1) **Proč je falešné sdílení (false sharing) problémem:**

   A. Překladač: Překladač není schopen pro paralelizaci vygenerovat optimální strojový kód
   B. Správnost: Paralelizace povede k nedeterministickým výsledkům
   C. Výkon: Zrychlení dané paralelizací bude nižší, než by bylo možné jinak očekávat. Škálovatelnost paralelizace tak bude omezená

2) **Uvažujme třídu std::atomic:**

   A. Má copy constructor: class_name (const class_name &)
   B. Nemá ani copy constructor, ani move constructor
   C. Má move constructor: class_name (class_name &&)

3) **Falešné sdílení (false sharing) můžeme odstranit:**

   A. Pomocí podmíněné proměnné
   B. Vytvořením lokální kopie sdílené proměnné
   C. Pomocí globálního zámku nad sdílenou proměnnou

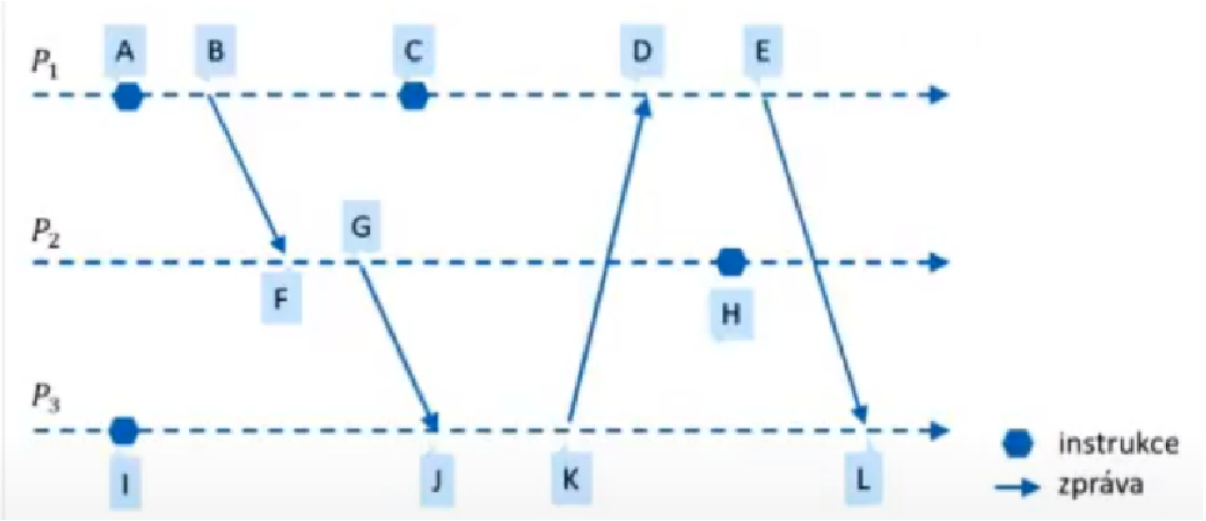
4) **Mějme sekvenční program. Předpokládejme, že výpočet trvá 20 % času, zbytek času je nevyužit či se čeká na I/O. Dále předpokládejme, že výpočet můžeme 10x zrychlit. Jak dlouho poběží paralelizace programu podle Amdahlova zákona?**

   A. O 21.95 % rychleji
   B. O 18 % rychleji
   C. O 19.05 % rychleji
   D. O 20 % rychleji
   E. O 2 % rychleji

5) **Uvažujme distribuovaný výpočet v asynchronním distribuovaném systému. Uvažujme následující tři vlastnosti distribuovaných výpočtů - *živost*, *bezpečnost* a *odolnost vůči selhání* – a uvažujme jakých garancí na tyto vlastnosti jsme schopni dosáhnout, pokud příslušný distribuovaný algoritmus vhodně navrhneme:**

   A. Lze současně garantovat bezpečnost a živost
   B. Lze současně garantovat bezpečnost, živost a odolnost vůči selháním
   C. Lze garantovat buď bezpečnost nebo živost, ale ne obě vlastnosti dohromady
   D. Lze garantovat buď bezpečnost nebo odolnost vůči selháním, ale ne obě vlastnosti dohromady
   E. Lze současně garantovat bezpečnost a odolnost vůči selháním
   F. Lze současně garantovat živost a odolnost vůči selháním
   G. Lze garantovat buď živost nebo odolnost vůči selháním, ale ne obě vlastnosti dohromady

6) **V distribuovaném systému vykonávajícím výpočet zachycený na obrázku běží *skalární* i *vektorové* logické hodiny. Na začátku výpočtu jsou všechny hodiny inicializovány na nulové hodnoty.**

