•

POS příprava na semestrálku

- Jako v minulem zkouskovem, pouzivejte komentare - namisto psani poznamek do textu.

Odkazy materiály:

- wiki: http://wiki.fituska.eu/index.php/Fulltextov%C3%A9_ot%C3%A1zky_POS
- semafory, synchronizační úlohy: http://wiki.fituska.eu/index.php/Semafory#Implementace_semaforu_v_POSIX
- nějaké poznámky, ale skoro nic tam není: https://docs.google.com/document/pub?id=1Vm-pLA9ZuyuRoF1-3XPsHKxiMGj-NOqvmC4FA2Cxb0s
- vypracované státnicové okruhy: http://wp.soulwasted.net/category/msz/pos DEAD!
- fituska 2008/2009 vypis: https://fituska.eu/download/file.php?id=3285

Otázky

2015
Zadání prvního termínu:
11 otázek za 6b, poslední příklad za 5b, žádné zákeřnosti, vpodstatě vše co už se někdy objevilo

- Jaké formě muze byt rozhrani. Příklady rozhranní a standardy.
- Jak vznikají a zanikají vlakná. Porovnat režii, co je jednotkou provádění.
- Co je to inverze priority a jak se dá řešit.
- Podle ruznych algoritmu nakreslit gantuv diagram, jak budou se stridat procesor pro 4 procesy. Je dano okamzik, kdy proces startoval, a delka kolik ma bezet.
- Metody prevence uváznutí při přidělování SR prostředků
- Spočítej počet výpadků stránek pro nahrazovací algoritmy OPT, LRU a FIFO ze zadaného sledu odkazů.
- organizace logického adresového prostoru a jádra systému, výhody a nevýhody
- Průběh fork() z hlediska správy paměti a zmenšení jeho režie.
- Organizace tabulky stránek, principy a vysvětlit jak to funguje u Intel x86-32b.
- Příklad na doplnění POSIX mutexu s condition. Byl tam čekající a signalizující. První proces měl signalizovat, že je něco připraveno, a pak čekat až to ten druhý zpracuje a.,
-
-
- pak skončit. Druhý měl počkat, až ten první zasignalizuje, zpracovat to a pak zasignalizovat prvnímu, že skončil.

Zadání 1. opravného termínu:

- Zakreslit Gantův diagram: FIFO, SRT, SJF + napsat průměrný čas zpracování procesů
- Algoritmy: FIFO, SCAN a circular SCAN - spocitat celkove posunuti hlavy
- Zapsat algoritmus vzájemného vyloučení s použítím swap
- Zapasat algoritmus čt enáři-písaři bez stárnutí písaře
- Metadata souboru, (jak se vyhodnocují?, kdy se zapisují?)
- Rozdíl implementace čekání u monitorů a semaforů

2014

1. V jaké formě muze byt rozhrani (binarni a ... , to bylo v prvni prednasce)

- binarni - assembler, HW zavislé; standardy: iBCS-2 (Intel), MIPS ABI (SGI)
- zdrojové - C/C++; standardy: SVID (Unix System V Interface Definition), X/Open, IEEE/ISO POSIX 1003.1 (pro jazyk C), Open Group

2. Jak se resi kritická sekce v jadre a v podprogramech pri preruseni. Lze k tomu vyuzit semafor. t

- zakázat přerušení
- zakázat preemptivní přepínání
- vzájemné vyloučení zamykaním datových struktur a povolení preemptivního přepínání jádra (jsem v KS, je mi odebrán CPU, ale zámek je pořád zamčený - nikdo nic nezmění)

3. Porovnejte rezii v prepinani kontextu v ruznych implementacich vlaken a procesu. Ve ktere z implementace lze vlakna pouzit spulu s procesy.

Implementace vláken

• N:N (1:1) - na úrovni jádra systému, vlákna na úrovni uživatelské jsou reprezentována v jádře (OS/2, AIX < 4.2, Windows/NT, LinuxThreads, NPTL – clone()):

− režie přepínání kontextu
− jádro musí evidovat všechna vlákna (datové struktury v jádře)
+ plné zakázat přerušení, zakázat přerušení využití více procesorů v jednom programu
+ volání jádra přímá (nemusí být zapouzdřena)

• N:1 - na úrovni knihoven, vlákna jsou plně implementována v rámci uživatelského procesu, jádro o nich nic neví (DCE, FreeBSD < 5.x)

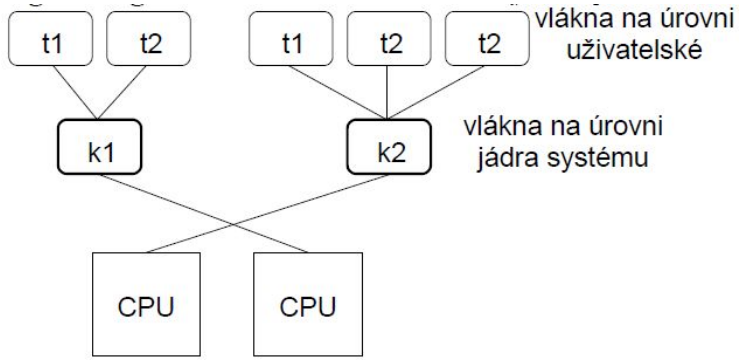
+ nízká režie jádra, plná kontrola nad plánováním
− všechna blokující volání jádra musí být zapouzdřena
− nelze využít více procesorů v jednom programu, jednotkou přidělování času procesoru je proces

• N:M (N≥M≥1) - kombinovaný přístup, důvody:

• prováděná vlákna musí být reprezentována v jádře pro správu procesorů, nicméně nemá smysl reprezentovat všechna běžící, stačí tolik, kolik je procesorů,
• čekající (pozastavená) a připravená vlákna nemusí být reprezentována datovými strukturami v jádře, jádro o nich vůbec nemusí vědět - menší režie

• přepínání kontextu přes jádro má větší režii než v rámci uživatelského procesu – dokud lze využít přidělený procesor, probíhá běh v režimu sdílení času pro všechna aktivní vlákna v daném procesu.
• lze simulovat N:1 až N:N nastavením max. počtu vláken na úrovní jádra (thr_setconcurrency()) SOLARIS (LWP), AIX, Irix 6.5, FreeBSD 5.x (KSE) - vlákna na úrovni uživatelské lze vázat dynamicky nebo pevně na vlákna jádra systému

Light Weight Process - vlákno na úrovni jádra systému



zakázat přerušeníImplementace modelu M:N je značně složitá, LWP mají v jádře obdobnou režii jako procesy (bez adresového prostoru). LWP jsou z hlediska jádra jednotkami přidělování procesorů. Klasický proces pak běží jako 1 LWP vlákno.

