Eerste Hertoets Concurrency

24 maart 2016, 11.00–13.00, Unnik-220.

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 vraag 1 en 2 op pagina 1, 3 en 4 op pagina 2, en 5 en 6 op pagina 3. Vraag 1, 5 en 6 zijn 3pt, vragen 3 en 4 elk 2pt, vraag 2 is 4pt. Cijfer T1 is totaal plus 1, gedeeld door 1,7.

1. **Amdahl and friends:** Een programma besteedt 5 seconden aan werk dat niet parallel uitgevoerd kan worden en 10 seconden aan werk dat wel parallel uitgevoerd kan worden.

   (a) Wat is volgens Amdahl's Law de maximaal haalbare versnelling van dit programma op een quadcore CPU? Laat hyperthreading buiten beschouwing.
   (b) Waarom kan dit volgens Gustafson en Barsis toch beter?
   (c) Welke informatie is nodig om met het work-span model een betere inschatting te maken van de te verwachten prestatie van een parallelle variant van het programma?

   Oplossing: (a) $Sp \leq 1/(f + (1-f)/P)$. In dit geval: $f = 5/(5+10) = 1/3$; $Sp \leq 2$.
   (b) Bij toenemende processorkracht is te verwachten dat de grootte van de parallelle taak toenoemt, terwijl de seriele code meestal constant blijft.
   (c) We hebben informatie over afhankelijkheden tussen deeltaken nodig een de span te kunnen berekenen.

   Beoordeling/Toelichting: Voor elk correct antwoord een punt. Bij (a): ook correct als $Sp = 2$ ipv kleiner of gelijk aan.

2. **Hardware:** (a) Wat is de relatie tussen speculative execution en branch prediction?
   (b) Wat is false sharing, en hoe beïnvloedt dit de performance van een parallel programma?

   Oplossing: (a) Branch prediction wordt door de processor gebruikt om te voorspellen hoe execution flow verder gaat na een conditional jump, zodat de pipeline speculatief gevuld wordt met instructies. Speculative execution voert de conditionele code onafhankelijk van de voorwaarde uit. Hierdoor kunnen conditie en aftakking(en) parallel behandeld worden. We ruilen hier de kosten van uitvoering van irrelevante code tegen de kosten van een foute voorspelling.
   (b) False sharing treedt op wanneer meerdere cores data binnen dezelfde cache line aanpassen. Aangezien niet alle cores dezelfde L1 cache hebben (alleen voor twee hyperthreaded cores is dit het geval) moeten de L1 caches met elkaar gesynchroniseerd worden, wat een tijdrovend proces is.

   Beoordeling/Toelichting: (a) Voor een correcte beschrijving van speculative execution en branch prediction: elk een punt. Relatie volgt vanzelf uit correcte beschrijving van beide concepten. (b) Voor een correcte uitleg van false sharing twee punten.

3. **Vectorizatie:** Twee vormen van paralellisme zijn instruction level paralellism en thread level paralellism.

   (a) Wat is het verschil tussen beide vormen?
   (b) Wat is het verschil tussen een thread en een stream (ook wel lane)?

   Oplossing: (a) Threads worden onafhankelijk van elkaar uitgevoerd, al dan niet op verschillende cores. Instruction level paralellism kan in een enkele thread worden uitgevoerd; de threads (streams?) lopen dan in lock-step en moeten hetzelfde algoritme uitvoeren, eventueel met masking (speculative execution) om verschillen in flow af te handelen. Heel kort: threads hebben elk een eigen program counter, streams niet.
   (b) Threads hebben elk een eigen program counter, streams niet. Streams worden in lock-step uitgevoerd.

   Beoordeling/Toelichting: Voor elk correct antwoord een punt.

4. **Fair executie:** In dit programma kijkt thread 1 steeds naar `w`, die door thread 2 afwisselend op `true` en `false` wordt gezet. Ga uit van read/write atomicity; initieel is `w=s=True` en `t=0`:

```
Thread 1:          Thread 2:
  while (w)          while (s)
  { t = t + 1 }        { w = !w }
  s = False           print t
```

Beschrijf een oneindige executie die mogelijk is onder een fair scheduler.

Oplossing: Een oneindige executie: Omdat thread 1 termineert als hij `w==False` ziet, kan thread 2 niet alleen thread 1 blokkeren maar ook die blokkade weer opheffen, nl. door `w` weer op `true` te zetten. Je moet dus kijken naar executies, waar thread 2 telkens een even aantal wijzigingen schrijft tussen lees-operaties van thread 1. Omdat fairness alleen iets zegt over het voorkomen van operaties, maar niet over de volgorde (of situatie waarin het gebeurt), sluit fairness dit niet uit.

Beoordeling/Toelichting: Max 2pt (inc uitleg).

A = Een bit Altijd true is niet hetzelfde als bit steeds true wanneer hij gelezen wordt.
B = "Beide threads doen iets" is niet voldoende voor Fairness, ze moeten oneindig vaak iets doen.
F = Geen verklaring waarom dit Fair is.

5. **LockOne:** Hier staan de lock en unlock van de LockOne klasse (voor thread i).

```
public void lock()            public void unlock()
{ flag[i] = true;                  { flag[i] = false ; }
  while (flag[j]) {}  }
```

(a) Aan welke drie eisen moet een lock implementatie voldoen?
(b) Welke van deze eisen is/zijn voor LockOne niet voldaan? Waarom?
(c) Als je de twee regels in `lock` verwisselt, is dan het probleem opgelost?

