KIV/PPR Ukázkový test *(celý bod 1 musí být splněn na minimálně 15 bodů)*

1. Vysvětlete stručně:

[2b] Vysvětlete rozdíl mezi *non-real time*, *soft-real time* a *hard-real time* výpočtem.

non-real time → Vlákna nemají stanovenu žádnou dobu, deadline, do kdy musí dokončit výpočet.
Real time → Dokončení výpočtu ve stanoveném termínu je kritické, termín musí být dodržen bez ohledu na zátěž systému (brzdy v autě, vojenské systémy, podpora života, jaderné zařízení, mobilní telefony, …).
Hard Real Time → Dokončení výpočtu po termínu se považuje za chybu a výsledek za bezcenný – strict deadline. Nedodržení termínu může vést k celkovému selhání systému.
Soft Real Time → Překročení termínu se toleruje, systém reaguje zhoršenou kvalitou poskytovaných služeb.

[3b] Vysvětlit vztah mezi *procesem*, *threadem* a *fiberem* vč. správy zdrojů a plánování.

Proces → největší výpočetní entita plánovače; vlastní prostředky, paměť a další zdroje; v závislosti na OS možnost preemptivního multitaskingu
Thread → každý proces má alespoň jeden - primární - thread; vlákna sdílí adresový prostor procesu a jeho zdroje; každé vlákno má svůj vlastní kontext (id, registry, prioritu, atd.); v závislosti na OS možnost preemptivního multithreadingu
Fiber → terminologie MS; analogie threadu k procesu; Fiber plánuje některý thread procesu, ne plánovač operačního systému. Fiber běží v kontextu threadu, který ho naplánoval. Má pouze stav, zásobník a specifická data – např. prioritu má pouze thread. Fiber může ukončit běh vlákna v jehož kontextu běží. Ukončení aktivního fiberu jiným fiberem není OK – stack corruption => abnormal termination. Nevyužívá preemptivního multithreadingu OS, proces musí zajistit, že se fiber vzdá procesoru.

[1b] Jaký je rozdíl mezi *RTS* a *RTOS*?

Real Time Operating System

- Umožňuje vytvořit systém reálného času
- Sám o sobě nezaručuje, že výsledky budou vypočítány včas (to je úkol programátora)

RTS → jedná se vlastně o program reálného času, např. lednička

- Například pro ledničku, kde jenom monitorujeme teplotu a rozsvěcíme/zhášíme žárovku, není RTOS nezbytně nutný
- Dostatečně malý projekt se může obejít bez velkého RTOS, potřebujeme-li jeho funkčnost v množství menším než malém
- RTOS má také sám o sobě nějakou režii; stejně jste to vy jako programátoři, kdo se musí postarat, aby se stíhali deadlines; pokud se bude RTS někdy portovat, musí být v prvé řadě možné portovat RTOS; scheduler RTOS může pěkně zkomplikovat ladění; cena některých RTOS; v okamžiku, kdy si sami musíte naprogramovat většinu funkcí RTOS, je načas uvažovat o RTOS

[2b] Vysvětlete, proč a kdy se používá klíčové slovo *volatile* (=nestálý).

- Volatile je rezervované slovo C/C++; pokud je nějaká proměnná označena slovem volatile, nebude kompilátor žádným způsobem optimalizovat její užití.
- Deklarace volatile (nestálý) znamená, že k obsahu proměnné může přistupovat ještě nějaký jiný proces než ten, který je řízen aktuálním zdrojovým kódem. Může to být například souběžné běžící vlákno jiného procesu, hardwarové přerušení nebo samotný hardware.

- použití *volatile* pro objekt, který

- představuje vstupní/výstupní port
- je sdílen mezi několika současně běžícími procesy
- je modifikován pomocí přerušení

[2b] K čemu je dobrý objekt typu podmínková proměnná v interface vláken POSIX a jaké operace nad ní jdou provést (vše slovně, popřípadě typy operací v C).

