Internal Report 2010–08 February 2011

Universiteit Leiden
Opleiding Informatica

Multithreading

voor

iedereen

Jaron Viëtor

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Multithreading voor iedereen

Jaron Viëtor

Februari 2011

Samenvatting

Multithreading op systemen met meer dan één processor bracht de belofte om programma's sneller te laten werken met zich mee, maar zo'n configuratie blijkt veel lastiger te gebruiken te zijn dan verwacht. Deze scriptie beschrijft de problemen die multithreading met zich meebrengt en mogelijke oplossingen, introduceert de theorie voor een methode die werkt op zowel uni- als multiprocessormachines en sluit af met een beschrijving van een algemene library die multithreading toegankelijk maakt voor de gemiddelde programmeur. Deze library wordt tenslotte als voorbeeld uitgewerkt in C++ en gebruikt voor een viertal voorbeeldprogramma's.

1 Probleemstelling

De laatste 40 jaar was het mogelijk om bijna elk jaar het aantal transistoren op een chip te verdubbelen (ook wel bekend als Moore's law, genoemd naar Gordon Moore die dit in 1965 opmerkte [10]), maar sinds het begin van het nieuwe millennium lopen we tegen problemen aan. Het grootste probleem is dat de hitte die chips produceren een harde grens voor de kloksnelheid blijkt te zijn [25]. Om dit op te lossen maken fabrikanten tegenwoordig multi-core processors: in plaats van één hele snelle processor, worden er twee of meer minder snelle gebruikt. Zo wordt de geproduceerde hitte verdeeld over meer oppervlakte. Als bijkomend voordeel hebben deze nieuwe multi-core computers de mogelijkheid dat meer dan één taak tegelijkertijd uit kan worden gevoerd. Als voorbeeld kan je denken aan een webserver waarbij er volledig onafhankelijk van elkaar één processor bezig is met data doorsturen aan alle verbonden bezoekers en een andere processor constant wacht op nieuwe verbindingen van bezoekers en deze verbindingen klaar zet om data te gaan ontvangen van de andere processor.
Om dit gedrag te implementeren zou je simpelweg twee programma's kunnen schrijven waarbij één programma de data doorstuurt en een tweede programma binnenkomende verbindingen accepteert en daarna aan het andere programma doorgeeft. Dit vereist dus

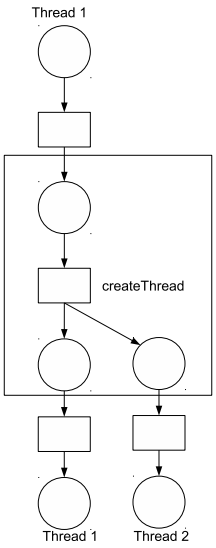
communicatie tussen deze programma's. De beschikbare opties voor deze communicatie zijn helaas nogal omslachtig en langzaam [29]. Een veel efficiëntere techniek is om slechts één programma te hebben dat zichzelf opsplitst en het werk over de processoren verdeelt. Dit wordt ook wel multithreading genoemd, en een gesplitst stuk programma heet een thread [20]. Deze threads delen hun computergeheugen met elkaar. Dit wil onder andere zeggen dat geheugen dat door één van de threads wordt gebruikt ook lees- en schrijfbaar is voor alle andere threads van datzelfde programma. Hierdoor kunnen de threads heel efficiënt gegevens met elkaar uitwisselen. Helaas brengt dit ook enkele moeilijkheden met zich mee: stel je voor dat er een nieuwe verbinding binnenkomt bij onze webserver precies op het moment dat de thread die de data verstuurt een volgende bezoeker wil gaan afhandelen. Het is best mogelijk dat de andere thread het nummer van deze verbinding nog aan het wegschrijven was en de data-thread het nummer half-geschreven uitleest. Dan is er dus een verkeerd nummer uitgelezen, en zal het programma fouten beginnen te maken.
Zo'n probleem heet een synchronisatieprobleem [20]. Synchronisatieproblemen zijn een groot struikelblok voor programmeurs die applicaties willen schrijven die gebruik kunnen maken van de voordelen van multithreading. Daarom zijn er volop kant-en-klare oplossingen te vinden die hiermee om kunnen gaan in de vorm van libraries. Een library is een algemeen stuk code dat simpel zonder aanpassingen te hoeven maken toe te voegen is als onderdeel van een programma. Eén van de meest gebruikte oplossingen voor de synchronisatie van verschillende threads is toegang tot gedeelde stukken geheugen slechts aan één thread tegelijk verlenen. Terugkerend naar het webserver voorbeeld zal er dan als er net een nieuwe verbinding binnenkomt, terwijl de data-thread de volgende verbinding wil gaan behandelen, moeten worden gewacht tot de nieuwe verbinding volledig is afgehandeld door de nieuwe-verbinding-afhandel-thread. Als de tweede thread bij dit stuk geheugen wil, moet deze dus wachten totdat de eerste thread klaar is en kan hij daarna pas verder gaan.
Dit wachten, ook wel "blocking" genoemd [20], is een erg nare eigenschap van programma's. Van een programma dat kan blokkeren is het namelijk erg lastig uitspraken te doen over of dit programma zal termineren binnen een bepaalde tijd of misschien wel "eeuwig" vast blijft zitten [23]. Een oplossing is het gebruiken van een methode die synchronisatie bewerkstelligt zonder dat er gewacht moet worden, waardoor er efficiënter met resources kan worden omgegaan. Zo kan de data-thread bijvoorbeeld de nieuwe verbinding overslaan totdat deze helemaal klaar voor gebruik is en hoeven de andere verbindingen dus niet te wachten. Het exacte verloop van het programma is dan wellicht niet meer zoals oorspronkelijk ontworpen, maar het resultaat van de uitgevoerde taken blijft gelijk aan de blocking variant [24]. Alle synchronisatiemethodes (en zelfs alle algoritmes[1] [24]) zijn

[1]Met een algoritme bedoel ik een methode om een probleem op te lossen die kan worden uitgedrukt als een (werkend) computerprogramma

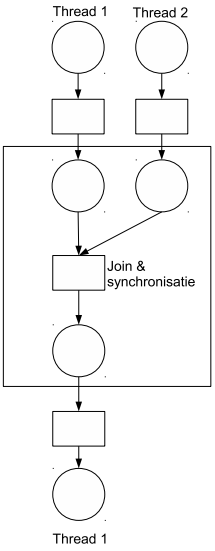
namelijk “non-blocking” te implementeren maar toch worden de blocking varianten nog het meest gebruikt: door een algoritme blocking te implementeren hoef je namelijk niet na te denken over (mogelijke) synchronisatieproblemen en ga je dus een boel problemen uit de weg.
Bijna alle moderne processoren (zie ook “Hardware-ondersteuning”) hebben speciale hard-ware-ondersteuning die het implementeren van synchronisatiemethodes bijna triviaal maken. Zulke ondersteuning wordt dan ook volop gebruikt. Er zijn echter processoren (vooral oudere processoren, maar ook moderne zoals de PA-RISC processor [12]) die deze hardware-ondersteuning missen. Bovendien is er geen gestandaardiseerde manier om deze hardware-ondersteuning aan te spreken (in C++ dan, nieuwere high-level talen vaak wel) en moet er gebruik worden gemaakt van compiler-specifieke instructies.
Er bestaan ook synchronisatiemethodes die deze hardware-ondersteuning niet nodig hebben zoals het Bakery Algorithm [20], maar alle threading libraries die ik kon vinden (zie ook “Libraries (praktijk)”) bieden alleen ondersteuning voor synchronisatiemethodes die hardware-ondersteuning vereisen. Een programma dat geschreven is om te profiteren van multithreading kan dus niet betrouwbaar worden gebruikt op processoren zonder hardware-ondersteuning. Zelfs als deze ondersteuning wel aanwezig is, zijn er aparte instructies nodig voor alle verschillende compilers.
Ik ben gaan uitzoeken wat er nodig is om een non-blocking oplossing voor synchronisatie te kunnen waarmaken die bovendien algemeen is voor alle soorten hardware. Tijdens dit onderzoek is er een idee ontstaan voor een set gereedschappen om op een soepele manier met de moeilijkheden van multithreading om te gaan, in plaats van een enkelvoudige kant–en–klaar oplossing. Dit idee heb ik uitgewerkt in C++, samen met een aantal simpele programma’s die de correcte werking en het gebruik van deze code demonstreren.