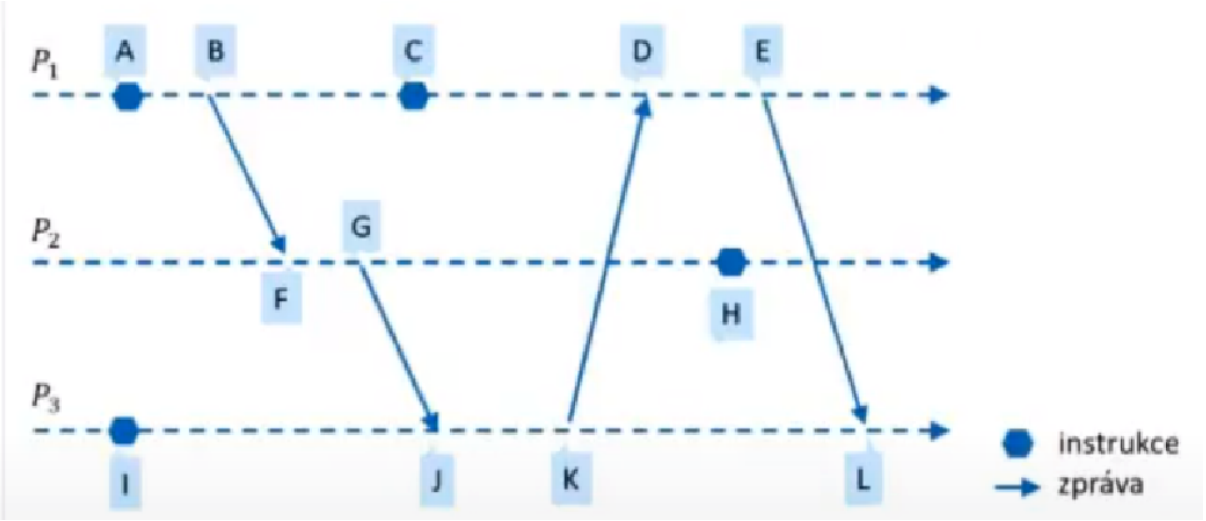


Označte všechna pravdivá tvrzení týkající se hodnot logického času v průběhu výpočtu

Pozn.: Jak pro skalární, tak pro vektorové hodnoty se dotazujeme na jejich hodnoty ve dvou různých okamžicích výpočtu. Pro každý z těchto okamžiků jsou v seznamu tři možné hodnoty, z nichž právě jedna je správná:

A. Hodnota vektorových hodin procesu P2 bezprostředně po události G bude (0,2,0)
B. Hodnota skalárních hodin procesu P3 bezprostředně po události L bude 9
C. Hodnota skalárních hodin procesu P3 bezprostředně po události L bude 6
D. Hodnota vektorových hodin procesu P2 bezprostředně po události G bude (2,2,0)
E. Hodnota vektorových hodin procesu P3 bezprostředně po události L bude (0,0,4)
F. Hodnota vektorových hodin procesu P3 bezprostředně po události L bude (5,3,4)
G. Hodnota vektorových hodin procesu P3 bezprostředně po události L bude (5,2,4)
H. Hodnota vektorových hodin procesu P2 bezprostředně po události G bude (2,2,1)
I. Hodnota skalárních hodin procesu P2 bezprostředně po události H bude 4
J. Hodnota skalárních hodin procesu P3 bezprostředně po události L bude 8
K. Hodnota skalárních hodin procesu P2 bezprostředně po události H bude 5
L. Hodnota skalárních hodin procesu P2 bezprostředně po události H bude 3

7) **V distribuovaném výpočtu zachyceném úplně na obrázku uvažujme uspořádání událostí. Označte všechny výroky, které jsou pravdivé vzhledem k uspořádání dle relace stalo se před:**



A. Událost **I** se stala před událostí **E**
B. O kauzálním vztahu událostí **I** a **H** nelze rozhodnout
C. **I** a **H** jsou souběžné události
D. Událost **C** se stala před událostí **K**
E. **C** a **G** jsou souběžné události
F. **D** a **L** jsou souběžné události
G. Událost **F** se stala před událostí **L**
H. O kauzálním vztahu událostí **B** a **G** nelze rozhodnout

8) Které z následující vlastností distribuovaného výpočtu jsou nutné k tomu, aby algoritmus Ricart-Agrawala garantoval *bezpečnost* a *živost*:

A. Ve výpočtu nedochází ke ztrátám zpráv.
B. Ve výpočtu nedochází ke zpoždění zpráv.
C. Doba odezvy procesů je kratší než doba strávená procesy v kritické sekci.
D. Ve výpočtu nedochází k haváriím procesů
E. Doba přenosu zpráv je kratší než time-out pro vstup do kritické sekce.

9) Jak v OpenMP nastavím počet běžících vláken pro jeden cyklus na nejvyšší úrovni na X. Vyberte všechny správné odpovědi:

A. Nastavením tzv. “internal control variable” nthreads-var na X,1,1
B. Spuštěním programu s parametrem “--threads X”
C. Direktivou #pragma omp parallel num_threads(X)
D. Příkazem omp_set_num_threads(X)
E. Odnastavením proměnné prostředí OMP_THREAD_LIMIT
F. Nastavením proměnné prostředí “export OMP_NUM_THREADS=X,1,1”
G. Nastavením proměnnou prostředí “export OMP_THREAD_MAX=X”

10) Direktiva #pragma omp parallel vytvoří:

A. Tým vláken, která se připojí k hlavnímu vláknu (join) po konci následujícího bloku
B. Tým vláken, která se připojí k hlavnímu vláknu (join) po konci tohoto bloku

C. Tým vláken, která je potřeba explicitně připojit (join) k hlavnímu vláknu

11) Uvažujme následující příklad:

```
std::mutex m;
#pragma omp parallel num_threads(2)
    {
#pragma omp single nowait
        {
            m.lock();
#pragma omp task
            {
                m.lock();
                Hello();
                m.unlock();
            }
        }
        Hello();
        m.unlock();
    }
```

A. Dojde k uváznutí
B. Nedojde k uváznutí

12) Pro použití podpory paralelismu v algoritmu standardní šablonové knihovny STL je potřeba:

A. Volat objekt třídy std::execution::seq s parametrem algoritmu
B. Volat objekt třídy std::execution::par s parametrem algoritmu
C. Instanciovat objekt třídy std::execution::seq a ten předat algoritmu
D. Předat objekt std::execution::seq algoritmu
E. Předat objekt std::execution::par algoritmu
F. Instanciovat objekt třídy std::execution::par a ten předat algoritmu

13) Uvažujme následující příklad:

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
    int a = 42, b = 1;
#pragma omp parallel num_threads(thread_count) shared(a) firstprivate(b)
```

Která tvrzení platí:

A. Ve dvou vláknech vytvořených direktivou parallel není zřejmé, jakou bude mít proměnná b hodnotu. Bude inicializovaná na 42

B. Ve dvou vláknech vytvořených direktivou parallel není zřejmé, jakou bude mít proměnná b hodnotu. Bude inicializovaná na 1
C. Ve dvou vláknech vytvořených direktivou parallel není zřejmé, jakou bude mít proměnná b hodnotu. Bude ale sdílena.
D. V hlavním vlákně není zřejmé, jakou bude mít proměnná b hodnotu před během bloku za direktivou parallel.
E. Ve dvou vláknech vytvořených direktivou parallel není zřejmé, jakou bude mít proměnná b hodnotu. Nebude ale sdílena.
F. V hlavním vlákně není zřejmé, jakou bude mít proměnná b hodnotu po běhu bloku za direktivou parallel

14) Uvažujme kód:

```
std::mutex m1;
void f(int id) {
std::lock(m1);
std::lock_guard lock1(m1, std::adopt_lock);
std::cout << "Thread " << id << " says hi." << std::endl;
}
int main(int argc, char* argv[]) {
std::thread t1(f, 1);
std::thread t2(f, 2);
t1.join();
t2.join();
}
```

A. Nedojde k uváznutí, protože mutex nebude odemčen
B. Nedojde k uváznutí, protože mutex m1 bude odemčen dvakrát
C. Dojde k uváznutí, protože mutex nikdy nebude odemčen
D. Dojde k uváznutí, protože mutex bude odemčen dvakrát
E. Nedojde k uváznutí, protože mutex nikdy nebude zamčen

15) Pokud pracujeme s std::thread exportovanou hlavičkou thread:

A. Vlákno začne běžet po zavolání konstruktoru a metody call
B. Vlákno začne běžet bezprostředně po zavolání konstruktoru
C. Vlákno začne běžet po zavolání konstruktoru a metody join
D. Vlákno začne běžet po zavolání konstruktoru a metody run

16) Návratovou hodnotu vlákna mohu získat:

A. Při použití hlavičky thread, pomocí method get() třídy std::jthread
B. Při použití hlavičky future, pomocí metody get() třídy std::launch
C. Při použití hlavičky future, pomocí metody get() třídy std::future
D. Při použití hlavičky thread, pomocí metody get() třídy std::thread
E. Při použití hlavičky future, pomocí metody get() třídy std::async
F. Při použití hlavičky future, pomocí metody get() třídy std::thread

17) Uvažujme následující příklad:

```
omp_nest_lock_t countMutex;
struct CountMutexInit {
  CountMutexInit() { omp_init_nest_lock (&countMutex); }
  ~CountMutexInit() { omp_destroy_nest_lock(&countMutex);
};
struct CountMutexHold {
  CountMutexHold() { omp_set_nest_lock (&countMutex); }
  ~CountMutexHold() { omp_unset_nest_lock (&countMutex);
};
A();
CountMutexHold a;
B();
CountMutexInit b;
```

A. Kód nepovede k uváznutí, pokud dojde k neošetřené výjimce ve volání A(). Neošetřená výjimka ve volání B() to nemůže ovlivnit
B. Kód povede k uváznutí. Nezmění na tom nic ani neošetřená výjimka vyvolaná voláním v A(), ani neošetřená výjimka ve volání B()
C. Kód povede k uváznutí, pokud dojde k neošetřené výjimce ve volání A(). Neošetřená výjimka ve volání B() to nemůže ovlivnit.
D. Kód povede k uváznutí, pokud dojde k neošetřené výjimce ve volání B(). Neošetřená výjimka ve volání A() to nemůže ovlivnit.
E. Kód nepovede k uváznutí. Nezmění na tom nic ani neošetřená výjimka vyvolaná voláním v A(), ani neošetřená výjimka ve volání B()

18) V distribuovaném systému sestávajícím ze čtyř procesů běží algoritmus Bully pro volbu lídra, přičemž jako volební kritérium slouží identifikátor procesu (tj. Proces P4 má nejvyšší hodnotu volebního kritéria). V systému došlo k selhání dosavadního lídra P4 a proces P4 zůstává nadále nedostupný.

Předpokládejme, že selhání procesu P4 detekuje nejdříve proces P2, který následně spustí volbu lídra pomocí algoritmu Bully. Označte tvrzení týkající se následného průběhu algoritmu Bully:

A. P2 odešle zprávu ELECTION procesu **P1**
B. Ihned po přijetí zprávy ELECTION proces **P3** pošle procesu **P1** zprávu ELECTION
C. Po vypršení volebního time-out pošle proces **P3** procesu **P1** zprávu COORDINATOR(P3)
D. P2 odešle zprávu ELECTION procesu **P3**
E. Ihned po přijetí zprávy ELECTION proces **P3** pošle procesu **P4** zprávu ELECTION
F. Po vypršení volebního time-out pošle proces **P3** procesu **P2** zprávu OK

G. Po vypršení volebního time-out pošle proces **P3** procesu **P1** zprávu OK
H. Ihned po přijetí zprávy ELECTION proces **P3** pošle procesu **P1** zprávu OK
I. Po vypršení volebního time-out pošle proces **P3** procesu **P2** zprávu COORDINATOR(P3)
J. Po vypršení volebního time-out pošle proces **P3** procesu **P4** zprávu COORDINATOR(P3)
K. Ihned po přijetí zprávy ELECTION proces **P3** pošle procesu **P2** zprávu OK

19) (Pozn.: omylem jsem napsal čtyř procesů místo pěti. Tato otázka v testu nebyla. Ta, co byla, je níže jako otázka 20) V distribuovaném systému sestávajícím ze čtyř procesů běží algoritmus Bully pro volbu lídra. V systému došlo k selhání lídra, které bylo detekováno jedním z procesů a tento proces se chystá zahájit volbu lídra. Předpokládejme, že v systému od tohoto okamžiku nebude docházet k dalším selhání (procesu ani kanálu). Timeout T_{OK} označuje časový interval, po který proces po odeslání zprávy ELECTION čeká na zprávu OK předtím, než se prohlásí za kandidáta na lídra a odešle zprávu COORDINATOR. V systému není k dispozici nativní multicast.

Označte pravdivá tvrzení týkající se komunikační složitosti a latence následného průběhu algoritmu Bully:

A. Během volby lídra bude v *nejlepším* případě v systému odesláno **5** zpráv
B. Během volby lídra bude v *nejhorším* případě v systému odesláno **11** zpráv
C. Během volby lídra budou v *nejlepším* případě v systému odeslány **2** zprávy
D. Volba lídra může v nejhorším případě trvat **1** komunikační latenci + T_{OK}
E. Během volby lídra budou v *nejlepším* případě v systému odeslány **4** zprávy
F. Volba lídra může v *nejhorším* případě trvat **2** komunikační latence + T_{OK}
G. Během volby lídra bude v *nejhorším* případě v systému odesláno **31** zpráv
H. Volba lídra může v *nejhorším* případě trvat **3** komunikační latence + T_{OK}
I. Během volby lídra budou v *nejhorším* případě v systému odesláno **19** zpráv

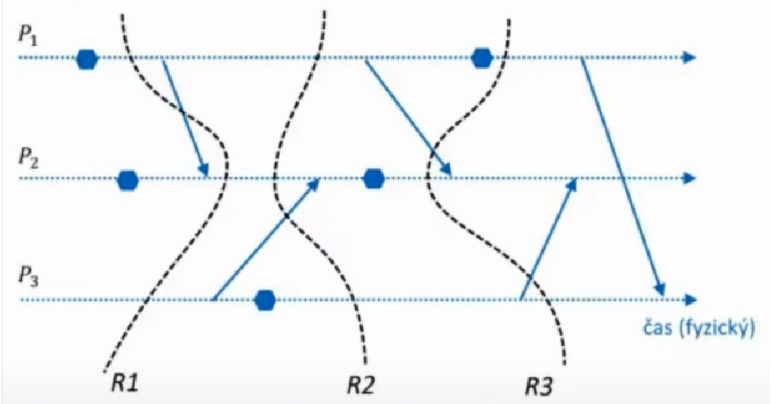
20) V distribuovaném systému sestávajícím z pěti procesů běží algoritmus Bully pro volbu lídra. V systému došlo k selhání lídra, které bylo detekováno jedním z procesů a tento proces se chystá zahájit volbu lídra. Předpokládejme, že v systému od tohoto okamžiku nebude docházet k dalším selhání (procesu ani kanálu). Timeout T_{OK} označuje časový interval, po který proces po odeslání zprávy ELECTION čeká na zprávu OK předtím, než se prohlásí za kandidáta na lídra a odešle zprávu COORDINATOR. V systému není k dispozici nativní multicast.

Označte pravdivá tvrzení týkající se komunikační složitosti a latence následného průběhu algoritmu Bully:

A. Během volby lídra bude v *nejlepším* případě v systému odesláno **5** zpráv
B. Během volby lídra budou v *nejhorším* případě v systému odesláno **11** zpráv
C. Během volby lídra budou v *nejlepším* případě v systému odeslány **2** zprávy
D. Volba lídra může v nejhorším případě trvat **1** komunikační latenci + T_{OK}
