Vakoverschrijdend Project
Academiejaar 2015-2016

Unigornel

Maxim Bonnaerens, David Vercauteren en Henri Verroken
Groep 3

Begeleiders:
Bjorn De Sutter, Bart Coppens en Jonas Maebe

Project in het kader van het Vakoverschrijdend Projectvak
in de Bachelor Computerwetenschappen

Inhoudsopgave

1 Inleiding

De voorbije jaren is heel wat vooruitgang geboekt inzake virtualisatie. Een van de meest recente technologieën in dit segment zijn de bibliotheekbesturingssystemen of unikernels. Dit zijn minimale besturingssystemen specifiek ontwikkeld voor één doeleinde. In dit project bouwen we een zo minimaal mogelijk besturingssysteem voor de recent ontwikkelde, maar heel populaire programmeertaal Go. Met een aangepaste versie van de Go compiler kunnen we zo programma's geschreven in Go rechtstreeks uitvoeren op de Xen hypervisor zonder gebruik te maken van een traditioneel besturingssysteem.

1.1 De programmeertaal Go

Go is een vrij jonge statisch getypeerde programmeertaal die ongeveer zes jaar geleden werd ontwikkeld door Google [1]. Sindsdien heeft de taal ongelooflijk aan populariteit gewonnen en heeft ze zich in vele bedrijven genesteld als vervanger van Python, Ruby, PHP, C++, Java, etc [1, 2, 3, 4]. Ze laat haar sterktes zien bij het ontwikkelen van onder meer REST API's, maar ook in grote en complexe softwareprojecten zoals Docker, etcd, Caddy, LXD en nog vele andere.

1.1.1 Een groeiende populariteit

De redenen waarom we Go hebben gekozen zijn talrijk. Een minder technisch maar daarom niet minder belangrijk argument is de almaar groeiende populariteit van Go. Hierdoor is er een grote en zeer actieve gemeenschap, waardoor veel open broncode beschikbaar is. Dit betekent dat er een groot aantal bibliotheken beschikbaar zijn, dewelke onmiddellijk in bestaande projecten, inclusief het onze, kunnen gebruikt worden.

De grote belangstelling voor Go heeft uiteraard ook een rechtstreekse invloed op de grootte van het doelpubliek dat denkbaar interesse zal tonen voor ons project. Aangezien we plannen de resultaten van dit project na afloop publiek te maken, hopen we dat de vruchten van ons werk positief zullen ontvangen worden door de Go- en unikernelgemeenschap.

1.1.2 Technische argumenten

Verder zijn er ook uitgebreide technische argumenten die van Go de ideale programmeertaal maken waarvoor men een unikernelsysteem zou ontwerpen. Een aantal van de belangrijkste eigenschappen die onze doelstellingen vergemakkelijken, worden hieronder opgesomd.

Ten eerste wordt een Go applicatie gecompileerd tot één groot statisch gelinkt uitvoerbaar bestand [5]. Hierin zit alle benodigde functionaliteit die niet wordt aangeleverd door de kernel waarop de applicatie wordt uitgevoerd. Dit betekent dat de uitvoerbare bestanden volledig onafhankelijk kunnen worden verplaatst van de ene machine naar de andere, zonder gebonden te zijn aan de aanwezigheid van gedeelde bibliotheken zoals `libc`, `libm` en `libssl`. We krijgen aldus één binair bestand dat uitgevoerd moet worden op de hypervisor.

Tevens wordt een Go applicatie gecompileerd naar machinecode [5]. Er is geen interpreter nodig om Go code uit te voeren. Dit vereenvoudigt het uitvoeringsproces sterk. Aangezien de Go applicatie rechtstreeks op de processor kan draaien die de virtuele machine monitor ons aanbiedt, moet onze unikernel geen interpreter meezeulen om de applicatie uit te voeren.

