



KIV Operační systémy

Meziprocesová synchronizace



Meziprocesová synchronizace

- Meziprocesová nebo mezivláknová?
- V Linuxu je thread uvnitř OS reprezentován s PCB, u jiných OS s TCB
- Takže lze univerzálně říci, že se dále budeme bavit o synchronizaci operačním systémem plánovatelných entit
- Výhodou je, že s tímto přístupem pokryjeme i synchronizaci threadu uvnitř procesu
 - Tj. synchronizujeme thready v alespoň jednom procesu



Celočíselný semafor

- Abstraktní datový typ, který kontroluje přístup ke sdílenému prostředku (sdílí ho více vláken)
 - O vlastním, sdíleném prostředku ale neví nic
 - Má limit, kolik threadů může naráz přistupovat ke sdílenému prostředku
 - Binární semafor má tuto hodnotu nastavenou na 1
 - Má počítadlo, kolik threadů už prostředek sdílí
 - Má frontu čekajících threadů, které by chtěly prostředek sdílet



Acquire

- ... též známé jako wait, down či P(passering v originále, „*P*ust' mě dovnitř“)
- Thread žádá, aby byl vpuštěn semafor a počítadlo semaforu bylo sníženo o n , kde n bývá zpravidla 1
- Operační systém, který semafor poskytuje, musí atomicky zajistit:
 - Test, zda může být počítadlo sníženo o n a zůstat nezáporné
 - Pokud ano, sníží se počítadlo a thread běží dál
 - Pokud ne, počítadlo se nesníží a thread se zablokuje



Acquire – uniprocessor

- Operace acquire musí proběhnout atomicky, takže OS musí nějakým způsobem implementovat kritickou sekci, ve které změní stav semaforu a procesu
- Na uniprocessoru může rovnou měnit příslušné struktury
 - Je však třeba pohlídat, aby během kritické sekce nedošlo k přepnutí kontextu
 - Dočasně lze pozastavit přepínání kontextu (interrupt od hodin se bude ignorovat – buď se nastaví sw vlajka, že jeho obsluha nebude ovlivňovat přepínání kontextu, nebo se hw interrupt zamaskuje)



Acquire – multiprocessor

- Na multiprocesoru nestačí zamaskovat přerušení pro jeden procesor – muselo by se to udělat pro všechny, což opět vyžaduje synchronizaci => tudy cesta nevede
- sw vlajka by musela být v globální paměti všech procesoru a nastavovala by se pomocí atomických operací => to už s nimi můžeme rovnou změnit počítadlo semaforu
- Kód snížení počítadla je založený na cyklu spinlocku

Acquire – spinlock

```
mov eax, [counter]
```

spin:

```
mov edx, eax
```

```
sub edx, 1          ; n=1
```

```
jns trylock        ; pokud je číslo záporné, není místo a musíme
```

```
call doblock       ;thread zablokovat, dokud se neuvolní
```

trylock: lock cmpxchg [counter], edx ; zkusíme nastavit novou hodnotu

;počítadla (edx) je-li stále eax==[counter]

;nepředběhnul nás jiný procesor?

```
jnz spin           ; předběhnul-li, aktuální [counter] je teď v eax
```



Acquire – uspání threadu

- Není-li možné thread vpustit dále za semafor, OS ho musí uspat
 - Stav threadu se změní na blokováný
 - Thread se přidá do seznamu threadů čekajících na daný semafor
 - Pokud by bylo $n > 1$, musí se do seznamu přidat i n
 - A do TCB se přidá semafor do seznamu entit, nad kterými je thread blokováný
- TryAcquire - Namísto toho, aby se thread v Acquire uspal, TryAcquire vrátí příslušnou chybovou hodnotu



TryAcquire - SpinCount

- Spinlock acquire lze vykonávat v uživatelském adresovém prostoru
- Lze se tedy pokoušet o získání přístupu přes semafor předem stanovenou dobu, bez přepnutí do režimu jádra, a pak
 - Acquire zavolá jádro a to thread uspí
 - TryAcquire vrátí řízení uživatelskému kódu threadu bez volání jádra, a to může dělat jinou, uživatelskou činnost
 - Např. viz `RTL_CRITICAL_SECTION.SpinCount` u WinAPI



Release

- ... též známé jako signal, up či V(vrijgave v originále, „pusť mě Ven“)
- Thread informuje, že opouští kritickou sekci, a že se má počítadlo semaforu zvětšit o nějaké m , zpravidla $m=n=1$
- Funkce OS analogicky k Acquire atomicky zvýší počítadlo o m , ale pak se ještě podívá, zda na opouštěném semaforu není blokován nějaký thread, který by mohl pokračovat



Release – vzbuzení threadu

- V základě by stačilo:
 - z neprázdné fronty čekajících threadů na daném semaforu vyjmout ten první
 - z příslušné v TCB odkazované fronty tohoto threadu vyjmout daný semafor
 - a nastavit stav threadu na runnable
- Jenže...
 - Co když thread žádal o Acquire s $n > 1$?
 - Co když je thread uspán ještě z jiného důvodu?