2 Wat zijn threads nou precies?

Zoals eerder vermeld is een thread een “afgesplitst” stuk programma. Dit splitsen gebeurt door middel van een programma-instructie die het besturingssysteem opdracht geeft de splitsing uit te voeren. Let op: het betreft hier **niet** de zogenaamde “fork” instructie — deze splitst ook het programma maar maakt een identieke kopie van het actieve proces, dus een heel nieuw proces en niet slechts een nieuwe thread! De instructie die aan het besturingssysteem wordt gegeven (welke instructie dit is, is — helaas — afhankelijk van het gebruikte besturingssysteem en (nog) niet gestandaardiseerd) krijgt (onder andere) als argument het adres van een instructie uit het programma mee. Dit adres is het punt waar de nieuwe thread begint met programmacode uit te voeren. Vanaf dat moment zijn de threads volkomen gelijkwaardig en kunnen effectief worden gezien als twee losse processen met als belangrijk verschil dat ze hun address space met elkaar delen en niet zoals normale processen hun eigen address space hebben. Er is geen sprake van hiërarchie bij threads —



Figuur 1: Petri-deelnet weergave van opsplitsende threads



Figuur 2: Petri-deelnet weergave van samenkomende threads

ze zijn allemaal siblings van elkaar — en het stoppen van een enkele thread zal dus niet de andere threads beïnvloeden. Ter illustratie is in Figure 1 een schematische weergave van dit splitsen weergegeven in de vorm van een fragment van een Petri net. Aangezien threads volledig gelijkwaardig aan elkaar zijn, kan elk van de twee threads die er op dit punt bestaan op een later punt weer meer threads afsplitsen. Threads worden meestal net zo afgehandeld als processen — ze kunnen dus verdeeld zijn over een willekeurig aantal processoren of processing cores, maar de exacte verdeling hangt af van de implementatie die wordt gebruikt. Het is ook mogelijk om twee threads weer samen te voegen — dit heet een "join". Een thread[2] kan gebruikt worden als argument voor een join-call. De thread die deze call maakt zal vervolgens blokkeren totdat de thread die als argument werd gebruikt termineert. Als dit nooit gebeurt, zal de thread die de join-call maakte dus ook nooit meer verder gaan vanaf dat punt. Let wel: het uitvoeren van een join is zeker niet verplicht! Een programma wordt vanzelf beëindigd als alle threads van het programma beëindigd zijn. Een join ziet er als Petri net dus uit zoals in Figure 2. Belangrijk om op te merken is dat er geen zachte limiet is aan het aantal keer dat er gesplitst wordt, maar wel een implementatie-afhankelijke harde limiet. Er is dus geen beperking tot slechts één of twee threads, maar er kunnen er in theorie willekeurig veel zijn. Verder kan een join worden uitgevoerd door eender welke thread van hetzelfde proces. Een thread hoeft dus niet per se gejoind te worden door dezelfde thread die deze thread gemaakt heeft, hoewel dat in de praktijk wel gebruikelijk is. Een programma dat gebruik maakt van meer dan één thread heet ook wel een multithreaded programma.

3 Problemen

3.1 Thread Safety

Een multithreaded programma is thread safe als het altijd correct wordt uitgevoerd. Indien er geen resources worden gedeeld door de threads is dit geen probleem, maar als dit wel het geval is kunnen er bijvoorbeeld race conditions ontstaan. Een race condition is een software "bug" [27] waarbij de volgorde waarin commando's worden uitgevoerd het eindresultaat beïnvloedt. Dit probleem doet zich niet alleen voor op assemblyniveau in de vorm van data die half-geschreven gelezen of overschreven kunnen worden, maar ook op hogere niveaus. Als er meer dan één thread is, kunnen gebeurtenissen in onvoorziene volgordes plaatsvinden. Dit laatste is eigenlijk het resultaat van een structuurfout in het programma, en kan dus niet eenvoudig worden opgelost maar vereist gedisciplineerd werken van de programmeur of een gespecialiseerde taal voor dit doel [19]. Race conditions kunnen echter zelfs optreden als de code er goed uit lijkt te zien en zich pas voordoen

[2]Aan de hand van een unieke identifier voor de thread, die in de meeste implementaties wordt teruggegeven als resultaat van de instructie voor het afsplitsen van een nieuwe thread.

in bepaalde zeldzame gevallen[3]. Een mooi voorbeeld hiervan is de grote stroomuitval in Noord-Amerika van 2004 [14], waarbij een race condition ervoor zorgde dat 2 processen tegelijkertijd schrijf-toegang tot data kregen en het hele alarmsysteem vastliep en geen binnenkomende gegevens meer verwerkte. Was deze software thread-safe uitgevoerd dan was de stroomuitval misschien nooit gebeurd. Thread safety is dus extreem belangrijk voor de stabiliteit en veiligheid van software — aangenomen natuurlijk dat de rest van de software ook in orde is.
In principe kan er tussen elke assembly-instructie die wordt uitgevoerd een context switch[4] optreden naar een andere thread, en op een multi-processor systeem kunnen er zelfs instructies simultaan uitgevoerd worden. Elke serie van operaties die wordt uitgevoerd op of met behulp van resources die gedeeld worden door verscheidene threads dient dus beschermd te worden zodat deze niet halverwege onderbroken wordt door een andere thread die de data gebruikt voordat consistentie[5] is bereikt. Een andere optie is dat alle threads rekening houden met niet consistente data, maar correctheid bewijzen van code gebruikmakend van een dergelijke constructie is natuurlijk erg lastig.
Om correctheidseisen te kunnen stellen aan een synchronisatiemethode die thread safety bewerkstelligt, is het handig om threads in drie secties op te delen:

- Het deel waarin de code staat die niet onderbroken mag worden: de “kritieke sectie”. Met andere woorden: een stuk waarvan we graag willen dat het “atomair” wordt uitgevoerd.

- Het deel direct voor de kritieke sectie, waar wordt gecontroleerd of de kritieke sectie mag worden binnengegaan of niet: de “entry”.

- Het deel direct na de kritieke sectie, waar eventuele clean-up kan worden gedaan: de “exit”.

De rest van de thread, die niets te maken heeft met gedeelde data, negeren we hier. Nu we deze secties hebben gedefinieerd kunnen we de volgende regels gebruiken voor het bewijzen van correctheid van de synchronisatiemethode [20]:

[3]Het is mogelijk om systematisch te controleren op thread safety, maar hiervoor is een representatie nodig van het programma met informatie over de constraints waaraan alle resources dienen te voldoen [30]. Aangezien het maken van deze representatie nogal tijdrovend is wordt dit in de praktijk dus niet veel gedaan hoewel het zeker mogelijk is.

[4]Bij een context switch worden onder andere alle registers naar het geheugen weggeschreven, en vervangen door de registers van de andere thread (of proces). In feite “schakelt” de processor dus eigenlijk naar een andere thread.

[5]Consistent wil hier zeggen dat de resource in een staat is waarin hij veilig kan worden gebruikt — niet meer bezig met veranderen dus, bijvoorbeeld, maar er kunnen meer eisen zijn afhankelijk van hoe er gebruik wordt gemaakt van de resource.

- Als een thread in zijn kritieke sectie bezig is, moeten alle andere threads niet in hun kritieke sectie zijn.
- Het moet mogelijk zijn voor een thread om zijn kritieke sectie binnen te gaan, als geen andere thread in zijn kritieke sectie is of wil binnengaan.
- Indien meer dan één thread zijn kritieke sectie binnen wil gaan, moet maar één thread dat daadwerkelijk kunnen doen.
- Als een thread zijn kritieke sectie gaat verlaten en er zijn andere threads die hun kritieke sectie binnen willen gaan, moet de "exit" code zorgen dat dit wordt toegestaan voor één andere thread.

Let wel: We doen hier de aanname dat alle threads slechts één kritieke sectie hebben, die allemaal invloed op elkaar uit kunnen oefenen. Het is mogelijk om deze regels aan te passen zodat er verscheidene "groepen" van kritieke secties zijn waarbij er nooit meer dan één kritieke sectie uit eenzelfde groep tegelijk actief mag zijn, zoals in de praktijk gebruikelijk is. Verderop in dit document wordt hier aandacht aan besteed.