Het derde en laatste pluspunt dat we willen bespreken, is de ondersteuning voor vele verscheidene platformen die reeds is ingebouwd in de Go programmeertaal. Elke applicatie geschreven in Go wordt gebouwd door het commando `go build` uit te voeren [6]. De compiler zal automatisch de bestanden vinden die moeten gecompileerd worden, wat leidt tot een enorm eenvoudig compilatieproces. Wanneer men nu hetzelfde programma voor een ander besturingssysteem of zelfs een andere processorarchitectuur wil bouwen, kan men heel eenvoudig `GOOS=linux GOARCH=arm GOARM=5 go build` ingeven, waarna men een uitvoerbaar bestand krijgt voor het gewenste platform [7].

De ondersteunde besturingssystemen en architecturen zijn talloos. Go is op dusdanige wijze geschreven dat platformafhankelijke code zeer eenvoudig te bewerken en toe te voegen is [8]. Hierdoor zal het toevoegen van ons eigen platform, namelijk de Xen hypervisor, niet heel problematisch blijken.

1.2 Virtualisatie

In het huidige softwarelandschap zijn er tal van mogelijkheden om een applicatie uit te rollen in een moderne cloudomgeving. Al deze alternatieven hebben één gemeenschappelijk kenmerk. Ze zijn namelijk allemaal gebaseerd op een of andere vorm van virtualisatie.

1.2.1 Doelen van virtualisatie

De motieven om virtualisatie te gebruiken en de voordelen die hiermee gepaard gaan zijn legio [9]. Bij virtualisatie wordt een virtuele versie gemaakt van een of meerdere systeemmiddelen, zoals hardware, werkgeheugen en opslagruimte. Zo kan een systeemmiddel door meerdere gebruikers of toepassingen aangewend worden, ofschoon isolatie tussen elk van hen gegarandeerd blijft.

Wanneer men een softwaretoepassing uitrolt, is het immers van groot belang dat deze geen rechtstreekse invloed kan uitoefenen op en evenzeer niet kan beïnvloed worden door andere applicaties [10]. Elke applicatie kan immers gecompromitteerd worden van buitenaf en wanneer dit gebeurt, is het van groot belang dat andere applicaties en hun data toch afgeschermd blijven. Hetzelfde geldt indien meerdere onafhankelijke gebruikers zich bedienen van eenzelfde computersysteem. Men wil in dat geval ten allen tijde vermijden dat gebruikers elkaar kunnen hinderen of ongeoorloofd toegang kunnen verkrijgen tot elkaars gegevens.

Een eenvoudige oplossing is voor iedere toepassing een afzonderlijke fysieke machine te voorzien. Dit is echter weinig kostenefficiënt. Hardware is immers kostbaar en dit zorgt ervoor dat dure hardware onderbenut blijft, aangezien applicaties niet onophoudelijk de volledige rekenkracht van een moderne server behoeven. Virtualisatie zorgt er aldus voor dat hardware op een meer efficiënte wijze kan ingezet worden terwijl isolatie gegarandeerd blijft [11].

1.2.2 Klassieke vormen van virtualisatie

Om het onderwerp van dit project goed te kunnen plaatsen is het nodig de ontwikkelingen omtrent virtualisatietechnieken kort te schetsen. Een van de oudste manieren om systeemmiddelen te delen is uiteraard het klassieke besturingssysteem zelf. Een traditioneel besturingssysteem zal verschillende hardwarecomponenten zoals de processor, het werkgeheugen en de netwerktoegang beheren zodat meerdere processen kunnen draaien op hetzelfde systeem. Doorgaans is de behaalde isolatie tussen de verschillende processen echter slecht. Vaak delen processen het bestandssysteem en weten ze van elkaars bestaan [12].

Om betere afscherming te bekomen tussen verschillende toepassingen kan men ervoor opteren om meerdere virtuele machines uit te voeren op dezelfde hardware. Elk van deze virtuele machines draait in dat geval een eigen besturingssysteem of kernel, afgezonderd van de andere besturingssystemen. Een hypervisor of virtuele machine monitor biedt een virtuele versie aan van de onderliggende hardware en beheert de verschillende virtuele machines [9]. Voorbeelden van populaire hypervisors zijn Xen, VMware, KVM en Hyper-V. De isolatie aangeboden door de hypervisor is heel sterk, waardoor de toepassingen goed afgeschermd zijn.

