus přerušení je definován v ISA

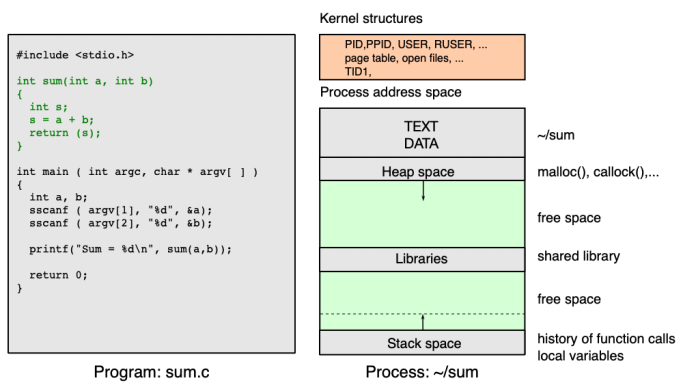
**Systémové volání** – pokud chce aplikace použít prostředek systému musí požádat jádro OS pomocí systémového volání

**Program**: posloupnost instrukcí definující chování procesu

### Procesy

* instance spuštěného programu/aplikace
* entita, v rámci, které jsou alokovány prostředky (paměť, vlákna, otevřené soubory, zámky, semafory, sokety atd.)
* implicitně má každý proces jedno výpočetní “main” vlákno
* jádro OS si pro každý proces udržuje celou řadu datových struktur nezbytných pro:
  + identifikaci: číslo procesu (PID), číslo rodič. Procesu (PPID), …
  + bezpečnost: identita procesu (USER, RUSER), …
  + správu paměti: informace pro překlad virtuálních adres (page table), …
  + správu FS: tabulka deskriptorů souborů, …
* Při vzniku nového procesu je **část datových struktur zděděna** od rodičovského procesu (tabulka deskriptorů souborů, …) a **část je nastavena na nové hodnoty**, které jsou specifické pro nový proces (číslo procesu, …)

**Proces s jedním vláknem**



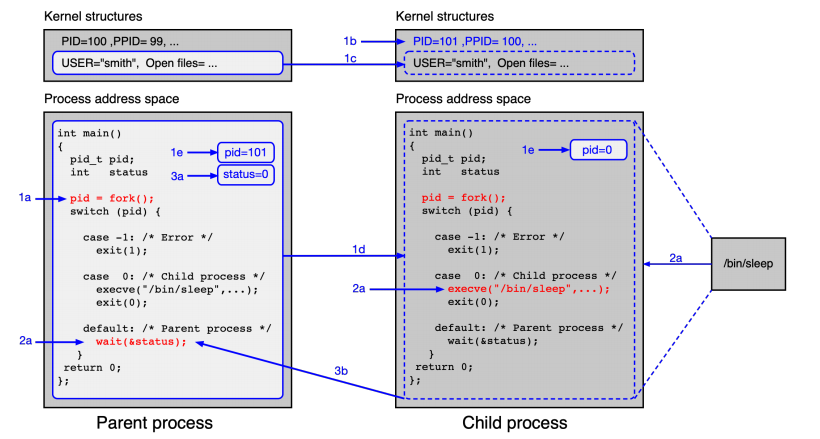
TEXT = spustitelný binární kód

DATA = proměnné a jejich hodnoty

#### Vytvoření nového procesu

* Nový proces se vytvoří, když existující proces zavolá příslušné systémové volání
  + V OS unixového typu: fork(), execve(), …
  + V MS Windows: CreateProcessA(), …
* Nově vzniklý proces může představovat
  + **Kopii/klon původního procesu** (např. po zavolání fork())
    - Jádro alokuje nové datové struktury pro nový proces (část z nich bude nově zinicializována, část bude kopií od rodiče)
    - Adresový prostor procesu bude zkopírován od rodiče (TEXT, DATA, zásobník, halda, …)
    - Čítač instrukcí bude ukazovat na následující instrukci za fork()
  + Úplně nový proces (např. po zavolání fork() a execve())
    - Jádro alokuje nové datové struktury pro nový proces (část z nich bude nově zinicializována, část bude kopií od rodiče)
    - Adresový prostor procesu bude nově zinicializován (prázdný zásobník, halda, …) a TEXT, DATA, knihovny budou načteny ze souboru
    - Čítač instrukcí bude ukazovat na první instrukci programu
* Metody v Unixu
  + Fork
    - vytvoří nový proces, který je kopií procesu, z kterého byla tato funkce zavolána
    - v rodičovském procesu vrací číslo potomka (v případě chyby -1), v potomkovi vrací vždy 0
  + execve – adresový prostor procesu, ze kterého je funkce volána, je přepsán obsahem souboru, který se začne vykonávat od začátku
  + wait – zablokuje rodičovský proces, ve kterém je zavolána, dokud se jeden jeho potomek neukončí