- používá se pro pasivní čekání `vlákna` (stav *waiting*) na splnění podmínky (typicky pro vzájemné vyloučení - např. producent-konzument) - podmínková proměnná je svázána s mutexem.
- Kombinací *mutexu* a *podmínkové proměnné* lze vytvořit **semafor**, **monitor** nebo **bariéru**.
- Operace nad podmínkovou proměnnou **x** sdruženou s mutexem **y**:
  - `pthread_cond_wait (&x, &y);` /* čekání */
  - `pthread_cond_broadcast (&x);` /* vzbuzení všech vláken */
  - `pthread_cond_signal (&x);` /* vzbuzení jednoho vlákna, které je dle své priority a plánovací kategorie "nejprivilegovanější" */
  - `pthread_cond_init(&x,&attr);` /* inicializace podm. proměnné předanými atributy */
  - `pthread_cond_destroy(&x);` /* zruší podm. proměnnou */
  - `pthread_cond_t x = PTHREAD_COND_INITIALIZER;` //stejné jako init s attr NULL

[2b] Uveďte příklad, kdy může dojít k superlineárnímu urychlení.

- Efekt cache-paměti – Rozdělením výpočtu mezi více procesorů může dojít za příznivých podmínek k daleko častějšímu uplatnění lokálních cache-pamětí (častější cache-hit než je obvyklé). Každý lokální výpočet je pak prováděn rychleji než v případě výpočtu jedním procesorem.
- Anomálie při prohledávání – paralelizované prohledávací algoritmy metodou „ořezávání“ pracují často rychleji, než by odpovídalo lineárnímu urychlení. Je to hlavně díky tomu, že paralelizovaný algoritmus je vlastně odlišný od sekvenčního a umožňuje většinou rychlejší upřesňování průběžného výběrového kritéria.

[2b] Celočíselný semafor (co to je, k čemu slouží, jaké jsou základní operace nad celočíselným semaforem).

- semafor je synchronizační primitivum, které obsahuje celočíselný čítač. Využívá se zejména jako ochrana proti souběhu tím, že chrání přístup do kritické sekce *(ideální pro producent-konzument)*.
- V() - *verhogen (= zvýšit)* - Při výstupu z kritické sekce je vyvolána operace V(), která odblokuje vstup do kritické sekce pro další (čekající) proces (tj. zvýší čítač o 1).
- P() - *proberen (= zkusit)* - otestuje stav čítače a v případě že je nulový, zahájí čekání. Je-li nenulový, je čítač snížen o jedničku a vstup do kritické sekce je povolen
- pseudokód:

```
P(Semaphore s){ wait_while_not(s > 0); then s--;/*atomická
operace */}
V(Semaphore s { s = s+1; /* musí být atomické */}
Init(Semaphore s, Integer v) { s = v; }
```

[3b] Popište tři různé synchronizační objekty WinAPI.

- Event, Mutex, Semafor
- objekt je buď signaled, nebo nonsignaled

- čeká se pomocí tzv. wait funkcí
    - **Single-Object** → `SignalObjectAndWait, WaitForSingleObject/Ex`
    - **Multiple-Object** → `WaitForMultipleObjects/Ex, MsgWaitForMultipleObjects/Ex`
    - **Alertable Wait** → Čekání pomocí `SignalObjectAndWait` a `*Ex` se ukončí i v případě, že systém dokončil I/O operaci, nebo má nastat APC (asynchronous procedure call) pro dané vlákno
    - **Registered Wait** (určeno pro thread pool) → `RegisterWaitForSingleObject, UnregisterWaitEx`

[2b] Popište, k čemu je dobrá kritická sekce při synchronizaci.

Kritická sekce je nejmenší část kódu, kde dochází k přístupu ke sdílenému prostředku (např. sdílená data), ke kterému nemohou současně přistupovat dva nebo více procesů či vláken.

