Eerste Toets Concurrency

Donderdag 22 december 2016, 8.30 – 10.30, Educ-Γ.

Motiveer je antwoorden *kort*! Zet je mobiel uit. Stel geen vragen over deze toets; als je een vraag niet duidelijk vindt, schrijf dan op hoe je de vraag interpreteert en beantwoord de vraag zoals je hem begrijpt. **Cijfer:** Maak vragen 1 en 2 op pagina 1, 3 en 4 op pagina 2, en vraag 5 op pagina 3. Vragen 1, 2 en 4 zijn 3pt, vragen 3 en 5 zijn 2pt. Resultaat is totaal plus 1, gedeeld door 1,3 (max 10).

1. **Het Peterson Lock:** Het verkrijgen van een lock (de `lock()`-methode voor thread `i`) gaat bij Petersons algoritme zo:

   ```
   1.  flag[i] = T;
   2.  vic = i;
   3.  while (flag[j] && vic == i) { }
   ```

   (a) Hoe luidt de `unlock()`?
   (b) Hoe luidt de **No deadlock** eis op een lock?
   (c) Waarom is het onmogelijk dat een thread die het lock wil, oneindig vaak door de andere thread wordt ingehaald? Is er een maximum op het aantal keren dat hij wordt ingehaald?

   Oplossing: (a) `flag[i] = False;` (terugzetten van `vic` is niet nodig).
   (b) Als een of meer threads het lock willen (dus in de `lock` methode zitten) en het lock is vrij, dan krijgt eentje in eindige tijd het lock.
   (c) Als je rekent vanaf het beginnen aan `lock()` is er geen maximum. Zolang thread i nog bezig is met het zetten van `flag[i]`, kan de ander die nog als `False` lezen en willekeurig vaak de kritieke sectie in en uit gaan. Maar oneindig vaak kan niet, omdat de thread (dankzij fairness) na een poosje klaar is met stap 1, het zetten van `flag[i]`. Noem de tijd waarop stap 1 klaar is t_1. Op t_1 kan de andere thread al bezig zijn met de `lock()` en wellicht ook doorgaan. Maar vanaf dat moment, tot na de kritieke sectie van i, blijft `flag[i] True`. Dus thread j kan, als hij na t_1 aan `lock()` begint, alleen nog door als hij `vic` op `i` ziet staan, nadat hij hem zelf op j heeft gezet. Dat kan eenmaal gebeuren doordat thread i stap 2 uitvoert, maar daarna niet meer. Conclusie: de andere thread kan maximaal 1x binnenkomen en kritiek worden nadat `i` stap 1 afheeft en voordat `i` kritiek wordt.

   Beoordeling/Toelichting: Tot 3pt, 1 per deelvraag. Beoordelingscodes:

   D = "Na `unlock` van de ander kan hij Door" is niet genoeg! Hetzelfde geldt bv. ook voor het TaS-lock, dat juist om Starvation bekend staat. Je hebt hier wel echt nodig dat je ook beargumenteert dat een thread die unlockt en weer wil locken, niet kan voorgaan, dus zal blijven wachten.
   E = Dat het van de Eis moet, wil nog niet zeggen dat het in dit algoritme ook gebeurt.
   F = Dat een thread het lock niet oneindig lang mag houden volgt uit de Fairness eigenschap, maar is geen eis op het lock.
   J = Je unlockt met `flag[J]`, -1/2.
   M = De deadlock eis gaat ook over een aanvraag door Meerdere threads.
   O = Threads gaan *niet* per sé Om en om.
   S = Dit is niet de **no deadlock** maar de **no Starvation** eis.
   V = Unlockt met Vic, onjuist, 0pt.
   W = "Niet op elkaar Wachten" is nog geen **no deadlock**; zeg wat er *wel* moet gebeuren! De eis gaat ook over wat er moet gebeuren als maar en thread het lock vraagt.
   X = Dit is niet de **no deadlock** maar de **mutual eXclusion** eis.

2. **Valentie in Consensus-objecten:** De executieboom van een consensus-object bevat configuraties en beschrijft welke configuraties elkaar kunnen opvolgen in een executie.

   (a) Waarom heeft een configuratie doorgaans meerdere opvolgers?
   (b) Wanneer is een configuratie *univalent*?
   (c) Kunnen van een bivalente configuratie *alle* opvolgers univalent zijn? Kunnen van een bivalente configuratie *alle* opvolgers 1-valent zijn?