**Příklad vytvoření nového procesu v Unixu**



Obsah obrázku text

Popis byl vytvořen automaticky

#### Ukončení procesu

* jádro OS při ukončení procesu
  + se pokusí předat „návratový kód“ rodiči,
  + ukončí všechna vlákna, který existují v rámci daného procesu
  + uvolní adresový prostor procesu a příslušné datové struktury v jádře související s daným procesem
* varianty ukončení procesu
  + proces se ukončí sám
    - pokud dojde na konec programu (např. return v „main“ vláknu)
    - zavolá příslušnou knihovní funkci (např. exit)
  + proces je ukončen jádrem
    - pokud dojde k fatální chybě (např. dělení nulou, špatná manipulace s pamětí, …)
    - na základě např. přijatého signálu

#### Implementace procesů

**Tabulka procesů**

* jádro OS si udržuje informace o procesech pomocí zřetězeného seznamu struktur, který můžeme nazývat „tabulkou procesů“

**Process control block (PCB)**

* jedna položka tabulky (struktura) reprezentuje všechny nezbytné informace, které si jádro OS musí pamatovat pro jeden proces
* obsahuje všechny nezbytné informace, které si OS musí pamatovat o procesu
* Informace pro identifikaci procesu
  + Číslo procesu (PIS), číslo rodičovského procesu (PPIS)
  + Číslo seance (SID), číslo úlohy, číslo projektu
  + Jméno procesu, …
* Informace související se identitou procesu/bezpečností
  + Vlastník procesu
  + Příslušnost ke skupinám
  + Privilegia přiřazená procesu, access token
* Informace o alokovaných prostředcích
  + Paměť
    - Informace o přidělené fyzické paměti
    - Informace nutné pro překlad logických adres na fyzické
  + Otevřené soubory
  + Prostředky pro mezi procesovou komunikaci
    - Synchronizační nástroje (semafory, signály, …)

**Thread control block**

* Pro každé vlákno daného procesu si OS musí pamatovat řadu informací, které jsou uloženy v různých datových strukturách. Tyto struktury budeme označovat jako thread control block a obsahují následující informace.
  + Informace pro identifikaci vlákna – číslo vlákna (TID)
  + Informace pro přepínání kontextu
    - Hodnoty viditelných registrů CPU
    - Hodnoty řídících a stavových registrů
    - Ukazatel na zásobník
  + Informace pro plánování vláken
    - Typ plánovacího algoritmu
    - Priorita
    - Stav vlákna
    - Informace o událostech, na které vlákno čeká, …

### Vlákna

* výpočetní entita (proud instrukcí), které je přidělováno jádro CPU
* vlákna vytvořená v rámci procesu sdílí většinu prostředků alokovaných v tomto procesu
* jádro OS si pro každé vlákno udržuje celou řadu datových struktur nezbytných pro
  + identifikaci: číslo vlákna (TID), …
  + zásobník: lokální proměnné, historie volání, …
  + informace nutné pro přepínání kontextu: čítač instrukcí, aktuální hodnoty registrů, …
  + informace pro plánování vláken: priorita, čas strávený na CPU
  + a další

**Proces s více vlákny**

Obsah obrázku stůl

Popis byl vytvořen automaticky

#### Vytvoření a ukončení vlákna

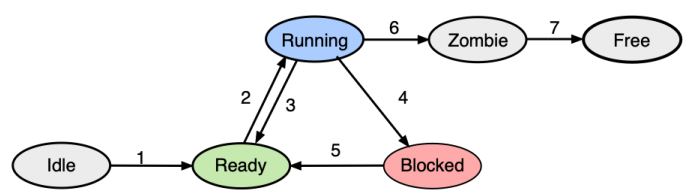
* procesy se implicitně vytváří s jedním „main“ vláknem
* pokud chceme vytvořit v rámci procesu další vlákna, pak můžeme použít
  + OS API/knihovny – např. POSIX thread library (MS i linux)
  + Programovací jazyky s vestavěnou podporou vláken – c++11, java

