pamäť a dvojková sústava (to by mal vedieť už každý)
1 bajt = 8 bitov = 2^0
1024 bajtov = 1 kilobajt = 2^{10}
1024 kilobajtov = 1 megabajt = 2^{20}
... (kilo, mega, giga, tera... = 2^{10}, 2^{20}, 2^{30}, 2^{40}...)

dvojková sústava ⇔ hexa
hodí sa vedieť to hlavne pri počítaní fyzických adries, ktoré sú hexa čísla...
veľkosť 2 kilo = 2^{11} = 1000 0000 0000 = po štvoriciach od konca z dvojkovej do hexa: 0 0 8 => teda 2 kilo je 0x800
-> pri čísle 1000 0000 000, pripomínam, že tá jednotka je MSB (most significant bit), podobne 0 na konci LSB (least)
-> často sa to dá predstaviť a vypočítať rýchlo z hlavy, napr. 1 giga do hexa:
giga = 2^{30} -> od čísla 30 je najbližší násobok 4ky 32 (8 * 4 = 32) teda hexa číslo bude končiť siedmimi nulami a prvá štvorica má na začiatku jednu nulu (v dvojkovej sú z ľava 31. bit a 30. bit), už si len stačí predstaviť prvú štvoricu, čo je 0100, teda 4, takže 2^{30} = 0x40000000 sa dá dať z hlavy v pohode -> btw kalkulačka je na skúške povolená...

prednáška 7 príklad na slajde 38/44
16 bitová logická adresa => je 2^{16} logických adries dokopy
veľkosť stránky 1k = 2^{10} = 0x400 => počet „framov“ je $2^{16-10} = 2^6$
logická adresa 0xE5C -> v dvojkovej 0000 1110 0101 1100 (prvých 6 bitov je číslo stránky, posledných 10 bitov je offset)
číslo stránky = 3, offset = 0x25C -> je dobré si uvedomiť, že číslo stránky nie je číslo framu !
pozrieme sa do „Process B page table“ kde pod číslom stránky 3 nájdeme číslo framu 10
frame 10 začína na fyzickej adrese 10*veľkosť stránky -> 10*0x400 = 8*0x400 + 2*0x400 = 0x2000 + 0x800 = 0x2800
fyzická adresa našej logickej adresy je teda adresa začiatku framu + offset = 0x2800 + 0x25C = 0x2A5C
poznámka: 8*0x400 = 0x ‘32’00, ale 32 v hexa je 0x20 -> 0x2000 ☺

S Y N C H R O N I Z Á C I A

1) Unisex toaleta (alebo sprcha, sauna, …)

Navrhnite synchronizačné riešenie pre unisex toaletu (určenú pre mužov aj ženy) s nasledovnými obmedzeniami:
- na toalete sa nesmú nachádzať muži a ženy zároveň
- na toalete sa nesmú nachádzať viac ako traja ľudia súčasne

Riešenie by sa malo vyhnúť zablokovaniu. Starvácia – vyhladovanie procesov sa neberie do úvahy.
Môžeme predpokladať, že toaleta je vybavená všetkými potrebnými semaformi.

riešenie:
-> máme semafor, ktorý dovoľuje ľuďom v rade nakuknúť do záchoda, že čo sa tam deje.. neuvažujeme starváciu, to znamená, že ak tam dôjde žena, ktorej treba srať a dlho nebude môcť vojsť, tak sa neposerie na chodbe (vo vnútri sú dvaja chlapi, príde tam žena, ktorá sa začne točiť v tom while... lenže keď tam príde ďalší muž, ktorý sa postaví za ženu, tak ho to uspí, žena ho na konci whilu zobudí a uspí samú seba, žena následne spí až kým ju nezobudí posledný chlap, ktorý vyjde zo záchoda...)

```
semafor sem;
sem.value = 1;
int pohlavie = -1;  // 0 = muz; 1 = zena; -1 = nezalezi
int num = 0; // pocet ludi na toalete
#define MAX 3 // max ludi

void vstup( int moje_pohlavie ){      // vstupný protokol človeka, ktorý ide srať
  while(1){
    sem.wait();                       // ak môže vstúpiť, tak ho pustí a dek. value, inak ho postaví do radu
    if( pohlavie == moje_pohlavie && num < MAX )
    {
      num++;                 // num je premenná s ktorou semafor nerobí
      sem.signal();          // vojde dnu ( sú tam teraz dvaja alebo traja ) -> dovolí ďalšiemu nakuknúť
      break;
    }
    if( pohlavie == -1 && num == 0 )  // ak tam vchádza prvý
    {
      pohlavie = moje_pohlavie;       // tak označí pohlavie záchodu za svoje
      num = 1;
      sem.signal();                   // a dá signál ďalšiemu, že môže nakuknúť
      break;
    }
    sem.signal();                     // tu len inkrementuje value a točí sa v cykle (obsadz. čakanie)
  }
}
void vystup( int moje_pohlavie ){
  sem.wait();                // dekrementuje si value, aby medzitým nikto nezmenil num ( seba neuspí ! )
  num--;
  if( num == 0 )
    pohlavie = -1;           // tu nám je jedno ako pomalé je to priradenie
  sem.signal();              // pri odchode zo záchoda „dá signál“ ďalšiemu, že môže nakuknúť
}
```

3) Autobusová zastávka

Napíšte synchronizačný kód, ktorý simuluje nasledovnú situáciu so všetkými uvedenými obmedzeniami:
Na autobusovú zastávku postupne prichádzajú ľudia, ktorí čakajú na autobus. Keď autobus dorazí, všetci čakajúci vyvolajú procedúru BoardVehicle(), pričom pasažier nemôže nastúpiť do autobusu na zastávke (musí počkať na ďalší autobus), ak dorazil na zastávku až po príchode autobusu. Kapacita autobusu je 50 ľudí, ak sa na zastávke v čase príchodu autobusu nachádza viac ľudí ako je tento počet, niektorí z nich musia počkať na ďalší autobus. Po nastúpení všetkých pasažierov autobus vyvolá procedúru Departure(). Pokiaľ na zastávke nikto nečaká, vyvolá túto procedúru okamžite.

riešnie:

-> dva semafory -> jeden kontroluje čakanie na zastávke (mutex), druhý kontroluje nastupovanie do autobusu (bariera)

```
semafor bariera;
bariera.value = 0;
semafor mutex;
mutex.value = 1:
#define CAPACITY 50
int waiting = 0;

void process_human(){
  mutex.wait();               // prišiel na zastávku a čaká na signál, ktorý mu dovolí postaviť sa do radu
  waiting++;                  // počet čakajúcich
  mutex.signal();             // dá signál ďalšiemu, že sa môžu postaviť do radu ^^
  bariera.wait();             // čaká na signál, ktorý mu dovolí postaviť sa do radu pred dvere busu
  BoardVehicle();             // toto sa vykoná až keď mu autobus dá signál
  // tu je už človek v buse
}

void process_bus(){
    mutex.wait();           // sám sa akoby postavil do radu, aby zablokoval ľudí, ktorí prídu na zástavku
    int temp = waiting;     // počet čakajúcich si dáme do pomocnej premennej
    for ( int X = 0 ; X < temp && X < CAPACITY ; X++ )
    {
      waiting--;
      bariera.signal();     // dá signál čakajúcemu, aby nastúpil
    }
    mutex.signal();         // dá signál ostatným, aby sa mohli postaviť do radu na zastávke
    Departure();            // odíde
}
```

-> toto riešenie je trochu fail, pretože nie je splenené „ keď autobus dorazí, všetci čakajúci vyvolajú procedúru BoardVehicle() “

4) Jedáleň

Napíšte synchronizačný kód, ktorý simuluje nasledovnú situáciu so všetkými uvedenými obmedzeniami:
Každý študent po príchode do jedálne vyvolá procedúru Jedlo() a následne Odchod().
Medzi týmito dvomi procedúrami sa nachádza v stave “pripravený na odchod”. Synchronizačné obmedzenie pre túto situáciu je, že študent nikdy nesmie sedieť pri stole sám, pričom študent sedí pri stole sám vtedy, keď všetci ostatní, ktorí vyvolali Jedlo(), vyvolajú Odchod() pred tým, ako bola ukončená procedúra Jedlo() pre tohto študenta.

riešnie:

-> predstavíme si najjednoduchšiu situáciu: v jedálni je iba jeden stôl so 4mi stoličkami

```
semafor mutex;
mutex.value = 1;
semafor table;                      // analógia so semaforom na unisex hajzli
table.value = 1;                    // prvé volanie wait ho pustí do KO, nepostaví do radu
int eating = 0;

Student(){
  // vstúpi do jedálne
  mutex.wait();                     // ide si vziať jedlo, alebo čaká kým sa uvoľní miesto pri stole
  eating++;
  if( eating < 4 )
    mutex.signal();                 // keď si tam prvý sadne a zistí, že je miesto, tak pustí ďalšieho

  Jedlo();
  // tu dojedol a je už pripravený na odchod
  while(1){
    table.wait();                   // pustí ho to aby sa točil v cykle ( obsadzujúce čakanie ) alebo uspí
    if ((eating - 1) % 4 != 1) {    // ak by po jeho odchode neostal pri stole 1 študent
      eating--;
      table.signal();               // pustí ďalšieho, ktorý dojedol aby sa chlapec začal točiť v cykle
      break;                        // jemu je to už ale jedno, on má už v pi**
    }
    table.signal();                 // tu si len inkrementuje value aby sa sám mohol točiť v cykle
  }
  Odchod();                         // tu už odchádza z jedálne
  mutex.signal();                   // dáva signál ďalšiemu, ktorý si vezme jedlo a pôjde ku stolu
}
```

SPRÁVA PAMÄTI

1) Fragment programu I.

Fragment programu:
for(i=0; i < n; i++) A[i] = B[i] + C[i]; // nech n = 1024
je po skompilovaní na počítači s registrami procesora R1, ..., R8 umiestnený vo virtuálnom adresovom priestore nasledovne:

Adresa	Inštrukcia	Komentár
0x0040	(R1) <- ZERO	R1 bude register pre i
0x0041	(R2) <- n	R2 bude register pre n
0x0042	compare R1,R2	porovnanie hodnôt i a n
0x0043	branch if gr or eq 0x0049	ak i≥n choď na 0x0049
0x0044	(R3) <- B(R1)	do R3 i-ty prvok poľa B
0x0045	(R3) <- (R3) + C(R1)	pričítaj C[i]
0x0046	A(R1) <- (R3)	súčet daj do A[i]
0x0047	(R1) <- (R1) + ONE	inkrementuj i
0x0048	branch 0x0042	choď na 0x0042
...		
...		
0x1800 .. 0x1BFF	storage for A	
0x1C00 .. 0x1FFF	storage for B	
0x2000 .. 0x23FF	storage for C	
0x2400	storage for ONE	
0x2401	storage for ZERO	
0x2402	storage for n	

Veľkosť stránky pamäti je 1k. Proces má pridelené 4 stránkové rámy. Koľko výpadkov stánok nastane počas behu fragmentu s použitím algoritmu výberu obete
a) LRU b) FIFO c) Optimálny
Pred začatím vykonávania uvedeného fragmentu sa nepracuje so stránkami, ktoré tento fragment používa.

riešenie:
-> adresový priestor 0x0040 až 0x2402 je potrebné rozdeliť na intervaly po jedno kilo: 1k = 2^{10} = 0x400 v hexa, čím získame postupnosť stránok v pamäti

0x0000 -> 0x03FF 0 ← tu máme program
0x0400 -> 0x07FF 1
0x0800 -> 0x0BFF 2
0x0C00 -> 0x0FFF 3
0x1000 -> 0x13FF 4
0x1400 -> 0x17FF 5
0x1800 -> 0x1BFF 6 ← storage for A
0x1C00 -> 0x1FFF 7 ← storage for B
0x2000 -> 0x23FF 8 ← storage for C
0x2400 -> 0x27FF 9 ← storage for ONE, ZERO, n