   Oplossing: (a) Vanwege het non-determinisme in de executie: het is onbepaald, van welke thread de instructie als volgende effect heeft, en elke thread leidt tot een andere opvolgende configuratie.
   (b) Een configuratie C is univalent als alle executiepaden onder C dezelfde uitkomst hebben.
   (c) Ja, je kunt zelfs bewijzen dat er zeker een bivalente configuratie is met alleen 0- en 1-valente kinderen (kritieke configuratie). Nee, als een configuratie alleen 1-valente opvolgers heeft, is hij zelf ook 1-valent.

   Beoordeling/Toelichting: Tot 3pt, eentje per deelvraag. Beoordelingscodes:
   C = Conditionals geven *geen* vertakking in de executieboom! Als programmeur denk je wel dat een conditional (**if**) tot verschillende vervolgstappen kan leiden, maar welke dat is wordt helemaal door de toestand van dat moment bepaald.
   K = 'Definieert' univalentie met verwijzing naar de Kind-nodes. Dat zegt helaas nog niet wat het is, max 1/2.
   M = Zonder Motivatie max 1/2 voor c.
   N = Omdat er Nog geen return is geweest, heeft een bivalente configuratie *altijd* n opvolgers.
   V = De Volgorde van operaties binnen een thread ligt vast.

3. **Worker Threads:** Worker threads bieden betere performance en meer controle dan de generieke mechanismen van de Parallel Task Library.

   (a) Beschrijf wat "work stealing" is.
   (b) Hoe kunnen worker threads het gebruik van caches verbeteren?

   Oplossing: (a) Worker threads eisen taken op totdat alle taken uitgevoerd zijn. Het aantal threads wordt hierbij niet aangepast aan het aantal taken, maar aan de beschikbare hardwarebronnen.
   (b) De worker threads kunnen ieder aan een vaste core toegewezen worden. Hiermee voorkomen we dat gecachete data in een volgende time slice opnieuw opgehaald moet worden.

   Beoordeling/Toelichting: Voor elk correct antwoord een punt.
   Bij (a) is niet correct:
   Een thread die klaar is steelt werk van andere threads.
   Wanneer een thread het werk van een andere thread doet.
   Een thread die van de ene core naar de andere springt.
   Bij (b) is half goed:
   Context switching alleen uitleggen voor hyperthreading.

4. **Thread Parallellisme:** Windows en Linux zijn voorbeelden van preemptive operating systems. Oude versies van Windows maakten gebruik van cooperative multitasking.

   (a) Wat is cooperative multitasking?
   (b) Wat is een context switch?
   (c) Wat is een logische core?

   Oplossing: (a) Taken zijn zelf verantwoordelijk voor het teruggeven van de controle aan het OS.
   (b) Het wisselen van taken op een core. De state van de taak die actief was wordt opgeslagen, en de state van de te herstarten taak wordt hersteld.
   (c) Een hyperthreaded core doet zich naar het OS voor als twee logische cores. Het delen van de hardwarebronnen en het toewijzen van timeslices aan threads die het OS op deze logische cores plaatst gebeurt door de CPU zelf.

   Beoordeling/Toelichting: Voor elk correct antwoord een punt.
   Bij (a) is niet goed:
   Threads communiceren onderling over multitasking.
   Bij (b) is niet goed:
   Het overzetten van een thread naar een andere core.

5. **SIMD:** Voor het efficient uitvoeren van code met behulp van SIMD instructies is het een goed idee om de gebruikte data om te zetten van AoS naar SoA.

   (a) Wat is het verschil tussen AoS en SoA?
   (b) Waarom is de SoA structuur beter voor SIMD code?

   Oplossing: (a) AoS: Array of Structures; in het geheugen liggen de objecten achter elkaar waarbij de data van een enkel object bij elkaar blijft. SoA: Structure of Arrays; hierbij worden de objecten zodanig verweven dat individuele floats of ints van alle objecten in ononderbroken arrays liggen.
   (b) SIMD instructies hebben steeds 4 of 8 data nodig in een 128bit of 256bit register. De SoA maakt het mogelijk deze bits zonder conversie uit het geheugen te lezen.

   Beoordeling/Toelichting: Voor elk correct antwoord een punt.
   Bij (a) is niet correct:
   SoA is structure of arrays, AoS is array of structures (opschrijven wat de afkortingen betekenen is niet hetzelfde als het verschil tussen de twee uitleggen).
   Een SoA is een enkel object met daarin de data voor heel veel objecten.
   Bij (a) is half goed:
   'Ieder dataveld wordt een array' (niet goed, want: een 'dataveld' of property kan zelf ook weer een object zijn; het gaat hier om 'primitiven', bijv. float of int.