Process-based multitasking – více procesů, v každém procesu je jedno vlákno, mají svoje i sdílené resources

Thread-based multitasking – multithreading – více procesů, každý proces má více vláken, mají svoje i sdílené resources

#### Stavy vláken

* Stav vlákna popisuje, co se s daným vláknem právě děje
* Mezi základní stavy patří
  + **Idle**: vznik nového vlákna
  + **Ready**: vlákno čeká až mu bude přiděleno jádro CPU
  + **Running**: vlákno je zpracováváno jádrem CPU
  + **Blocked**: vlákno čeká na událost (dokončení operace, příchod signálu, …)
  + **Zombie**: vlákno je ukončováno, ale zatím ještě nebylo vše dokončeno
  + **Free**: vlákno bylo kompletně zrušeno (pouze teoretický stav)



#### Implementace vláken

* 2 způsoby: v uživatelském prostoru a v jádru OS

**v uživatelském prostoru**: (user-level threads)

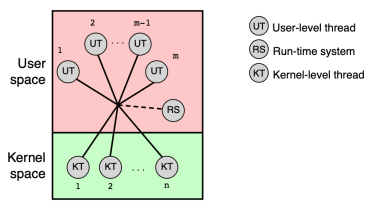
* jsou spravována v uživatelském prostoru pomocí run-time systému
* používají kooperativní plánování (po určitém čase vlákno předá řízení zpět run-time systému pomocí příslušné funkce)
* jádro OS nemá „žádnou“ informaci o uživatelských vláknech v jednotlivých procesech a spravuje je jako proces s jedním vláknem
* Jednoduchá/rychlá správa: vytvoření, přepínání, synchronizace vláken je v uživatelském prostoru bez zásahu jádra OS
* Vlákna jednoho procesu jsou mapována na jedno jádro CPU
* Blokující volání jednoho vlákna zablokuje všechny ostatní vlákna procesu
* model many-to-one: víc uživatelských vláken namapováno na 1 kernel

**v jádře OS**:

* typická implementace v současných OS s podporou vláken
* vlákna jsou spravována přímo jádrem OS
* jádro OS
  + spravuje jeden PCB pro každý proces
  + spravuje jeden TCB pro každé vlákno
  + poskytuje systémové volání pro správu vláken z uživatelského prostoru
* jádro má informace o všech vláknech ⇒ jádro OS přiděluje jednotlivá jádra CPU jednotlivým vláknům na určitou dobu
* blokující systémové volání na vlákno zablokuje pouze toto vlákno
* vytvoření/ukončení vlákna, systémové volání, přepnutí kontextu ⇒ představuje přepnutí z user modu do kernel modu
* model one-to-one – jedno uživatelské vlákno namapováno na jedno kernel vlákno

**hybridní implementace**

* model many-to-many, kombinace předchozích implementací, kdy *m* uživatelských vláken je namapováno na *n* kernel vláken



#### Plánování vláken

* jádro OS a vytvořená vlákna sdílí omezený počet jader výpočetního systému
* jedno vlákno (instrukční proud) může být v jednom okamžiku zpracováváno jedním „logickým“ jádrem CPU
* aby se vlákna „rozumným“ způsobem podělila o omezený počet jader, tak se vlákna na jádrech střídají nejčastěji pomocí preemptivního plánování
* **preemptivní plánování vláken**:
  + Vláknu je přiděleno volné jádro CPU, pokud ho jádro OS vybere na základě plánovacích kritérií (např. priority)
  + Jádro OS vláknu přidělí **časové kvantum**, během kterého vlákno bude zpracováváno jádrem CPU
  + Vláknu je jádro CPU odebráno pokud
    - Dojde k uplynutí časového kvanta (přerušení od časovače)
    - Vlákno provede systémové volání
    - Dojde k přerušení (např. dokončená operace)
* V okamžiku, kdy se uvolní jádro CPU, musí OS vybrat „vhodné“ vlákno ve stavu Ready, a umožní mu používat toto jádro po určitou dobu
* V OS je obvykle implementováno několik různých plánovacích strategií a uživatel/administrátor může pro různé aplikace nastavit vhodnou strategii, popřípadě určí, na kterých jádrech CPU poběží
* typy aplikací z pohledu plánování: dávkové úlohy (př. zpracovávání dat), interaktivní aplikace (reagují na události), úlohy reálného času (aplikace řídící technologický proces)