- Při řízení přístupu do kritické sekce musí být dodrženy tři podmínky:
    - výhradní přístup – vstup do kritické sekce je povolen nejvýše jednomu procesu
    - vývoj – rozhodování o vstupu je pouze na procesech, které o něj usilují
    - omezené čekání – rozhodnutí o vstupu nesmí být pro některého čekajícího odkládáno do nekonečna

[2b] Uveďte, jakou strojovou instrukci potřebujete pro realizaci spinlocku, kdy se spinlock používá a kdy se spinlock používat nemá.

- instrukci TSL (Test and Set Lock), která nastaví proměnnou na true a vrátí její předchozí hodnotu (jedná se o atomickou instrukci, v C: `int __sync_lock_test_and_set(int *, int))`
- strojová instrukce `lock cmpxchg`
- spinlock je aktivní čekání, proces tedy spotřebování systémové prostředky `spinlock_t lk = SPIN_LOCK_UNLOCKED; spin_lock(&lk); spin_unlock(&lk);`
- spinlock se používá všude tam, kde se nesmí čekat v uspaném stavu (obsluha přerušení, obsluha události časovače apod.), ale i v případech, kde se jedná o velice krátký chráněný úsek (spinlock má menší režii než jiné synchronizační objekty).

[2b] Uveďte, jaké jsou základní výhody a nevýhody dynamického load-balancingu oproti statickému load-balancingu.

statický:
- výpočet přiřazení procesů na uzly je proveden ještě před spuštěním distribuované aplikace
    - výpočet může běžet libovolně dlouho, abychom dosáhli požadované přesnosti předpovědi – pokud ji metoda umožňuje dosáhnout
- nelze reagovat na dynamické změny v prostředí
- vyžadují předem spoustu informací o chování sítě a aplikace (např. kom. zpoždění, doby běhu procesů)
    - nereálné požadavky nelze splnit
        - vliv na přesnost a tedy i rychlost výpočtu

dynamický:
- výpočet přiřazení procesů na uzly sítě se provádí za běhu distribuované aplikace
    - výpočet se odehrává v reálném čase a nemůže si proto dovolit konzumovat příliš mnoho zdrojů
- umí se vyrovnat s dynamickými změnami
    - procesy musí umět pre-empci
- potřebné informace lze zjistit až za běhu aplikace
    - nebo si jich část vyžádat předem

[2b] Jaké jsou možnosti konstrukce rendez-vous v Adě, uveďte příklad(y).

- rendezvous je přímo podporované synchronizační primitivum pro **task** (task je paralelizační typ)
- Ada má chráněnné celé objekty (Protected Objects, Protected Types):
  - *entry* slouží přímo jako vstup do kritické sekce a mění stav objektu, task je proveden pouze pokud je podmínka u entry true
  - *procedure* je bez ošetření KS, tedy normální standardní metoda
  - *function* je metoda, která nemění stav objektu, ale pouze jeho stav vrací
- *entry* vytváří meeting point a *accept* vyvolá čekání (tast se uspí) na další **task**

```
task type Simple_Task is //bez type použitelné jen pro jendu instanci
        entry Start(Num : in Integer);
        entry Report(Num : out Integer);
   end Simple_Task;

task body Simple_Task is
   Local_Num : Integer;
begin
   //čeká na vložení čísla – entry call
   accept Start(Num : in Integer) do
        Local_Num := Num;
   end Start;
   //normálně pokračuje v běhu
   Local_Num := Local_Num * 2;
   //čeká na vyzvednutí spočítané hodnoty
   accept Report(Num : out Integer) do
        Num := Local_Num;
   end Report;
end Simple_Task;
```

Může být nezbytné, aby úkol mohl reagovat na několik vstupních volání (entry calls) – pokaždé na jiné dle okolností – tj. ne v předem určeném pořadí.

```
// Vynutíme si inicializaci a další se už pak může vykonávat v libovolném pořadí.
accept Init(Item : in Integer) do
   Local_Item := Item;
end Init;
loop
   select
        accept Stop;
             exit;
   or
        when podmínka = > //může i nemusí být
        accept Put(Item : in Integer) do
             Local_Item := Item;
        end Put;
        Local_Item := Local_Item * 2;
   else
        Put_Line("No entry call at this time");
   end select;
   delay 0.01;
end loop;
```