3.2 Hardware-ondersteuning

Een oplossing voor de synchronisatieproblemen van multithreading is gebruik te maken van ondeelbare operaties. Dit wil zeggen dat zo'n operatie in één ononderbreekbare stap wordt uitgevoerd. Sterker nog — in het geval van een multiprocessor-systeem mag geen van de andere processors deze operatie onderbreken. Zulke operaties heten ook wel atomair, en zijn een uitermate geschikt gereedschap ter voorkoming van race conditions.
In het geval van meer dan één processor is ondeelbaarheid simpelweg niet haalbaar zonder speciale ondersteuning van buitenaf. Deze ondersteuning kan hardware zijn in de vorm van een processorinstructie die atomair uitgevoerd wordt, of een softwarematige oplossing [22]. Dit is simpel te bewijzen door middel van een gedachtenexperiment: stel je 2 threads voor die dezelfde instructies bevatten, op hetzelfde moment in hun voortgang (proces counter) zijn, en op twee verschillende processors worden uitgevoerd die exact even snel zijn. Elke instructie zal dan door beide processors precies tegelijk op dezelfde data worden uitgevoerd, en dus lezen en schrijven ze dezelfde data op hetzelfde moment. Hierdoor is het onmogelijk om een volgorde te bepalen van welke thread "eerst" toegang tot een resource krijgt. Deze beslissing moet op een punt gemaakt worden — aangezien niet beide threads tegelijk gebruik maken maken van de betreffende resource.
Dit probleem uit zich erg snel en dus hebben ontwerpers van processors ermee rekening gehouden. Vrijwel alle processors die meer dan één core bevatten en/of verkrijgbaar zijn voor gebruik in multiprocessor-systemen hebben dan ook speciale atomaire instructies die dit software-probleem omzeilen door hardware ondersteuning voor atomaire instructies toe

te voegen. Het beslisprobleem van wie er eerst gaat wordt dan dus doorgeschoven naar de hardware, die op een of andere (al dan niet eerlijke) manier de beslissing neemt. Een van de meest gebruikte atomaire processorinstructies is de zogenaamde "Compare And Swap" (CAS) instructie. Deze instructie biedt een atomaire operatie die vrij simpel is: een bepaalde geheugenlocatie wordt vergeleken met een verwachte waarde en in dezelfde atomaire operatie vervangen door een nieuwe gegeven waarde als de verwachte waarde inderdaad aangetroffen werdt. Alle Intel x86 en IA-64 processoren vanaf de i486 bevatten deze instructie in de vorm van de CMPXCHG (Compare, Exchange) instructie [9]. ARM processoren bevatten vanaf ARMv6 de LDREX (load reserved) en STREX (store reserved) instructies [1] die samen het gedrag van een CAS instructie kunnen nabootsen. Ook SPARC processors hebben een Compare And Swap instructie [15]. De PowerPC [13] en MIPS [11] architecturen hebben conditional load en conditional store instructies die ook samen het gedrag van een CAS instructie kunnen nabootsen. Kortom — vrijwel alle multiprocessor targets hebben een CAS instructie of kunnen een gelijkwaardige operatie opbouwen uit andere instructies. Bij mijn weten is de enige uitzondering de PA-RISC architectuur [12], die helemaal geen atomaire instructies heeft behalve een "clear word" instructie. Het is hiermee niet mogelijk iets gelijkwaardigs aan een CAS-instructie op te bouwen. Deze architectuur wordt echter al sinds 2008 niet meer verkocht en support ervoor loopt in 2013 af [8], dus ik zie dit niet als een groot probleem voor het ontwikkelen van een algemene oplossing.

Systemen waar maar één processor in zit hebben geen "echte" threads — er is immers op deze machines maar één fysieke processor aanwezig en dus kan er maximaal één thread tegelijkertijd actief zijn. Dit wil zeggen dat als er verscheidene threads bestaan op het systeem deze afwisselend een timeslice krijgen toegekend op de processor. Om deze reden is het hier wel mogelijk om een volgorde te bepalen tussen normaal gesproken gelijkwaardige threads, aangezien er altijd één van de threads als eerste "aan de beurt" is. De scheduler[6] maakt hier dus de keuze voor ons. Er is per definitie altijd minimaal een rudimentaire scheduler beschikbaar als er gebruik wordt gemaakt van multithreading, ongeacht in welke omgeving er wordt gewerkt.
Een voorbeeld van hoe dit kan werken: Stel er draaien weer dezelfde gelijke threads uit het gedachtenexperiment op ditmaal een systeem met maar één processor. Als verschil laten we hier bij het starten van de thread deze zichzelf eerst nummeren. Dit kan altijd — een thread is immers zelf de eerste thread, of wordt gestart door een andere thread die al een nummer heeft. Dit uitdelen van nummers kan dus uitsluitend gebeuren door threads die reeds een nummer hebben. Ongeacht welke volgorde van uitvoer er wordt gekozen door de scheduler, zal er altijd één van de threads als eerste een resource willen gebruiken (want ze zijn nooit tegelijkertijd actief!). Deze thread laat dit aan de andere

[6]De scheduler is de mechaniek die dicteert welke taak wanneer gebruik mag maken van welke processor.

threads weten door in de "lock" variabele van de betreffende resource zijn threadnummer te schrijven. Het is best mogelijk dat een andere thread zijn nummer hier geschreven heeft nadat deze locatie gecontroleerd werd en voordat het nummer weggeschreven werd (bijvoorbeeld als de thread een context switch onderging precies tussen deze twee operaties in), dus vlak na het wegschrijven wordt dit nummer nog eens uitgelezen en vergeleken met het nummer dat net geschreven is. Mocht het nummer veranderd zijn in de tussentijd, dan wordt de operatie afgebroken en krijgt de thread alsnog geen toegang tot de resource. Als alles wel goed gaat, is er na dit punt met zekerheid geen mogelijkheid meer dat een andere thread per ongeluk de resource "kaapt" (want we draaien op een systeem met maar één processor, dus er is geen simultane uitvoer van instructies mogelijk) en kan er veilig gebruik worden gemaakt van de resource. Zodra alle operaties klaar zijn en de resource beschikbaar is voor een andere thread, wordt er een 0 geschreven naar de "lock" variabele om de resource weer klaar voor gebruik te markeren voor andere threads. Deze methode garandeert thread safety op systemen met maar één processor[7].

3.3 Blocking

Blocking is wat er gebeurt als een thread (of proces) niet verder kan gaan omdat deze geen toegang krijgt tot een resource die nodig is. Ofwel treedt er een context switch op naar een andere tread, ofwel blijft de thread wachten (door middel van busy-waiting, waarbij de processor no-op operaties uitvoert) totdat hij verder kan. In beide gevallen wordt natuurlijk niet optimaal gebruik gemaakt van de beschikbare hardware. Het liefste wil je kunnen zien of een resource beschikbaar is, en zo niet een andere taak uitvoeren en later nogmaals de resource proberen te gebruiken. Bij blocking kan deze aanpak niet.
Er zijn drie eigenschappen die een non-blocking geïmplementeerd algoritme kan hebben, waarbij de zwakste eigenschap obstruction-freeness is. Een implementatie is obstruction-free als elke thread op elk moment, mochten alle andere threads worden onderbroken, in zijn eentje zijn bewerkingen kan voltooien binnen een van te voren bekend aantal stappen.
Iets sterker dan obstruction-freeness is lock-freeness. Een implementatie is lock-free als op elk moment mits je lang genoeg wacht minimaal één van de threads vooruitgang boekt.
Wait-freeness is de sterkste van deze drie eigenschappen en eist dat elke losse operatie (van elke thread) van de implementatie binnen een van te voren bekend aantal stappen voltooit.
Alle wait-free implementaties zijn lock-free, alle lock-free implementaties zijn obstruction-free. Het is belangrijk dat een synchronisatiemethode wait-free wordt uitgevoerd omdat geen algoritme sterkere non-blocking eigenschappen kan hebben dan de onderdelen

[7]Aangenomen dat er een besturingssysteem draait met een zinnige scheduler, uiteraard! Als er een scheduler is die bijvoorbeeld de tweede thread pas begint uit te voeren als de eerste is afgelopen, kan het nog mis gaan als de eerste thread output van de tweede thread nodig heeft en daarop (dus automatisch eeuwig) wacht.

waaruit het gebouwd is. Als de synchronisatiemethode dus obstruction-free is maar niet lock-free, kan het algoritme dat deze methode gebruikt nooit beter dan obstruction-free worden uitgevoerd. Als niet wordt voldaan aan een eis van één van deze eigenschappen binnen de synchronisatiemethode, kan dit dus zeker niet het geval zijn voor het gehele algoritme!
Het is gewenst om een algoritme zo sterk mogelijke eigenschappen mee te geven, dus moeten de eigenschappen van een onderdeel dat zo belangrijk is als synchronisatie zeker de sterkste zijn om dit mogelijk te houden.

4 Huidige oplossingen

4.1 Algoritmes (theorie)

Synchronisatiemethodes zijn niets nieuws — dat blijkt wel uit de eerder genoemde regels voor het bewijzen van de correctheid van synchronisatiemethodes, die samen met een tweetal andere bewijsmethodes stammen uit de jaren negentig [20]. Er zal dan ook geen verrassing zijn dat er reeds algoritmes bestaan om synchronisatiemethodes te implementeren. Hieronder volgen een aantal voorbeelden van synchronisatiemethodes waarvan reeds bewezen is dat ze correct zijn.