-> pozrieme sa ako beží program (for(i=0; i < n; i++) A[i] = B[i] + C[i]) -> prečítanie inštrukcie je zrejme jasne idúce z prvej stránky (**0**), prečítaná inštrukcia vykoná čítanie ZERO ktoré leží zrejme na poslednej stránke (**9**), potom sa zase číta inštrukcia z prvej stránky (**0**) a potom z poslednej stránky premenná n (**9**). Porovnávame i a n (**0**), prejde aj tá branch inštrukcia na 0x43 (**0**) if ale nezbehne, 0x44 (**0**), čítame z miesta, kde je B (**7**), 0x45 (**0**), čítanie C (**8**), **0**, **6**, **0**, **9**, choď na 0x42 (**0**), keď pôjde cyklus posledný krát, tak porovná R1 s R2 (**0**), branch(**0**) podmienka zbehne takže to skočí na 0x49, kde sa vykoná ďalšia inštrukcia (**0**). Výsledkom je postupnosť čítaní stránok pri činnnosti programu (postupnosť čísiel stránok)

```
0 9 0 9 0 0 0 7 0 8 0 6 0 9 0 | 0 0 0 7 0 8 0 6 0 9 0 | 0 0 0
                              |<---------1023x------->|
```

-> už len treba nakresliť tabuľku podľa postupnosti čísel a spočítať výpadky.

a) LRU

```
      0 9 0 9 0 0 0 7 0 8 0 6 0 9 0 | 0 0 0 7 0 8 0 6 0 9 0 | 0 0 0
-------------------------------------------------------------------------
1     0 9 0 9 0 0 0 7 0 8 0 6 0 9 0   0 0 0 7 0 8 0 6 0 9 0   0 0 0
2       0 9 0 9 9 9 0 7 0 8 0 6 0 9   9 9 9 0 7 0 8 0 6 0 9   9 9 9
3               9 9 7 7 8 8 6 6       6 6 6 9 9 7 7 8 8 6 6   6 6 6
4                   9 9 7 7 8 8       8 8 8 6 6 9 9 7 7 8 8   8 8 8
-------------------------------------------------------------------------
      * *         *   *   *   *         *   *   *   *
```

teda 6 + 1023x4 = 2 + 1024*4 = 2 + 4096 = 4098 výpadkov (budeme bez kalkulačky, takže som tam tie 4 najebal k 1023ke naschvál, lepšie sa počíta)

b) FIFO

```
      0 9 0 9 0 0 0 7 0 8 0 6 0 9 0 | 0 0 0 7 0 8 0 6 0 9 0 | 0 0 0
-------------------------------------------------------------------
1     0 9 9 9 9 9 9 7 7 8 8 6 0 9 9   9 9 9 7 7 8 8 6 0 9 9   9 9 9
2       0 0 0 0 0 0 9 9 7 7 8 6 0 0   0 0 0 9 9 7 7 8 6 0 0   0 0 0
3                   0 0 9 9 7 8 6 6   6 6 6 0 0 9 9 7 8 6 6   6 6 6
4                       0 0 9 7 8 8   8 8 8 6 6 0 0 9 7 8 8   8 8 8
-------------------------------------------------------------------
      * *           *   *   * * *         *   *   * * *
```

teda 7 + 1023*5 = 2 + 1024*5 = 2 + 4096 + 1024 = 5096 + 26 = 5122 výpadkov

c) optimálny -> obeť je stránka, ktorá bude v budúcnosti najneskôr použitá
-> začiatok bude rovnaký, načítať 0,9,7,8 musíme tak či tak, takže potiaľ máme 4 výpadky (0978), // 4
pri 6tke sa pozrieme, ktoré číslo budeme najneskôr potrebovať (8čku), ďalší výpadok * (0976) // +1 -> obsah rámov 0976 pred cyklom
v cykle pri 8čke sa pozrieme ďalej (najďalej je 7čka pri ďalšom zbehnutí), takže výpadok * (0986) // +1 – cyklus // i = 1
pri 7čke obetujeme 9tku *(0786) a pri 9tke obetujeme 6tku *(0978) // +2 – cyklus // i = 2
pri 6tke obetujem 8čku *(0976) -> čo je obsah rámov s ktorým som vstupoval na začiatku // + 1 – cyklus // i = 3

```
             i | 1  2  3  4  5  6  7  8  ...1023
počet výpadkov | 1  2  1  1  2  1  1  2  ... -> každé 3 zbehnutia cyklu sú 4 výpadky
```

-> celkový počet výpadkov je teda 4 + 1 + 1023*(4 / 3) = 5 + 990*(4/3) + 33*(4/3) = 5 + 330*4 + 11*4 = 49 + 120 + 1200 = 1369 výpadkov

2) Multiprogramovanie s oblasťami pevnej dĺžky, bez výmen - swapovania

Počítačový systém má v hlavnej pamäti priestor pre štyri programy. Tieto programy v priemere 50% času čakajú na V/V operácie. Aká časť času procesora (CPU) je v priemere nevyužitá?

riešenie:

-> Počas vykonávania každého programu sa strieda samotný výpočet (procesor vykonáva inštrukcie programu) s čakaním na V/V. Keby bol v pamäti len jeden program, bolo by nevyužité v priemere 50% jeho času (pravdepodobnosť, že tento jeden program čaká na V/V je p = 0.5 -> tu je dobré si uvedomiť, že toto je už priemerný prípad, pretože v zadaní je „v priemere 50%“) V prípade štyroch programov je to pravdepodobnosť, že všetky štyri programy čakajú na V/V operácie = $p^n = 0.5^4 = 0{,}0625$, teda 6,25%

3) Multiprogramovanie s oblasťami pevnej dĺžky, bez výmen - swapovania

Nech počítač má 2MB pamäte, operačný systém zaberá 512kB a aj každý bežiaci program zaberá 512 kB. Ak všetky programy trávia 60% času čakaním na V/V, o koľko percent by sme zvýšili priepustnosť (využitie CPU) pridaním 1MB pamäte?
Koľko musíme pridať pamäte, aby sme zvýšili priepustnosť aspoň na 99% ?

riešenie:

-> v prípade 2MB môžu byť v pamäti 3 programy (3*512 + 512 OS), nevyužitý čas bude $(0.6)^3 = 21.6\%$. Ak pridáme 1 MB, môžeme do pamäti umiestniť 5 programov a nevyužitý čas bude $(0.6)^5 = 7.776\%$, t.j. dosiahli sme zvýšenie o 13.824%.
-> chcem zvýšiť priepustnosť nad 99%, teda chceme aby bolo nevyužité najviac 1% času CPU, teda riešime otázku koľko programov by muselo byť v pamäti, aby pravdepodobnosť toho, že všetky čakajú na V/V operácie bola menšia alebo rovná ako 0.01 → $(0.6)^n \leq 0.01$ → $\log((0.6)^n) \leq \log(0.01)$ → $n.\log(0.6) \leq \log(0.01)$ → $n \leq \log(0.01)/\log(0.6) = 9{,}015$ → n = 10, teda by sme potrebovali pridat pamäť pre ďalších 7 programov, čo je 7*512 = 3,5MB

4) Multiprogramovanie so swapovaním

Niektoré systémy so swapovaním sa snažia eliminovať externú fragmentáciu pomocou kondenzácie (kompakcie) pamäti. Nech systém s 1MB používateľskej pamäti robí kondenzácie raz za sekundu. Nech prenos (copy) 1 bytu trvá 0.5 mikrosec. a priemerná dĺžka voľného úseku je 0.4 krát veľkosť priemerného segmentu. Aká časť celkového času CPU je použitá na kondenzácie?
Ako často treba robiť kondenzáciu, aby sa nespotrebotrebovalo viac ako 10% času CPU ?

riešenie:

-> v systéme je 1 MB pamäti, čo je zaplnená a voľná pamäť dohromady, teda zaplnená + 0.4 * zaplnená = 1 MB → 1.4 z = 1MB → z = 1/1.4 MB
-> prenos 1 bytu trvá 0.5 mikrosec, takže prenos $(1/1.4)*2^{20}$ B bude trvať $(1/1.4)*2^{20} * 0.5 = (1/1.4)*2^{19}$ mikrosec = 374491 mikrosec, čo je 0.3745 sec a teda 37.45 % z jednej sekundy, teda na kondenzácie je použitých 37.45% času CPU
-> ako často treba robiť kondezáciu, aby na ňu nebolo použitých viac ako 10% času CPU? z akého času bude 0.3745 sekundy maximálne 10 % ? kondenzáciu treba robiť najčastejšie raz za 3.745 sekundy

5) Virtuálna pamäť so stránkovaním I.

Nech vykonanie jednej inštrukcie trvá 1 mikrosec., ale v prípade odvolávky sa na neprítomnú stránku pamäti (page fault) ďalších n mikrosec.
Aký je efektívny (priemerný) čas vykonávania jednej inštrukcie, ak sa page fault vyskytuje priemerne každých k inštrukcií?

riešenie:

-> page fault sa vyskytuje priemerne každých k inštrukcií, teda čas jednej z k inštrukcií nebude 1 ale 1+n mikrosec. → k inštrukcií teda trvá k+n mikrosec
-> priemerný čas vykonania jednej inštrukcie je (k+n)/k = 1+n/k mikrosec.

6) Virtuálna pamäť so stránkovaním II.

Logický adresový priestor každého procesu má 8 stránok po 1024 slov a mapuje sa do fyzickej pamäti s 32 stránkovými rámami.
a) Koľko bitov má logická adresa?
b) Koľko bitov má fyzická adresa?

riešenie:

a) proces má 8 stránok, teda 2^3, aby sme ich vedeli adresovať potrebujeme 3 bity, stránka má 1024 slov, teda 2^{10}, takže 10 bitov nám treba na offset
-> logická adresa bude mať 3 bity číslo stránky + 10 bitov offset, teda 13 bitov
b) pri fyzickej adrese offset ostane nezmenený, ide len o to, že každá z tých 8 stránok bude mať v sebe adresu jedného z tých 32 stránkových rámov, na adresovanie ktorých nám treba 5 bitov ($32 = 2^5$), teda fyzická adresa bude mať 15 bitov

7) Oneskorenie spôsobené kopírovaním tabuľky stránok

Počítač má 32-bitový adresový priestor a 8kB stránky. Tabuľka stránok pre práve bežiaci proces je v hardvéri a každá jej položka má 32 bitov. Keď proces štartuje (alebo sa prepína), tabuľka stránok sa kopíruje z pamäti do hardvéru rýchlosťou 1 položka/100 nsec. Ak každý proces beží 100 msec (vrátane napĺňania tabuľky), akú časť času CPU bude zaberať napĺňanie?

riešenie:

-> 8kB je 2^{13} bajtov, teda nám treba 13 bitov na adresovanie 8kB veľkého priestoru (týchto 13 bitov bude predstavovať offset v logickej adrese)
-> adresa má 32 bitov, teda 19bitov na adresu stránky + 13 bitov offset -> počet všetkých stránok je 2^{19} (524288)
-> tabuľka stránok sa kopíruje rýchlosťou jedna za 100 nanosekúnd, čiže skopírovanie celej tabuľky zaberie 2^{19} * 100 nanosec = 52 428 800 nanosec, čo je 52,429 ms -> každý proces kopíruje tabuľku a každý proces beží 100ms, teda napĺňanie zaberie 52,429% času CPU

8) Dvojúrovňové tabuľky stránok

Počítač s 32-bitovými adresami používa dvojúrovňové tabuľky stránok (ako napr. 80386): 9 bitov 11 bitov offset
1. Aké veľké sú stránky a koľko ich je vo virtuálnom adresovom priestore jedného procesu?
2. Akú časť adresového priestoru je možné adresovať pomocou jednej položky stránkového adresára, aká je s tým spojená réžia?
3. Aká by bola réžia na celý adresný priestor a pri jednoúrovňovej tabuľke stránok?

riešenie:

-> 9 bitov + 11 bitov + 12 bitov offset
-> 12 bitový offset znamená, že na určenie posunu v stránkovom ráme na ktorý odkazuje položka tabuľky druhej úrovne nám treba 12 bitov, teda veľkosť jednej stránky je 2^{12} = 4kB
-> 9 bitov predstavuje číslo položky stránkového adresára, čo sú v podstate tabuľky stránok druhej úrovne... je to ako pri jednoúrovňovom stránkovaní, ale keď sa dostaneme k fyzickej adrese stránkového rámu, tak tam nás čakajú ďalšie tabuľky stránok (položka stránkového adresára), z ktorých dostaneme fyzickú adresu stránkového rámu, v ktorom už bude to, čo chceme... ak 9 bitov je na číslo položky stránkového adresára, tak ich je 2^9
-> 2^9 položiek stránkového adresára, pričom v každá položka predstavuje tabuľku stránok, ktorých čísla sú zapísané v 11tich bitoch, teda 2^{11} stránok na každú položku stránkového adresára → dokopy $2^9 * 2^{11} = 2^{20}$, teda 1 mega stránok
-> položká stránkového adresára je v podstate tabuľka stránok, ktorá je v stránkovom ráme na ktorý odkazuje tabuľka stránok prvej úrovne, teda jedna položka stránkového adresára môže adresovať 2^{11} stránok veľkých 4kB → teda 2kB * 4kB = 8MB adresového priestoru
-> aká je s tým spojená réžia = aká je réžia spojená s adresovaním pomocou jednej položky stránkového adresára? teda koľko adresovateľných jednotiek predstavuje jedna položka stránkového adresára? → položka stránkového adresára predstavuje 2^{11} položiek a s tým spojená réžia je 2 K
-> aká by bola réžia na celý adresný priestor? teda koľko adresovateľných jednotiek predstavuje tabuľka prvej úrovne a následne položka stránkového adresára? → $2^9 + 2^{11}$ položiek → réžia na celý adresný priestor je 2,5 K
-> aká by bola réžia pri jednoúrovňovej tabuľke stránok? -> vieme, že je 2^9 položiek stránkového adresára a každá jeho položka má 2^{11} položiek, čo je dokopy 2^{20} stránok, ktoré by predstavovali jednu jednoúrovňovú tabuľku stránok – réžia s tým spojená by bola 1 M

9) Page faults I.

Zistilo sa, že počet inštrukcií programu medzi dvoma výpadkami stránky (page faults) je priamo úmerný počtu pridelených stránkových rámov (t.j. väčšia časť vo fyzickej pamäti = väčšie intervaly medzi výpadkami). Nech inštrukcia trvá normálne 1 mikrosec. a s výpadkom stránky 2001 mikrosec. Program trval 60 sec. a mal 15000 výpadkov.
Ako dlho by trval výpočet s dvojnásobným počtom stránkových rámov?

riešenie:

-> 15000 výpadkov trvá 30,015 s (pozor v tom čase 30,015 s už zbehlo aj 15000 inštrukcií, ktoré výpadky vyvolali – viď. zadanie „ inštrukcia s výpadkom stránky 2001 mikrosec. „) -> zvyšný čas 29,985 sec bežali inštrukcie každú mikrosekundu jedna -> dokopy bol počet inštrukcií PRESNE 30 miliónov
-> dvojnásobný počet stránkových rámov znamená zdvojnásobenie počtu inštrukcií vykonaných medzi dvoma výpadkami a teda zníženie počtu výpadkov presne o polovicu -> t.j. 7500 výpadkov -> zdržanie pri inštrukcii, ktorá vyvolá výpadok je presne 2000 mikrosekúnd, teda ak sme znížili počet výpadkov o 7500, celkový čas bude pôvodných 60 sec – 7500 * 2 ms = 45 sekúnd (presne !)

10) Asociatívna pamäť (T.L.B.)

V počítači majú logické adresové priestory procesov 1024 stránok. Tabuľky stránok sú držané v pamäti. Čítanie slova z tabuľky trvá 500 nsec. Na zníženie tejto réžie má počítač asociatívnu pamäť, ktorá drží 32 párov (virtuálna stránka, fyzický stránkový rám) a dokáže vyhľadať položku za 100 nsec.
Aká úspešnosť asociatívnej pamäti (hit rate) je potrebná na redukciu priemernej doby vyhľadávania na 200 nsec?

riešenie:
-> asociatívna pamäť sa používa na zrýchlenie prekladu lineárnej adresy stránky na fyzickú adresu stránkového rámu, obsahuje položky v tvare (stránka, stránkový rám), pričom kľúčom pre vyhladávanie je stránka. Adresovací hardvér hľadá adresu stránky vo všetkých položkách paralelne a iba ak nenájde správnu položku, tak musí zisťovať adresu stránkového rámu pomocou tabuľky stránok – stránka sa hľadá v asociatívnej pamäti (100 nsec) a ak sa nenájde tak sa číta z tabuľky stránok (500 nsec) – teda stránky, ktoré nie sú v asociatívnej pamäti sa čítajú 600 nsec.
-> hit rate = pravdepodobnosť, že adresa sa nachádza na takej stránke, ktorá je v asociatívnej pamäti
-> Nech AP je počet (úspešných) vyhľadaní adresy cez asociatívnu pamäť, TS je počet vyhľadaní cez tabuľku stránok. Priemerná doba vyhľadávania bude t_p = (100*AP+600*TS)/(AP+TS) = 100*AP/(AP+TS) + 600*TS/(AP+TS) nsec.
Pri dostatočne veľkom počte prístupov bude hit rate p = AP/(AP+TS) a 1 – p = TS/(AP+TS), teda úspešnosť asociatívnej pamäti získame riešením rovnice: 200 = 100*p + 600*(1-p) = 100p + 600 – 600p -> 500p = 400 -> p = 4/5

11) Prideľovanie pamäti pomocou „Buddy“ algoritmu I.