Release – vzbuzení threadu

- Co když thread žádal o Acquire s $n > 1$?
 - Pak je třeba vybrat thread, který byl uspán s n menším nebo rovným počítadlu semaforu
 - Jenže, co když ho předběhne thread s menším?
 - Pak se musí fronta uspaných threadů projít znovu hledat thread s vyhovujícím n .
 - Takže to má ve výsledku takovou režii, že je lepší podporovat $n=m=1$



Release – vzbuzení threadu

- Co když je thread uspán ještě z jiného důvodu?
 - Např. je-li uspán z debuggeru
 - Thread nebude převeden do stavu runnable, dokud ho bude něco blokovat
 - A tím pádem musí OS z fronty uspaných threadů vybrat další, který by bylo možné zkusit odblokovat
 - Teoreticky by bylo možné odblokovat všechny na semafor čekající thready, protože by se zase v případě neúspěchu uspaly – ale je to moc velká a zbytečná režie navíc
 - => metoda vzbouzení má vliv na celkovou režii



Producent-konzument

- Kruhový buffer, binární semafor pro přístup k `buffer[bufsize]` , semafor pro zápis a semafor pro čtení
 - Oba inicializovány na limit `bufsize`, počítadlo pro čtení na 0 a pro zápis na `bufsize`
 - Binární semafor lze nahradit atomickými operacemi

Producent:

`P(pro_zápis), P(buffer)`

`vlož_do_bufferu`

`V(pro_čtení), V(buffer)`

Konzument:

`P(pro_čtení), P(buffer)`

`vyber_z_bufferu`

`V(pro_zápis), V(buffer)`



Mutex

- Sice má navenek tu samou funkcionalitu jako binární semafor, ale:
 - Může mít vlastníka – jenom ten thread, který ho zamknul ho může odemknout
 - Může poskytovat inverzi priorit
 - Může zabránit ukončení threadu, který mutex uzamknul



Roura

- Roura je buffer, který má dva souborové deskriptory, jeden pro zápis a jeden pro čtení
- A když už má roura souborový deskriptor, může mít i souborové jméno
 - Pojmenovaná roura je pak persistentní, jinak roura zaniká s posledním procesem, který ji mohl používat
- Roura se často využívá k přesměrování výstupu jednoho konzolového programu na vstup druhému



Roura – zápis a čtení

- Buffer roury má omezenou velikost, takže musíme ošetřit, aby thready zapsaly jenom tolik, kolik je v ní místa
- Aplikujme úlohu producent konzument
 - Buffer bude kruhový
 - Producent bude zapisovat n bytů
 - Konzument bude vybírat m bytů
 - => a známe řešení/implementaci na bázi semaforů
 - Které ovšem musíme ošetřit pro specifické případy – např. když producent bude chtít zapsat více bytů, než kolik je velikost bufferu
 - Producentů i konzumentů může být několik



stdin, stdout, stderr - Linux

- V POSIXovém systému uděláme close požadovaného handle
- OS pak jeho číslo použije jakmile vytvoříme nový souborový deskriptor, anebo budeme duplikovat handle

```
dup2(fileno(newstdinopenedfile), STDIN_FILENO);
```

```
dup2(fileno(newstdoutopenedfile), STDOUT_FILENO);
```

```
dup2(fileno(newstderropenedfile), STDERR_FILENO);
```

```
fclose(newstdinopenedfile);
```

```
fclose(newstdoutopenedfile);
```

```
fclose(newstderropenedfile);
```

stdin, stdout, stderr - WinAPI

```
PROCESS_INFORMATION piProcInfo;
```

```
STARTUPINFO siStartInfo;
```

```
BOOL bSuccess = FALSE;
```

```
ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
```

```
siStartInfo.cb = sizeof(STARTUPINFO);
```

```
siStartInfo.hStdError = g_hChildStd_OUT_Wr;
```

```
siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
```

```
siStartInfo.hStdInput = g_hChildStd_IN_Rd;
```

```
siStartInfo.dwFlags |= STARTF_USESTDHANDLES;
```

```
bSuccess = CreateProcess(NULL, szCmdline, NULL, NULL, TRUE, 0, NULL,  
NULL, &siStartInfo, &piProcInfo);
```