4.1.1 Semaforen en het bankiersalgoritme

Semaforen en het bankiersalgoritme stammen uit de jaren zestig [21]. Dijkstra zag in de tijd dat computers nog met ferrite cores werkten al de problemen met synchronisatie en probeerde er oplossingen voor te vinden. Het bankiersalgoritme maakt gebruik van semaforen om te garanderen dat klanten van een bank (threads) altijd uiteindelijk de florijnen (resources) krijgen die ze nodig hebben. Een semafoor telt het aantal vrije resources dat nog over is, waarbij alle toegang tot deze teller altijd via ondeelbare operaties gebeurt. Het bankiersalgoritme is dus een voorbeeld van het gebruik van ondeelbare operaties om het synchronisatieprobleem op te lossen.

4.1.2 Petersons algoritme

Petersons algoritme uit de jaren tachtig [28] gebruikt een “turn” variabele als beslisser voor welke thread een resource krijgt. Nadat thread 0 klaar is met zijn kritieke sectie zet deze de turn variabele op 1 om aan te geven dat thread 1 nu zijn kritieke sectie binnen mag. Als thread 1 klaar is zet deze de turn variabele weer op 0 en zo verder. Dit heeft natuurlijk een paar problemen. Zo loopt alles direct vast als één van de threads zijn taken heeft voltooit of crasht. Bovendien is deze oplossing slechts geschikt voor twee threads (hoewel uitbreiden naar n threads vrij simpel is). Belangrijk om op te merken is

dat Petersons algoritme in tegenstelling tot het bankiersalgoritme niet afhankelijk is van ondeelbare operaties, en dus geen hardware-ondersteuning hiervoor nodig heeft.

4.1.3 Bakkerijalgoritme en tickets

Het bakkerijalgoritme[26] is een voorbeeld van een algoritme dat het synchronisatieprobleem oplost met nummertjes (tickets). In deze klasse van oplossingen krijgt elke thread die zijn kritieke sectie binnen wil gaan een soort "wachtnummer" toegewezen en mag daarna in de volgorde van de wachtnummers de kritieke sectie binnen gaan. Aangezien dit opvragen van het wachtnummer een operatie is die eigenlijk beschermd zou moeten zijn maar dat niet is, is het mogelijk dat twee threads hetzelfde wachtnummer krijgen toegewezen. Het bakkerijalgoritme lost dit probleem op door het threadnummer als een soort tiebreaker te gebruiken bij gelijke wachtnummers. Net als Petersons algoritme heeft dit algoritme geen ondeelbare operaties nodig om te werken, maar omdat alle wachtnummers van alle threads moeten worden nagekeken bij elke poging tot toegang tot een resource is het niet een bijzonder efficiënt algoritme en daarom geen goede keuze voor een algemene synchronisatiemethode.

4.2 Libraries (praktijk)

Net zoals er theoretische oplossingen bestaan voor synchronisatie zijn er ook vele libraries beschikbaar die één of meer synchronisatiemethodes implementeren in een programmeertaal. Ter herhaling: een library is een algemeen stuk code dat simpel zonder aanpassingen te hoeven maken toe te voegen is als onderdeel van een programma. In de hierop volgende paragrafen worden er een aantal uitgelicht, maar het is zeker geen volledige opsomming — daarvoor bestaan er simpelweg te veel. Zo worden bijvoorbeeld stdthread [16] en threadslib [17] niet behandeld omdat deze puur in details van de implementatie verschillen van de andere oplossingen.

4.2.1 Boost Thread Library

De Boost Thread library [4] maakt threading algemeen voor alle ondersteunde targets[8] en besturingssystemen. Hij bevat ook non-blocking manieren om locks te verkrijgen en te controleren, maar is deel van de enorm grote Boost libary, en daardoor geen lichtgewicht oplossing. Bovendien neemt het ook de taak van threads aanmaken en samenvoegen op zich, en niet alleen het veilig houden van de data. Voor eenvoudige toepassingen overschiet Boost dus al snel zijn doel. Verder staat Boost bekend als erg lastig om te cross-compilen en is het niet practisch om de library statisch in je applicatie te verwerken — waardoor

[8]Een target is een doel waarvoor een programma kan worden gecompileerd. Over het algemeen is een target een type processor.

applicaties die hem gebruiken onnodig groot worden of de library als voorgeinstalleerde vereiste nodig hebben. Niet bepaald een elegante oplossing dus.

4.2.2 Het atomic_ops project

Het atomic_ops project [3] biedt een wrapper rond de atomische operaties zoals die zijn geïmplementeerd in vrijwel alle bestaande compilers, en bied een aantal handige manieren om hier gebruik van te maken, maar ondersteunt geen uniprocessor-systemen die deze operaties niet ondersteunen (nogal logisch, deze operaties zijn daar immers niet aanwezig). Bovendien worden de atomische functies simpelweg in één standaard formaat gegoten, en dus niet makkelijker te gebruiken maar alleen bruikbaar op (bijna) alle targets. atomic_ops is dus slechts een manier om de atomische operaties die processors al ondersteunen op één manier aan te spreken, en geen kant-en-klare oplossing voor de problemen die in de inleiding werden beschreven.

4.2.3 Atomic

Atomic [2] levert transactie ondersteuning. Hiermee wordt het mogelijk om bepaalde meegeleverde datatypes "terug te rollen" naar een punt in het verleden. Dit is echter ondanks wat de naam doet vermoeden, een zaak die helemaal los staat van threading, en meer van toepassing is op databases en andere dergelijke toepassingen.[9]

4.2.4 Ondersteuning voor threading in standaarden

Er zijn voorstellen gedaan om atomaire operaties[6] en threading in de C++-standaard op te nemen, en zowel de C++0x [7] als de C1X [5] standaarden, die binnenkort af zullen zijn, bevatten ondersteuning voor atomaire operaties en threading. Dit betekent dat een library die deze functies makkelijk te gebruiken maakt zonder verlies van de kracht ervan erg nuttig zal zijn in de nabije toekomst.

5 Multithreading voor iedereen

Hopelijk zijn door de hoofdstukken hierboven de problemen met multithreading and synchronisatie een beetje duidelijk geworden. Ik stel een nieuwe (theoretische) library voor die omgaan met deze problemen makkelijker maakt.

[9] Deze library is hier toch vermeld, omdat de naam kan doen vermoeden dat hij atomaire operaties met zich meebrengt, terwijl dit niet zo is.

5.1 Waarom een nieuwe library?

Deze library is bedacht omdat er nog geen algemene oplossing voor het synchronisatie-probleem was. Ik heb de betekenis van "algemeen" hier zo ver mogelijk opgerekt: niet alleen algemeen in de zin van hardware en software compatibiliteit, maar ook algemeen in de zin van hoe de library werkt. De bestaande threading libraries implementeren allemaal slechts één enkele synchronisatiemethode. In deze nieuwe opzet krijg je alle gereedschappen aangereikt die nodig zijn om een werkende, correcte en gepersonaliseerde synchronisatiemethode in elkaar te zetten, zonder gedwongen te zijn specifiek een bepaalde methode toe te passen. Het fijne hiervan is dat er niet nagedacht hoeft te worden over de fijne details van de synchronisatiemethode (die zijn immers verwerkt in de functies van de library), maar hij in grote lijnen kan worden bedacht en daarna relatief eenvoudig geïmplementeerd, met als extra voordeel dat het bewijzen van de correctheid ervan makkelijker wordt door de reeds gegeven garanties van de library.
De library is dus een soort toolkit voor synchronisatie en niet een kant-en-klaar oplossing. Dit staat toe om in elk project een heel specifieke synchronisatiemethode toe te passen die helemaal geoptimaliseerd is voor de situaties die daarin voorkomen.

5.2 Eisen aan de library

Ik had een aantal eisen voor ogen waar volgens mij deze library aan zou moeten voldoen:

- De library moet simpel maar krachtig te zijn om vooral een hulp te zijn voor programmeurs en geen last. Dit wil zeggen dat programmeurs de keuze hebben om hun algoritme wait-free uit te voeren — of juist niet. Ze moeten leestoegang terwijl er geschreven wordt naar een datastructuur kunnen toestaan — of juist niet. De kracht zit hem dus vooral in de eenvoud en het duidelijke gedrag van de library, die goede mogelijkheden op een heldere manier aanbiedt maar alle opties open laat.

