Synchronisatie en sequencing.

Toen de eerste operating systems gemaakt moesten worden, werd men met problemen uit twee toen nieuwe gebieden geconfronteerd. Het feit, dat men toen tegelijkertijd met twee soorten van "nieuwigheid" te maken kreeg heeft er waarschijnlijk in niet geringe mate toe bijgedragen, dat de opgave een goed operating system te maken toen als bijzonder moeilijk ervaren werd, zo moeilijk dat de opgave nog steeds een bijzonder aureool heeft: zelfs in 1974 worden er nog symposia aan gewijd!

De eerste klasse problemen vallen onder wat bekend is geworden onder de naam "scheduling". Als men een groot aantal programma's door een grote machine wil laten uitvoeren, programma's waarvan de uitvoering beslag legt op verschillende gedeelten van de installatie, dan wil men een "rooster" maken, dat zonder te grote vertragingen in de uitvoering van individuele programma's alle onderdelen van de installatie zo goed mogelijk bezighoudt. Dit verlangen, dat een duidelijk economische motivering heeft, is voor de bouw van operating systems de primaire aanleiding geweest.

Ietwat simplificerend kan men hier twee scholen onderscheiden. De eerste school stelt zich op het standpunt, dat we over het quantitatieve gedrag van programma's allerlei gegevens hebben, zodat het "rooster" betrekkelijk lang van te voren en tamelijk rigide kan worden opgesteld. Terwijl de rigiditeit van het rooster de individuele verwerkingssnelheid der programma's ten goede kan komen, is de prijs die hiervoor betaald moet worden, dat de programmeur de quantitatieve gegevens over het toekomstig gedrag moet opgeven --en als hij dat niet goed kan, moet hij er maar een slag naar slaan-- terwijl het operating system al deze gegevens verwerken moet. De tweede school probeert het rooster zo flexibel mogelijk te houden, de van te voren op te geven karakterisering minimaal te houden en baseert zich op de verwachting dat het gedrag van een programma een zekere continuiteit zal vertonen en dat het gedrag in de nabije toekomst wel zal lijken op dat in het recente verleden. Het is als een KNMI dat voor morgen in principe hetzelfde weer als vandaag voorspelt.

Gepréoccupeerd met de quantitatieve gevolgen van allerlei strategieen zijn velen verrast door de soms te late ontdekking, dat er bij dit soort toekenningsstrategieen zich naast quantitatieve problemen ook zuiver logische problemen voordoen. Wij noemen het verschijnsel van "de dodelijke omarming" --later als "deadlock" bekend geworden--: in een poging om momentaan de installatie zo effectief mogelijk te gebruiken, kan het zich in een situatie manoevreren, waarin twee of meer processen cyclisch op elkaar gaan staan wachten. Voorts noemen we de "After you, after you"-blokkade, waarin de beslissing wat van twee elkaar in de tijd uitsluitende gebeurtenissen eerst zal gebeuren ten onrechte oneindig lang kan worden uitgesteld. En tenslotte het probleem van de "individuele uithongering", wanneer een programma nooit --of: onacceptabel lang niet-- aan bod komt. Mits tijdig onderkend hoeven deze problemen geen onoverkomelijke moeilijkheden op te werpen.

De tweede klasse problemen ontstond door de zg.ingreep --die later onder de naam "interrupt" bekend is geworden--, waardoor randapparaten op onvoorspelbare en niet reproduceerbare momenten voltooing van autonoom uitgevoerde handelingen aan het centrale rekenorgaan meldden. Van de vertrouwde sequentiele en volledig deterministische automaat moest men overschakelen op de non-deterministische machine, die zich in bepaalde aspecten irreproduceerbaar zou gedragen. Vooral dit laatste eiste een grote omschakeling, niet in de laatste plaats, omdat er in die tijd nog nauwelijks twijfel werd geuit over het geloof

dat men een programma moest "testen" om het goed te krijgen en te geloven, dat het goed was. Het was de irreproduceerbaarheid van de machine, samen met de centrale plaats, die operating systems innemen, die de vraag naar een correctheidsbewijs onontkoombaar pousseerde.