Zprávy

- Významná forma synchronizace MS Windows
 - Zejména u GUI
 - Všechny vizuální prvky jsou window, která přijímají a posílají zprávy
 - Přičemž v main je hlavní smyčka zpráv
 - Konzolové aplikace ji nepotřebují, ale mohou použít
- Jeden thread doručí zprávu druhému threadu
 - Může i nemusí čekat, až ji příjemce zpracuje
- OS spravuje frontu příchozích zpráv per thread

Zprávy – hlavní smyčka

```
int wmain() {  
    CreateWindow(....)  
    while(GetMessage( &msg, NULL, 0, 0 )) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
    return msg.wparam;  
}
```

- Každé window má svou WindowProcedure, která zprávy přijímá

Zprávy – WindowProcedure

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM
IParam) {
    switch (uMsg) {
    case WM_SIZE: {
        int width = LOWORD(IParam); // Macro to get the low-order word.
        int height = HIWORD(IParam); // Macro to get the high-order word.

        // Respond to the message:
        OnSize(hwnd, (UINT)wParam, width, height);
    }
    break;

        return DefWindowProc(hwnd, uMsg, wParam, IParam);
    }
}
```



Zprávy – odesílání

- PostMessage – odešle zprávu, nezajímá ho výsledek
- SendMessage – odešle zprávu, ale je blokován, dokud ji příjemce nezpracuje a nevrátí výsledek (int)
- WM_COPY data – jeden z parametrů je ukazatel na blok paměti, který je při doručení do jiného procesu přístupný v paměti procesu příjemce
 - Lze použít při SendMessage
- Jak to implementovat?



PostMessage – implementace

- OS volá WindowProcedure a serializuje zprávy jí zpracovávané
- Po dokončení WindowProcedure musí OS provést vyjmutí zprávy z fronty zpráv
- Při PostMessage
 - Zpráva se pouze přidá do fronty příjemce, odesílatel pokračuje s vykonáváním kódu
 - Až ji příjemce zpracuje, OS ji vyjme z jeho fronty zpráv



SendMessage – implementace

- Se zprávou musí být svázaný nějaký synchronizační prostředek, nad kterým se odesílatel uspí/bude blokován, dokud příjemce nezpracuje odesílanou zprávu
 - V principu jde o to samé, jako u semaforu
- Až příjemce zprávu zpracuje, tj. kód OS dostane řízení po návratu z WindowProcedure, OS překopíruje eax příjemce do eax odesílajícího (tj. zkopíruje návratovou hodnotu), a zruší blokaci odesílajícího nad SendMessage



WM_CopyData– implementace

- Speciální zpráva, umožňující předat velké množství dat
- WinAPI říká, že:
 - odesílající nemá modifikovat odesílaný blok paměti, dokud SendMessage neskončí
 - Příjemce nemá tento blok paměti modifikovat, jako by pro něj byl read-only
- Windows jsou sice privátní OS s uzavřeným kódem, ale možnou implementaci si už lze dovodit



WM_CopyData– implementace

- Při SendMessage OS dočasně namapuje stránky odesílajícího procesu do adresového prostoru příjemce, a nechá je jako read-only
- Před voláním WindowProcedure příjemce ale OS musí upravit pointer ukazující na předávaná data tak, aby tento pointer ukazoval na předávaný blok paměti na adrese, kam byly stránky namapovány
- Po návratu z WindowProcedure OS stránky odmapuje
- Chce-li si příjemce data ponechat, musí si je sám zkopírovat – což říká i dokumentace WinAPI



Signály

- Významná forma synchronizace v POSIXu
- Obsluha signálu je rutina, identifikovaná číslem, která se vyvolá při události, jíž toto číslo odpovídá
 - Podobnost s tabulkou vektorů přerušení
- Např. uživatel konzolové aplikace stiskne Ctrl+C
 - OS transformuje stisk této klávesy na signál SIGINT (signal to interrupt the process) a naplánuje vykonání příslušné obsluhy
 - Co se stane dále, to závisí na tom, co daná obsluha signálu dělá



Signály – default handler

- Co by se stalo, kdyby chtěl OS vykonat obsluhu signálu, ale proces by pro něj nenastavil obslužnou rutinu?