1.2.3 Evoluties in virtualisatie

Wanneer men virtuele machines gebruikt, zal men meestal een of meerdere applicaties in een eigen virtuele machine draaien. Dit wil zeggen dat men hiervoor ook een eigen volwaardig besturingssysteem zal voorzien. Dit is een groot nadeel, aangezien veel functionaliteit gedupliceerd wordt en veel diensten onnodig zijn voor verscheidene applicaties. Dit leidt tot een grote werklast voor de onderliggende hardware. Teneinde deze werklast te verminderen kan men gebruik maken van een meer recente virtualisatietechnologie, namelijk containers.

Bij containers delen de verschillende virtuele machines eenzelfde Linuxkernel. Aan de hand van diensten aangeboden door Linux, zoals het `namespaces`-mechanisme en het `cgroups`-mechanisme, worden verschillende applicaties of applicatiegroepen van elkaar gescheiden zodat het lijkt alsof ze allen op een apart besturingssysteem draaien. Containers beschikken over een eigen volwaardig bestandssysteem, beheren hun eigen netwerkinterfaces en kunnen elkaars processen niet beïnvloeden, maar delen wel dezelfde

kernel [13]. Voorbeelden van containerisatiesoftware zijn onder meer het verouderde OpenVZ en meer recent Docker en LXC/LXD.

We zien dat de virtualisatietechnologie evolueert naar zeer lichte virtualisatievormen waarbij toch nog afdoende isolatie kan gegarandeerd worden. Het delen van een kernel is eenvoudiger, sneller en vereist minder systeemmiddelen dan het opstarten van een volwaardige virtuele machine. Waar het opstarten van zo'n virtuele machine tot enkele minuten kan duren, neemt het opstarten van een container veelal slechts enkele seconden in beslag.

1.3 Unikernels

Een volgende stap in het virtualisatieproces is de ontwikkeling van de zogeheten unikernels [14, 14]. Bij unikernels doet de hypervisor opnieuw zijn intrede om de isolatie tussen de verschillende applicaties te waarborgen, maar wordt het volwaardig besturingssysteem, dat gebruikt wordt bij klassieke op hypervisor gebaseerde virtualisatie, vervangen door een uiterst minimale versie. Dit minimaal bibliotheekbesturingssysteem bevat enkel de meest essentiële zaken die vereist zijn om de applicatie uit te voeren, waardoor de unikernel uitermate gespecialiseerd is.

1.3.1 Xen als hypervisor

Wij hebben Xen gekozen als hypervisor waarvoor we ons unikernelsysteem willen bouwen. Xen is een open broncode virtuele machine monitor die hoofdzakelijk gebruik maakt van paravirtualisatie om te communiceren met de virtuele machines [15].

Waar bij klassieke hardwarevirtualisatie vaak virtuele hardware wordt aangeboden aan de bovenliggende besturingssystemen die niet te onderscheiden is van de echte hardware, wordt bij paravirtualisatie gewerkt met eigen hardwareinterfaces. Opdat dit mogelijk is, moeten besturingssystemen die men op Xen wil uitvoeren, aangepast worden. Zo worden hypercalls gebruikt om met de hypervisor te communiceren en geprivilegieerde instructies uit te voeren.