Problém modelů N:N a N:M – podpora vláken na úrovni uživatelské nemá dostatečné informace o akcích na straně jádra, plánování je problém.

4. Spocitejte kolik vypadku stranek maji pri pouziti algoritmu ...(ktere presne tam byly nepamatuji).
5. Jsou dane zdroje, nakreslit grafy, provest redukce a zjistit jestli system je ve stavu uvaznuti.
6. Podle ruznych algoritmu nakreslit gantuv diagram, jak budou se stridat procesor pro 4 procesy. Je dano okamzik, kdy proces startoval, a delka kolik ma bezet.
7. napsat nejaky programek, doplnit pouzitim pthread_wait,pthread_cond atd. Master proces pripravi neco, posle signal synovi a ceka, jak se docka pokracuje, a pak ceka dokud syn neskonci. Syn ceka na master od zacatku, zacina jakmile obdrzi sigóonal od master,pokracuji, posila master signal, ze skoncil.
8. ostatni byly ze seznamu na fitwiki.

2013

5, 9, 11, 12, 13(tusim,ze mierne upravena), 18, 23 + bonusovka bol napisat kod: obecne riesenie pisari/ctenari pomocou mutexov a conditionov (predpis funkci ako wait,unlock,atd boli zadane)

2012

Rozdíl mezi semaforem a monitorem, proč se doporučuje v dnešní době používat semafor?
Podmínky pro uváznutí v systému SR. Jmenujte postačující podmínku (nebo tak nějak)
Segmentace vs. stránkování, typy fragmentace které u každého vznikají, organizace paměti.
Problémy signálu v klasickém Unixu, jak to řečí POSIX.
Víc si nepamatuji

Doplním co tam ještě bylo, nezkoumal jsem co se opakovalo z minula:

-- něco o multiprogramování, technické prostředky nutné pro implementaci jádra

-- Principy implementace krátkodobého vzájemného vyloučení

-- ganttův diagram a určit průměrnou dobu zpracování pro plánování FIFO, SJF, SRT a byly dány 4 procesy, jejich start a doba trvání

-- organizace logického adresového prostoru a jádra systému

-- počet výpadků stránek pro nahrazovací algoritmy OPT, LRU a FIFO, a zadán sled odkazů

-- průběh fork z hlediska správy paměti, pak co zlepší execl()

-- Synchronní a asynchronní provádění V/V operací v rozhraní jádra systému

-- Metadata v systémech souborů, co vše zahrnují, problém nekonzistence metadat při výpadku a způsoby řešení

-- implementace podmíněné kritické sekce pomocí POSIX 1003.1c, zadány nějaké deklarace fcí a ještě něco, to nevím co to bylo ...

1. Operační systém, služby jádra

Operační systém (OS)

- most mezi hardwarem a běžícím programem
- most mezi hardwarem a uživatelem
- vytváří prostředí pro běh programů (procesů, vláken)

2. Architektura počítačů, hierarchie zdrojů, průběh V/V a multiprogramování

Architektura počítačů

• jeden nebo více procesorů,
• V/V řadiče a procesory komunikují každý s každým (sběrnicí nebo křížovým přepínačem) nebo omezeně (V/V pouze jeden),
• procesory a V/V řadiče pracují nezávisle a paralelně.

Počet procesorů a organizace paměti (synchronizace):

• jednoprocesorové
• víceprocesorové:
    - jedna sdílená paměť (UMA),
    - rozprostřená sdílená paměť (NUMA),
    - distribuované systémy (bez sdílené paměti)

3. Typy operačních systémů, požadavky na technické vybavení

Typy operačních systémů

Plánování:

• dávkové (batch),
• sdílení času (timesharing),
• systémy reálného času (real-time)

Použití:

• univerzální
• specializované (souborový server, databázový server, RT)

Počet uživatelů:

• jednouživatelské,
• víceuživatelské

A) Monoprogramové - MS-DOS:

• aktivní pouze jeden proces,
• jednodušší implementace - nenastává souběžnost provádění,
• využití zdrojů slabé, obvykle jen jeden uživatel.

B) Multiprogramové (multitasking, multiprogramming):

• aktivních více procesů současně,
• efektivnější využití prostředků,
• jednodušší implementace vyšších vrstev (GUI, síť. rozhraní),
• nutnost pro víceuživatelský systém.

B-1) Jednoprocesorové (uniprocessor, UP)

B-2) Víceprocesorové, paralelní (multiprocessor, MP)

B-2-a) Symetrické multiprocesorové systémy (SMP)

• kód jádra i kód procesů je prováděn na všech procesorech,
• procesory mají rovnocenný přístup k operační paměti, V/V zařízením a přerušovacímu systému.

B-2-b) Nesymetrické multiprocesorové systémy

Jak realizovat současný běh více procesů než je počet procesorů?

Ochrana operačního systému:

1. znemožnění modifikace kódu a datových struktur jádra OS,
2. zabránění provádění V/V operací,
3. zabránění přístupu do paměti mimo přidělený prostor,
4. odolnost vůči chybám (odebrání procesoru při zacyklení).