 - Bud' by došlo na Segmentation Fault (což je signál SIGSEGV, u MS výjimka Access Violation), anebo by se začal vykonávat náhodný kód, což by nejspíš také skončilo výjimkou
- OS každému procesu při jeho vytvoření poskytne tzv. default handler pro každý signál
 - Dokud proces nenastaví svou obsluhu, vykonává se obsluha OS



Signály – user handler

- Např. default handler SIGINT ukončí proces
- Když ale proces bude provádět nějakou činnost, např. kopírování souboru, jeho programátor může chtít, aby Ctrl+C pouze ukončilo aktuální činnost
- => proces si nainstaluje vlastní obsluhu SIGINT, která pouze ukončí aktuální činnost



Signály – ignorování

- Většinu signálů lze také ignorovat – tj. pokud nastanou, neprovede se žádná obsluha
 - Ani uživatelská, ani OS
- Vyjímkou jsou dva signály, které nelze ignorovat, ani pro ně nastavit vlastní obsluhu
 - SIGKILL – ukončí proces
 - SIGSTOP – zastaví proces



Signály – Process-Directed

- Podle POSIXu všechny jádrem plánované thready mají mít stejný PID
- Takže pokud se má vykonat obsluha signálu, OS vybere thread, který ji vykoná – Process-Directed signal
- Nicméně, thread má možnost pomocí `pthread_sigmask` možnost ignorovat signály pro sebe a své potomky
 - Takže je možné zpracovat signály v jednom, konkrétním threadu
 - Množina threadů, ze které OS vybírá, se omezí na jeden prvek
 - Bude-li prázdná, ukončí se celý proces



Signály – Thread-Directed

- Vedle Process-Directed signálů jsou také Thread-Directed
- Funkce `*kill` má jako jednu z parametrů číslo signálu
- Funkce `kill` pošle signál procesu
 - Např. `SIGKILL`
- Funkce `pthread_kill` pošle signál konkrétnímu threadu
 - Např. `SIGSEGV` je signál pro konkrétní thread, ve kterém došlo k neoprávněnému přístupu do paměti
 - Jestliže thread nemá definovanou obsluhu pro signál, jehož defaultní akcí je ukončení procesu, není ukončen thread, ale celý proces



Signály – fork, exec

- Potomek nebude mít signalizován žádný signál, i kdyby jeho rodič měl
- Uživatelské obsluhy a ignorování signálů je zděděno
- Exec přepíše stávající kód novým kódem
 - Přepíše i kód stávajících uživatelských obsluh signálů
 - => všechny obsluhy signálů jsou nastaveny do výchozího stavu
 - Jinak by se stalo to, co je popsáno na slidu „Signály – default handler“



Signály – implementace

- Per thread/proces, jádro si udržuje
 - seznam obsluh signálů,
 - spinlock, který chrání přístup k nim,
 - 64-bitovou masku ignorovaných signálů
 - 64-bitovou masku signálů čekajících na obsluhu
 - Obousměrný spojový seznam signálů čekajících na obsluhu
 - Každá položka ještě obsahuje OS-specifické info
- Standardní signál vždy čeká jenom jeden
 - Ještě jsou rt-signály



Real-Time signály

- Existuje 32 standardních signálů (0-31) s předdefinovaným významem a 32 real-time signálů (32-63) bez předdefinovaného významu; akce neobslouženého rt signálu je terminate process
 - Real-time indikuje, že mají být obslouženy ASAP – ne real-time
 - Používá je např. Native POSIX Thread Library pro běh programů využívajících POSIX threads
- Od standardních signálů se liší
 - Ve frontě může být několik instancí jednoho rt signálu
 - Jsou doručovány v garantovaném pořadí
 - Obsahují další systémové info



Signály vs. zprávy

- Konzolová aplikace ve Windows může nastavit obsluhu např. pro Ctrl+Z
- Ekvivalentem SIGKILL je ve Windows GUI zpráva WM_QUIT
- Linux má 64 signálu, navržené s ohledem na konzolové aplikace, Windows $2^{\text{sizeof(int)}}$ zpráv navržené s ohledem na GUI aplikace
- GUI v Linuxu používá jiné mechanismy, např. signály a sloty dle Qt, které se dají provozovat i pod Windows