E. Během volby lídra budou v *nejlepším* případě v systému odeslány **4** zprávy
F. Volba lídra může v nejhorším případě trvat **2** komunikační latenci + T_{OK}
G. Během volby lídra bude v *nejhorším* případě v systému odesláno **31** zpráv
H. Volba lídra může v nejhorším případě trvat **3** komunikační latenci + T_{OK}

I. Během volby lídra budou v *nejhorším* případě v systému odesláno **19** zpráv

21) Na obrázku jsou zachyceny tři řezy distribuovaného výpočtu:



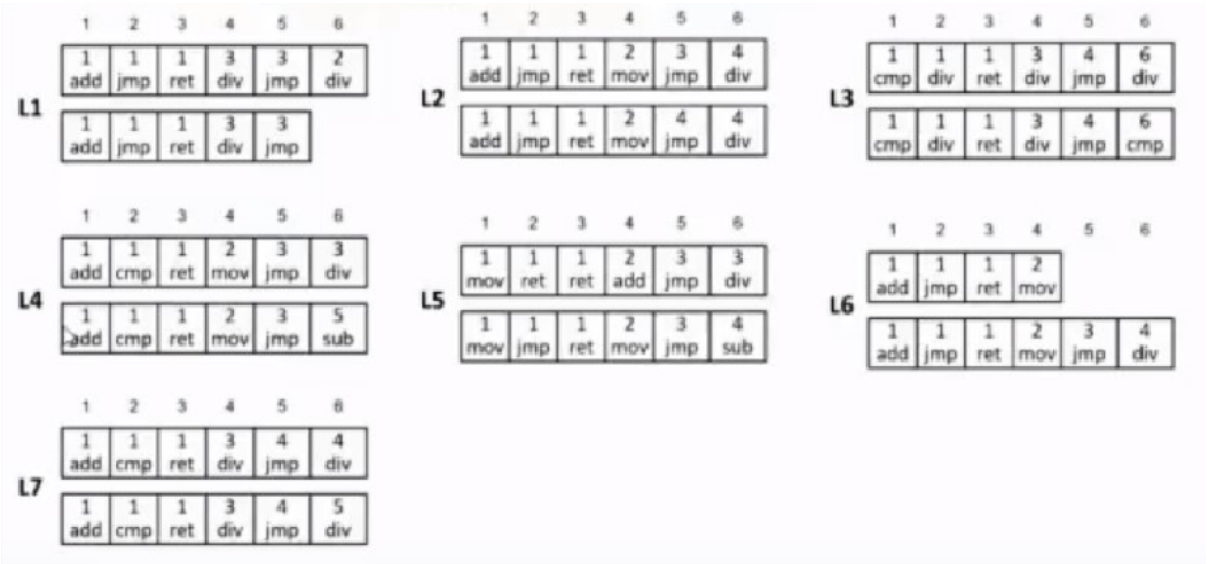
Předpokládejme, že externí pozorovatel má v jakémkoliv fyzickém časovém okamžiku okamžitý přístup ke stavu všech procesů a všech kanálů. Označte všechna pravdivá tvrzení:

A. Řez R1 je konzistentní řez
B. Řez R2 mohl být pozorován externím pozorovatelem
C. Řez R3 je konzistentní řez
D. Řez R2 je konzistentní řez
E. Řez R3 mohl být pozorován externím pozorovatelem
F. Řez R1 mohl být pozorován externím pozorovatelem

22) Označte pravdivá tvrzení týkající se možného využití Chandy-Lamportova algoritmu pro výpočet globálního snapshotu. (Pozn.: předpokládejte, že ve výpočtu nedochází k selháním procesů ani kanálů). Chandy-Lamportův algoritmus umožňuje:

A. Spolehlivě z vypočteného snapshotu identifikovat objekty, na které aktuálně v systému globálně neexistuje žádná reference (např. za účelem následné garbage collection)
B. Spolehlivě z vypočteného snapshotu detekovat, že ve výpočtu probíhá volba lídra
C. Spolehlivě z vypočteného snapshotu detekovat, že výpočet uváznul
D. Spolehlivě z vypočteného snapshotu detekovat, že v algoritmu RAFT mají procesy vzájemně nekonzistentní logy
E. Spolehlivě z vypočteného snapshotu detekovat, že výpočet skončil (ukončení výpočtu)
F. Spolehlivě z vypočteného snapshotu detekovat, že ve výpočtu čeká alespoň jeden proces na vstup do kritické sekce

23) Označte všechny dvojice logů, které se mohou vyskytnout v průběhu algoritmu RAFT.



A. L7
B. L6
C. L4
D. L1
E. L5
F. L2
G. L3

24) Skupina 4 procesů P1, P2, P3, P4 vykonává algoritmus Ricart-Agrawala pro vyloučení procesů. Žádný z procesů není aktuálně v kritické sekci (KS) a ani o vstup nepožádal a v přenosu není žádná zpráva. Uvažujme, že proces P1 právě vykonal operaci enter() pro vstup do KS. Předpokládejme, že doba přenosu zpráv je přesně 100ms, že procesy reagují na příchozí zprávy okamžitě a že v systému nejsou posílány žádné jiné zprávy než zprávy samotného algoritmu Ricart-Agrawala a že systém *nepodporuje* nativní multicast. Označte všechna pravdivá tvrzení:

A. P1 bude muset na vstup do KS od vyvolání operace enter() počkat **300** ms.
B. Od vyvolání operace enter() pro vstup do KS do ukončení operace exit() pro opuštění KS budou v algoritmu odeslány celkem **3** zprávy
C. P1 bude muset na vstup do KS od vyvolání operace enter() počkat **100** ms
D. Po vstupu do kritické sekce musí proces P1 odeslat všem ostatním procesům zprávu **HELD**.
E. Od vyvolání operace enter() pro vstup do KS do ukončení operace exit() pro opuštění KS budou v algoritmu odesláno celkem **6** zpráv
F. P1 bude muset na vstup do KS od vyvolání operace enter() počkat **200** ms
G. Před vstupem do kritické sekce musí proces P1 odeslat všem ostatním procesům zprávu **WANTED**.
H. Před vstupem do kritické sekce musí proces P1 odeslat všem ostatním procesům zprávu **REQUEST**.
I. Od vyvolání operace enter() pro vstup do KS do ukončení operace exit() pro opuštění KS budou v algoritmu odesláno celkem **9** zpráv

J. Před vstupem do kritické sekce musí proces P1 obdržet od všech ostatních procesů zprávu **OK**.
K. Po výstupu z kritické sekce musí proces P1 odeslat všem ostatním procesům zprávu **OK**.
L. Před vstupem do kritické sekce může proces P1 obdržet od některého z ostatních procesů zprávu **WAIT**.

25) Následující tvrzení se týkají úplnosti třech různých detektorů selhání běžících v distribuovaném systému sestávajícího z 10 procesů. Pro každý z detektorů definujme stupeň robustnosti jako maximální číslo N takové, že daný detektor zůstane úplný, pokud v systému neselže současně více než N libovolných procesů (a tedy že pokud selže N+1 procesů, tak úplnost již garantovat nelze)

Označte výroky, které označují správnou hodnotu stupně robustnosti pro každého z detektorů:

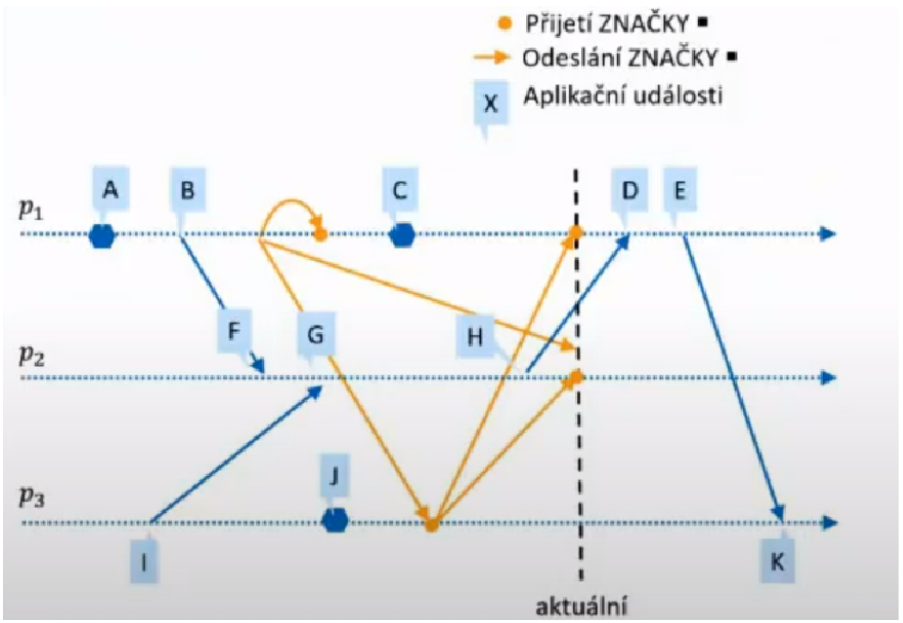
A. Stupeň robustnosti *oboustranného kruhového heartbeat* detektoru je **3**
B. Stupeň robustnosti *oboustranného kruhového heartbeat* detektoru je **1**
C. Stupeň robustnosti *SWIM* detektoru s K=3 je **1**
D. Stupeň robustnosti *oboustranného kruhového heartbeat* detektoru je **2**
E. Stupeň robustnosti *centralizovaného heartbeat* detektoru je **1**
F. Stupeň robustnosti *centralizovaného heartbeat* detektoru je **9**
G. Stupeň robustnosti *SWIM* detektoru s K=3 je **3**
H. Stupeň robustnosti *centralizovaného heartbeat* detektoru je **0**
I. Stupeň robustnosti *SWIM* detektoru s K=3 je **9**

26) Uvažujme příklad