Prideľovanie pamäti sa deje pomocou deliaceho “Buddy” algoritmu. Pamäť má veľkosť 1024. Päť procesov obsadí pamäť postupne s požiadavkami na veľkosť priestoru 52, 100, 198, 40 a 132. Koľko pamäti zostáva voľnej a v akých veľkých úsekoch?
riešenie:
-> veľkosti úsekov musia mať pri buddy algoritme vždy veľkosť druhej mocniny, obsadené úseky značím tučným písmom, riešenie podľa prednášky:
1024 <- chceme alokovať 52 -> najbližšia druhá mocnina je 64, pamäť sa delí, kým nevznikne úsek dĺžky 64
512 + 512
256 + 256 + 512
128 + 128 + 256 + 512
64 + 64 + 128 + 256 + 512 <- alokácia 52 (52/64)
64 + 64 + **128** + 256 + 512 <- alokácia 100 (100/128)
64 + 64 + **128** + **256** + 512 <- alokácia 198 (198/256)
64 + **64** + **128** + **256** + 512 <- alokácia 40 (40/64)
64 + **64** + **128** + **256** + **256** + 256 <- alokácia 132 (132/256)
-> k dispozícii ostal úsek o veľkosti 256, úspešne sme splnili všetky požiadavky (alokovalo sa všetko tak ako sa malo)
-> vnútorná fragmentácia: 64 – 52 + 128 – 100 + 256 – 198 + 64 – 40 + 256 – 132 = 246

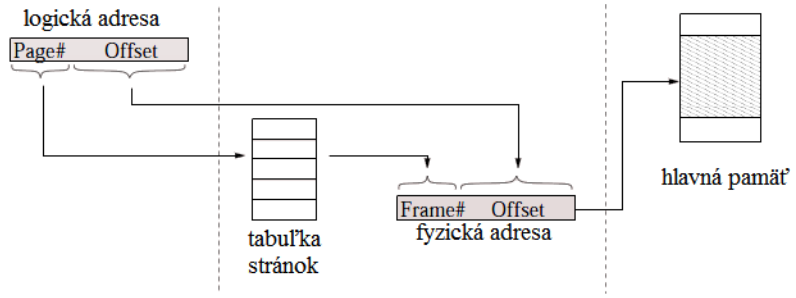
12) Veľkosť tabuľky stránok

Pamäť je organizovaná stránovaním. Veľkosť stránky je 4 KB. Virtuálna pamäť má veľkosť 4 GB, fyzická pamäť má veľkosť 256 MB. Akú veľkosť potrebuje mať tabuľka stránok? Aká je pravdepodobnosť chyby stránkovania pri náhodne zvolenej virtuálnej adrese?
riešenie:
-> 4KB = 2^{12} B, 4GB = 2^{32} B, 256MB = 2^{28} B -> počet stránok je 2^{32} / 2^{12} = 2^{20} – 1 mega stránok
-> počet stránkových rámov vo fyzickej pamäti je 2^{28} / 2^{12} = 2^{16} – 64 kilo stránkových rámov
-> tabuľka stránok nám hovorí, kde vo fyzickej pamäti je ktorá stránka zo všetkých, čo dokážeme adresovať. Má teda toľko položiek, koľko stránok je virtuálnych -> v našom prípade má tabuľka stránok 1 mega položiek -> tabuľku stránok si predstavme ako klasické jednorozmerné pole, pričom indexy poľa sú čísla stránok a hodnoty na konkrétnych indexoch sú čísla stránkových rámov, prípadne fyzické adresy stránkových rámov -> berme, že každá položka obsahuje číslo stránkového rámu, ktorých je 2^{16} teda nám treba 16 bitov (2 bajty) na každú položku -> veľkosť tabuľky je teda 2 MB
-> chyba stránkovania nastane, ak si zvolíme stránku, ktorá nie je vo fyzickej pamäti -> p = počet stránok, ktoré nie sú v pamäti / počet stránok, teda p = (2^{20} - 2^{16}) / 2^{20} = 1 – 2^{16} / 2^{20} = 1 – 1/16 = 15/16 = 93,75% presne

13) Schéma výpočtu adresy

Nakreslite schému výpočtu adresy pri adresovaní s použitím segmentácie a aj stránkovania.

riešenie:
-> preklad adresy pri stránkovaní



-> ako „schému výpočtu“ by som možno napísal takéto niečo:

číslo stránky + posunutie → číslo stránkového rámu x jeho veľkosť + posunutie

-> tento obrázok však odmietam: https://www.fiitkar.sk/wiki/images/b/b3/OS_sprava_pamate_13.png

-> pri segmentácií sa využíva tabuľka segmentov – analógia tabuľky stránok – obsahuje fyzické adresy začiatkov segmentov
-> index do tabuľky segmentov je číslo segmentu z logickej adresy
-> fyzická adresa je súčtom bázovej adresy segmentu a posunutia

19) Fragmentácia

Veľkosť stránky je 512 bajtov. V systéme sa vykonáva alokácia len v súvislých intervaloch adries. Proces vykoná nasledujúcu sekvenciu operácií:

```
A = malloc(1000);
B = malloc(500);
free(A);
A = malloc(2000);
C = malloc(1000);
D = malloc(500);
free(C);
C = malloc(2000);
```

Na alokáciu pamäte sa používa algoritmus First Fit. Koľko bajtov zostane nevyužitých v rámci vnútornej fragmentácie a koľko v rámci vonkajšej fragmentácie stránok? Predpokladajte, že sa pamäť alokuje od adresy 0. Odpoveď zdôvodnite.

riešenie:

-> First Fit znamená, že sa pohybujeme od začiatku pamäťového priestoru a vezmeme prvý úsek, do ktorého sa vôjdeme a alokujeme
-> pamäť je rozdelená na 512 bajtov dlhé úseky, ktoré budem označovať 1 (obsadený) a 0 (voľný)
11 | ← vezmú sa prvé dva úseky a alokuje sa do nich A (1000 / 1024)
11 | 1 | ← do ďalšieho úseku sa alokuje B (500 / 512)
00 | 1 | ← free(A)
00 | 1 | 1111 | ← alokuje sa nové A do 4 úsekov (2000 / 2048)
11 | 1 | 1111 | ← C sa zmestí do prvých dvoch voľných – „first fit" (1000 / 1024)
11 | 1 | 1111 | 1 | ← alokovanie D (500 / 512)
00 | 1 | 1111 | 1 | ← free C
00 | 1 | 1111 | 1 | 1111 | ← alokovanie nového C (2000 / 2048)
-> v rámci vonkajšej fragmentácie zostane nevyužitých 1024 bajtov (prvé dva voľné úseky)
-> v rámci vnútornej fragmentácie zostane nevyužitých 12 + 48 + 12 + 48 = 120 bajtov („diery" v obsadených úsekoch)

23) Buddy alokácia

V systéme je veľkosť jednej stránky pamäte 16 a pre alokáciu je k dispozícii 8 stránok. Samotné stránky sú nedeliteľné a prideľuje ich alokačný mechanizmus typu Buddy (metóda rekurzívneho delenia).