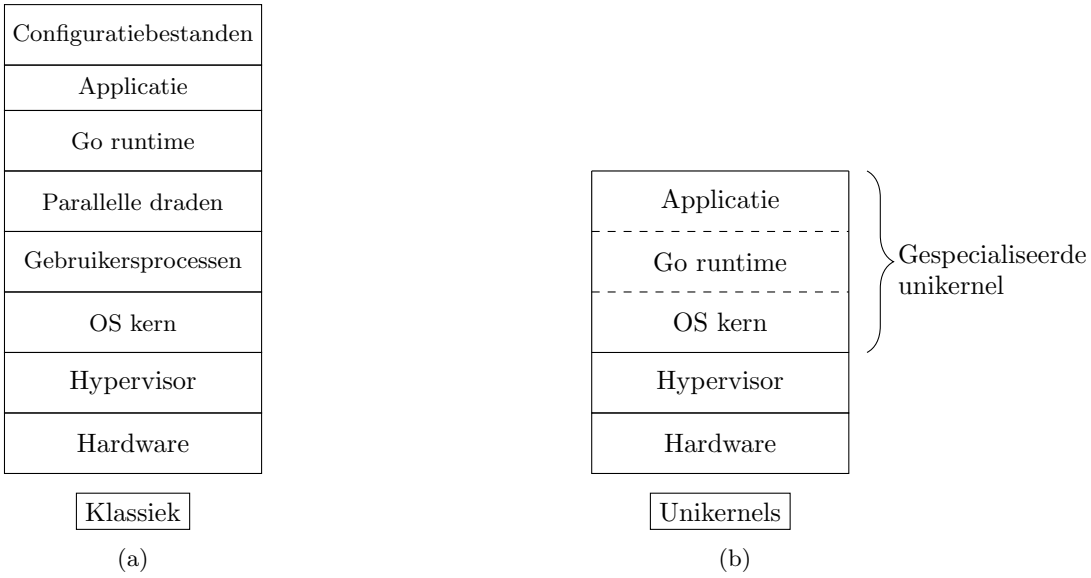
Het gebruik van paravirtualisatie brengt enkele niet te onderschatten voordelen met zich mee. Naast een denkbare prestatietoename, die te danken is aan het vermijden van vele inefficiënte technieken die nodig zijn om volwaardige hardwarevirtualisatie te implementeren, moet een gastbesturingssysteem slechts één type hardware ondersteunen, namelijk de virtuele hardware aangeboden door Xen. De echte drivers voor de vele soorten hardware worden geïmplementeerd in de hypervisorlaag, waardoor de implementatie van gastbesturingssystemen sterk vereenvoudigd wordt.

Xen is een heel populaire hypervisor. Wanneer we kijken naar bestaande unikernelprojecten zien we dat Xen vaak het primair ondersteunde platform is. Dit is het geval bij MirageOS, Rumprun, LING, HaLVM en vele anderen, die ook besproken worden in Sectie 1.3.4. Xen is ook een gevierde virtuele machine monitor in de cloud. Zo wordt Xen gebruikt als hypervisor in de Amazon Elastic Compute Cloud (EC2), IBM SoftLayer en de Rackspace Cloud [16].

1.3.2 Het minimaal besturingssysteem

Wanneer we een volwaardig besturingssysteem willen vervangen door een minimaal exemplaar, moeten we ons afvragen welke diensten dit minimaal besturingssysteem moet aanbieden. Dit hangt echter sterk af van het type applicatie dat we bovenop ons minimaal besturingssysteem willen uitvoeren.

Het klassieke besturingssysteem en de bijhorende distributie bieden vele diensten aan, zoals ondersteuning voor gebruikersprocessen en parallelle draden, een bestandssysteem, een standaardcollectie van uitvoerbare programma's zoals een interactieve shell, en nog vele andere. Men merkt onmiddellijk dat de meeste toepassingen slechts een heel beperkte verzameling van deze diensten nodig heeft. Bij unikernels herleidt men het besturingssysteem tot zijn meest essentiële vorm. Dit wordt ook getoond in Figuur 1.



Figuur 1: In deze figuur tonen we het verschil tussen de klassieke serveraanpak, die te zien is in Figuur 1a en de vernieuwende aanpak die unikernels met zich meebrengen, die te zien is in Figuur 1b. Een klassiek besturingssysteem bevat vele diensten die niet nodig zijn voor elke applicatie. Een unikernel is gespecialiseerd en bevat enkel de nodige functionaliteit.