**Cíle plánování**

* z hlediska systému
  + spravedlivost: každý uživatel/proces/vlákno získá úměrnou část CPU výkonu
  + vyváženost: rozložit zátěž rovnoměrně na všechny části systému
  + prosazení strategie: možnost uplatnit zvolenou plánovací strategii
* dávkové úlohy
  + doba zpracování: čas, který uplyne od spuštění do ukončení úlohy
* interaktivní aplikace
  + doba odezvy: čas, který uplyne od spuštění do ukončení úlohy
  + přiměřenost: splnit očekávání uživatele
* úlohy reálného času
  + garantování odezvy: zabránění ztrátě dat
  + předvídatelnost: zamezení degradaci výkonu

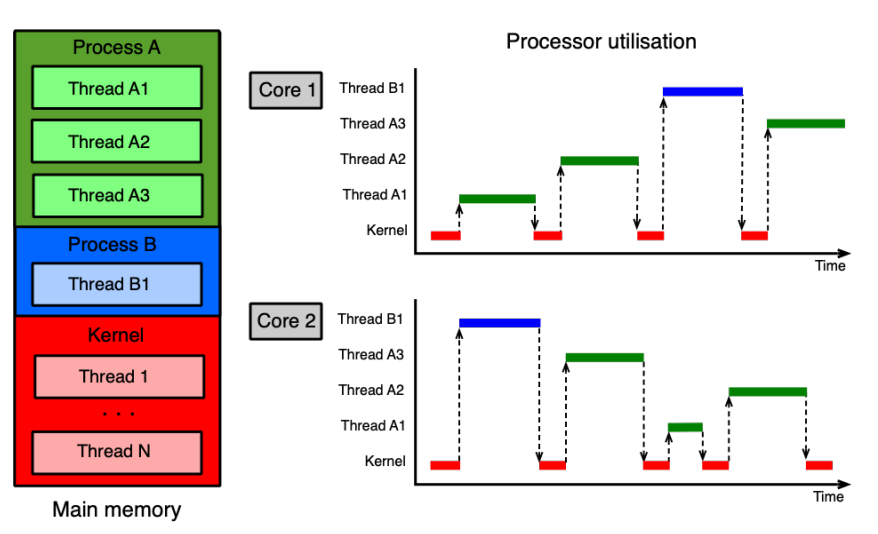
**Strategie plánování**

* *Plánování s odnímáním* (preemtive scheduling)
  + Jádro OS přidělí vláknu jádro CPU pouze na určitou dobu (časové kvantum) a po uplynutí této doby mu ho odebere
  + Vlastnosti
    - Strategie vhodná pro vlákna v interaktivních systémech
    - Umožňuje „rozumným“ způsobem sdílet CPU výkon systému
    - Dochází zde k častějšímu přepínání kontextu – horší využití CPU/lepší doba odezvy
* Vlákna čekají ve frontě FIFO, až jim bude přiděleno jádro CPU
* Časové kvantum je volitelný parametr
  + Krátké čas. kvantum – horší využití CPU/krátká doba odezvy
  + Dlouhé čas. kvantum – dlouhá doba odezvy/lepší využití CPU
  + Rozumný kompromis je 10-50ms
* *Kooperativní plánování* (cooperative scheduling)
  + Jádro OS přidělí vláknu jádro CPU a vlákno ho používá, dokud ho samo neuvolní (např. dokončení výpočtu, blokující funkce, …)
  + Vlastnosti
    - Vlákno může blokovat systém a musí „spolupracovat“ pouze když žádá službu jádra → tato strategie je vhodná pouze pro „prověření/kernel“ vlákna a není vhodná pro běžná uživatelská vlákna
    - Minimalizuje se počet přepnutí kontextu → lepší využití CPU/horší doba odezvy

**Přepínání kontextu**

* mechanismus, při kterém se vlákna vystřídají v používání jádra CPU
* **kontextem** se rozumí všechny nezbytné informace, které jsou nutné pro pozdější spuštění přerušeného vlákna od okamžiku přerušení (např. čítač instrukcí, obsahy registrů, …)
* samotná vlákna žijí v iluzi, že běží bez přerušení od začátku do konce
* přepínání kontextu probíhá v několika krocích:

1. uloží se kontext původního vlákna
2. jádro OS naplánuje další vlákno
3. nastaví se kontext tohoto vlákna