- De library moet zo lichtgewicht als maar mogelijk is zijn, zonder functionaliteit te kort te doen uiteraard. Omdat het hier draait om programma's waarvan er geprobeerd wordt ze sneller te laten lopen, moet de library zelf natuurlijk het geheel niet trager maken.

- De library moet wait-free zijn geïmplementeerd. Zoals eerder in dit document al beschreven is, is het erg belangrijk dat een synchronisatiemethode wait-free is zodat de algoritmes die hem gebruiken niet beperkt zijn en zelf ook wait-free kunnen worden uitgevoerd als de programmeur dat wil.

- Kunnen terugvallen op een software-implementatie indien hardwareondersteuning niet beschikbaar is. Er moet dus niet alleen code zijn geschreven voor zowel mét

als zonder hardwareondersteuning, maar deze code moet in beide gevallen hetzelfde gedrag vertonen.

- Tenslotte moet de library enkele garanties bieden waarmee het mogelijk moet zijn gemakkelijker correctheid te bewijzen van de synchronisatiemethode.

5.3 Garanties

De library maakt het thread-safe implementeren van algoritmes dus gemakkelijker, maar voorkomt zeker geen fouten in de implementatie of denkwijze van de programmeur! Het kan wegens de gestelde eisen aan de library raar genoeg geen eis zijn dat simpelweg de Thread Safety library gebruiken het algoritme bij definitie thread-safe maakt. Indien de library juist wordt gebruikt zal dit zeker zo zijn, maar aangezien er zoveel vrijheid is kan een programmeur best "fouten" maken waardoor het geheel niet meer thread-safe wordt. De library maakt een aantal publiekelijk toegankelijke operaties (functies) beschikbaar op het datatype waarop hij gebruikt wordt (vanaf hier: de resource), die bepaalde garanties bieden als ze aangeroepen worden. Dit maakt het makkelijker te bewijzen dat de synchronisatiemethode correct is. Wat deze garanties precies zijn wordt hier besproken.

5.3.1 Garanties bij standaard gebruik

De library kan worden aangeroepen met een aantal opties. Als de library wordt aangeroepen zonder enige verdere opties mee te geven, is dit het standaard gedrag. Het standaard gedrag is de meeste restrictieve, maar biedt daardoor ook de meeste garanties. Als je goed kijkt, zal je zien dat het standaard gedrag van de library precies het gedrag van een mutex is, en dus ook niets speciaals doet. Het gedrag als mutex is alleen van toepassing als getAccess, dropAccess, getVal en putVal alleen worden aangeroepen met het *threadnummer* van de thread die de aanroep doet. Bij het overtreden van deze regel vervallen alle garanties.[10] De library bestaat uit zes functies, maar slechts vijf functies hebben garanties. Dat zijn de volgende:

haveAccess(*threadnummer***)** Deze functie voert een controle uit die kijkt of *threadnummer* momenteel exclusieve toegang heeft verkregen naar de resource. De functie geeft als return-waarde een boolean *true* indien dit waar blijkt te zijn, en een boolean $false$ indien dit niet zo is. Er is enkel een garantie op correcte data als een thread zijn eigen *threadnummer* opvraagt, andere *threadnummer* waardes moeten worden gezien als data met beperkte geldigheidsduur en als direct verlopen worden beschouwd — dit omdat direct op het

[10] Helaas kunnen we dit gedrag niet forceren, aangezien er geen gestandaardiseerde manier is om een uniek nummer te genereren voor de "current thread" die werkt voor alle soorten threading.

moment dat de functie afgelopen is (en zelfs terwijl hij nog bezig is) natuurlijk een andere thread de waarde kan beïnvloeden. In het geval dat het eigen *threadnummer* wordt opgevraagd is de enige thread die de waarde kan beïnvloeden (dit door het gedrag van getAccess en dropAccess) de thread die de aanvraag doet, en alleen dan is er dus de garantie dat de waarde correct is.

getAccess(*threadnummer***)** Deze functie wordt gebruikt om exclusieve toegang aan te vragen naar de resource, voor thread *threadnummer*. Als toegang wordt verleend geeft de functie als return-waarde een boolean *true*, en zo niet een boolean $false$. Er is de garantie dat deze return-waarde altijd correct is, en dus alleen *true* is als toegang daadwerkelijk is verleend en alleen $false$ is als er echt geen toegang is verleend. Let wel — er is dus niet de garantie dat $false$ betekent dat de resource in gebruik is door een andere thread! Dit omdat de thread die tijdens de functieaanroep de resource in gebruik had, best tussendoor dropAccess kan hebben aangeroepen. Dit gedrag wordt bereikt door een CAS-instructie te gebruiken als die wordt ondersteund door de hardware. Mocht de CAS-instructie niet ondersteund worden, wordt er de aanname gedaan dat het om een single-processor systeem gaat en er dus geen threads simultaan actief kunnen zijn, en een variatie gebruikt op het bakkerijalgoritme om altijd toegang aan slechts één thread tegelijk te geven.

dropAccess(*threadnummer***)** Deze functie wordt gebruikt om de reeds verkregen exclusieve toegang weer op te geven. Deze functie heeft geen return-waarde, maar er is de garantie dat hij altijd succesvol is als er inderdaad exclusieve toegang was voor het betreffende *threadnummer* naar de betreffende variabele. Als deze toegang er niet was, doet de functie niets. Het is dus altijd veilig om hem aan te roepen, wat het effect heeft dat toegang wordt opgegeven indien deze verkregen was.

getVal(*threadnummer***,** *val***)** Deze functie vult *val* met de inhoud van de betreffende variabele en geeft als return-waarde *true* terug. Dit gebeurt alleen als er exclusieve toegang is voor dit *threadnummer*, zo niet doet deze functie niets en is de return-waarde $false$. Er is de garantie dat deze functie geen actie onderneemt zonder exclusieve toegang. Dit wordt bereikt door voor elke aanpassing intern eerst haveAccess aan te roepen.

putVal(*threadnummer***,** *val***)** Deze functie vult de inhoud van de betreffende variabele met de waarde van *val* en geeft als return-waarde *true* terug. Dit gebeurt alleen als er exclusieve toegang is voor dit *threadnummer*, zo niet doet deze functie niets en is de return-waarde $false$. Er is de garantie dat deze functie geen actie onderneemt zonder exclusieve toegang. Dit wordt bereikt door voor elke aanpassing intern eerst haveAccess aan te roepen.

5.3.2 Garanties bij `INSECURE_READ` gebruik

Als de library wordt aangeroepen met de optie `INSECURE_READ`, gelden de onderstaande garanties in plaats van de standaard garanties. Dit gedrag is iets minder restrictief — om precies te zijn staat het lezen (maar niet schrijven!) zonder exclusieve toegang toe. In het geval dat er in het programma zelf mee rekening wordt gehouden dat de data mogelijk niet consistent is, kan zo het lezen van inconsistente data worden toegestaan terwijl exclusieve schrijftoegang is vergeven aan een (andere) thread.

haveAccess(*threadnummer***)** Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

getAccess(*threadnummer***)** Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

dropAccess(*threadnummer***)** Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

getVal(*threadnummer***,** *val***)** Deze functie vult *val* met de inhoud van de resource en geeft als return-waarde *true* terug. Als er geen exclusieve toegang is voor dit *threadnummer* is de return-waarde $false$, maar wordt *val* nog steeds gevuld met de (mogelijk inconsistente) data. Als er dus *true* wordt teruggegeven is er de garantie dat de data consistent is, maar als er $false$ wordt teruggegeven niet. Het is dus belangrijk in dit geval om de return-waarde van deze functie te controleren, en er niet zomaar vanuit te gaan dat deze *true* is!

putVal(*threadnummer***,** *val***)** Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

5.3.3 Garanties bij `ALLOW_STEALING` gebruik

Als de library wordt aangeroepen met de optie `ALLOW_STEALING`, gelden de onderstaande garanties in plaats van de standaard garanties. Dit gedrag is veel minder restrictief — om precies te zijn staat het het stelen van exclusieve toegang toe. Om een algoritme obstruction-free of beter te implementeren is er een mechaniek nodig waarmee de acties van andere threads kunnen worden "teruggerold", alsof ze nooit gebeurd waren. Dit kan door middel van bijvoorbeeld transacties, maar in vele gevallen kan het ook veel simpeler door gebruik te maken van domeinkennis over het gekozen algoritme. Vaak heeft het merendeel van operaties geen effect op de andere threads, en hoeft dus ook niet te worden

teruggerolt. Programmeurs willen hier wellicht slim gebruik van maken, en daarom is het ook mogelijk gemaakt binnen de library om te controleren of een lock nog actief is of niet. Bovendien kunnen threads de locks van andere threads breken en zelf overnemen. Zo kan er dus door rekening te houden met dit gedrag code worden geschreven waarbij een thread kan worden gecancelled door een andere thread, zelfs als deze niet meer actief is (bijvoorbeeld door een crash of deadlock), en relatief gemakkelijk obstruction-free of beter worden uitgevoerd. Helaas heeft dit natuurlijk wel effect op de garanties:

haveAccess(*threadnummer*) Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

getAccess(*threadnummer*, *force*) De nieuwe parameter *force* is een boolean. Als *force* gelijk is aan *false* gedraagt deze functie zich gelijk aan het standaardgedrag. Is *force* echter gelijk aan *true*, is het gedrag anders — dan wordt toegang **altijd** verleend, maar is de return-waarde *false* als de toegang is gestolen van een andere thread, en *true* indien dit niet nodig was om toegang te verlenen.