2. [6b] Napište fragment kódu, který by realizoval spinlock.

```
void spinlock_lock(int *lock){
  while(test_and_set(lock) == 1); //test_and_set() je atomická operace
}
void spinlock_unlock(int *lock){ *lock = 0; }
void spinlock_init(int *lock){ *lock = 0; }
```

3. [10b] Popište, v čem spočívá optimalizace následujícího fragmentu kódu:

```
for i:=1 to High(Items) do Items[i]:=Items[i] + Items[i-1];
```

- jde o součet prefixů viz přednáška 3b Shared SPMD *(příliš dlouhé a lehce zmatené abych to sem celé přepsal)*

4. [6b] Napište MPI program, jehož jeden proces vypíše na standardní výstup (uzlu, na němž běží proces s rankem MPI_COMM_WORLD 0) zprávu "Nazdar z procesu xx!" ze všech procesů v konfiguraci systému. Komentujte, co dělají jednotlivé použité MPI funkce.

```
MPI_Init(&argc, &argv);

  int processes_count;
  MPI_Comm_size(MPI_COMM_WORLD, &processes_count);

  int process_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &process_id);

  if(process_id == 0) {
    int msg_size, i;
    MPI_Status status;

    for(i = 1; i < processes_count; i++){
      MPI_Probe(0, 0, MPI_COMM_WORLD, &status); //ziska status prichozi
     msg
      MPI_Get_count(&status, MPI_BYTE, &msg_size); //ziska velikost dat
      char *msg[msg_size]; //alokace bufferu
      MPI_Recv(msg, msg_size, MPI_BYTE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
          &status); //prijmu zpravu a ulozim do bufferu
      std::cout << msg << std::endl;
    }
  }else {
    int msg_size;
    char *msg[100];
    msg_size = sprintf(str, "Nazdar z procesu %d!", process_id);
    MPI_Send(&msg, msg_size, MPI_BYTE, 0, 0, MPI_COMM_WORLD);
  }

  MPI_Finalize();
```

5. V prostředí MPI potřebujete implementovat svou vlastní bariéru, tj. ekvivalent funkce MPI_Barrier.

a) [10b] Uveďte slovně a doplňte příslušným fragmentem kódu, jak byste bariéru realizovali.

- root zavolá **recv** pro všechny ostatní nody a čeká až se ozvou (root je na bariéře a čeká na ostatní)
- každý node, který dokončí svou práci pošle **send** do root nodu a zablokuje se na **recv** (tj. čeká na bariéře na zbytek)
- až root přijme od vsech zprávu (tj. ∀ jsou na bariéře), tak všem pošle zprávu že mohou pokračovat

```
MPI_Init(&argc, &argv);

int processes_count;
MPI_Comm_size(MPI_COMM_WORLD, &processes_count);

int process_id;
MPI_Comm_rank(MPI_COMM_WORLD, &process_id);

MPI_Status status;

if(process_id == 0) { //root
  int i;

  for (i = 1; i < processes_count; ++i) {
    //wait for all others nodes done their jobs
    MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, MY_TAG_FOR_BARRIER,
             MPI_COMM_WORLD, &status);
  }

  //release all nodes blocked on barrier
  for(i = 1; i < processes_count; i++){
    MPI_Send(NULL, 0, MPI_BYTE, i, MY_TAG_FOR_BARRIER, MPI_COMM_WORLD);
  }
}else { //others
  /* … do some job … */
  //signal when you are done
  MPI_Send(NULL, 0, MPI_BYTE, 0, MY_TAG_FOR_BARRIER, MPI_COMM_WORLD);
  //blocking on barrier
  MPI_Recv(NULL, 0, MPI_BYTE, 0, MY_TAG_FOR_BARRIER, MPI_COMM_WORLD,
        &status);
}
```

b) [3b] Popište slovně funkce MPI (co dělají, parametry, které jste potřebovali v bodě a.)