### 3. Synchronizační nástroje

* deterministický algoritmus – vždy ze stejných výchozích (vstupních) podmínek svým během vytvoří stejné výsledky
* časově závislé chyby – situace, kdy dvě nebo několik vláken používá (čte/zapisuje) společné sdílené prostředky (např. sdílení proměnné, sdílené soubory, …) a výsledek deterministického algoritmu je závislý na rychlosti jednotlivých vláken, které používají tyto prostředky
* kritická sekce – část programu, kde vlákna používají sdílené prostředky (např. sdílený soubor)
* synchronizace je nutná, aby v případě přístupu několika vláken k jednomu prostředku nedocházelo k časově závislé chybě
* při použití synchronizace mohou nastat chyby:

1. *deadlock (uváznutí)*: situace, kdy několik vláken čeká na událost, kterou může vyvolat pouze jedno z čekajících vláken
2. *livelock*: situace, kdy několik vláken vykonává neužitečný výpočet (mění svůj stav), ale nemohou dokončit výpočet
3. *hladovění (starvation)*: situace, kdy je vlákno ve stavu „Ready“ předbíháno a nedostane se po „dlouhou“ dobu k prostředkům

* pouze jedno vlákno může být uvnitř kritické sekce
  + ostatní vlákna musí počkat, dokud se kritická sekce neuvolní
* toho lze docílit dvěma základními způsoby
  + pomocí aktivního čekání
  + pomocí blokujících systémových volání/knihovních funkcí

#### Aktivní čekání (busy waiting, spinning)

* sdílená proměnná indikuje obsazenost kritické sekce (zamčená/odemčená)
* před vstupem do
  + *zamčené sekce*: vlákno ve smyčce **„aktivně“ testuje** aktuální hodnotu proměnné do okamžiku, než se sekce uvolní
  + *odemčené sekce*: vlákno změní hodnotu sdílené proměnné (**zamkne** kritickou sekci) **a vstoupí do sekce**
* po opuštění kritické sekce
  + vlákno změní hodnotu sdílené proměnné (**odemkne sekci**)

**Instrukce TSL (test-and-lock)**

* pro implementaci potřebujeme atomickou instrukci pro zamykání/odemykání
  + žádné jiné vlákno během provádění této instrukce nemůže přistupovat ke slovu v paměti
  + její implementace závisí na konkrétní HW architektuře (např. zamčení paměťové sběrnice)
  + toto je korektní řešení ve více-jádrových systémech se sdílenou pamětí
* TSL je hypotetická instrukce sloužící ke konkrétní implementaci aktivního čekání na HW úrovní
  + Je atomická
  + Skládá se ze dvou kroků
    - Načte obsah slova z dané adresy v paměti do registru
    - Nastaví obsah slova v paměti na nenulovou hodnotu (zamkne kritickou sekci)

**Vlastnosti:**

* Výhody
  + Minimální režie, pokud vlákno nemusí čekat nebo čeká krátkou dobu před vstupem do kritické sekce
* Nevýhody
  + Vlákno, které čeká před vstupem do kritické sekce, zatíží jedno jádro CPU na 100%
  + V jistých situacích může aktivní čekání skončit uváznutím – inverzní prioritní problém
  + Inverzní prioritní problém
    - Nutné podmínky
      * OS používá prioritní plánování vláken s fixní prioritou (priorita přiřazená vláknu se během existence nemění)
      * CPU má omezený počet *n* jader
    - Uváznutí nastane pokud
      * Vlákno A má nízkou prioritu a nachází se v kritické sekci
      * Vlákno B má vyšší prioritu a čeká pomocí aktivního čekání na vstup do kritické sekce
      * Všechna jádra v systému jsou obsazena vlákny s vyšší prioritou
      * Neboli: když vlákno s malou prioritou je v kritické sekci a s velkou čeká na vstupu do ní, OS přiděluje čas pouze tomu s velkou prioritou, a to může jen aktivně čekat
* Aktivní čekání se vyplatí, pokud se očekává krátká doba čekání na prostředek a nehrozí inverzní prioritní problém (např. synchronizace v jádru OS)