dropAccess(*threadnummer*) Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

getVal(*threadnummer*, *val*) Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag. Let wel dat het belangrijk is in dit geval om de return-waarde van deze functie te controleren, en er niet zomaar vanuit te gaan dat deze *true* is!

putVal(*threadnummer*, *val*) Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag. Let wel dat het belangrijk is in dit geval om de return-waarde van deze functie te controleren, en er niet zomaar vanuit te gaan dat deze *true* is!

5.3.4 Garanties bij `ALLOW_STEALING` en `INSECURE_READ` gebruik

Het is ook mogelijk om beide opties tegelijk te gebruiken. Dan zijn de garanties de logische combinatie van de twee opties, en zien er als volgt uit:

haveAccess(*threadnummer*) Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

getAccess(*threadnummer*, *force*) Deze functie gedraagt zich in dit geval gelijk aan het `ALLOW_STEALING`-gedrag.

dropAccess(*threadnummer***)** Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag.

getVal(*threadnummer***,** *val***)** Deze functie gedraagt zich in dit geval gelijk aan het `INSECURE_READ`-gedrag. Let wel dat het belangrijk is in dit geval om de return-waarde van deze functie te controleren, en er niet zomaar vanuit te gaan dat deze *true* is!

putVal(*threadnummer***,** *val***)** Deze functie gedraagt zich in dit geval gelijk aan het standaardgedrag. Let wel dat het belangrijk is in dit geval om de return-waarde van deze functie te controleren, en er niet zomaar vanuit te gaan dat deze *true* is!

5.3.5 Garanties bij het gebruik van de *Data* functie

De eerder genoemde zesde functie zonder garanties is de *Data*(*threadnummer*) functie. Deze functie geeft als return-waarde het daadwerkelijke adres van de resource in het geheugen, maar doet dit alleen als er exclusieve toegang is voor het betreffende *threadnummer*. Dit is natuurlijk een enorm veiligheidsrisico, omdat het adres van de "echte" gegevens een vrijkaart is om alle ingebouwde regels van de library te breken. Het gebruik van deze functie laat dan ook in principe alle garanties vervallen, maar hij kan toch zeer nuttig zijn. Door bijvoorbeeld de library te gebruiken onder de standaard gedragsinstellingen kan je door simpelweg het adres weg te gooien op het moment dat de exclusieve toegang wordt opgegeven toch nog alle gewoonlijke garanties en het mutex-gedrag behouden. Omdat dit buiten de library om dient te gebeuren kan de library zelf hier dus niets meer garanderen, maar als je je zelf aan deze regel houdt kan je toch alle garanties die je normaal hebt behouden (en daarbij ook het makkelijker kunnen bewijzen van de correctheid van de synchronisatiemethode).

5.3.6 Het nut van de garanties

Eerder werden de volgende regels genoemd voor het bewijzen van correctheid:

- Als een thread in zijn kritieke sectie bezig is, moeten alle andere threads niet in hun kritieke sectie zijn.
- Het moet mogelijk zijn voor een thread om zijn kritieke sectie binnen te gaan, als geen andere thread in zijn kritieke sectie is of wil binnengaan.
- Indien meer dan één thread zijn kritieke sectie binnen wil gaan, moet maar één thread dat daadwerkelijk kunnen doen.

- Als een thread zijn kritieke sectie gaat verlaten en er zijn andere threads die hun kritieke sectie binnen willen gaan, moet de "exit" code zorgen dat dit wordt toegestaan voor één andere thread.

Laten we deze regels nu zoals eerder besproken algemener maken, zodat elke kritieke sectie die niets te maken heeft met een andere kritieke sectie ook daadwerkelijk onbeïnvloed gelijktijdig met deze andere kritieke sectie actief kan zijn. Om te beginnen moeten we onze definities iets aanpassen:

- Het deel waarin de code staat die niet onderbroken mag worden: de "kritieke sectie". Met andere woorden: een stuk waarvan we graag willen dat het "atomair" wordt uitgevoerd.

- Het deel direct voor de kritieke sectie, waar wordt gecontroleerd of de kritieke sectie mag worden binnengegaan of niet: de "entry".

- Het deel direct na de kritieke sectie, waar eventuele clean-up kan worden gedaan: de "exit".

We gaan hier "niet onderbroken mogen worden" op een andere manier zien — we willen niet dat de resources waarop de code betrekking heeft "aangeraakt" (dus gelezen of geschreven) worden door andere threads terwijl we in de kritieke sectie zijn. Een deel dat niet onderbroken mag worden is dan elk deel waarin deze resources voor een bepaalde tijd inconsistent zijn. Onder inconsistent verstaan we dat een andere thread mogelijk de resource verkeerd interpreteert terwijl deze wordt gebruikt en/of gewijzigd door de thread die ermee bezig is. De bestaande regels veranderen we dan naar de volgende regels:

- Als een thread in een kritieke sectie bezig is gebruikmakend van een gedeelde resource X, moeten alle andere threads niet in een kritieke sectie zijn gebruikmakend van deze gedeelde resource X.

- Het moet mogelijk zijn voor een thread om een kritieke sectie binnen te gaan gebruikmakend van een gedeelde resource X, als geen andere thread in een kritieke sectie gebruikmakend van deze gedeelde resource X is of wil binnengaan.

- Indien meer dan één thread een kritieke sectie binnen wil gaan gebruikmakend van een gedeelde resource X, moet maar één thread dat daadwerkelijk kunnen doen.

- Als een thread een kritieke sectie gebruikmakend van een gedeelde resource X gaat verlaten en er zijn andere threads die een kritieke sectie binnen willen gaan gebruikmakend van deze gedeelde resource X, moet de "exit" code zorgen dat dit wordt toegestaan voor één andere thread.

Dit is niet meer dan een generalisatie van de regels naar meer dan één gedeelde resource en meer dan één kritieke sectie. Nieuw hier is dat het best mogelijk is dat een programma in meer dan één kritieke sectie tegelijk bezig is, gebruik makend van verschillende resources. Deadlock is dus wel degelijk nog mogelijk — bijvoorbeeld als een thread α in een kritieke sectie zit die gebruik maakt van gedeelde resource X en een thread β in een kritieke sectie zit die gebruik maakt van gedeelde resource Y, waarbij α een kritieke sectie gebruikmakend van Y nodig heeft om zijn kritieke sectie te kunnen verlaten, en β een kritieke sectie gebruikmakend van X nodig heeft om zijn kritieke sectie te kunnen verlaten. Het enige wat deze constructie voorkomt is dan ook de corruptie van data, niet het mogelijk zijn van deadlocks of livelocks. Het verbieden van kritieke secties binnenin andere kritieke secties voorkomt dit wel, maar is veel minder efficiënt omdat blocking dan nodig is in de meeste gevallen.
Nu dit alles is gedefinieerd valt het direct op wat het nut van de bovenstaande garanties is. Door deze garanties als building blocks te gebruiken wordt het erg simpel om om te gaan met de regels die we nu hebben staan. Sterker nog — het gebruik van enkel de library volgens zijn standaard gedrag, waarbij de data dus **niet** buiten de library om wordt aangeraakt (gelezen of geschreven) zorgt ervoor dat automatisch aan al deze regels wordt voldaan (en dit is wat het tot een mutex maakt in dit geval)! Door bij de minder restrictieve vormen van de library in eigen code ervoor te zorgen toch te voldoen aan de bovenstaande regels, is het mogelijk een zeer efficiënt programma te maken dat geen last heeft van datacorruptie, waarvan de code toch nog leesbaar blijft voor anderen.
Met andere woorden — deze library (en natuurlijk elke andere library of code die zich aan deze regels houdt) biedt een set gereedschap aan waarmee het relatief gemakkelijk wordt om algoritmes uit te voeren op een zo efficiënt mogelijk manier (bijvoorbeeld obstruction free of beter), terwijl hun correctheid gemakkelijk te bewijzen wordt (ik bedoel hier natuurlijk het bewijzen van de correctheid van de omgang met (en niet corrupteren van) de data, niet bewijzen van obstruction-freeness of correctheid van de berekening, wat volgens de traditionele methodes zal moeten blijven gebeuren).