Nutná podpora na úrovni hardware:

1. Dva režimy činnosti procesoru,
2. privilegované instrukce povolené pouze v systémovém režimu (V/V, změna režimu, zpracování přerušení),
3. ochrana paměti – definuje přístupné úseky adresového prostoru pro běžící proces,
4. přerušovací systém, přerušení převede procesor do systémového režimu,
5. generátor pravidelných přerušení (časovač),
6. pro efektivní V/V nutné DMA nebo inteligentní řadič (přenosy velkých objemů dat)

4. Režimy činnosti procesoru, přechody mezi režimy, volání jádra

Uživatelský režim – provádění uživatelských procesů

Systémový režim – provádění kódu jádra OS

Přechody mezi režimy:

a) Přerušení – vždy do systémového režimu, na definovanou adresu

b) Návrat z obsluhy přerušení – návrat zpět (IRET)

c) Volání jádra – privilegovaná instrukce, přechod do systémového režimu, na pevnou adresu, parametry obvykle na zásobníku v uživatelském adresovém prostoru

d) Návrat z volání jádra – privilegovaná instrukce pro přechod do uživatelského režimu a nastavení programového čítače (PC)

Volání jádra

• volání jádra je realizováno speciální instrukcí (SVC, lcall, int, trap),
• jediný styk procesu s okolním prostředím,
• vše je zprostředkováno jádrem operačního systému,

• proces je zapouzdřen, operační systém pro něj vytváří iluzi virtuálního počítače .

5. Definice rozhraní jádra systému, standardizace

Definice rozhraní jádra

A) Na úrovni binární:

```
# na zásobníku fd, buf, length
read:   lea $0x3,%eax
        lcall $7,$0 (Linux - int $0x80)
        jb error
        ret
error:  movl %eax,_errno
        movl $-1,%eax
        ret
```

Binární rozhraní je systémově závislé (procesor, verze systému)
Standardy: iBCS-2 (Intel), MIPS ABI (SGI), apod.

B) Na úrovni zdrojové:

```
read(fd, buffer, length); /* pro C/C++ */
```

Historicky také systémově závislé (např. datový typ length)

Standardiazce

Sjednocování rozhraní řeší *přenositelnost* – signály, terminály, synchronizace,…

SVID – UNIX System V Interface Definition

- definice rozhraní originálního komerčního UNIXu z AT&T

X/Open (Open Group) – X/Open Portability Guide (XPG)

- sdružení výrobců, širší definice rozhraní jádra, bez omezení na konkrétní verzi UNIXu

IEEE POSIX 1003.1

- rozhraní jádra systému pro jazyk C
- součást standardů POSIX
- mělo více revizí, přidávání vlastností (vlákna)

6. Techniky strukturování jádra operačních systémů

1. Monolitické jádro (monitor)

• bez vnitřní struktury (big mess)
• volání mezi moduly voláním podprogramů
• žádné omezení volání a vztahů mezi moduly
• uživatelský proces = podprogram jádra
• často bez rozdělení na systémový/uživatelský režim

2. Jádro (kernel)

• jádro běží v systémovém režimu
• procesy běží v uživatelském režimu a volají jádro (jádro je pasivní), striktní rozhraní mezi procesy a jádrem
• jádro vytváří pro proces abstrakci virtuálního počítače

3. Mikrojádro (Mach)

• Služby jádra částečně v systémovém režimu (mikrojádro), částečně v uživatelském režimu (systémové procesy)
Minimální jádro - úkoly:
• přepínání kontextu
• přidělování paměti
• ochrana paměti, nastavení adresového prostoru
Ostatní služby řešeny samostatnými procesy nad mikrojádrem:
• prostředí procesů, spouštění procesů
• autentizace, autorizace, účtování
• virtualizace paměti, odkládání, zavádění
• V/V
• síťové vrstvy
• systém souborů

4. Exokernel

Služby jsou poskytovány jako podprogramy uvnitř uživatelského procesu. Jádro v systémovém režimu je voláno pouze pro synchronizaci a přidělování prostředků.

5. Virtuální počítač (Virtual Machine)

• jádro běží zcela v uživatelském režimu
• v systémovém režimu běží pouze monitor virtuálního počítače - zachytává a emuluje privilegované instrukce
• plně virtualizuje všechny prostředky

Princip - vrstvy s definovanou funkcí, volání pouze podřízených vrstev.
1. virtuální procesor - přepínání kontextu, synchronizace
2. přidělování paměti, uvolňování

3. plánování - přidělování procesoru, zastavení, synchronizace na vyšší úrovni
4. přidělování paměti na vyšší úrovni, DTA
5. V/V - zahájení, zpracování přerušení
6. Síťové vrstvy
7. spouštění a ukončování procesů
8. systém souborů
9. virtualizace paměti
10. rozhraní jádra, prostředí procesu

- Makrojádro
- Makrojádro s moduly
- Mikrojádro se službami

7. Paralelní systém procesů, precedenční relace, graf pokrytí, posloupnost provádění

8. Deterministický paralelní systém, nezávislost, interference, Bernsteinovy podmínky

Bude výsledek paralelního systému při paralelním provádění vždy stejný bez ohledu na posloupnost provádění?
Pokud ano, pak nazýváme paralelní systém časově nezávislým, deterministickým.

Def.: Paralelní systém je deterministický, jestliže pro daný počáteční stav s0 je $Vx(\alpha) = Vx(\alpha')$, $1 \leq x \leq m$, pro všechny posloupnosti provádění α a α'.
Jinak: Posloupnost hodnot zapisovaných do všech proměnných závisí pouze na počátečním stavu proměnných.

Def.: Bernsteinovy podmínky neinterference:
Dva procesy Pi a Pj jsou neinterferující, jestliže platí:
1. Pi < Pj nebo
2. Pj < Pi nebo
3. $R(Pi) \cap W(Pj) = W(Pi) \cap R(Pj) = W(Pi) \cap W(Pj) = \varnothing$
Věta: Paralelní systém skládající se ze vzájemně neinterferujících procesů je deterministický.