#### Blokující volání

* Blokující systémové volání/knihovní funkce je implementování obvykle pomocí datových struktur, které umožňují
  + Si pamatovat stav kritické sekce (odemčená/zamčená)
  + Udržovat seznam vláken, která čekají na vstup do kritické sekce
* Před vstupem do kritické sekce
  + *Zamčená sekce*: vlákno provede systémové volání/knihovní funkci, které ho zablokuje (přepne jeho stav do stavu „Blocked“), a tím pádem vláknu přestane být přidělován procesor → **pasivně čeká** na uvolnění sekce
  + *Odemčená sekce*: vlákno provede systémové volání/knihovní funkci, které ho nezablokuje ale pouze si zapamatuje, že sekce je zamčená, a vlákno „vstoupí“ do sekce
* Po opuštění kritické sekce
  + Vlákno pomocí systémového volání/knihovní funkce probudí čekající vlákno/vlákna
  + V případě, že již žádná vlákna nečekají, pak si zapamatuje, že sekce je odemčená
* Implementace pomocí: zámků (mutex), podmíněných proměnných, semaforů, bariér

**Zámky**

* Implementujeme pomocí zámku označovaný jako **mutex**, který si pamatuje:
  + Svůj stav (zamčený/odemčený)
  + Kdo je jeho vlastníkem (vlákno, které ho zamklo)
  + Množinu vláken, která jsou na něm blokovaná
* Nad zámkem mutex jsou definovány atomické operace
  + *mutex\_lock*
    - pokud je mutex odemčený, tak ho zamkne. Volající vlákno se stane vlastníkem zámku
    - pokud je mutex zamčený, tak zablokuje volající vlákno
  + *mutex\_unlock*
    - měl by volat pouze vlastník zámku (jinak skončí chybou)
    - pokud jsou nějaká vlákna blokována zámkem, tak se jedno z nich probudí
    - pokud již žádné vlákno není blokováno, tak se odemkne mutex

Obsah obrázku text

Popis byl vytvořen automaticky

**Vlastnosti blokujících volání**

* výhody
  + čekání na vstup kritické sekce nepředstavuje žádnou režii
* nevýhody
  + začátek a ukončení představují určitou režii, neboť je nutné změnit stav vlákna v jádře OS
* vyplatí se, pokud se očekává delší doba čekání na prostředek

**Podmíněné proměnné (condition variable)**

* podmíněná proměnná si pamatuje, která vlákna jsou na ní blokována
* nad podmíněnou proměnnou var jsou typicky definovány operace
  + *cond\_wait*
    - funkce musí být volána se zámkem mutex, který je zamčený volajícím vláknem
    - funkce automaticky uvolní mutex a zablokuje volající vlákno, dokud nebude proměnná opět signalizována
    - po odblokování (návratu z funkce) je mutex opět zamčen
  + *cond\_signal*
    - odblokuje aspoň jedno ze zablokovaných vláken
* používají se např. u problému producent-konzument

**Semafory**

* datový typ semafor
  + obsahuje celočíselný čítač
  + pamatuje si množinu vláken, která jsou na něm zablokována
* na čítač se nastaví číslo, kolik může do sekce vláken, když je čítač 0 – blokuje vlákna, když vlákno opustí kritickou sekci, inkrementuje čítač
* nad semaforem jsou definovány atomické operace
  + *sem\_init*
    - nastaví čítač na hodnotu value (vstup) a vyprázdní frontu čekajících vláken
  + *sem\_wait*
    - pokud je čítač větší než 0, potom se sníží o jedničku
    - V opačném případě se volající vlákno zablokuje a uloží do fronty
  + *sem\_post*
    - pokud nějaká vlákna čekají ve frontě, potom se jedno z nich probudí
    - v opačném případě se čítač zvětší o jedničku

**Bariéry**

* bariéra umožňuje jednoduše synchronizovat iterační výpočty (např. výpočet n-té mocniny matice pomocí více vláken)
* bariéra obsahuje
  + čítač, který definuje sílu bariéry (počet vláken, který bariéru prolomí)
  + frontu vláken, která jsou na bariéře blokována
* pro bariéru jsou definovány atomické operace
  + *barrier\_init*
    - funkce nastaví čítač bariéry na *value* (vstupní hodnota) a vyprázdní frontu čekajících vláken
  + *barrier\_wait*
    - pokud je čítač bariéry větší než 1, pak se čítač sníží o 1 a volající vlákno se zablokuje
    - jinak se všechna blokovaná vlákna probudí a čítač se opět nastaví na *value*

### 4. Klasické synchronizační úlohy

#### Večeřící filosofové

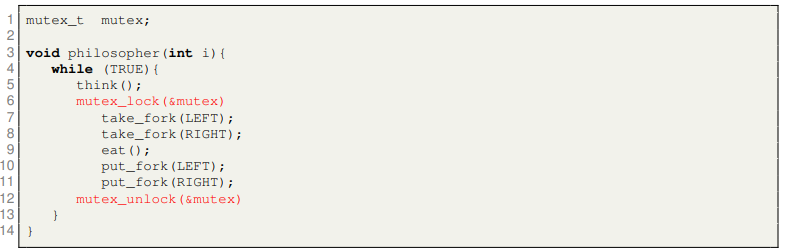
* Několik vláken soutěží o omezený počet prostředků.
* Je N filosofů kolem kulatého stolu a mezi talíři je vždy vidlička. (celkem N vidliček)
* Pokud se chce filosof najíst, musí mít 2 vidličky.
* Filosof buď přemýšlí, má hlad a pokouší se získat vidličky nebo jí.