```
#include <iostream>
#include <chrono>
#include <thread>
using namespace std::this_thread;
void A() {
  std::cout << "a";
  sleep_for(std::chrono::seconds(5));
  std::cout << "A";
}
void B() {
  std::cout << "b";
  sleep_for(std::chrono::seconds(1));
  std::cout << "B";
}
void C() {
  std::cout << "c";
  std::thread t(A);
  t.detach();
  std::thread u(B);
  u.join();
  std::cout << "C";
}
int main() {
  C();
  std::thread t(B);
  t.join();
  A();
}
```

A. První bude vypsáno písmeno c
B. První bude vypsáno písmeno B
C. První bude vypsáno písmeno a
D. První bude vypsáno písmeno b
E. První bude vypsáno písmeno A
F. První bude vypsáno písmeno C
G. První písmeno není možné určit

27) V distribuovaném systému byl spuštěn Chandy-Lamportův algoritmus pro výpočet globálního snapshotu. Aktuální stav běhu algoritmu je zachycen na obrázku níže:



Uvažujme akce, které jednotlivé procesy provedou bezprostředně jako další krok Chandy-Lamportova algoritmu (procesy mohou v dalším kroku provést i několik akcí najednou)

A. Proces **P1** spustí záznam příchozích zpráv na kanále **P1** -> **P3**
B. Proces **P1** zaznamenává svůj lokální stav
C. Proces **P1** odešle zprávu ZNAČKA procesu **P2**
D. Proces **P1** odešle zprávu ZNAČKA procesu **P3**
E. Proces **P1** spustí záznam příchozích zpráv na kanále **P3** -> **P1**
F. Proces **P2** odešle zprávu ZNAČKA procesu **P1**
G. Proces **P1** zastaví záznam příchozích zpráv na kanále **P2** -> **P1**
H. Proces **P2** zaznamenává svůj lokální stav
I. Proces **P1** zastaví záznam příchozích zpráv na kanále **P3** -> **P1**
J. Proces **P2** odešle zprávu ZNAČKA procesu **P3**

28) OpenMP je:

A. Knihovna libgomp
B. Organizace
C. Podfuk
D. Program
E. Překladač
F. Specifikace

29) Uvažujme dva kousky kódu:

```
int soucet;
void Inkrement() {
#pragma omp critical
  soucet += 1;
}
```

```
int soucet;
void Inkrement() {
#pragma omp atomic
  soucet += 1;
}
```

A. Chování obou příkladů bude stejné a nekorektní. Může dojít k problémům se souběhem (race condition)
B. Chování obou příkladů bude stejné a korektní. Nemůže dojít k problémům se souběhem (race condition)
C. Chování prvního příkladu bude nekorektní, zatímco ve druhém příkladu může dojít k problémům se souběhem (race condition)
D. Chování druhého příkladu bude nekorektní, zatímco v prvním příkladu může dojít k problémům se souběhem (race condition)

30) Pokud chceme v C++ pracovat s mutexy, které konstrukce umožní s nimi pracovat tak, aby případná výjimka v kódu nezpůsobila uváznutí?

A. std::lock_guard
B. std::unique_lock
C. std::mutex
D. Vlastní implementace schématu “Resource Acquisition Is Initialization” (RAII)

31) Co je špatně na následujícím kousku kódu (se standardními hlavičkami iostream, thread, vector):

```
const int thread_count = 10;
void Hello(long my_rank);
int main(int argc, char* argv[]) {
std::vector <std::jthread> threads;
for (int thread=0; thread < thread_count; thread++) {
     threads.push_back(std::jthread(Hello, thread));
}
std::cout << "Hello from the main threadn";
return 0;
}
```

A. Kód je celý v pořádku
B. Kód má používat standardní hlavičku jthread, nikoli thread
C. Kód nepoužije volání metody join, a tudíž neuvolní paměť
D. Vlákna nikdy nezačnou běžet, protože po konstruktoru nebude zavolána metoda run
E. Více vláken než počet jader procesorů může vést k uváznutí

32) Vyberte všechny správné odpovědi. Pokud pracujeme s objektem třídy std::thread exportovanou standardní hlavičkou thread, objekt je:

A. CopyConstructible
B. CopyAssignable
C. Nic z toho
D. MoveConstructible
E. TriviallyCopyConstructible

33) Uvažujme kód se standardními hlavičkami iostream, thread:

```
int i = 3;
void decrement() { i = i - 1; }
void print() {
int j;
do { j = i; } while (j == 0);
std::cout << j << std::endl;
}
int main() {
std::thread first(decrement);
std::thread second(decrement);
std::thread third(print);
first.join();
second.join();
third.join();
return 0;
}
```

A. Hodnota 3 nebo 2 nebo 1
B. Hodnota 3
C. Hodnota 2 nebo 1
D. Žádný výstup
E. Hodnota 0
F. Hodnota 3 nebo 2
G. Hodnota 2 nebo 1 nebo 0

34) Uvažujme následující kód

```
#pragma omp parallel  num_threads(thread_count)
    {
        method(1);
#pragma omp sections
        {
#pragma omp section
            {
                method(2);
                method(3);
            }
#pragma omp section
            { method(4); }
        }
    }
```

Která tvrzení jsou správně:

A. Není jasné, zda se provedou method(1) a method(4) souběžně
B. Není jasné zda se method(2) a method(3) provedou souběžně. Závisí to na thread_count
C. method(1) doběhne před spuštěním method(3)
D. Není jasné, kolikrát bude spuštěna method(2). Závisí to na thread_count
E. Není jasné, zda se provedou method(2) a method(4) souběžně. Závisí to na thread_count
F. Není jasné, kolikrát bude spuštěna method(1). Závisí to na thread_count
G. method(1) doběhne před spuštěním method(2)
H. method(1) bude spuštěna právě jednou a nezávisí to na thread_count
I. method(1) doběhne před spuštěním method(4)
J. method(2) bude spuštěna právě jednou a nezávisí to na thread_count
K. Není jasné, zda se provedou method(1) a method(2) souběžně

35) Uvažujme příklad s hlavičkami iostream, “omp.h”

```
void work(int n) { std::cout << n; }
void sub3(int n) {
    work(n);
#pragma omp barrier
    work(n);
}
void sub2(int k) {
#pragma omp parallel num_threads(2) shared(k)
    sub3(k);
}
void sub1(int n) {
    int i;
#pragma omp parallel num_threads(2) private(i) shared(n)
    {
#pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}
int main() {
    sub1(1); sub2(1); sub3(1); return 0;
}
```

Která tvrzení jsou správně:

A. Výstup může být 010111
B. Výstup může být 001111
C. Výstup může být 0101
D. Výstup může být 0011
E. Výstup může být 00111111
F. Výstup může být 01011111

36) Uvažujme příklad s hlavičkami iostream, “omp.h”