X = 1; while(1) { if(malloc(X) == NULL) break; X = X + 7; }

Odpovede na nasledujúce otázky vysvetlite a znázornite schémou (kreslením obsahu ôsmich stránok):

Koľko alokačných operácií sa vykoná úspešne?

Aká bude na záver súhrnná veľkosť internej fragmentácie?

Aká bude na záver súhrnná veľkosť externej fragmentácie?

riešenie:

-> obsadené úseky sú tučným písmom, na začiatku máme 8 úsekov o veľkosti 16 :

```
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |  ← malloc(1)  (1/16)
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |  ← malloc(8)  (8/16)
| 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |  ← malloc(15) (15/16)
| 16 | 16 | 16 | 32 | 16 | 16 | 16 |        ← malloc(22) (22/32)
| 16 | 16 | 16 | 32 | 32 | 16 |             ← malloc(29) (29/32)
                                             ← 36 sa už nealokuje
```

-> vykoná sa 5 úspešných alokácií (1, 8, 15, 22, 29)
-> v rámci internej fragmentácie ostane nevyužitých 15 + 8 + 1 + 10 + 3 = 37 bajtov
-> v rámci externej fragmentácie ostané nevyužitých 16 bajtov (1 voľný úsek)
-> najbližší voľný úsek sa delí rekurzívne na polovice až kým neplatí, že $2^{i-1} < s \leq 2^i$ -> tu je však delenie vyslovene v zadaní zakázané !

25) Model virtuálnej pamäti

Fragment programu a jeho dáta sú vo virtuálnom adresovom priestore procesu umiestnené podľa doleuvedenej schémy. Veľkosť stránky je 1k a proces má vo fyzickej pamäti pridelené len štyri stránkové rámy označené 0xA, 0xB, 0xC, 0xD. Na začiatku sú tieto rámy neobsadené a obsadzujú sa postupne v poradí A,B,C,D s využitím algoritmu výberu obete stránky typu NRU (Not Recently Used, dlho nepoužívaná stránka). Jeden záznam v tabuľke stránok pozostáva zo štyroch položiek: číslo stránky, bit M, bit R (pre uvedený algoritmus) a bit Absent (stránka nie je v pracovnej pamäti). Periodické nulovanie bitu R nastáva po dokončení každej tretej inštrukcie programu. R1 až R5 sú registre v procesore. Stránka obsahujúca konštantu programu ZERO je nastavená ako rezidentná. Znázornite riešenie pre odpovede na nasledujúce otázky:

Aká bude postupnosť referencií stránok počas vykonávania prvých desiatich inštrukcií programu a koľko výpadkov stránky vtedy nastáva?

Aký bude stav kompletného obsahu tabuľky stránok po vykonaní prvých piatich inštrukcií programu?

Aký bude stav kompletného obsahu tabuľky stránok po vykonaní prvých desiatich inštrukcií programu?

```
0x0c00 …0x0cff   pole A     (stranka 1)
0x1000 …0x10ff   pole B     (stranka 2)
0x1400 …0x14ff   pole C     atd...
0x1800 …0x18ff   pole D
0x1c00 …0x1c01   cislo ZERO      program odtialto:
0x2000       (R1)<-ZERO               prenos
0x2001       (R2)<-A(R1)              prenos (R1: index v poli)
0x2002       (R3)<-B(R1)              prenos
0x2003      B(R1)<-(R2)               prenos
0x2004       (R4)<-C(R1)              prenos
0x2005       (R2)<-D(R1)              prenos
0x2006       (R5)<-(R2)+(R4)          sucet
0x2007       (R5)<-(R5)+(R3)          sucet
0x2008       (R2)<-(R5)+B(R1)         sucet
0x2009      branch 0x2abc             skok
```

riešenie:

-> najskôr, čo je to M a R bit :)

-> M (Modified bit): pri zavedení stránky sa vynuluje, pri modifikovaní stránky sa nastavuje; indikuje, že pred uvoľnením stránkového rámu v ktorom sa stránka nachádza ju treba skopírovať do záložnej pamäti

-> R (Referenced bit): pri každom prístupe k stránke ho HW nastavuje, OS ho preiodicky nuluje (prerušenie od časovača)

-> NRU algoritmus vyberie stránku z určitej skupiny podľa čo najnižšieho R bitu a následne podľa M bitu: priorita výberu („RM“ – 00, 01, 10, 11)

-> stránka rezidentná sa nevyhadzuje a natvrdo tam zostáva a po každej tretej inštrukcii treba bity R vynulovať (**PO** inštrukcii–65 až za 5 končí prvá inšt.)

-> stránky sú: 1 (A) , 2 (B), 3 (C), 4 (D), 5 (ZERO), 6(program) -> 5 je rezidentná, tá je tam stále v jednom ráme

-> postupnosť referencií: 6 5 6 1 6 2 (nuluj) 6 2M 6 3 6 4 (nuluj) 6 6 6 2 (nuluj) 6

```
         6   5   6   1   6   2 | 6   2   6   3   6   4 | 6   6   6 | 2   6
-------------------------------------------------------------------------
0xA      6   5   5   5   5   5 | 5   5   5   5   5   5 | 5   5   5 | 5   5
0xB          6   6   1   6   2 | 6   2   2   2   2   2 | 6   6   6 | 2   2
0xC              6   1   6   | 2   6   6   3   6   4 | 2   2   2 | 6   6
0xD                  1   1   | 1   1   1   6   3   6 | 4   4   4 | 4   4
-------------------------------------------------------------------------
                                 5       6       7   8   9      10          ← číslo inštrukcie
         *   *       *       *               *       *
```

-> pri inštrukcii číslo 5 má stránka číslo 6-10 (RM), 2-01, 1-00 -> ako obeť sa vyberie 1tka

-> obsah stránkových rámov po vykonaní 5tej inštrukcie je 5326 (2-11, 3-10, 6-10, 5-00)

-> obsah stránkových rámov po vykonaní 10tej inštrukcie je 5264 (2-11, 6-10, 4-00, 5-00)

