Debugging

- Čistě sériový kód se dá odladit snadno, protože posloupnost vykonávání instrukcí je pokaždé stejná
- U paralelních výpočtů to neplatí
    - Odladit netriviální, navíc je-li optimalizovaný, paralelní výpočet je jedna z nejtěžších věcí

Testování

- I když vypadá kód bez chyby, stejně ho nechte testovat víc než dostatečně dlouho za různých podmínek

Sériový kód

- Stále je třeba odladit sériový kód, resp. kód vlákna, zda např. provádí správně např. I/O operace
    - Jde o kontrolu té části výpočtu, kdy nedochází k interakci s ostatními vlákny, ani se nepřistupuje ke sdíleným proměnným
- Lze s ním odladit pouze omezenou množinu všech možných chyb paralelního programu
    - Většinou jde o chyby, které je vidět přímo ze zdrojového kódu

Kontrolní výpisy

- Výsledek paralelního programu s chybnou synchronizací závisí i na tom, jak rychle poběží jeho vlákna
    - o Krokování v debuggeru IDE způsobuje velké prodlevy

    - Může tak docházet k dodatečné synchronizaci, jejímž následkem se chyba neprojeví
  - Kontrolní výpis sice neposkytne tolik informací jako použití debuggeru, ale zase je rychlost vykonávání vláken bližší jejich běžné rychlosti
    - Kontrolní výpis také způsobuje dodatečnou synchronizaci, která může zabránit chybě, aby se projevila
  - Fce volající jádro způsobují dodatečnou synchronizaci
    - Např. v kontrolním výpisu GetThreadID

Randseed

- Rychlost programu, tj. i možné potlačení chyby, ovlivní
  - Debug verze vs. release verze
    - Release verze by minimálně díky optimalizaci měla běžet rychleji
    - V debug verzi může být něco od překladače, co způsobí dodatečnou synchronizaci

  - Použitý hardware
    - Pentium Pro vs. fce Delay Borland Pascalu
      - CRT unit počítal, kolikrát zvládne procesor vykonat smyčku za jeden tik hodin -55ms
      - Pentium Pro od 200MHz výš jich zvládlo více 65536
      - Tj. 16-bitové dělení přeteklo, a následně se to projevilo jako chyba dělení nulou

  - Zátěž systému

        - V systému může běžet několik aplikací, uzamykatelné zdroje nemusí být vždy stejně dostupné, využití systémových zdrojů se mění
            - viz zátěžové testy simulující nedostatek systémových zdrojů
- Je možné do debug verze programu přidat náhodné zpoždění – sleep(rand)
    - A na začátku inicializovat proměnnou randseed, tak aby se případná chyba dala zopakovat
        - Pokud jí ovšem způsobují náhodné prodlevy, opakované ve stejném pořadí

Lamportovy hodiny

- Vymyšlené pro distribuované systémy
    - Pro sdílenou paměť budeme vlákno považovat za proces distribuované aplikace

- Časové značky, které umožňují částečné uspořádání událostí
    - Ale neumožňují implikovat závislost
- Časová značka je monotónně inkrementující se counter
    - Jeden na proces/vlákno
    - integer

- Vlastnosti
    - Proces/vlákno inkrementuje counter před každou událostí
    - Kdykoliv proces pošle zprávu, vlákno provede interakci, přiřadí se k tomu časová značka – counter
    - Proces přijímající zprávu, druhý účastník interakce, nastaví svůj vlastní counter na maximum z přijatého

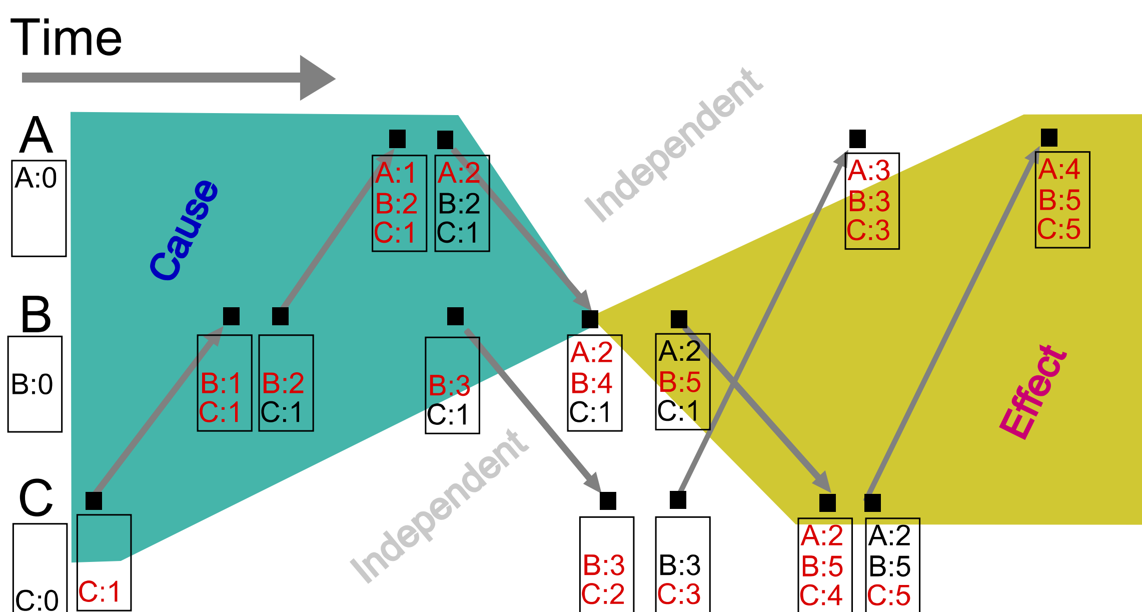
counteru a svého counteru, než došlo k přijetí cizího counteru