Zaken die altijd moeten geïmplementeerd worden, zelfs voor de meest elementaire toepassingen, zijn onder meer virtueel geheugenbeheer, ondersteuning voor draden en netwerktoegang opdat men kan communiceren met de buitenwereld. Al deze diensten moeten, samen met systeemoproepen nodig om een applicatie uit te voeren, geïmplementeerd worden in het minimaal besturingssysteem. Welke hoogstnoodzakelijke diensten ons minimaal besturingssysteem moet aanbieden om programma's geschreven in Go uit te voeren en hoe we dit minimale besturingssysteem hebben ontworpen, leest u verder in Sectie 6.3.

1.3.3 Voordelen van unikernels

Unikernels brengen vele voordelen met zich mee [14]. Dit is vooral te danken aan het weglaten van vele diensten uit het besturingssysteem die niet nodig zijn om de bovenliggende applicatie te draaien, zoals hierboven besproken wordt.

Alle onnodige en ongebruikte code aanwezig in een klassiek besturingssysteem zit niet vervat in de unikernel. We krijgen aldus zeer kleine gecompileerde bestanden, die toch alles bevatten om bovenop de hypervisor te kunnen draaien. Bovendien zijn de geheugen- en rekenvereisten van de applicatie geoptimaliseerd. Geheugen en processortijd die voorheen verkwist werden door onnodige componenten van het besturingssysteem, komen nu vrij om meer applicaties te draaien op hetzelfde hardwaresysteem. Unikernels leiden aldus tot een meer adequaat en efficiënt gebruik van de onderliggende hardware.

Voorts leiden al deze optimalisaties tot zeer korte opstarttijden. Gezien de kleine hoeveelheid componenten die moeten geïnitialiseerd worden alvorens de applicatie aan het werk kan, start een unikernel merkbaar sneller op dan een klassiek besturingssysteem. We krijgen opstarttijden in de orde van milliseconden, wat vooral van belang is in het domein van on-demand computing in een elastische cloudomgeving.

Als laatste is het ook belangrijk te vermelden dat unikernels een verhoogde veiligheid met zich meebrengen. Er is niet alleen een veel kleinere aanvalsoppervlakte, maar wanneer een applicatie alsnog gecompromitteerd is, is het onmogelijk conventionele technieken te gebruiken, zoals het opstarten van een shell, omdat deze gewoonweg niet aanwezig zijn in een unikernel.

We zien dus dat het gebruik van unikernels vele voordelen met zich meebrengt. Vooral met het oog op het alsmaar verhoogde niveau van computersysteemvirtualisatie en on-demand computing, ook wel de cloud genoemd, is het zeer belangrijk de toepassingsdensiteit op hardware te verhogen.

1.3.4 Bestaande unikernelprojecten

Wanneer we een project als het onze willen verwezenlijken, kan het lonen om eens te kijken naar en inspiratie te putten uit bestaande gelijkaardige projecten. In dit onderdeel bespreken we twee unikernelprojecten die relevant zijn en kunnen helpen om onze doelen te bereiken, namelijk MirageOS en gorump.

MirageOS [14, 17, 18], ook ontwikkeld voor Xen, is een cloudbesturingssysteem dat vrijwel volledig geschreven is in OCaml. Met MirageOS kan men unikernels ontwerpen geschreven in OCaml, die rechtstreeks draaien op de Xen hypervisor. Het project gelijkt op het onze daar het ook een zo groot mogelijk deel van het minimaal besturingssysteem en de ondersteunende bibliotheken schrijft in een hogere programmeertaal. MirageOS is veruit het meest gevorderde unikernelproject.

Zowaar nog interessanter is gorump [19], een aangepaste versie van Go die draait op Rumprun [20, 21]. Rumprun is een kernel gebaseerd op de NetBSD-kernel waarmee men unikernels kan bouwen voor een platform naar keuze. Rumprun bouwt weinig gespecialiseerde unikernels maar ondersteunt een breed gamma aan toepassingen en talen. Aangezien de ontwikkelaars van gorump Go hebben moeten aanpassen om te draaien op de aangepaste NetBSD-kernel, kunnen we uit dit project veel inspiratie putten.