MPI_Comm_size → získá celkový počet processů
MPI_Comm_rank → získá ID aktuálního proccesu (0 je hlavní proces)
MPI_Recv(data, velikost, typ, odesílatel, TAG = označení zprávy, nevim, status) → přijme zprávu
MPI_Send(data, velikost, typ, adresát, označení zprávy, nevim) → odešle zprávu
MY_TAG_FOR_BARRIER → konstanta označující zprávy týkající se bariéry

Z testů z minulých let

Napište bariéru v PVM, bez použití funkce pvm_barier + popište jednotlivé pvm funkce, které jste použili

```
Master:
   taskCount = 10;
   int *tids; //pole s IDckama vzniklejch procesu
   int cc = 0;
   int msg = 1;
   cc = pvm_spawn("slave", (char**)0, PVMTaskDefault, 0, taskCount, tids);

   //přiděl práci a zablokuj se na recv
   for (i = 0; i < cc; i++) { pvm_recv(tids[i], -1); }
   //az vsichni skonci, tak je odblokuj
   for (i = 0; i < cc; i++) {
     pvm_initsend(PvmDataDefault);
     pvm_pkint(&msg, 1, 1);
     pvm_send(tids[i], 0);
   }

slave:
   int ptid = pvm_parent();
   int msg = 1;
   //...dokončili jsme nějakou svojí činnost...
   pvm_initsend(PvmDataDefault);
   pvm_pkint(&msg, 1, 1);
   pvm_send(ptid, 0); //synchronizace na barieru
   //zablokovani, master uvolni az budou na bariere vsichni
   pvm_recv(ptid, -1);
```

`pvm_spawn` - vytvoří nějaký počet procesů podle zadaných kritérií, do parametru int* tids šoupne pole intů s ID vytvořených procesů, vrací délku tohoto pole, první parametr-cesta k binárce programu

`pvm_recv(tid, msgtag)` - blokující volání, čeká na data z procesu s ID tid

`pvm_initsend()` - vyprázdní buffer a nastaví kódování

`pvm_pkint(int *ip, int pocetpolozek, int step)` - pack fce, vloží data do bufferu

`pvm_send(int tid, int msgtag)` - odeslani zapackovanych dat

`pvm_parent()` - vrátí tid rodiče, mastera, prostě procesu který nás tyvvořil v pvm_spawn

Napište v MPI kus kódu, aby každý proces vypsal na stand. výstup "Ahoj já jsem proces XX!", popište kadou pouitou funkci

```
MPI_init(&argc, &argv); //Inicializace MPI
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); //zjištění čísla procesoru
cout << "Ahoj já jsem proces XX!" << rank << "!" << endl;
MPI_finalize();
```

Napsat v pseudokodu 4 ruzne algoritmy vecericich filozofu (3x4 body), které nevedou na deadlock/livelock a jsou paralelní.

- **“Rebel Inside, Rebel for Life”**
  - Jeden z filozofů bude mít jinou vnitřní logiku (bude si brát vidličky v opačném pořadí než ostatní) → jenomže už nemáme všechny komponenty identické
- **Ostraha jídelny**
  - Jídelnu bude hlídat semafor, který bude inicializován na hodnotu <1, M-1>
  - Komponenty jsou identické, ale přibyl nám do programu další prvek
- **Ohodnocení vidliček (zdrojů) – priorita**
  - Filozof si vždy vezme nejprve vidličku s nejvyšší prioritou
  - Všechny komponenty jsou identické, stále stejný počet prvků v programu
- **Aloha, CSMA/CD, …**
  - Filozof se pokusí vzít si obě vidličky postupně
  - Když to nevyjde, uvolní, co si vzal, a počká náhodnou dobu, než to zkusí znovu
  - Sice to lze těžko formálně zaručit, ale v praxi to funguje u Ethernetu do překročení hraniční hodnoty aktivity vysílačů

[2b] Asychronní komunikace (princip, základní operace).

- odesilatel vklada zpravu do vyrovnavaci pameti a prijemce si ji vyzvedne az ji bude potrebovat
- mezi vlakny pomoci zprav `send` a `recv;` jen receive je blokujici (v synchr. je i send blokujici)
- lze udělat i recv asynchronní, třeba pomocí boost.asio

[2b] Co to je asymetrické rendez-vous?