- Synchronizuje svoje hodiny s vysílajícím procesem, iniciátorem interakce

- Nechť *a* a *b* jsou události
- *C(x)* je čas události *x*
- Pak platí
    - Pokud *a* způsobilo *b*, pak *C(a)* < *C(b)*

Vektorové hodiny

- Vektorové hodiny umožňují detekci porušení příčinnosti mezi událostmi v distribuovaném systému
- Lamport umí detekci pro dva procesy



http://upload.wikimedia.org/wikipedia/commons/5/55/Vector_Clock.svg

- Máme vektor Lamportových hodin
    - Hodin všech procesů/vláken
    - Proces si drží nejmenší potřebnou kopii vektoru
    - Všechny procesy/vlákna vždy inkrementují o stejné, konstantní množství – 1

- Postup
    - Vynulovat všechny hodiny
    - Proces/vlákno inkrementuje svoje hodiny při vnitřní události
        - Např. když se chystá odeslat zprávu, ve které bude odesílaný vektor s jeho hodinami již inkrementovanými
    - Kdykoliv proces přijme zprávu, vlákno se dostane do interakce, inkrementuje svoje logické hodiny a updatuje zbývající, známé hodiny ostatních procesů/vláken na maxima z toho, co zná, a z toho, co přijal

- Nechť je *VC(x)* časový vektor události x
- Pak pro události *a* a *b* platí, že *VC(a) < VC(b)* tehdy, když:
    - for i:=0 to vectorsize-1: VC(a)[i]<=VC(b)[i]
    - a pro alespoň jedno *i* platí, že VC(a)[i]<VC(b)[i]

- Pak opět platí
    - Pokud *a* způsobilo *b*, pak *VC(a) < VC(b)*

Další možnosti

- Maticové hodiny
    - chronologická a příčinná souvislost událostí
    - zobecnění vektorových hodin

- Vektor verzí – verzování replik dat

- Co se začalo odvíjet od Lamportových hodin, je spíše vhodné pro studium doktoranda - ale je dobré to znát i tak

Příklad využití Lamportových hodin u vláken

- Rozhraní jsou nadefinována analogicky ke konceptu IAdviseSink z Windows COM

```
type
  PEventMedium = ^TEventMedium;
  TEventMedium = record
    Clock:integer;   //Lamportovy hodin
    Sender:TSenderID;
    Code:integer;    //kód zprávy
    WParam,          //parametry
    LParam:integer;
  end;

  IEventAdviseSink = interface(IUnknown)
  ['{FBCA3BDD-9E2A-4AA7
     -961E-742336109CD0}']
     procedure OnEvent(const
            event:PEventMedium); stdcall;
  end;

  IEventAdviseHolder =interface(IUnknown)
  ['{33675443-A096-40B2
     -B076-1DCD5475DAEF}']
    function Advise(
             const Sink:IEventAdviseSink;
               out Connection:integer
              ):HRESULT; stdcall;
    function Unadvise(const
    Connection:integer):HRESULT; stdcall;
    function SendOnEvent(const Code,
                   WParam, LParam:integer
             ):HRESULT; stdcall;
  end;
```

Pozor na to, že vše není chyba synchronizace vláken

- Pokud se zdá, že algoritmus je napsaný správně, ale přesto to nefunguje, nemusí být chyba v synchronizaci vláken, ale v inicializaci proměnných a dat

- Překladače se obvykle chovají různě, když produkují debug verzi a release verzi výsledného kódu

- Debug verze obvykle inicializuje proměnné a přidělenou paměť na nuly
    - Snadná detekce nil pointerů
    - Snadnější detekce dělení nulou
    - Při přičítání se jakoby nic neděje – čísla se nemění
    - Při násobení se jakoby nic nepočítá – výsledkem je 0

- Release verze nemusí nic inicializovat
    - Proč do proměnné přiřazovat nulu, když se vzápětí bezpodmínečně přepíše něčím jiným?