*Správné řešení*: nebude docházet k časově závislým chybám, uváznutí, livelocku, hladovění, …

*Optimální řešení*: v jeden okamžik může jíst až dolníCeláČást(N/2) filosofů.

**Správné řešení**

* celý proces od sebrání vidliček, najedení po položení vidliček je kritická sekce – mutex\_lock, mutex\_unlock
* může jíst pouze jeden filosof – není optimální



**Optimální řešení**

* pomocí mutexu a N podmíněných proměnných
* musíme vzít atomicky obě vidličky, upozorňujeme sousedy po skončení a sousedi jsou mezitím blokování mutexem, dokud je neupozorní. Optimálně může jíst N/2 filosofů

Obsah obrázku text

Popis byl vytvořen automaticky

#### Čtenáři-písaři

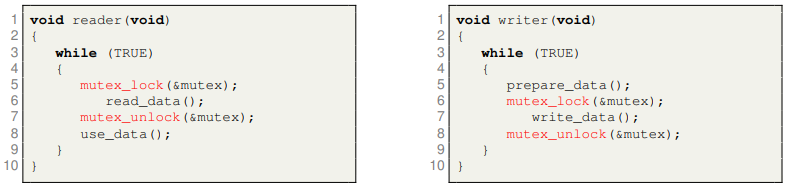
* 2 typy vláken soutěží o přístup ke společnému prostředku
* V systému máme jeden sdílený prostředek a 2 typy vláken:
  + Čtenáři – pouze čtou
  + Písaři – mohou modifikovat
* Pouze jeden písař může modifikovat v jeden okamžik

*Optimální řešení*: Více čtenářů může číst současně pokud žádný písař nepřistupuje k prostředku

*Spravedlivé řešení*: pokud písař/čtenář čeká na sdílený prostředek, pak by ho žádný jiný čtenář ani písař neměl předběhnout

**Správné řešení**

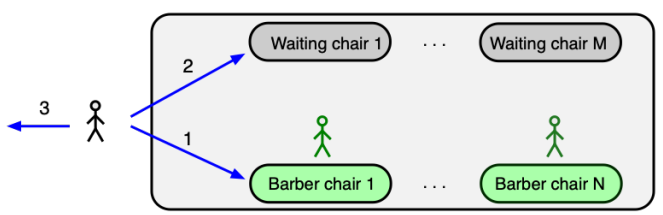
* Sdílený prostředek je kritická sekce – zalockujeme to
* Neobsahuje časově závislé chyby ani uváznutí – správné
* Pouze jeden čtenář nebo pouze jeden písař může přistupovat v jeden okamžik – není optimální



**Optimální spravedlivé řešení**

* Musíme si pamatovat v jakém pořadí čtenáři a písaři začínají čekat
* Použijeme mutex, counter a frontu. Pokud do fronty přijde více čtenářů za sebou, mohou se zmergovat a mít přístup najednou. Až odejdou, přijde písař a zablokuje si to pro sebe
* Zdrojový kód v přednášce č. 4 str. 15

#### Spící holiči

N holičů a N křesel k holení, M křesel na čekání. Pokud tam nikdo není, holič spí. Až přijde zákazník, mohou nastat 3 situace:

1. holič je volný – zákazníka ostříhá
2. holič není volný, ale je místo v čekárně – počká
3. jinak odejde

* snažíme se napárovat holiče a klienta

*Řešení*: mutex a 2 semafory

**Správné optimální řešení**

* pomocí 1 mutexu a 2 semaforů
* zákazníci/holiči jsou simulováni vlákny, která vykonávají funkci customer() / barber()

**Obsah obrázku text

Popis byl vytvořen automaticky**

* fce customer – zamkne mutex a koukne se jestli je volno – pokud není, tak odchází
  + sem\_post(customers) (dá signál barberovi, který je uspany pomocí sem\_wait) probudí uspaného holiče – odemkne mutex a jde si seknou do čekárny – sem\_wait(barbers)