- že volaný (ten co dal accept) nemusí znát toho kdo jej zavolal

[2b] Realizujete paralelní součet vektoru s rozměrem n na symetrickém multiprocesoru s N procesory (n>>N). Jaké mezní urychlení můžete dosáhnout a co pro to musí být splněno?

[2b] Operace P() nad semaforem (jaká data má objekt semaforu, co operace P() dělá).

- P() = *proberen (= zkusit)* - otestuje stav čítače a v případě že je nulový, zahájí čekání. Je-li nenulový, je čítač snížen o jedničku a vstup do kritické sekce je povolen
- semafor má jen celočíselnou hodnotu udávající max. počet současně přistupujících vláken

```
P(Semaphore s){
      wait_while_not(s > 0);
      then s--; /*atomická operace */
}
```

[2b] Jak se vyrobí v Javě monitor? Jaké má skryté atributy a k čemu tyto atributy slouží?

- monitor se skládá z dat, ke kterým je potřeba řídit přístup, a množiny funkcí, které nad těmito daty operují.
- jako podporuje monitory přímo konstrukcí "synchronized" lze synchronizovat blok, instanční metodu nebo statickou metodu (dnes lepší používat fce z balíku `java.util.concurrent`)
- skryté atributy (zřejmě) jsou fronta čekajících vláken pro získání monitoru a fronta spících vláken uspaná voláním `wait()`

[2b] Jaké skryté atributy má kritická sekce?

- musí mít frontu čekajících vláken pro vstup do kritické sekce, metody pro získání a uvolnění “zámku” a něco co definuje je-li zamčeno

[2b] Fyzická komunikační topologie multiprocesoru s distribuovanou pamětí (uvést příklady).

[2b] K čemu je dobrý objekt typu "mutex" pro vlákna v normě POSIX, jaké se s ním dají dělat operace

- mutex se používá pro vzájemné vyloučení
- můžeme jej inicializovat, zamknout a odemknout (v kombinaci s podm. proměnnou lze vytvářet např. semafory, atp.)

[2b] Urychlení algoritmu (definice + příklad)?

- Speedup = Poměr doby výpočtu referenčního algoritmu a porovnávaného algoritmu
- S(p) = E(1) / E(p), >1 znamená urychlení
- perfektní urychlení → poměr je přesně roven počtu procesorů (nelze dosáhnout)

[2b] Účinnost paralelního algoritmu (definice + nějaký příklad).

- Efficiency = urychlení dělené počtem procesorů (uvažujeme urychlení proti sekvenčnímu algoritmu)

```
Sekvenční výpočet trvá 10s, paralelní algoritmus na 4 procesorech trvá 5s
⇒ urychlení = 2
⇒ účinnost  = 0,5
```

[2b] Model MPSD (proudové zpracování dat), co to je, jaké urychlení lze dosáhnout.

- **M**ultiple Instruction (**P**rogram), **S**ingle **D**ata Stream
- Používáno pro výpočty odolné proti poruchám (**Fault Tolerant**)
  - Několik různých systémů zpracovává ty samá data a musejí se shodnout na výsledku – např. řízení letu raketoplánu, letadla, atd.
- Používá se v tzv. **Pipeline architektuře**
  - několik procesů zpracovává data v jednom datovém proudu
  - analogií je montážní linka v továrně
  - Např. instrukční pipeline procesoru

[1b] Jaká je maximální komunikační vzdálenost dmax pro 2D mřížku/toroid/3D mřížku/ s rozměrem M.

```
dmax(2D)     = 2 * (M - 1)
dmax(toroid) = M - 1
dmax(3D)     = 3 * (M - 1)
```

[3b] Jaké jsou redukční operace v MPI a jake jejich parametry?