6 Praktijkgebruik

Een oplossing die in theorie werkt is natuurlijk mooi, maar het kan nooit kwaad het een en ander in praktijk te testen! Daarom heb ik een voorbeeldimplementatie van de library geschreven in C++, en een viertal programma's die deze implementatie van de library gebruiken om te demonstreren wat hij kan en dat hij werkt.

6.1 De library zelf

Als voorbeeld van een implementatie van de hierboven vermelde eigenschappen heb ik een thread safety template library in C++ geschreven die elk type datastructuur relatief makkelijk thread-safe kan maken. De voorbeeldcode is als bijlage bij deze scriptie gevoegd, als een C++ header file. Een template library kan worden gezien als een soort gereedschap dat zichzelf aanpast aan het vereiste datatype. Dit betekent dat de library kan worden gebruikt als een wrapper rondom elk bestaand datatype, van een simpele byte tot de meest ingewikkelde user-defined structuur. Bij het schrijven van de library zijn er een aantal keuzes gemaakt die misschien niet voor de hand liggen. Hieronder worden deze nog even toegelicht.

6.1.1 Lock als een void-pointer

De lock-variabele wordt gebruikt om op te slaan welke thread toegang tot een resource krijgt, en als datatype voor deze variabele heb ik gekozen voor een pointer. Normaal gesproken zou je deze variabele (en de threadcounter) wellicht als een integer kiezen. Het aantal bytes geheugen dat een integer gebruikt is echter verschillend op verschillende architecturen. Bovendien is de CAS-instructie (of iets gelijkwaardigs) niet altijd voor alle groottes geheugenblokken beschikbaar. Deze twee problemen kunnen in één klap worden opgelost door als teller en als lock een void pointer te gebruiken. GCC (GNU Compiler Collection) bevat namelijk ingebouwde defines die de grootte in bytes van een pointer bevatten en andere defines die aangeven of voor een bepaalde grootte geheugenblok een CAS-instructie beschikbaar is. Zo kunnen we dus compile-time controleren of we een CAS-instructie kunnen gebruiken of niet. Verder is er nog het probleem dat bijvoorbeeld 8-bit processors maar erg weinig geheugen kunnen aanspreken, en tellers en lockvariabelen van 4 bytes hier een enorme verspilling zouden zijn. Het gebruik van een void pointer lost dit op, omdat pointers altijd een mooi gekozen grootte hebben voor de architectuur die ze gebruikt[11]. Zo schaalt deze library dus automatisch mee met de gekozen architectuur. Hiernaast is er nog het extra voordeel dat pointers op vrijwel alle processoren in een enkele assembly-instructie kunnen worden geschreven en gelezen, dus valt het probleem van de niet-CAS-oplossing waarbij dit in zo min mogelijk instructies moet gebeuren[12] weg.

[11] Per definitie kan een pointer immers niet meer geheugen aanspreken dan waar de architectuur toegang tot heeft, dus heeft hij een grootte waarbinnen al het aanspreekbare geheugen past — en over het algemeen de kleinste grootte waarvoor dit geldt.

[12] Want bij één instructie is succes gegarandeerd — als er meer dan één instructie nodig is om te lezen (of preciezer — te vergelijken) is deze garantie niet meer aanwezig omdat de thread een context switch kan hebben ondergaan tussen de instructies.

6.1.2 Volatile private variabelen

Alle private variabelen zijn voorzien van het volatile keyword. Programmeurs bekend met het volatile keyword zullen direct weten waarom dit nodig is, maar voor degenen die dit niet zijn hier een verduidelijking. Als verscheidene threads dezelfde data gebruiken is het best mogelijk dat thread A een stuk data inleest naar een register, en het later weer leest. De compiler ziet deze "dubbele" read, en zal voor de tweede read het register gebruiken in plaats van de waarde opnieuw uit het geheugen te lezen. Een prima optimalisatie — behalve als thread B ondertussen de inhoud van dit stuk geheugen heeft veranderd. Het volatile keyword voorkomt dit soort optimalisaties en dwingt de compiler om assembly code te genereren die elke keer opnieuw de waarde uit het geheugen inleest.

6.1.3 De variatie op het bakkerijalgoritme

De library gebruikt (voor processoren die geen hardwareondersteuning bieden in de vorm van een CAS-instructie) een variatie op het bakkerijalgoritme. In plaats van tickets en een thread nummer om de knoop door te hakken, wordt er zo veel mogelijk geprofiteerd van de scheduler die aanwezig is: de scheduler bepaald in principe welke thread "eerst" mag. Dit betekent dat de voorkeur voor bepaalde threads kan worden bepaald aan de hand van de voorkeur die wordt opgegeven aan de scheduler — een methode die veel algemener is dan een of andere obscure waarde in de library zelf. Een nadeel van deze keuze is dat als de scheduler niet bepaald intelligent is, de threading implementatie dat dan natuurlijk ook niet is.

6.1.4 De TCount class

De TCount class is een simpele singleton[13] class met slechts één functie, namelijk AddThread. Deze functie doet niets meer dan als return-waarde een nieuw threadnummer afgeven elke keer dat hij wordt aangeroepen. Omdat er geen eenduidige manier is om threadnummers te maken (de meeste soorten threading hebben wel een threadnummer of iets soortgelijks, maar deze hebben verscheidene datatypes) heb ik deze class toegevoegd aan de Thread Safety Library. Hij geeft threadnummers terug in de vorm van void pointers, het type dat de library zelf verwacht te krijgen als nummers en ook intern mee werkt. Het is natuurlijk niet verplicht om de TCount class te gebruiken voor het genereren van unieke nummers, hij is slechts toegevoegd als aanvulling op de library zelf en kan gemakkelijk worden vervangen door een andere (wellicht betere?) manier van threadnummers uitdelen.

[13] Een singleton kan slechts eenmaal bestaan — het aanmaken van meer dan één van deze resulteert in een verwijzing naar de eerste in plaats van een echte nieuwe instantie.

6.2 Een viertal programma's

Ik heb de voorbeeldlibrary zoals hierboven beschreven gebruikt om een aantal voorbeeldprogramma's die gebruik maken van de POSIX standaard voor threads [18] te implementeren. De keuze voor POSIX Threads is eigenlijk niet relevant voor deze implementaties, en er is slechts voor POSIX Threads gekozen omdat ik daar reeds mee bekend was. De volledige broncode van deze voorbeeldprogramma's is bij deze scriptie gevoegd, inclusief een Makefile. De voorbeeldprogramma's zouden moeten kunnen compileren op elk POSIX-compatible besturingssysteem (zoals bijvoorbeeld Linux) met de GCC compiler.

6.2.1 Mutex

Mutex is een programma dat de mutex-werking van het standaardgedrag van de library demonstreert. Als het programma start wordt er als gedeelde resource een integer aangemaakt, beschermd door mijn Thread Safety library, en geïnitialiseerd op nul. Vervolgens worden er n threads gestart die allemaal dezelfde code uitvoeren: ze proberen toegang tot de resource te krijgen en blijven in een loop wachten totdat ze toegang krijgen, waarna de resource wordt uitgelezen, één seconde gewacht, daarna de uitgelezen waarde met één verhoogd, en teruggeschreven naar de resource. Als het goed is, is de waarde van de resource nadat alle threads zijn afgelopen dus n. Nadat alle threads zijn beëindigd schrijft het programma de waarde van de resource naar het scherm en geeft deze waarde als return-waarde voor het hele programma. Als command-line opties heeft het programma alleen de optie "n", voor het instellen van n. Indien niet opgegeven is $n = 42$.

6.2.2 Readable Mutex

Readable Mutex is een programma dat het `INSECURE_READ`-gedrag van de library demonstreert. Het programma doet in principe hetzelfde als het Mutex programma, met als aanpassing dat de library met de `INSECURE_READ`-flag wordt aangeroepen, en na het maken van de n threads, er gedurende n seconden, elke seconde wordt afgedrukt wat de waarde van de resource op dit moment is. Dit laat het lezen van mogelijk niet consistente data zien in praktijk. De in– en uitvoer van het programma is verder gelijk aan die van Mutex.