Oplossing: (a) Een lock moet voldoen aan (1) **Mutual Exclusion**: hoogstens 1 thread heeft het lock (2) **No Deadlock**: Als threads het lock willen krijgen, en het is vrij, moet eentje het krijgen (3) **No Starvation**: elke thread die het lock vraagt, zal het ooit krijgen.
(b) LockOne voldoet aan **mutual exclusion** omdat een thread eerst zijn flag op true zet, en dan nog de andere op false moet zien voordat hij kritiek wordt (principe van Safe Sluice). LockOne voldoet niet aan **No deadlock**: als beide threads de `lock` methode beginnen, kunnen ze beide elkaar blokkeren en is er deadlock. De deadlock impliceert ook starvation; een tread kan overigens niet willekeurig vaak worden ingehaald.
(c) Nee, dan krijg je de code van LockZero die geen **mutual exclusion** kent.

Beoordeling/Toelichting: Voor a, b en c elk 1pt. Beoordelingscodes:

A = Bij (b) wel iets over deadlock, maar Andere eisen niet genoemd; -1/2.
N = Bij (a) worden de Namen genoemd maar geen omschrijving van de eisen; 1/2pt.
S = Zegt NoStarvation bij (b). Omdat NoStarvation *sterker* is dan NoDeadlock, is schending van NoStarvation zwakker dan schending van NoDeadlock, daarom is dit een minder sterk antwoord; 0,5pt.
W = Bij (b) wordt wel de geschonden eis genoemd, maar niet Wanneer die geschonden wordt (scenario); 1/2pt.
Z = Ziet bij c de schending van ME over het hoofd, max 1/2.

6. **Multivalued register:** Je kunt een m-waardig *regular* register maken uit een array van m regular bits. De Writer schrijft waarde x door bit x op 1 te zetten en de lagere bits op 0, de reader zoekt vanaf positie 0 naar de eerste 1:

```
Write(x):                    Read:
  r[x].write(1)                for(i=0; i<m; i++)
  for (i=x-1; i>=0; i--)         if (r[i].read == 1) return i;
    r[i].write(0)              // A
```

(a) Laat zien dat de Reader kan falen wanneer de Writer ook de hogere bits op 0 zet (met een loopje `for (i=x+1; i<m; i++) r[i].write(0)`).
(b) Is het gebouwde register ook atomic? Leg uit!
(c) Bobs compiler weigert de `Read` code wegens executiepaden zonder `return`. Daarom wil Bob als default op plek `A` een `return ...` zetten; met welke waarde achter de `return` is de methode correct?

Oplossing: (a) Stel dat de Writer eerst Write(4) doet en dan Write(2). Na de Write(4) zijn alle bits voor positie 4 op 0, neem aan dat de Reader begint en de 0len leest tot positie 3. De Writer voert zijn complete Write(2) uit, daarna staan alle bits voorbij positie 2 op 0. De Reader gaat verder en leest alleen maar 0len.
(b) Het is niet atomic, een inversie is mogelijk, omdat de gebruikte bits zelf alleen maar regular zijn. (Scenario hiervoor: stel de waarde is 2 maar de Writer doet een Write(3). De Writer begint met r[3] op 1 te zetten, en gaat dan r[2] op 0 zetten met `r[2].write(0)`. Tijdens deze write kan de Reader een volledige Read afwerken, waarbij de read op r[2], die met de r[2].write(0) overlapt, de nieuwe waarde al geeft zodat de Read resultaat 3 geeft. Daarna kan de Reader weer een volledige Read afwerken, waarbij de r[2].read, die weer met de r[2].write(0) overlapt, de oude waarde nog teruggeeft zodat deze Read 2 oplevert.
We *vermoeden* dat de constructie *wel* een atomair register zou opleveren als de gebruikte bits atomair zijn. Als dat klopt, kun je een scenario alleen maar geven door niet-atomair gedrag van bits erin te bouwen.)
(c) Met elke waarde, omdat deze return statement nooit wordt uitgevoerd.

Beoordeling/Toelichting: Voor a, b en c elk 1pt. Beoordelingscodes:

A = Motiveert een nee bij (b) met een Atomair scenario; 0pt. *Bijvoorbeeld:* een "inversie" bij twee overlappende reads, dit kan niet want als twee Reads overlappen kan nooit sprake zijn van inversie.
B = Beredeneert bij (b) dat het atomair is, maar vanuit Atomaire bits. Slim, maar de gegeven bits zijn slechts regular; 1pt.
D = Noemt alleen Definitie van atomic maar bewijst er niets over, 0pt.
L = Beweert dat een Latere writewaarde kan worden geretourneerd. Kan niet, zou ook niet regular zijn; 0pt.
M = Beredeneert bij (b) incorrectheid met een MultiWriter scenario, terwijl deze constructie alleen bedoeld is voor SingleWriter; 0pt.
R = Beredeneert Regular ipv atomair, 0pt.
S = Zegt bij (b) nee en noemt inversie, maar zonder Scenario te geven; 1pt.
T = Beweert bij (b) dat het niet atomair is omdat Timestamps nodig zijn; 1/2pt.
W = Voert bij (a) een scenario op waarin het fout gaat met twee Writers; zie M, 1pt.
Z = Bij (c) zijn natuurlijk oneindig veel goede antwoorden, maar voor vol punt moet je Zien dat het niet uitmaakt.