De eerste stap op deze lange weg is geweest om bijna alle activiteiten van het operating system onder te brengen in een aantal "sequentiele processen". Ik zei "bijna alle activiteiten": de overschakeling van de central processor van het ene sequentiele proces naar het andere valt er nl. buiten. Deze sequentiele processen zijn "zwak gekoppeld", dwz. behoudens duidelijk geiolsserde punten, waar ze elkaar kunnen beinvloeden, zijn ze verder onafhankelijk. Het voordeel van deze beschouwingswijze was tweeerlei: in de eerste plaats was nu een groot gedeelte van de activiteit weer interpretabel als de activiteit van een sequentieel proces, verder kon het non-determinisme van de installatie als geheel gevat worden in het postulaat, dat van de onderlinge snelheidsverhouding der individuele sequentiele processen "niets bekend was". Het organiseren van de correcte samenwerking van deze sequentiele processen was nu een duidelijk gelocaliseerde verplichting van een heel duidelijk type: het werd een synchronisatieopgave. Het was in de synchronisatie, dat het non-determinisme van de installatie grotendeels geabsorbeerd diende te worden, opdat er een bestuurbaar systeem zou ontstaan.

De synchronisatieopgave bleek twee kanten te hebben, een functionele en een implementatie-technische kant. Functioneel bleek de synchronisatie-opgave geformuleerd te kunnen worden als "synchroniseer de processen onderling zodanig, dat een bepaalde relatie invariant blijft". Als bijvoorbeeld twee processen onderling gekoppeld moeten worden in een "producent-consument" relatie via een eindige buffer, dan luidt de relatie, waarvan de invariantie gegarandeerd moet worden zoiets als

$$R: \qquad 0 \leq n \leq N$$

waarbij "n" het aantal porties in de buffer is en "N" de capaciteit van de buffer, uitgedrukt in porties. De producent --tengevolge van wiens voortgang "n:= n + 1" kan moeten worden uitgevoerd-- en de consument --tengevolge van wiens voortgang "n:= n - 1" plaats kan moeten vinden-- kunnen tijdelijk tegengehouden moeten worden (nl. als $n = N$, respectievelijk $n = 0$).

Hoe dit synchronisatieprobleem implementatietechnisch opgelost wordt, hangt ernstig af van de onderlinge communicatiemogelijkheden tussen de sequentiele processen. Een systematische oplossing bleek mogelijk onder de voorwaarde, dat de processen exclusief toegang kon worden gegeven tot enig gemeenschappelijk geheugen. Op ieder moment, dat een proces mogelijk de invariantie kan verstoren, moet eerst het stukje gemeenschappelijk geheugen geinspecteerd worden. In dit gemeenschappelijke geheugen wordt nl. de huidige toestand van het hele systeem --voorzover relevant-- bijgehouden. De inspectie kan dan vaststellen, of de critieke handeling nu wel of niet kan plaatsvinden: zo ja, dan wordt de handeling uitgevoerd, zo nee, dan gaat het proces in questie "slapen". In beide gevallen wordt er in het gemeenschappelijk geheugen voldoende spoor achtergelaten, voordat het exclusief toegankelijke geheugen weer wordt vrijgegeven en weer toegankelijk wordt voor de andere processen. De structuur van dit protocol heeft inmiddels zijn weerslag gevonden in diverse voorstellen voor "system implementation languages" en hun implementeerbaarheid is een goede test voor de adequaatheid van gekozen hardware specificaties.

De functionele specificatie van de synchronisatie-eis in de vorm van het invariant houden van een relatie zoals R heeft het voordeel, dat de synchronisatievoorwaarden formeel kunnen worden afgeleid. Zij

$$wp(S, R)$$

de "weakest pre-condition" voor de statement S, zodat activering van S tot resultaat zal hebben, dat na afloop aan de post-conditie R voldaan is. In ons voorbeeld zijn we geinteresseerd voor de producent in

$$wp("n:= n + 1", 0 \leq n \leq N)$$

en voor de consument in

$$wp("n:= n - 1", 0 \leq n \leq N) \quad .$$

Volgens het axioma van assignment reduceren deze twee weakest preconditions zich tot

$$0 \leq (n + 1) \leq N \quad \text{of} \quad -1 \leq n \ \underline{\text{and}} \ n < N$$

respectievelijk

$$0 \leq (n - 1) \leq N \quad \text{of} \quad 0 < n \ \underline{\text{and}} \ n \leq N+1 \quad .$$

Het is evenwel niet de taak om de waarheid van R te creeren, het is slechts de taak, om de waarheid van R niet te verstoren. Om de daartoe nodig en voldoende bijvoorwaarde --de zg. zynchronisatievoorwaarde-- te vinden mogen we uit de gevonden weakest pre-conditions in de conjunctie alle termen weglaten, die door R worden geimpliceerd; de synchronisatievoorwaarden vereenvoudigen zich dan tot

$$n < N$$

respectievelijk

$$0 < n \quad .$$

M.a.w. --in dit simpele geval niet verrassend!-- de producent mag iets in de buffer stoppen, mits deze aanvankelijk niet vol is, de consument mag iets uit de buffer halen, mits deze aanvankelijk niet leeg is. Bij ingewikkeldere invariante relaties --en in de praktijk zijn ze aanzienlijk ingewikkelder-- is deze techniek van "berekening" van de synchronisatievoorwaarden een belangrijk hulpmiddel bij de programmacompositie, waarvan bij ontwerp van multiprogrammeringssystemen dankbaar gebruik gemaakt kan worden.