Nezávislé procesy mají $R(Pi) \cap W(Pj) \neq \varnothing$

9. Procesy a vlákna, vztah, použití, metody implementace vláken

Proces je instance programu v paměti (vlastní adresový prostor), která se vykonává. Má jednoznačnou identifikaci (PID). Může být více procesů pro jeden program.
Vztahy: nové procesy vznikají duplikací běžícího rodičovského procesu (fork()) existuje vztah otec-syn. Nejvyšším prarodičem je proces init. Při ukončení otce se synové přesouvají k init. Při ukončení syna si otec vybere stav. Pokud otec na stav nečeká, stav visí v paměti a ze syna se stává zombie.
Vlákno je samostatně prováděná část programu v rámci jednoho procesu. Takto může jeden proces běžet na více procesorech paralelně. Vlákna jednoho procesu sdílí logický adresový prostor a systémové prostředky. Registry, zásobník a stav provádění programu se uchovává pro každé vlákno samostatně. *Použití*: I/O vlákno vedle výpočtů, GUI, u více procesorů nutnost pro výkon. *Výhody*: rychlejší než fork(), sdílení celého adresového prostoru (bez ochrany), pro zásobník nemusí řešit vícenásobný přístup.
Proces s vlákny: je pak jen obalová jednotka pro vlákna, která jsou "procesy" z pohledu procesoru proces, tak pouze uchovává vnější stav (PID, deskriptory apod.) společný všem vláknům. V UNIXu je tedy proces jednotkou přidělování prostředků a vlákno jednotkou přidělování procesoru.
Implementace: (pocet_procesu : pocet_vlaken)

- *N:N (1:1)* – vlákna na úrovní OS,
    - (+) volání je přímé a rychlé, plné využití multiprocessingu.
    - (-) evidence všech vláken v paměti jádra, větší režie během přepínání.
- *N:1* – OS vidí jen procesy, vlákna jsou v userspace pomocí knihoven.
    - (+) nízká režie vlákna (paměťová i časová), plná kontrola plánování (nezasahuje OS).
    - (-) blokující volání jádra se zapouzdřují, nejde použít pro více procesorů (CPU, dostane celý proces)
- *N:M* – kombinovaný přístup, OS vidí M vláken z OS.
    - (+) OS vidí jen potřebná vlákna (většinou podle počtu CPU), čekající vlákna už OS nevidí (bez režie), zároveň lze přepínat kontext i v userspace.
    - (-) velmi problematická implementace.

10. Vlákna POSIX 1003.1c - vytváření vláken, ukončení, přebrání stavu

```
int global; /* sdílená proměnná */
void *vlakno(void *arg)
{
        int local; /* privátní prom.vlákna */
        static global; /* sdílená proměnná */
        ... /* kód vlákna */
        return stat; /* ukončení vlákna */
        /* pthread_exit(void *stat); /* ekvivalentní*/
}
```

Vytvoření a spuštění vlákna – pthread_create():

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr, /* atributy */
        void *(*func)(void *), /* vlákno */
        void *arg); /* parametr */
```

Čekání na ukončení vlákna a převzetí stavu ukončení:

```
int pthread_join(pthread_t thread, void **st);
```

11. Současnost, souběžnost, atomické operace, synchronizace

Přepínání kontextu → **souběžný** běh více procesů
Def.: **Atomická operace** - nedělitelná operace, nemůže být přerušena uprostřed.
Význam: Pokud jsou prováděny všechny operace nad sdílenou datovou strukturou atomickými operacemi, zůstává stav struktury konzistentní i při paralelním přístupu.
Synchronizace: zajištění kooperace mezi paralelně (souběžně) prováděnými procesy.

12. Vzájemné vyloučení, podmínky, které musí splňovat obecné řešení, vztah k přidělování procesoru

Synchronizace: zajištění kooperace mezi paralelně (souběžně) prováděnými procesy
Vzájemné vyloučení (mutual exclusion) – základní úloha synchronizace: pouze jeden proces může provádět danou operaci. Vytváříme tím složitější atomickou operaci (nedělitelná operace, nemůže být přerušena uprostřed).
Kritická sekce – kód, jehož provádění je vzájemně vyloučené.

aOmezující podmínky:
1. Sekce start a výpočet mohou být libovolně dlouhé, trvat libovolnou dobu, včetně nekonečné (proces zde může skončit).
2. Kritická sekce je provedena vždy v konečném čase (proces zde nesmí zůstat čekat, ani skončit).
3. Proces musí mít zaručen nekonečně krát vstup do kritické sekce v konečné době (liveness).
4. Přidělování procesoru je nestranné, spravedlivé (weak fairness).

Fairness (spravedlnost) přidělování procesoru:
• unconditional fairness (nepodmíněná) - každý aktivní nepodmíněný atomický příkaz bude někdy proveden (může být dlouhodobě prováděn pouze jeden proces)
• weak fairness = unconditional fairness + každý aktivní podmíněný atomický příkaz bude proveden za předpokladu, že podmínka nabude hodnoty TRUE a nebude se měnit.
• strong fairness = unconditional fairness + každý podmíněný atomický příkaz, jehož podmínka se nekonečně častokrát mění, bude nekonečně krát proveden (prakticky nerealizovatelný plánovací algoritmus, musel by střídavě provádět po jedné atomické instrukce jednotlivých procesů).

Formální požadavky na hledané řešení:
• bezpečnost (safeness), v daném případě zaručuje vyloučení
• živost (liveness):
- nedochází k uváznutí (deadlock)
- nedochází k blokování (blocking)
- nedochází k stárnutí (starving)

13. Živost, uváznutí, blokování a stárnutí (hladovění)

Živost (liveness) - algoritmus je živý, pokud je bezpečný a nedochází k uváznutí, blokování a stárnutí (je zaručeno jeho dokončení v konečné době).
Uváznutí (deadlock) - procesy čekají v synchronizaci na stav, který by mohl nastat, kdyby jeden z nich mohl pokračovat.

Blokování (blocking) - proces čeká v synchronizaci na stav, který generuje jiný proces, a toto čekání není nutné z hlediska synchronizace (kritická sekce je volná). Postup procesu je blokován jiným procesem.
Stárnutí (starving) - proces může čekat v synchronizaci na stav, který nemusí být nikdy pravdivý v okamžiku testování. Není striktně omezena horní mez čekání (jinak jako blokování). V praxi se obvykle toleruje, závisí na plánovacím algoritmu.

14. Vzájemné vyloučení u jednoprocesorových systémů

Atomický kód – úsek kódu, kde nemůže dojít k přepnutí kontextu (multiprogramování) nebo přerušení (souběžný běh ovladače a procesů)
Vzájemné vyloučení mezi:

1) **procesy/vlákny a obsluhou přerušení** (zakázání přerušení). zaručuje, že obsluha přerušení nenaruší KS komunikace s řadičem trvá moc dlouho (vypnutí a zapnutí má vysokou režii)
2) **různými procesy/vlákny v jádře** (zakázání přepnutí kontextu): synchronně (zahájení čekání, spuštění jiného procesu); asynchronně, na externí událost (přerušení od časovače, apod.)
    a) zakázat přerušení – blokuje přepínání kontextu, ale i I/O.
    b) zakázat preemptivní přepínání kontextu v jádře - funguje explicitní přepínání, ale není výkonově vhodné pro systém.

c) vzájemné vyloučení zamykáním datových struktur (bin. semafor) a povolení preemptivního přepínání jádra

3) **uživatelskými procesy.** futex (fast user-space mutex). dříve používaly nástroje v jádře – semafory, apod.

15. Vzájemné vyloučení u víceprocesorových systémů

Vždy nutná synchronizace, nelze obejít (zakázání přerušení nestačí pro zamezení přepnutí kontextu – ostatní procesory běží současně a mohou také provádět kód jádra).

1) Krátkodobé vzájemné vyloučení (spin_lock).

Může střežit pouze kritické sekce, které jsou krátké, neblokující a bez preempce (proces/vlákno nesmí být pozastaveno, nesmí na nic zahájit čekání). Aktivní čekání je v tomto případě přijatelné, protože pak může být kritická sekce obsazena pouze procesem běžícím na jiném procesoru a ten ji brzy uvolní. Pozastavení procesu by bylo náročnější než krátké aktivní čekání (a vyžadovalo by opět vzájemné vyloučení). Krátkodobé vyloučení je nutné pro implementaci synchronizačních nástrojů (mutex, semafor, atd.).

a) Implementace pouze čtením/zápisem

• elegantní algoritmy pouze pro malý počet procesů, složitost

• dostupnost lepších speciálních atomických instrukcí

b) Speciální atomické instrukce

Nutná atomická instrukce nedělitelného čtení a zápisu (RMW) do paměti (musí korektně fungovat ve víceprocesorovém systému se sdílenou pamětí!)

Probltakémy

cyklus test&set zatěžuje pam. sběrnici (stále čte a zapisuje)

u HT se blokuje druhý kontext aktivním čekáním (také možnost stárnutí)

-> řešením je využít cache a pro HT spec. instrukce

2**) Dlouhodobé vyloučení.** Využívá bin. zámek (první sekce zamkne a odemkne při odchodu). Impl. pomocí test&set.

16. Binární semafor, operace, implementace, použití

Operace

- init(sem, v) – inicializace semaforu sem na hodnotu v = 0, 1
- lock(sem) – zamčení, čekání na nulovou hodnotu a nastavení v na 1
- unlock(sem) – odemčení, nastavení v na 0 a odblokování procesů čekajících v lock()

nelze číst hodnotu (může se změnit)

čekání v lock(sem) je pasivní

lock(sem) a unlock(sem) jsou atomické

odemykat může jiný proces než zamknul (předávání zámku)

silný/slabý semafor – nepodléhá/podléhá stárnutí
Mutex (mutual exclusion) – speciální binární semafor určený pouze pro vzájemné vyloučení, při zamčení má identifikovaného vlastníka, pouze vlastník ho může odemknout (nutné pro řešení inverze priority)

Použití
a) vzájemné vyloučení
init(sem, 0); /* volný */
while (1) {
lock(sem); ENTRY
kritická sekce
unlock(sem); EXIT
výpočet
}
b) signalizace událostí
nevhodné
řešeno obecným semaforem

17. Obecný semafor, operace, implementace, použití

Počáteční hodnota určuje „kapacitu“ semaforu – kolik jednotek zdroje chráněného semaforem je k dispozici. Jakmile se operací down() zdroj vyčerpá, jsou další operace down() blokující, dokud se operací up() nějaká jednotka zdroje neuvolní.

- interní hodnotou je celé číslo
- pokud je nula, je zamčen a čeká se na navýšení hodnoty

Operace
init(sem, v) – inic. sem. sem na hodnotu v >= 0
down(sem) – zamčení, atom. op. čekání na hodnotu > 0 a pak sníží o 1 a pokračuje
up(sem) – odemčení, zvýší hodnotu a případně odblokuje další čekající proces

jiné použití jako P() a V() nebo wait() a signal().
pamatuje si počet up() a down()

Použití
a) vzájemné vyloučení - ekvivalentní bin. semaforu
b) signalizace událostí

init(sem, 0);
P1: P2:
... up(sem);
down(sem); /* čeká až P2 provede up() */

oproti bin. sem. bezpečné (žádná událost se nemůže ztratit)

c) hlídání zdroje s definovanou kapacitou N:

```
init(sem, N);
Pi:
down(sem); /* pokud je volno, pokračujeme dále */
... /* nejedná se o kritickou sekci, je zde až N procesů současně! */
up(sem); /* uvolníme místo */
```

Implementace v UNIXových sys. není -> používá se monitor. Obecně implementován pomocí spinlocku a testování hodnoty čítače pozastavené procesy se přidávají na konec fronty.

Simulace
binární sem. lze simulovat dvěma číselnými se sdílenými proměnnými
obecný sem. lze simulovat třemi binárními a sdílenou hodnotou

18. Inverze priority, rekurzivní zámky

Inverze priority - zvýšení priority procesu v kritické sekci. Inverze priority je treba kdyz proces s nižší prioritou blokuje provádění procesu s vyšší prioritou. Proces s vyšší prioritou (h) nemůže vstoupit do kritické sekce, protože je v kritické sekci proces s nižší prioritou (l) a neběží, protože jsou v systému procesy s prioritou p > l (pokud p < h, jedná se o inverzi priority těchto procesů proti h).

Řešení

Dědění priority (*priority inheritance)*

- při vstupu procesu do KS je zvýšena priorita tohoto procesu na úroveň max. priority čekajících procesů na KS
- + priorita se nemění, pokud je proces osamocen
- - potřeba přepočítat max. prioritu při každém blokujícím čekání

Horní mez priority (*priority ceiling)*

- po dobu provádění KS je nastavena statická pevná priorita
- + jednoduchá implementace – pevná priorita
- - priorita procesu se zvyšuje vždy (i když to není nutné)

Použito pro…

- **Ano:** bin. semafor, mutex, monitor
- **Ne:** obecný semafor, condition (není jasné, kdo blokuje)

Rekurzivní zámek - složité knihovny, např. standardní V/V pro C:
printf() - musí zamknout stdout. printf() volá putchar() - musí zamknout stdout, nastává uváznutí
→ pokud je zámek zamčen stejným procesem, pouze se inkrementuje čítač úrovně rekurze, při odemykání se zmenšuje, zámek se odemkne až při 0

19. Klasické synchronizační úlohy, řešení různými nástroji

1. Vzájemné vyloučení (Mutual Exclusion)
2. Producent/konzument (Producer/Consumer, Bounded Buffer) producenti produkují data do sdílené paměti, konzumenti je z ní odebírají
• konzumenti musí čekat, pokud nic není vyprodukováno
• producenti musí čekat, pokud je paměť plná
• operace s pamětí musí být synchronizovány
3. Čtenáři/písaři (Readers/Writers)
• přístup ke sd.leným datům
• čtenář pouze čte data
• písař čte a zapisuje
• vzájemné vyloučení všech je příliš omezující:
• více čtenářů současně
• pouze jeden písař
4. Pět filozofů (Dining philosophers)
5 filozofů, 5 vidliček, 5 talířů
Problémy:
• vyhladovění – je třeba zajisti, že když chce jíst, dostane v konečném čase najíst a nebude systematicky předbíhán jinými filozofy
• uváznutí – všichni přijdou ke stolu a uchopí levou vidličku, nikdo nemůže jíst

20. Monitory, čekání, řešení uvolnění čekajících, srovnání se semafory

Monitor je abstraktní datový typ, který zajistí vzájemné vyloučení operací nad monitorem = všechny operace monitoru jsou atomické. Skládá se z dat, ke kterým je potřeba řídit přístup, a množiny funkcí, které nad těmito daty operují. Odděluje čekání a operace se sdílenými proměnnými.

Podmíněné proměnné
slouží k čekání uvnitř monitoru (proces v monitoru čeká na splnění nějaké podmínky)
Když funkce monitoru potřebuje počkat na splnění podmínky, vyvolá operaci ***wait*** na podmíněné proměnné, která je s touto podmínkou svázána. Tato operace proces zablokuje, zámek držený tímto procesem je uvolněn a proces je odstraněn ze seznamu běžících procesů a čeká, dokud není podmínka splněna. Jiné procesy zatím mohou vstoupit do monitoru (zámek byl uvolněn). Pokud je jiným procesem podmínka splněna, může funkce monitoru „signalizovat“,

tj. probudit čekající proces pomocí operace ***notify***. Operace notify budí jen ty procesy, které provedly wait na stejné proměnné.

Srovnání

Používá se namísto obecných semaforů v UNIXu (vyloučení i signály). Snazší používání (vyšší úroveň abstrakce), z pohledu programátora transparentní (jen volá funkce, zamykání řeší monitor).

21. Synchronizační nástroje POSIX 1003.1c, příklady řešení klasických synchronizačních úloh

22. Podmíněné kritické sekce (await) a bariéry

Podmíněné kritické sekce - vymezení kritické sekce a sdilených proměnných

```
resource sem::value: integer;
procedure down;
        begin
           region sem when value > 0 do value:=value-1;
        end;
procedure up;
        begin
           region sem do value:=value+1;
        end;
```

Implementace pomocí semaforů

```
lock(mutex); /* binární/obecný */
++waiting;
while (!C) {
   unlock(mutex);
   down(block); /* obecný */
   lock(mutex);
}
--waiting;
KS;
for (i = 0; i < waiting; i++) up(block);
unlock(mutex);
```

Bariéra - čekání na n procesů

Implementace pomocí semaforů:

• čítač s počátečním stavem N
• proces v bariéře dekrementuje čítač a pokud není nula, čeká na obecný blokující semafor
• až přijde poslední, čítač je nula, musí odblokovat všechny čekající (N-1 krát operace up())

Problém:
• blokovaný proces nemusí stihnout provést down(b), další odblokuje bariéru cyklem up() a bude tam o jednu signalizaci více, takže vstup do následující bariéry nebude blokující!
Řešení - dva čítače a dva blokující semafory, jednou použít jeden pár, pak druhý pár (nebo jeden pro vstup, druhý pro výstup)

23. Signály, problémy klasických signálů, v čem spočívá problém čekání na příchod signálu.

Typy signálů:
• chybové (dle PDP) = SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS a SIGSEGV, implicitně ukončení procesu
• uživatelské = SIGHUP, SIGINT, SIGQUIT, SIGKILL, SIGTERM, SIGUSR1 a SIGUSR2, ukončení procesu (SIGKILL nelze ignorovat).
• systémové = SIGCHLD, SIGSYS, SIGPIPE a SIGALRM, ukončení procesu s výjimkou SIGCHLD.

24. Signály POSIX, použití, řešení synchronizace zasíláním signálů.

25. Zasílání zpráv, adresace, použití pro synchronizaci

synchronní - odesílatel čeká na přijetí zprávy příjemcem (CSP)
asynchronní - bez čekání, proces může pokračovat
Adresace:

explicitní (přímá):
send(p, msg), receive(q, msg) - p, q jsou procesy
implicitní:
send(msg), receive(msg) - komukoli, od kohokoli
nepřímá:
send(m, msg), receive(m, msg) - m je schránka, port

Problémy implementace:

• priorita zpráv a výběr podle priority
• velikost zpráv a efektivní kopie mezi adresovými prostory
• vyrovnávací paměť (0 = rendezvous)