#### Producent-konzument

* představuje situaci, kdy si několik vláken vyměňuje data prostřednictvím sdílené paměti s omezenou velikostí
* producent – produkuje data a vkládá je do sdílené paměti (fronty) s omezenou velikostí
* konzument – vybírá data ze sdílené paměti (fronty)
* problémy:
  + musíme zajistit výlučný přístup při vkládání/vybírání dat z fronty
  + pokud je fronta prázdná – musíme zablokovat konzumenta
  + pokud je fronta plná – musíme zablokovat producenta

**Řešení**

* 1 mutex a 2 cond\_variable (full, empty)
  + Full – wait u producenta, signál u konzumenta
  + Empty – wait u konzumenta, signál u producenta

**Obsah obrázku text

Popis byl vytvořen automaticky**

### 5. Uváznutí

* Uváznutí (deadlock) – situace, kdy několik vláken čeká na událost/prostředek, kterou může vyvolat/uvolnit pouze jedno z čekajících vláken
* Alokační graf – znázorňuje alokaci prostředků jednotlivými vlákny
  + Je to orientovaný graf s dvěma typy uzlů (prostředky/vlákna)
  + Každá smyčka představuje uváznutí

**Coffmanovy podmínky**

* Uváznutí nastane pouze pokud jsou splněny následující podmínky:

1. **Vzájemné vyloučení**: každý prostředek je buď přidělen právě jednomu vláknu a nebo je volný (prostředek nemůže být sdílen více vlákny)
2. **Podmínka neodnímatelnosti**: prostředek, který byl již přidělen nějakému vláknu, mu nemůže být násilím odebrán (musí být dobrovolně uvolněn daným vláknem)
3. **Podmínka „drž a čekej“**: vlákno, které má již přiděleny nějaké prostředky, může žádat o další prostředky (vlákno může žádat o prostředky postupně)
4. **Podmínka kruhového čekání**: musí existovat smyčka dvou nebo více vláken, ve které každé vlákno čeká na prostředek přidělený dalšímu vláknu ve smyčce

* První tři podmínky jsou nutné ale ne dostačující → k uváznutí může dojít. Poslední podmínka představuje samotné uváznutí
* Pokud alespoň jedna z podmínek není splněna, nemůže dojít k uváznutí

**Způsoby řešení uváznutí**

* Pštrosí strategie
  + Ignorování celého problému
  + Problém se neřeší nebo řeší částečně
  + Dobré řešení, pokud by bylo řešení uváznutí příliš drahé
* Prevence uváznutí
  + Pomocí nesplnění alespoň jedné z Coffmanových podmínek
  + Porušení „vzájemného vyloučení“ – může vést k časově závislým chybám
  + Porušení „neodnímatelnosti prostředku“ – musíme si pamatovat stav, příkladem je přepínání kontextu
  + Porušení drž a čekej – vede k horšímu využití prostředků
  + Porušení „kruhového čekání“ – každý prostředek má číslo – vlákno může žádat o prostředky pouze v rostoucím pořadí – vhodné očíslování nemusí vždy existovat
* Předcházení vzniku uváznutí
  + Na základě pečlivé alokace prostředků
  + Předem známe všechny požadavky vláken na prostředky a vytvoříme bezpečný stav, který garantuje postupné uspokojení potřeb všech vláken
* Detekce uváznutí a zotavení
  + K uváznutí může dojít, ale je detekováno a odstraněno
  + Detekce – v systému jsou prováděny pravidelné kontroly, které se snaží odhalit existující uváznutí
  + Zotavení – část prostředků se „uvolní“ a tím se poruší čekání vláken ve smyčce – uváznutí se odstraní

**Algoritmus pro detekci uváznutí**

1. Vytvoříme kopii C vektoru F (aktuálně volné prostředky)
2. Na začátku jsou všechna vlákna neoznačená
3. Označíme všechna vlákna, která nechtějí žádný prostředek
4. Existuje vlákno, které lze uspokojit prostředek z C?
   1. Vlákno existuje – označíme vlákno, alokované prostředky přičteme k C, opakujeme bod 4
   2. Vlákno neexistuje – pokračujeme na bod 5
5. Byla označena všechna vlákna?
   1. Ano – žádné uváznutí
   2. Ne – uváznutí nastalo