De indeling van activiteiten in sequentiele processen, die vervolgens op een of andere manier gesynchroniseerd moeten worden, is echter in hoge mate willekeurig, zo willekeurig, dat wij de techniek ook kunnen gebruiken ... voor de compositie van sequentiele programma's!

Laten wij eens het bekende programma voor de algorithme van Euclides bekijken, dat voor $A > 0$ en $B > 0$ de grootste gemene deler GGD(A, B) uitrekent:

```
a:= A; b:= B;
while a ≠ b do if a > b then a:= a - b else b:= b - a fi od;
print((a + b)/ 2)
```

(Omdat na afloop van de loop a en b aan elkaar gelijk zijn, hebben we om redenen van symmetrie het gemiddelde uitgeprint.) De invariant van deze loop is

$$R: \quad GGD(A, B) = GGD(a, b) \ \underline{\text{and}} \ a > 0 \ \underline{\text{and}} \ b > 0 \quad .$$

Een andere manier om tegen dit programma aan te kijken is de volgende. Kennelijk hebben we twee cyclische procesjes, die ik maar even beschrijf als

```
do a:= a - b od        en        do b:= b - a od
```

en na de initialisering van a en b moeten deze tween cyclische processen zo gesynchroniseerd worden, dat relatie R invariant blijft! Als wij, net als bij de producent/consument relatie, de weakest preconditions uitrekenen en vervolgens deze vereenvoudigen door uit de conjunctie alle termen, die door R al geimpliceerd worden, weg te laten, vinden we de synchronisatiecondities

$$a > b \quad \text{respectievelijk} \quad b > a \quad .$$

En we kunnen ons programmaatje nu als volgt opschrijven:

```
a:= A; b:= B;
do a > b → a:= a - b []
   b > a → b:= b - a
od;
print((a + b)/2)  .
```

Tussen do en od staan een of meer alternatieven --in ons voorbeeld twee--, onderling gescheiden door "[]". Elk alternatief bestaat uit een boolean expressie (waarin we de synchronisatievoorwaarde herkennen) gevolgd door een horizontaal pijltje " → " als separator en een statement (of algemener: een lijst van statements, onderling door een puntkomma gescheiden). De door "[]" gescheiden alternatieven zijn semantisch ongeordend, steeds bepaalt de waarde van de boolean expressie of het bijbehorende alternatief voor executie in aanmerking komt, de hele constructie eindigt, als geen van de alternatieven meer voor executie in aanmerking komt. Beeindiging van de loop --in uniprogrammering een doorgaans nastrevenswaard ideaal-- betekent dus, dat alle synchronisatievoorwaarden false zijn: uit multiprogrammering is die situatie maar al te bekend: hij heet ... deadlock! Hier zien we, dat wat in uniprogrammering nagestreefd wordt, bij de bouw van operating systems juist angstvallig vermeden wordt: deadlock en termination zijn twee verschillende namen voor hetzelfde probleem!

In ons voorbeeld sluiten de twee boolean voorwaarden elkaar uit; als resultaat valt er nooit te kiezen en is ons programma dus volledig deterministisch. Als we in geval van meer alternatieven boolean voorwaarden hebben, die elkaar niet uitsluiten, wordt de keuze in het midden gelaten en hebben we een mogelijk non-deterministisch programma. De ervaring lijkt uit te wijzen, dat de mogelijkheid op deze wijze non-deterministische programma's te schrijven, een bijzonder welkome uitbreiding van ons arsenaal uitdrukkingsmogelijkheden is. Doordat het ons in staat stelt irrelevante beslissingen niet vast te leggen kunnen vaak programma's geschreven worden, waarin oorspronkelijke symmetrie van de probleemstelling in de structuur van het programma volledig behouden blijft.

Het belangrijkste lijkt echter, dat dezelfde technieken, die ons in multiprogrammering in staat stelden, de synchronisatievoorwaarden te berekenen, ons bij uniprogrammering nu in staat stellen, de sequencing conditions te bepalen.