6.2.3 Breakable Mutex

Breakable Mutex is een programma dat het `ALLOW_STEALING`-gedrag van de library demonstreert. Het programma doet in principe hetzelfde als het Mutex programma, met als aanpassing dat de library met de `ALLOW_STEALING`-flag wordt aangeroepen, en na het maken van de n threads, er nog een extra thread wordt gestart die een ander code-pad

uitvoert. Deze thread heb ik de AnnoyThread genoemd, omdat hij de andere threads behoorlijk in de weg zit: om precies te zijn steelt hij een seconde lang toegang tot de resource, en wacht dan 2 seconden. Dit herhaalt hij $n/2$ keer. Dit programma laat het stelen van toegang zien in praktijk. De in– en uitvoer van het programma is verder gelijk aan die van Mutex.

6.2.4 ReadBreakable Mutex

ReadBreakable Mutex is een programma dat het `ALLOW_STEALING` en `INSECURE_READ`-gedrag van de library demonstreert. Het is de logische mix tussen Breakable Mutex en Readable Mutex. Dit wil zeggen dat het het gedrag van Breakable Mutex vertoont, met de extra output van Readable Mutex. Dit programma laat de combinatie van het stelen van toegang en lezen van inconsistente data zien in praktijk. De in– en uitvoer van het programma is verder gelijk aan die van Mutex.

6.2.5 Testen

Om bovenstaande vier programma's te kunnen testen is er in de makefile een "test" target opgenomen. Door het uitvoeren van het commando "make test" worden alle vier de programma's achter elkaar gedraaid met een aantal verschillende waardes voor n, en vervolgens gekeken of de return-waarde steeds n is. Als dat zo is, staat er aan het eind de tekst "Tests gelukt" in beeld.

7 Discussie

7.1 Reflectie

Het concept, de library en testprogramma's hadden nog zeker verbeterd kunnen worden op een aantal vlakken:

- De bedoeling was een algemene methode te ontwikkelen. De voorbeeldlibrary is echter gebonden aan de GCC C++compiler, en de testprogramma's aan POSIX Threads. Door gebruik te maken van bijvoorbeeld atomic_ops en/of C++0x zou deze beperking kunnen worden opgeheven. Maar als we nog algemener kijken — in het bijzonder buiten de programmeertaal C++ — is dat niet het probleem dat ik probeerde op te lossen. Het gaat erom of de oplossing zélf werkt binnen elke omgeving, niet of er een library kan worden gemaakt die de oplossing implementeert binnen elke omgeving. Het zou zeker mooi zijn om dat te hebben, maar zoiets is gewoonweg niet haalbaar binnen redelijke tijdsbeperkingen. Het draait hier dan ook meer om het concept, en niet om de daadwerkelijke implementatie.

- De template-library ondersteunt niet alle operaties die variabelen ondersteunen. Dit had gekund via bijvoorbeeld operator overloading en zou alles een stuk completer maken, maar is geen noodzakelijk iets voor de werking van het geheel en maakt bovendien de code een stuk lastiger te lezen en te begrijpen. Vandaar dat ik dat ook niet gedaan heb.

- De voorbeeldprogramma's zijn nogal simpel. Mijn bedoeling was een Open Source project te vinden dat gebruik maakt van multithreading, mijn library als vervanging van de bestaande multithreading code te gebruiken, en vervolgens daar wat metingen op uit te voeren. Helaas bleek al vrij snel dat een project dat gebruik maakt van multithreading, over het algemeen een vrij complex project is. Een dergelijke test van de library zou dan ook meer tijd hebben gekost dan de hele rest van dit project samen. Bovendien is het erg lastig om zinnige metingen te verrichten aan multithreaded software, en worden er zo veel optimalisaties toegepast dat verscheidene methodes al snel onvergelijkbaar zijn. Een simpeler, meer praktisch gericht voorbeeld leek me daarom meer geschikt.

- Er hadden zeker nog meer gedragingen kunnen worden bedacht behalve alleen de `ALLOW_STEALING` en `INSECURE_READ` gedragingen. Deze twee (en hun combinatie) leken me echter de meest voor de hand liggende, waarvoor ik direct nuttige toepassingen kon bedenken. Mijn doel was dan ook niet om een compleet concept voor multithreading te bedenken, maar meer een soort basis waaraan verder gewerkt en verbeterd kan worden.

7.2 Samenvatting en conclusie

De tijd van de uniprocessor computers voor thuisgebruik is duidelijk voorbij, maar om effectief gebruik te maken van multiprocessor computers is het nodig dat programmacode zich kan opsplitsen in threads. Simpelweg opsplitsen in threads is helaas niet voldoende, er moet rekening worden gehouden met het synchroniseren van de threads en de gegevens die worden gedeeld door de threads moeten voorzichtig behandeld worden. Dit alles maakt het implementeren van software met behulp van multithreading zeker een uitdaging. Er bestaan wel methodes die proberen om multithreading makkelijker te maken, maar die hebben weer andere nadelen — zo zijn ze bijvoorbeeld langzaam, slecht portable, erg restrictief, of simpelweg nog niet af. Het concept van de Thread Safety Library, hoewel bij lange na geen complete oplossing, biedt een kleine en simpele doch krachtige set gereedschappen aan waarmee een programmeur kan experimenteren met zijn eigen ideeën over multithreading en zelf bedachte concepten en optimalisaties, binnen elke denkbare omgeving. Dit alles maakt de Thread Safety Library een nuttige toevoeging aan het arsenaal van wapens waarmee multithreading te lijf kan worden gegaan.

Referenties

[1] Arm info center. http://infocenter.arm.com (retrieved August 20 2010).

[2] Atomic library. http://calumgrant.net/atomic/ (retrieved August 20 2010).

[3] atomic_ops library. http://www.hpl.hp.com/research/linux/atomic_ops/index.php4 (retrieved August 20 2010).

[4] Boost threads library. http://www.boost.org/doc/libs/1_43_0/doc/html/thread.html (retrieved August 20 2010).

[5] C standard. http://www.open-std.org/jtc1/sc22/wg14/ (retrieved August 20 2010).

[6] C++ atomic types and operations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2145.html (retrieved August 20 2010).

[7] C++ standard. http://www.open-std.org/jtc1/sc22/wg21/ (retrieved August 20 2010).

[8] Hp 9000 end of product sale frequently asked questions. http://www.hp.com/products1/evolution/9000/faqs.html#2 (retrieved August 20 2010).

[9] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference.* http://developer.intel.com/design/processor/manuals/253666.pdf (retrieved August 20 2010).

[10] Intel technology: Moore's law. http://www.intel.com/technology/mooreslaw/ (retrieved August 20 2010).

[11] Mips32®architecture for programmers volume ii: The mips32®instruction set. http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00086%2D2B%2DMIPS32BIS%2DAFP%2D03.00.pdf (retrieved August 20 2010).

[12] Pa-risc instruction set overview. http://h21007.www2.hp.com/portal/download/files/unprot/parisc20/PA_6_inst_overview.pdf (retrieved August 20 2010).

[13] Powerpc user instruction set architecture. http://download.boulder.ibm.com/ibmdl/pub/software/dw/library/es-ppcbook1.zip (retrieved August 20 2010).

[14] Security focus: Tracking the blackout bug. http://www.securityfocus.com/news/8412 (retrieved August 20 2010).

[15] Sparcv9 architecture manual. `http://www.sparc.org/standards/SPARCV9.pdf` (retrieved August 20 2010).

[16] stdthread library. `http://www.stdthread.co.uk/` (retrieved August 20 2010).

[17] threads library. `http://threads.sourceforge.net/` (retrieved August 20 2010).

[18] Ieee std 1003.1c-1995, 1995.

[19] Edwin Brady and Kevin Hammond. Correct–by–construction concurrency: Using dependant types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102:145–176, 2010.

[20] Richard H. Carver and Kuo-Chung Tai. *Modern Multithreading : Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*. Wiley-Interscience, 2005.

[21] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[22] Henk J.M. Goeman. The arbiter: an active system component for implementing synchronizing primitives. Technical Report, Leiden University 1976; also published in: Fundamenta Informaticae IV.3, 517-534, 1981.

[23] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. pages 123–136, 1996.

[24] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, 1988.

[25] Geoff Koch. Discovering multi-core: extending the benefits of moores law. In *Technology@Intel Magazine. Intel. 2005.*, 2005. `http://www.intel.com/technology/magazine/computing/multi-core0705.pdf` (retrieved August 20 2010).

[26] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[27] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1992.

[28] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[29] W. Richard Stevens. *UNIX Network Programming, Volume 2: Interprocess Communications (2nd Edition)*. Prentice Hall, 1998.

[30] Richard West and Gary T. Wong. Cuckoo: a language for implementing memory- and thread-safe system services, 2005.