```
/* init: zaslání zprávy do schránky lock */
send(lock, msg);
while (1) {
receive(lock, msg);
KS;
```

send(lock, msg);
...
}

26. Zápis paralelních systémů a paralelismu, příklad zápisu paralelního systému.

1. P/S

P(P1, P2) paralelní provádění P1 a P2
S(P1, P2) sériové provádění P1, P2

2. cobegin/coend

Dijkstra: parbegin/cobegin
cobegin co
P1|P2|P3; P1 // P2 // P3;
coend oc
Lamport: < A; B; C; > atomická operace

3. fork/join

fork label spuštění procesu od návěští
quit ukončení procesu
join m,label m=m-1, když m==0 skok na návěští

Priklad zapisu : S(P(S(P(t1=a*b, t2=c*d), t4=t1+t2), t3=a/e), r=t4+t3)

27. Metody verifikace paralelních sysémů, stavový prostor paralelního systému, konstrukce stavového prostoru a ověřování živosti ve stavovém prostoru

Stavový prostor

Kripkeho model (S, R, L) - stavový prostor paralelního systému
S - konečná množina stavů
R - množina přechodů mezi stavy S
L - značení, definuje hodnotu atomických tvrzení pro každý stav
cesta - posloupnost $\sigma = s1, s2, s3, ... (si, si+1) \in R$
Problém: počet stavů je exponenciální - redukce

Značení - (příkaz, příkaz, stav flag1, stav flag2)
Konstrukce - Začneme od počátečního stavu a přidáváme všechny stavy, do kterých se může systém dostat, přičemž stavy se stejným značením tvoří jeden uzel. Z každého uzlu vede cesta dvěma směry (máme 2 procesy, pro více vícerozměrný prostor),

směr vpravo = postup Q, dolů = postup P.
Bezpečnost = neexistuje cesta do stavu označeného EE**
Živost = neexistuje nekonečný cyklus obsahující alespoň jeden vertikální a současně alespoň jeden horizontální přechod a neprocházející kritickou sekcí v původním grafu, ani v grafu vzniklém po zablokování jednoho procesu v sekci výpočet (reprezentujeme změnou přechodů C->D*** na C->C***).
Uváznutí a stárnutí = nekonečný cyklus neprocházející kritickou sekci při postupu v obou procesech (obou směrech)
Blokování = nekonečný cyklus, kdy zůstává jeden proces v synchronizaci a druhý stojí v sekci výpočet (před příkazem C).

28. Přidělování procesoru, plánování, stavový diagram procesů

• **plánování** (scheduler) - volba strategie a řazení procesů
    • sdilený plánovač
    • samostatný plánovač
• **přidělování** (dispatcher) - přepínaní kontextu na základě naplánování

Cíle plánování:
• Minimalizace doby odezvy
• Efektivní využití prostředků
• Spravedlivé dělení času procesoru
• Doba zpracování
• Průchodnost (počet úloh/čas)

Přidělování procesoru = přepínání kontextu podle plánování

29. Univerzální plánovač

Založen na prioritě procesorů, tedy procesor je přidělen procesu s nejlepší prioritou. Definován třemi charakteristikami:

1. **Interval rozhodování**
    a. Nepreemptivní - proces běží do ukončení nebo čekání
    b. Preemptivní - procesu může být odebrán procesor v časových kvantech, odblokování procesu s vyšší prioritou nebo příchod nového procesu
    c. Selektivní preempce - procesy mohou přerušovat nebo být přerušeny
2. **Prioritní funkce**, priorita určena na základě několika parametrů:
    a. Paměťové požadavky
    b. Spotřebovaný čas procesoru
    c. Doba čekání na procesor
    d. Doba strávená v systému
    e. Externí priorita
    f. ...

3. **Výběrové pravidlo**
   Výběr z více procesů se stejnou prioritou pomocí výběrového algoritmu

30. Plánovací algoritmy pro dávkové systémy

prioritní funkce P(a,r,t,d)

vysvětlivky:

- a - spotřevovaný čas cpu
- r - čas strávený v systému
- t - celkový čas procesoru
- d - perioda opakování procesu

algoritmy:

1. FIFO: P(r) = r
2. LIFO: p(r) = -r
3. SJF (Shortest Job First): P(t) = -t
4. SRT (Shortest Remaining time): p(a,t) = a - t
5. Statická priorita: P(i) = konstanta i dle procesu
6. RR (Round Robin): P() = konstanta (vždycky následuje ten následující, např. 1->2, 2->3, 3->1)
7. MLF (MultiLevel Feedback): prasárna
8. RM (Rate Monotonic): P(d) = -d
9. EDF (Earliest Deadline First): P(r,d) = -(d - r%d)

31. Plánovací algoritmy pro víceuživatelské systémy

32. Plánovací algoritmy pro systémy reálného času

33. Uváznutí při přidělování prostředku, podmínky uváznutí, možnosti řešení

Uváznutí - čekání na událost, která nemůže nastat díky čekání

Máme dva druhy prostředků:

- SR (Serially Reusable): opakovaně použitelné
- CR (Consumable Resources): jednorázově použitelné

Uváznutí je stav, kdy dochází k čekání na událost/prostředek, která nikdy nenastane/se nepřidělí.

Nastává při:

- Zamykání semaforů, zámků, souborů, …
    - P1: lock(S1); lock(S2) … P2: lock(S2); lock(S1)
- Přidělování paměti
    - pokud P1 a P2 chtějí dvě jednotky z M, kdy M má právě dvě jednotky
- Zasílání zpráv
    - Všechny procesy čekají na zprávu, kterou nikdo nepošle kvůli čekání

Příčiny:

- Pouze jeden proces může využívat prostředek
- Proces se nevzdá přidělených prostředků při neúspěšné alokaci dalších prostředků
- Proces získává prostředky sekvenčně
- Prostředek nemůže být preemptivně odebrán, odevzdává jej proces

34. Model systému přidělování prostředků, graf systému, def. blokování a uváznutí

35. SR prostředky, graf alokace SR prostředků, redukce grafu, nutná a postačující podmínka uváznutí

36. Detekce stavu uváznutí z grafu alokace SR prosředků - příklad

37. Zotaveni a prevence uváznutí, metody

38. Graf SR prostředků s deklarovanými maximy, Bankéřův algoritmus

39. Uváznutí u CR prostředků - přehled

40. Správa paměti, vztah k multiprogramování

41. Organizace paměti, společný/oddělené adresové prostory

A. organizace LAP

1. monoprogramové systémy
2. multiprogramové systémy

a. společný LAP
b. oddělený LAP

B. mapování LAP na FAP

1. úseky
2. segmenty
3. stránky

A1) Jeden souvislý úsek paměti

LAP(Pi) = FAP, jeden program v paměti