Er zijn nog vele andere unikernelprojecten, waaronder HaLVM voor Haskellprogramma's, LING voor Erlang en Runtime.js voor JavaScript. Er zijn ook meer algemene projecten zoals OSv en ClickOS die ondersteuning bieden voor allerhande programmeertalen. Al deze projecten zijn nog in volle ontwikkeling en het is duidelijk dat unikernels nog volop in hun kinderschoenen staan.

1.3.5 Mogelijke toepassingen

Het toepassingsgebied van unikernels is bijzonder groot, in het bijzonder dankzij de opkomst van betaalbare elastische cloudclusters. Hierbij wordt rekenkracht aangerekend per tijdseenheid, waardoor het te gepasten tijde af- en aanschakelen van toepassingsinstanties kan leiden tot grote kostenbesparingen. Wegens de flexibiliteit en de geoptimaliseerde systeemvereisten zijn gespecialiseerde unikernels bijgevolg erg geschikt in een dergelijke omgeving [17].

Een mooi voorbeeld hiervan is het *Just-In-Time Summoning of Unikernels*- of *jitsu*-project [22], waarbij een DNS-server unikernels pas zal opstarten wanneer een DNS-aanvraag voor een bepaalde website binnenkomt. De webserver draait dus enkel wanneer nodig. Snelle opstarttijden van unikernels maken dit mogelijk.

Unikernels kunnen uiteraard gebruikt worden voor allerhande type applicaties. Zolang het onderliggend besturingssysteem tegemoet kan komen aan de vereisten van de toepassing, is alles mogelijk. Webapplicaties en databanken, maar ook hooggespecialiseerde netwerktoepassingen zoals VPN's of pakketfilters, zijn allen uitermate geschikt om in een unikernel te verwerken.

2 Analyse van de probleemstelling

In dit onderdeel analyseren we kort de grootste deelproblemen die we zullen tegenkomen bij het bouwen van ons project. Zoals reeds in de inleiding beschreven bouwen we een minimaal besturingssysteem dat een Go applicatie rechtstreeks kan uitvoeren op de Xen hypervisor. Verder moeten we ook de Go compiler en runtime aanpassen zodat die Go code kan compileren voor ons nieuw platform.

2.1 De Go runtime

Wanneer we een systeem willen bouwen dat ondersteuning biedt voor een bepaalde programmeertaal, is het nodig de structuur en de werking van deze programmeertaal te bestuderen. Een Go applicatie is steeds georganiseerd in twee delen, namelijk de Go runtime en de code geschreven door de ontwikkelaar.

De Go runtime implementeert alle functionaliteit die Go aanbiedt. Het bevat een geheugenbeheersysteem met garbage collection, een ingebouwde scheduler en staat in voor de communicatie met het onderliggend besturingssysteem. Opdat de communicatie met onze minimale kernel correct verloopt, zal de runtime aangepast moeten worden.

2.2 De Go compiler

De Go compiler compileert en linkt een programma geschreven in Go naar een uitvoerbaar programma of een statische bibliotheek. De Go compiler bevat de code van de Go runtime, dewelke door de compiler wordt toegevoegd aan het geschreven programma.

Naast de Go runtime moeten we dus ook de Go compiler aanpassen. We moeten ons platform toevoegen als ondersteund compilatiedoel om er zo voor te zorgen dat onze aangepaste versie van de Go runtime gebruikt wordt en correct gecompileerd wordt.

2.3 Het minimaal besturingssysteem

Onze minimale kernel moet rechtstreeks op Xen draaien en enkel de functionaliteit bevatten die nodig is om Go uit te voeren. Als basis voor ons besturingssysteem gebruiken we Mini-OS [23, 24], geschreven in C. Dit is een minimaal besturingssysteem speciaal ontwikkeld om bovenop Xen te draaien. Door Mini-OS aan te passen en functionaliteit toe te voegen bekomen we ons uiteindelijk minimaal besturingssysteem.

