Profiling (jen poznámky z pohledu PPR)

- Staticky
  - Analýza programového kódu, co by se kde dalo naprogramovat lépe, aniž by se program spustil
    - Relevantní je právě znalost, jak psát efektivní programový kód
    - A jak funguje systém, pro který se programový kód píše

- Dynamicky
  - Až je hotová statická analýza, provede se dynamická
  - Za běhu programu se měří, který kód se vykonává nečastěji
    - Je pak předmětem dalších optimalizací
      - Jsou-li ještě nějaké možné

    - Anebo je předmětem optimalizace nadřazený programový kód, aby zbytečně často nevolal pomalý programový kód

Přepínání kontextu

- Paralelní programy využívají synchronizační primitiva
- Některá jsou implementována jádrem operačního systému
  - Tj. volání musí přepnout z uživatelského režimu do režimu jádra
    - A to je drahá operace v porovnání s voláním podprogramu ve stejném adresovém prostoru

- TryEnterCriticalSection
  - Test se může odehrát v uživatelském adresovém prostoru

- EnterCriticalSection
  - Přepnutí do režimu jádra
    - Pokud jsou vlákna implementována s podporou operačního systému

- Interlocked
  - Atomické instrukce
  - Pokud přistupujeme do vlastní paměti, vše se odehraje v adresovém prostoru volajícího
  - Nejkratší doba synchronizace

- Read/Write Lock
  - Pozor na to, že implementace může používat spinlock při čekání na AcquireWrite z volání AcquireRead
    - Pozor u zamykání časově náročných operací!

Žádoucí stav

- V nejlepším případě běží všechny procesory na plný výkon (v závislosti na počtu vláken a jader procesorů)
- Nejfrekventovanější úseky kódu
  - Obsahují minimum skoků, zejména podmíněných
  - Pokud to má smysl, používají vektorové instrukce
  - Neprovádějí alokace a dealokace
  - Správná granularita synchronizace vláken
    - Co to vlastně je, viz další přednášky
  - Volací konvence __fastcall, register, ABI

Počet vláken

- Čím více vláken, tím větší režie
  - Čas spotřebovaný plánovačem
  - Paměť na struktury
- Se vzrůstajícím počtem vláken roste pravděpodobnost chyby

- Je rozdíl mezi vláknem plánovaným operačním systémem (thread), a vláknem z uživatelského prostoru (fiber)
  - Thread má větší režii než fiber

- Jsou programátoři, kteří se domnívají, že např. napsat webový server způsobem jedno vlákno na jedno TCP connection je v pořádku – i když jich je např. >10000
  - Není to v pořádku, takový kód je špatný kód
  - Autor přednášek si nemohl nevšimnout, že se s tímto přesvědčením setkal u Javistů

  - Z pohledu OS, thread Javy není nutně thread, ale může to být fiber – viz následující přednášky
    - Záleží na JVM, programátor to neovlivní
  - Efektivní postup je mít pool několika málo vláken (korelujících s počtem CPU), které budou obsluhovat jednotlivé TCP spojení
    - TCP spojení odpovídá nějaká struktura
    - Java thready jsou takové řešení, ale **jen dokud** je JVM vytváří jako fibers

- => efektivní kód přepíná jen mezi počtem threads, který koreluje s počtem CPU (vlákna se nevytváří „na parádu“)
- => neví-li programátor, že je vlákno fibre, má ho za thread

Ukazatele na úrovni CPU ohledně paralelizace vykonávání instrukcí

- Aneb jak poznat, zda sériový kód běží dostatečně efektivně
  - Paralelizovaný pomalý kód sice může běžet rychleji, než sériový pomalý kód, ale pořád je to špatně!

- Ukazatele se zjistí např. s Intel VTune Amplifier
  - Pro AMD je CodeAnalyst

- CPI aka Cycles Per Instruction
  - Instrukce prochází několika fázemi od načítání, fetch, až po zápis výsledku, write-back
  - Bez pipeline nemůže počet cyklů menší než počet fází
  - Na uniprocesoru, když všechny fáze instrukcí poběží paralelně, může být CPI=1
  - Ovšem když např. superskalární procesor zvládne najednou dvě instrukce, pak může být CPI=0,5

- LLC Miss aka Last Level Cache Miss
  - Poměr cyklů během LLC Miss vůči všem cyklům
  - Příčina může být práce s příliš velkým blokem paměti, případně jsou data rozházená po příliš velkém úseku paměti

- Instruction Starvation
  - Procesor čeká na instrukci, kterou by mohl vykonat
  - Poměr pročekaných cyklů na instrukci ke všem cyklům
  - Může být následek práce s příliš rozsáhlým blokem paměti, špatné predikce skoků, nebo příliš velkého množství virtuálních funkcí

- Branch Misprediction
  - Poměr špatně předpovězených skoků ke všem skokům
  - Skoky se dají buď eliminovat, nebo předělat
  - Pozor, že eliminace skoků za každou cenu může vést ke zpomalení kódu

- Dále viz např.:
  - http://software.intel.com/sites/products/documentation/doclib/iss/2013/amplifier/lin/ug_docs/GUID-C541FEEF-4F10-48B7-87DD-10DC26E1FE36.htm

Java Profiling

- Více profilerů, můžete pokaždé dát jiné výsledky
- Předpokládejme, že profiler má za úkol pro JVM identifikovat tzv. „hot“ metody, které je třeba přeložit do nativního kódu
    - V praxi se však může stát následující:
        - Profiler ignoruje nativní metody, takže ve výsledku se bude neustále zmenšovat množina metod v bytecode a zvětšovat množina metod v nativním kódu
            - Což sice možná ve výsledku urychlí program, pozor na náklady na překlad do nativního kódu, ale na druhou stranu profiler dojde k chybnému závěru, že všechny metody jsou „hot“ – což nejsou
        - K vzorkování dochází v průběhu tzv. safe-pointů (viz dále přednáška o Javě)
            - Safe-pointy si ovšem JVM vkládá do kódu dynamicky podle potřeby
                - Takže se některé metody vůbec nemusí vzorkovat
        - T. Mytkowicz, A. Diwan, M. Hauswirth, P.F. Sweeney, „Evaluating the Accuracy of Java Profilers“, Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation

- Takže jako workaround lze měřit celkovou dobu běhu a ukazatele na úrovni CPU
    - Jenže pak dost dobře nerozlišíte JVM od vašeho programu