- Může se stát, že přidělená paměť už obsahuje něco, s čím program zdánlivě běží správně
    - Např. lokální proměnné podprogramů v oblasti již dříve použitého zásobníku
    - Vlastní paměťový manažer s recyklací použitých paměťových bloků, ale bez jejich inicializace

    - Ale na jiném počítači, nebo spuštění programu mimo IDE, už v takovém případě může mít za následek jiné chování programu

- Ze stejného důvodu je třeba kontrolovat návratové hodnoty synchronizačních funkcí!

Cizí knihovny

- Pokud se kód chová divně bez zjevné příčiny, může být chyba i v knihovně, kterou používáte
  - Např. projekt zkompilovaný s maximální optimalizací, díky které se projevila v něm obsažená chyba
    - Pak se totiž může stát, že se s jedním překladačem chyba objeví, a s druhým ne

- Nicméně, úměrně k počtu lidí, kteří danou knihovnu používají, je mnohem víc pravděpodobnější, že je chyba ve vašem kódu
  - Např. danou knihovnu špatně používáte

```
#include <concrt.h>
class CRWLock {
private:
   Concurrency::reader_writer_lock
                                mcLock;
public:
  void AcquireRead(){mcLock.lock_read();}
  void AcquireWrite(){mcLock.lock();   }
  void Release()     {mcLock. unlock();}
};
```

  - Concurrency Runtime by MS & Intel
  - Lepší chování než SRW Lock WinAPI
    - Jenže potřebujete mít plánovač a asociovat s ním vlákna
      - Jinak memory leaks a možné vyjímky
      - Jenomže concrt může vytvořit implicitní plánovač, a kód tak může **zdánlivě** běžet OK za určitých podmínek

Specifické triky

- Např. pro MS VC a WinDbg lze pro diagnostické účely pojmenovat vlákna přes výjimku
  - Specifická záležitost, původně nedokumentovaná
    - Posléze dokumentovaná k MS VS 2003

```
#if defined(_MSC_VER) && defined(_DEBUG)

typedef struct tagTHREADNAME_INFO {
  DWORD dwType;      //must be 0x1000
  LPCSTR szName;     //pointer to name
  DWORD dwThreadID; //thread ID
  DWORD dwFlags;     //must be zero
} THREADNAME_INFO;

void SetThreadName( DWORD dwThreadID,
LPCSTR szThreadName) {
  THREADNAME_INFO info;
  {
    info.dwType = 0x1000;
    info.szName = szThreadName;
    info.dwThreadID = dwThreadID;
    info.dwFlags = 0;
  }
  __try  {
    RaiseException( 0x406D1388, 0,
                sizeof(info)/sizeof(DWORD),
                (DWORD*)&info );
  }
 __except EXCEPTION_CONTINUE_EXECUTION){}
}
#endif
```

- V kombinaci MS VC 2010 (.dll) a Delphi 2005 (.exe)
    - Memory leaks
    - Neočekávaná vyjímka, která se někdy tvářila
        - jako chyba výpočtu,
        - jindy jako chyba synchronizace
        - v dalších případech jako chyba pointerové aritmetiky
        - a většinu času to běželo, jako kdyby tam žádná chyba nebyla

    - Podle debuggeru Delphi byla vyjímka na straně VC
    - Podle debuggeru VC byla vyjímka na straně Delphi
        - A pak to odlaďte, když se k tomu nikdo nezná…

- Hrozí takové nebezpečí u Javy a JVM?
    - V prvé řadě jde o zodpovědnost programátora, že vůbec něco takového použije
        - Musí vědět, co dělá – v uvedeném příkladu má možnost výběru, jestli to použije, nebo ne

    - Kdyby se JVM spoléhal pouze na bytecode, tak by program běžel neúnosně pomalu, a proto se JVM snaží nahradit známé knihovní funkce implementacemi napsanými v assembleru, nebo jiném, pointerovém jazyku
        - Tj. splněn první krok
        - Druhým krokem je spustit kód, který nevytvořil tým JVM, ani žádný nástroj JVM

- JNA
  - Java Native Access
  - Přístup k DLL a Shared Objects bez JNI

```
//BOOL WINAPI EnumWindows(
//   __in  WNDENUMPROC lpEnumFunc,
//   __in  LPARAM lParam);

public interface User32
               extends StdCallLibrary {
  interface WNDENUMPROC
              extends StdCallCallback {
    boolean callback(Pointer hWnd,
                     Pointer arg); }
  boolean EnumWindows(WNDENUMPROC
               lpEnumFunc, Pointer arg);
}
User32 user32 = User32.INSTANCE;

user32.EnumWindows(new WNDENUMPROC() {
  int count;
  public boolean callback(Pointer hWnd,
                     Pointer userData) {
        System.out.println(
              "Found window " + hWnd + ",
              total " + ++count);
        return true;    }
}, null);
                 https://jna.dev.java.net/
```

- Navzdory původní myšlence, jen reference na objekty a monitory (synchronized), dnes už i v Javě máte možnost vytvořit chyby „specifických triků“
- Je to o to záludnější, že Java na to nebyla navržená