```
void work(int n) { std::cout << n; }
void sub3(int n) {
    work(n);
#pragma omp barrier
    work(n);
}
void sub2(int k) {
#pragma omp parallel num_threads(2) shared(k)
    sub3(k);
}
void sub1(int n) {
    int i;
#pragma omp parallel num_threads(2) private(i) shared(n)
    {
#pragma omp for
        for (i=0; i < n; i++)
            sub2(i);
    }
}
int main() {
    sub1(2); sub2(2); sub3(2); return 0;
}
```

Která tvrzení jsou správně:

A. Výstup může být 11002222
B. Výstup může být 10101010222222 (s povoleným vnořením)
C. Výstup může být 1010222222
D. Výstup může být 1100222222
E. Výstup může být 110022222222
F. Výstup může být 11110000222222 (s povoleným vnořením)
G. Výstup může být 10102222

37) Třída std::mutex je definována v hlavičce:

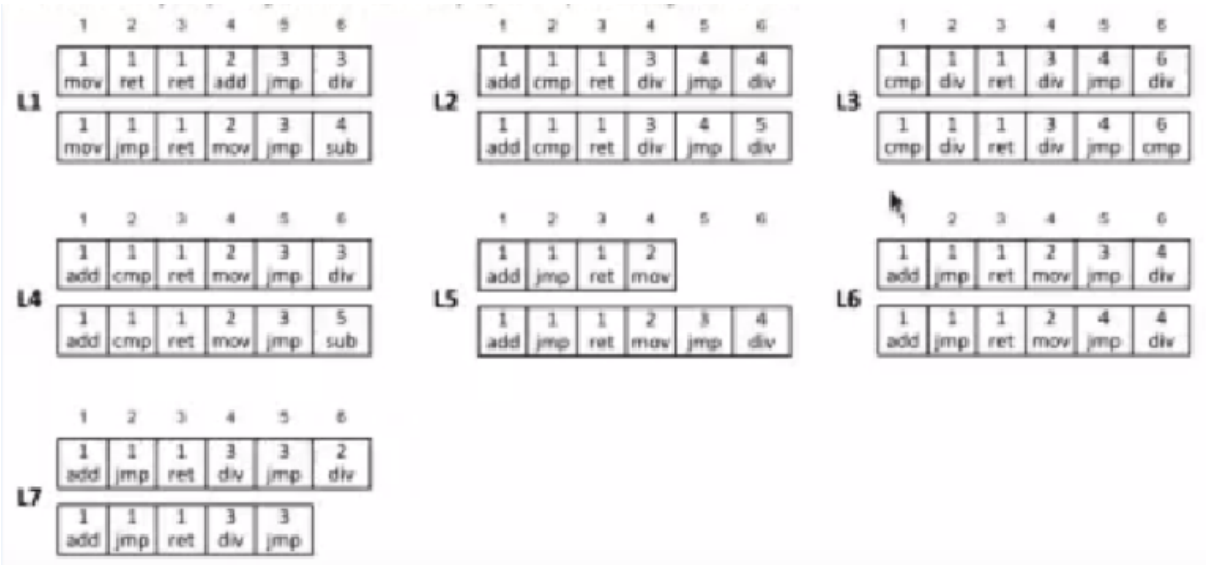
A. chrono
B. mutex
C. thread

38) Uvažujme použití hlavičky pthread.h a deklaraci pthread_t *thread_handles. Jak pokračovat, abychom správně inicializovali hread_handles:

A. pthread_attr_init(&thread_handles[0]);
B. thread_handles = pthread_create(NULL, Hello, (void *) thread);
C. thread_handles = (pthread_t*)malloc(thread_count * sizeof(pthread_t));
D. pthread_join(thread_handles(0), NULL);
E. pthread_create(&thread_handles(0), NULL);

F. (*thread_handles);

39) Označte všechny dvojice logů, které se mohou vyskytnout v průběhu algoritmu RAFT



A. L1
B. L2
C. L7
D. L4
E. L3
F. L6
G. L5

40) Následující tvrzení se týkají úplnosti třech různých detektorů selhání běžících v distribuovaném systému sestávajícího z 5 procesů. Pro každý z detektorů definujme stupeň robustnosti jako maximální číslo N takové, že daný detektor zůstane úplný, pokud v systému neselže současně více než N libovolných procesů (a tedy že pokud selže N+1 procesů, tak úplnost již garantovat nelze)

Označte výroky, které označují správnou hodnotu stupně robustnosti pro každého z detektorů:

A. Stupeň robustnosti *jednostranného kruhového heartbeat* detektoru je **0**
B. Stupeň robustnosti *jednostranného kruhového heartbeat* detektoru je **2**
C. Stupeň robustnosti *SWIM* detektoru s K=2 je **1**
D. Stupeň robustnosti *jednostranného kruhového heartbeat* detektoru je **1**
E. Stupeň robustnosti *centralizovaného heartbeat* detektoru je **1**
F. Stupeň robustnosti *centralizovaného heartbeat* detektoru je **4**
G. Stupeň robustnosti *SWIM* detektoru s K=2 je **4**
H. Stupeň robustnosti *centralizovaného heartbeat* detektoru je **0**
I. Stupeň robustnosti *SWIM* detektoru s K=2 je **2**

41) Uvažujme algoritmus RAFT pro distribuovaný konsenzus. Označte všechna pravdivá tvrzení týkající se průběhu algoritmu RAFT:

A. V každé epoše je v clusteru **právě jeden** proces ve stavu lídr
B. V každém fyzickém okamžiku je v clusteru **maximálně jeden** server ve stavu lídr
C. V každé epoše je v clusteru **maximálně jeden** proces ve stavu lídr
D. V každém fyzickém okamžiku je v clusteru **právě jeden** server ve stavu lídr

Další otázky co jsem našel na FEL discordu:

```
#pragma omp parallel
#pragma omp for
for(int i = 0 ; i < size ; i++) {
  if(is_solution(candidates[i])) {
    std::cout << candidates[i]
              << "is a solution" << std::endl;
    break;
  }
}
```

- Nepůjde pravděpodobně zkompilovat
- Paralelní blok skončí po nalezení prvního řešení
- Paralelní blok skončí, až všechna vlákna najdou řešení
- Aby blok skončil ihned po nalezení řešení, musíme (vhodně) doplnit `#pragma omp cancel for`
- Aby blok skončil ihned po nalezení řešení, musíme (vhodně) doplnit `#pragma omp cancelation point for`
- Měli bychom nastavit proměnnou prostředí `OMP_CANCELLATION=true`

Správné odpovědi:

- 1, 3-6