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,
           MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

- count – počet prvků v sendbuf
- recvbuf je jenom jedna hodnota – výsledek
- existuje i varianta `MPI_AllReduce`
  - výsledek se nepošle jenom rootu, ale všem procesům

[3b] Jaký je rozdíl v implementaci vláken v Javě a POSIX/rozhraní WinAPI?

[2b]Rozdily mezi SMP (Symetrický Multi Procesor) a AsMP (Asymetrický Multi Procesor)

- SMP
  - Všechny procesory jsou identické, vlákno může být alokováno na libovolný procesor
  - Pro plánovač OS je to jednodušší, než kdyby se procesory významně lišily
  - Procesory sdílejí jednu paměť – úzké hrdlo je sběrnice
- ASMP
  - Kromě sdílené paměti, každý procesor má vlastní lokální paměť a vlastní připojení I/O
  - Každý procesor může mít jinou instrukční sadu (v extrémním případě na každém z nich běží jiný OS) → OS nemůže alokovat libovolný proces na libovolnýprocesor

[3b] Co je základní abstraktní práce v TBB? Jak se používá?

- namísto vláken specifikuje **úlohy**, *tasks*, a jejich návaznosti
  - důvodem je mj. redukce efektu cache-cooling
- TBB vykonává úlohy paralelně, jak nejlépe to se současným know-how jde

[2b] uveďte 2 způsoby plánování vláken v RTOS.

- **událostně řízené**
  - úloha se přepne pouze tehdy, když nastane událost s vyšší prioritou, než má běžící úloha
  - kooperativní multithreading – úloha se po nějaké době dobrovolně vzdá procesoru
- **sdílení času** (tj. virtualizace procesoru)
  - úlohy se přepínají nejenom událostmi, ale i podle hodin
- **Earliest Deadline First**
  - Vlákna jsou v prioritní frontě, jakmile dojde k události jako vytvoření, či dokončení vlákna, fronta se prohledá a vyberou se vlákna s nejbližším termínem (z nich se pak vybere podle priority)
- **Monotonic scheduling**
  - Přiřazuje priority jednotlivým úlohám tak, aby stihly dokončit výpočet v termínu
  - Možným výsledkem je, že se zjistí, že to úloha nemůže stihnout

[2b] Fyzická komunikační topologie multiprocesoru s distribuovanou pamětí?

- Pevná – procesory jsou spojeny komunikačnímkanálem
  - každý s každým, 2D mřížka, Toroid, 3D mřížka (krychle), N-cube, Systolické pole
- Flexibilní
  - *Circuit switching*
  - *Packet switching*

[2b] Co je to anomální urychlení a jak ho lze vysvětlit?

- Distribuce rozsáhlých dat u distribuované aplikace může omezit nutnost stránkovat RAM
- S dostatečně rychlými komunikačními kanály pak dojde k rychlejšímu vykonávání programového kódu, protože odpadá čekání na zpomalující I/O operace provázející stránkování, včetně obsluhy příslušných přerušení (KIV/OS)
- Např. paralelizované vyhledávací algoritmy mohou mít větší než lineární urychlení
  - Lze rychleji upřesnit výběrová kritéria

Čeho se musíte vyvarovat při použití funkce pvm_spawn(), má-li být program snadno přenositelný do jiné instalace PVM?

- nesmí být natvrdo zadáno jméno stroje na kterém se mají procesy spustit → použití `PVMTaskDefault` jako `char *where`

```
int numt = pvm_spawn(char *task, char **argv, int flag, char *where,
           int ntask, int *tids )
```

vlakna, jedno vypisuje "Nazdar ", druhe "svete!\n" napiste program mainu a obou vlaken, aby vypsali 50x "Nazdar svete!\n" vlakna bezi celou dobu (tj. neuspavaji se), k synchronizaci se pouziva pouze instrukce TestAndSet

MAIN: deklaruje a inicializuje mutex1 na odemčeno a mutex2 na zamčeno a spustí obě vlákna