Monoprogramování bez ochrany paměti - jeden adresový prostor, program je zaváděn na pevnou nebo proměnnou adresu.

Jádro systému - pevně přidělena část paměti, zbytek dostupný pro procesy (MS-DOS), ochrana mezním registrem (HP RTE-II)

Segmentování (překryvné segmenty, overlay) - zavádění programu po částech, podle potřeby

A2a) Společný adresový prostor procesů

LAP(Pi) = LAP(Pj) = FAP pro všechny programy v paměti

Jeden adresový prostor, více programů v paměti na různých adresách, lze kombinovat s následujícími technikami zobrazování LAP do FAP

- vyžaduje dynamickou relokaci programů na danou adresu
- omezený sdílený LAP (ale pro 64bitové procesory nevadí)
+ jednoduchá správa
+ jediná tabulka stránek při kombinaci se stránkováním
+ přidělování paměti viz přidělování úseků, stránkování

Multiprogramování - nutná ochrana paměti:

• ochrana uživatelských programů navzájem
• ochrana jádra systému

Implementace ochrany:

• mezní registry
• chráněný režim činnosti procesoru

A2b) Oddělené adresové prostory procesů

LAP(Pi) != LAP(Pj) != FAP - zobrazení, mapování

Každý proces má k dispozici celý logický adresový prostor. Logický adresový prostor je tranformován (mapován) na fyzický některou z následujících transformací. Nutná podpora hardware (převod logické adresy na fyzickou je nutný pro každou instrukci).

Adresový prostor jádra:

a) oddělený - nutný přepočet adres parametrů volání jádra, přístup do jiného adresového prostoru pro čtení/zápis parametrů a dat (bufferů) každého volání jádra

b) sdílený s procesem, který volá jádro - horní část LAP rezervovaná pro jádro, v uživatelském režimu chráněná (i proti

čtení), v systémovém režimu zde běží jádro (Unix, Windows/NT)
- problém 32bitového adresového prostoru, rezerva pro jádro omezuje maximální velikost uživatelských dat, jádro nemá moc prostoru pro buffery (obvykle 512-1024 MB)
- není problém u 64bitových architektur

42. Správa úseků proměnné velikosti

Zdroj má omezenou kapacitu, je přidělován po úsecích proměnné velikosti. Při uvolňování vznikají volné úseky.

Cíle:

Efektivní využití zdroje o omezené kapacitě – pokud není zdroj zcela využit, musí být volné úseky použitelné k uspokojení požadavků.

Rychlá alokace zdroje – minimální čas nutný pro nalezení dostatečně velkého volného úseku.

Použití – přidělování systémové paměti, swap, LAP, apod.

Problém: Který volný úsek použít pro uspokojení požadavku o velikosti N? Pokud je větší, přidělit jej celý nebo rozdělit?

First Fit - přiděluje první postačující úsek

Next Fit - přiděluje první postačující úsek, příště pokračuje od místa posledního přidělení

Best Fit - přiděluje minimální postačující volný úsek, minimální interní fragmentace

Worst Fit - přiděluje vždy největší volný úsek, menší externí fragmentace malých úseků, ale nepříjemná fragmentace velkých úseků

Při uvolňování vznikají prázdné úseky o různých velikostech - problém evidence volných úseků:

- seznam řazený podle pořadí uvolnění – FIFO, LIFO (nejjednodušší)
- společný seznam organizovaný podle adres (vhodné pro spojování)
- samostatný seznam volných bloků
- seznam organizovaný podle velikosti (náročné zařazování)

Spojování volných úseků:

- nikdy
- při alokaci
- při uvolnění
- při neúspěšném přidělení (garbage collection)

43. Stránkování - princip, funkce, organizace tabulky stránek, TLB

Organizace tabulky stránek

Problém realizace tabulky stránek v HW
1. Transformace LAP na FAP se musí uskutečnit při každé instrukci a adresaci operandu (načtení instrukce, operandy)
− rychlost provádění instrukcí < 1 ns
− rychlost přístupu do paměti = 50 ns
− adresu nelze transformovat indexováním a čtením tabulky stránek z operační paměti!
2. Zobrazení plného 32bitového adresového prostoru - 1M položek (4 MB pro stránku 4 KB)
− malý program by vyžadoval plnou tabulku stránek (kód na začátku adresového prostoru, zásobník na konci)
− takto velkou tabulku stránek nelze umístit do rychlé interní paměti na procesoru (nehledě na problém jejího nastavování po přepnutí kontextu)
− pro 64bitový adresový prostor nereálné i uložení v paměti

44. Volba optimální velikosti stránky

cílem je minimalizovat režii interní fragmentace

s - priemerna velkost procesu
e - velkost polozky tabulky
p = sqrt(2se)

45. Virtuální paměť, obsah tabulky stránek, výpadek stránky a zpracování

46. Stránkovací algoritmus, definice, typy stránkovacích algoritmů

47. Nahrazovací algoritmy s pevným počtem rámců

48. Hodnocení stránkovacích algoritmů, Beladyho anomálie

49. Zásobníkové algoritmy, jejich vlastnosti

50. Thrashing, volba stupně multiprogramování a odkládání

51. Nahrazovací algoritmy s proměnným počtem rámců

52. Stránkování a paměťové nároky jádra, řešení u systémů BSD 4.4

53. Zamykání stránek, spouštění procesů, implementace fork()

54. Sdílení adresového prostoru, implicitní a explicitní, sdílené knihovny

55. Virtualizace souborů

56. Obsazení a využití logického adresového prostoru procesu, struktury jádra pro správu paměti

57. V/V na úrovni rozhraní jádra, adresace, funkce

58. Synchronní a asynchronní V/V, select, vyrovnávání diskových operací

59. V/V na úrovni jádra, adresace, rozhraní ovladačů, zpracování

60. V/V na úrovni komunikace s řadiči, optimalizace diskových operací

61. Systém souborů - logická struktura souboru, interní struktura souboru

62. Systém souborů - alokace diskového prostoru, optimalizace diskových operací

63. Systém souborů - metadata, adresářové struktury, odolnost proti výpadku

64. Systém souborů - přístup k souborům, čtení a zápis, ostatní operace

65. Příklad implementace systému souborů - UFS, FFS, Ext2

66. Autentizace a autorizace v klasickém Unixu

67. Ochrana systému a souborů v Unixu

https://www.youtube.com/watch?v=jpVF_YlwIVY :D