-> došlo k 6tim výpadkom

SÚBOROVÝ SYSTÉM

1) Vlastnosti súborového systému na diskoch

Ako dlho bude trvať prečítanie 64 kB programu z disku s priemernou dobou vyhľadávania stopy 30 msec, časom rotácie 20 msec a veľkosťou stopy 32 kB

a) pre veľkosť stránky 2kB

b) pre veľkosť stránky 4kB

Stránky sú náhodne roztrúsené po disku.

riešenie:

-> prečítanie jednej stránky sa skladá z troch činností:

1. vyhľadanie stopy = 30 ms
2. otočenie disku, kým hlavička „narazí“ na začiatok stránky -> pri najhoršom 20 ms, priemerne však 10 ms
3. prečítanie stránky -> ak celú 32 kB stopu číta 20 ms tak 2kB stránku číta (2/32)*20 = 1,25 ms, 4kB stránku číta 2,5 ms

-> prečítanie celého programu:

a) 32 stránok, každá 30 + 10 + 1,25 ms → teda 32 * 41,25 = 1320 ms

b) 16 stránok, každá 30 + 10 + 2,5 ms → teda 16 * 42,5 = 680 ms

2) Súbor na disku I.

Malý (veľmi malý) disk má jeden povrch, 40 stôp a 9 sektorov na stopu. Na tomto disku je uložený súbor, pričom priradenie fyzických sektorov logickým blokom je opísané reláciou

T = {<0,137>, <1,45>, <2,277>, <3,211>, <4,69>, <5,109>, <6,185>, <7,226>}.

Pre operáciu sekvenčného prečítania celého súboru z disku uveďte, v akom poradí sa budú čítať jednotlivé fyzické sektory, ak sa pre výber požiadavky používa politika „Shortest Seek First“ a na začiatku je čítacia hlavička nad stopou 21. Stopy, sektory a bloky sú číslované od 0.

riešenie:
-> bloky súboru 0,1,2,3,4,… sú na povrchu disku rozptýlené na pozíciách 137, 45, 277, 211,…
-> nás však zaujíma aká je to stopa -> stopa číslo 0 obsahuje sektory 0 – 8, stopa č. 1 sektory 9 – 17, 2 sektory 18 – 26, ... atď...
-> chceme vedieť na akej stope je pozícia 137 -> spravíme 137/9 = 15,2 -> 15*9 = 135, takže stopa 15 obsahuje sektory 135 až 143 -> 137 je na 15. stope
-> pozícia – stopa : 137 – 15, 45 – 5, 277 – 30, 211 – 23, 69 – 7, 109 – 12, 185 – 20, 226 – 25 -> stopy vzostupne: 5 7 12 15 20 23 25 30
-> začíname hlavičkou na stope 21 -> „shortest seek first“, takže ideme k najbližšej stope: 20, odtiaľ je 5 k 15, 3 k 23 takže ideme do 23...
-> postupnosť stôp ktoré prejde hlavička bude: 21, 20, 23, 25, 30, 15, 12, 7, 5 -> poradie fyzických sektorov je: 185, 211, 226, 277, 137, 109, 69, 45

3) Evidencia voľných blokov

Disk s veľkosťou 10 MB má veľkosť bloku 1 KB. Na čísla blokov sú použité 2 B. Určte počet blokov potrebných na evidovanie voľných blokov pri prázdnom a pri plnom disku, keď sa na evidenciu voľných blokov používa
a) bitová mapa
b) zreťazené bloky indexov

riešenie:
-> počet blokov je 10MB / 1 KB = 10 k = 10240 blokov
-> ak je disk plný tak evidovať voľné bloky nemá zmysel, keďže žiadne nie sú -> potrebujeme 0 blokov
a) ak je disk prázdny, tak máme 10240 prázdnych blokov, čiže potrebujeme uložiť 10240 bitov, na čo nám bude treba 2 bloky: 2*1024*8 = 16384 bitov
-> keď plánujeme do 2 blokov dať bitovú mapu, tak počet voľných blokov bude už len 10238 -> potrebujeme 2 bloky
b) ak je disk prázdny tak nám treba 10240*2 =20480 bajtov, čo je 20 blokov
-> ak minieme 20 blokov na evidenciu, ostane 10220 voľných blokov, ktoré vieme v pohode s tými 20 blokmi evidovať

4) Súbor na disku II.

Disk má 1 povrch, 5 stôp a 10 sektorov na stopu. Na disku je uložený súbor, pričom priradenie fyzických sektorov logickým (blokom) je opísané reláciou T = {<0,13>, <1,2>, <2,18>, <3,28>, <4,15>, <5,16>, <6,39>, <7,40>}.
Používateľ požiada operačný systém o načítanie celého súboru do pamäti. Napíšte v akom poradí sa budu čítať jednotlivé fyzické sektory, ak sa pre výber požiadavky používa politika "Elevator" (výťah) a na začiatku je čítacia hlavička nad stopou 3. Stopy, sektory a bloky su číslované od 0.

riešenie:
-> stopy: 0: 0 – 9, 1: 10 – 19, 2: 20 – 29, 3: 30 – 39, 4: 40 – 49
-> blok-stopa: (0 – 1), (1 – 0), (2 – 1), (3 – 2), (4 – 1), (5 – 1), (6 – 3), (7 – 4)
-> plánovanie algoritmom výťahu (elevator algorithm) je princíp zhodný s riadením výťahu. Najprv ide výťah jedným smerom a postupne sa zastavuje na všetkých momentálne požadovaných poschodiach (stopy na disku), potom sa obráti a ide opačným smerom...
-> čítacia hlavička začína na 3. stope, teda pôjde najskôr „hore“ k 4 a potom jedným smerom dole cez 2, 1 až 0
-> postupnosť blokov je teda: 6, 7, 0, 4, 5, 2, 3, 1
-> postupnosť fyzických sektorov je: 39, 40, 28, 13, 15, 16, 18, 2
-> hlavička ide „zhora“ cez stopy zostupne, no sektory v rámci stopy sa budú čítať vzostupne...

→ príklady na procesy som už nedával žiadne, tie na bankárov alogirtmus a round robin sú mega easy, tie „producent a konzument“ sú na dlho...

LihO