Vlakno 1	Vlakno 2

`for(1..50)`
`  while(TSL(mutex1) == 1);`
`  print("Ahoj ");`
`  unlock(mutex2);`
`  lock(mutex1);`
`endfor;`
	`for(1..50)`
	`  while(TSL(mutex2) == 1);`
	`  print("světe!");`
	`  unlock(mutex1);`
	`  lock(mutex2);`
	`endfor;`

Mate hodne dlouhy retezec v kodovani UCS-2 (2 byte na znak), ktery je zakonceny znakem \0. Vysledkem ma byt opet UCS-2 retezec zakonecny \0. Napiste paralelni program, ktery veme 2 retezce a vrati 1 retezec, ktery bude mit na kazde pozici spojene znaky z teze pozice obou retezcu. Pri mergovani se budou znaky radit abecedne.
Napr. AUBCD a CDF -> ACDUBF

- 2 byte na znak znamená, že se to nedá ukládat do pole charů
- ideálně každý proces dostane stejnou cast z obo retezcu, nad nimi udela merge a posle je bud masterovi nebo třeba sousedovi, který je spojý se svými a pošle svému sousedovi až se to dostane stejně k masterovi - ten to spojí a vše hotovo

Specifikujte intel TBB na prikladech.

```
tbb::task* CMasterCalculationTBBLogic::execute() {
	tbb::task_list list;
	for (int i=gaiFirst; i<=gaiLast; i++)
		list.push_back(*new(allocate_child()) CTask(i));

	set_ref_count(gaiLast-gaiFirst+2); //počet úloh +1
	spawn_and_wait_for_all(list);
	return NULL;
	//non-NULL by byla úloha, která by měla být spuštěna
	//jako další/závislá na téhle
}
```

Napiste program, ktery scita 2 vektory pomoci tbb::parallel_reduce.

```
class CMulVect {
	floattype *mA, *mB;
	int mLen;
public:
	floattype mProduct;
	CMulVect(floattype *a, floattype *b, int len) :
		mA(a), mB(b), mLen(len), mProduct(0.0) {}
	CMulVect(CMulVect& x, tbb::split) : mA(x.mA),
		mB(x.mB), mLen(x.mLen), mProduct(0.0) {}
	void join(const CMulVect& y) {
		mProduct += y.mProduct;
	}

	void operator()( const tbb::blocked_range<size_t>& r ) {
		int r_end = r.end();
		floattype *a = mA;
		floattype *b = mB;
		floattype sum = 0.0;
		for (int i=r.begin(); i!=r_end; ++i)
			sum += a[i]*b[i];
		mProduct += sum;
	}
}; //class CMulVect END

floattype MulVect(floattype *a, floattype *b, int len) {
	floattype result = 0.0;
	//Jsou data tak velká, aby se je vůbec vyplatilo počítat paralelně?
	if (len<=rmSerialThreshold) {
```

```
            for (int i=0; i<len; i++)
                  result += a[i]*b[i];
      } else {
            CMulVect mul(a, b, len);
            tbb::parallel_reduce(
            tbb::blocked_range<size_t>(0,len), mul);
            result = mul.mProduct;
      }
      return result;
} // floattype MulVect END
```

Máte bitmapu bludiste o rozmerech m x n; znaky 'x' ozancuji "zed", 'A' vstup, 'B' vystup a '*mezera*' volne misto pro pruchod. Spocitejte paralelne min vzdalenost cesty z A do B tak, aby byly vsechny procesory zatizene.
[7b] PVM realizujte programový model MPSD. Datové prvky jsou celočíselné vektory s rozměrem n. Rozměr se může měnit! Proud prochází přes N serverů. Napište proces, který realizuje jeden z vnitřních serverů. Pro vlastní zpracování prvku máte k dispozici funkci.
[10b] Máte vektor čísel. Napište paralelní program, který v poli najde nejdelší sub-vektor kladných čísel.
[8b] Popiště jak se na architektuře AMD či IA32 implementuje multithreading.
[5b] Máte pole čísel. Napište program pro distribuovaný systém, který spočítá průměr prvků pole.
[5b] Máte pole čísel. Napište program pro systém se sdílenou pamětí, který spočítá průměr prvků pole.