Mini-OS bevat reeds vele zaken, zoals een primitieve pagina-allocator, een coöperatieve scheduler, een interface voor de console en functies om te communiceren met de hypervisor. Dit betekent dat wij ons kunnen concentreren op onze primaire taak, namelijk Go draaiende te krijgen bovenop deze kernel.

3 Doelstellingen

De doelstelling van dit project is tweevoudig. Ten eerste bouwen we een minimaal besturingssysteem, gebaseerd op Mini-OS, dat de nodige diensten aanbiedt aan de Go runtime opdat toepassingen geschreven in Go kunnen worden uitgevoerd op de Xen hypervisor. Ten tweede passen we de Go compiler en runtime aan opdat Go kan communiceren met de onderliggende kernel om zo de runtime succesvol uit te voeren. Een unikernel die “Hello World” kan uitprinten naar de console, zien wij als een succesvolle uitkomst van dit project.

Verder is er ook extra functionaliteit die we kunnen implementeren nadat “Hello World” met goed gevolg naar het scherm kan worden afgedrukt. Onze intentie is een deel van deze extra’s te verwezenlijken indien er na het implementeren van de basisfunctionaliteit nog tijd over zou zijn. Voorbeelden van extra’s zijn het ontwikkelen van een TCP/IP stack, het lezen van de console en het beschikbaar stellen van een minimaal read-only bestandssysteem. Wat er uiteindelijk bereikt werd leest u in Sectie 5.2 en Sectie 7.

4 Taakverdeling

Het project vereist kennis over zaken zoals de Xen hypervisor en zijn hypercalls, het compilatieproces in Go en de werking van besturingssystemen. Omdat veel van deze kennis verworven wordt tijdens de loop van het project en omdat deze kennis onmisbaar is voor alle groepsleden, worden bijna alle taken in groep uitgevoerd. Dit houdt in dat er geen expliciete taakverdeling is. Dit wil evenwel niet zeggen dat er geen taken individueel zijn gerealiseerd. Zo zijn er bijvoorbeeld zaken waarvan meerdere implementaties zijn ontwikkeld, elk door een ander groepslid. Op deze manier kunnen meerdere verschillende inzichten met elkaar vergeleken en vervolgens gecombineerd worden. In Sectie 9 wordt dieper ingegaan op de samenwerking.

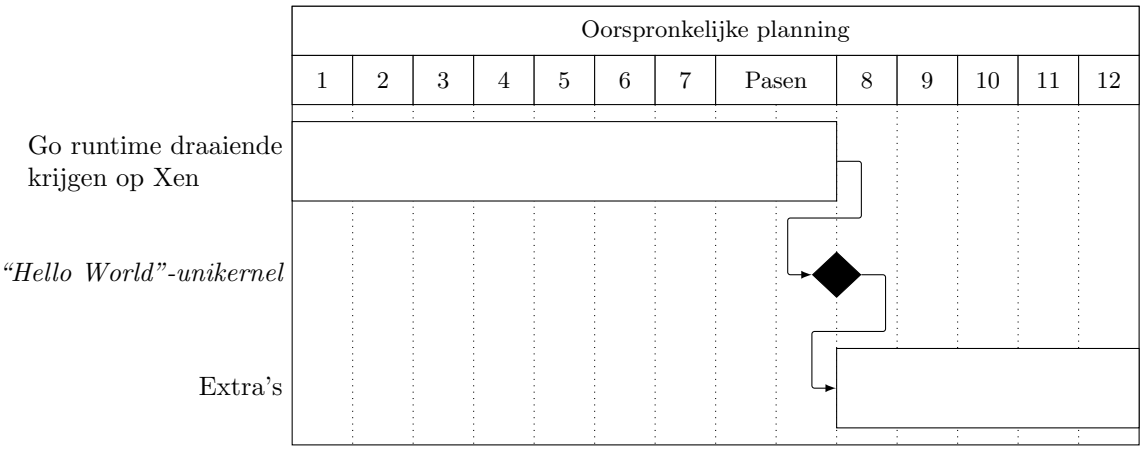
5 Planning

In dit onderdeel bespreken we kort de vooropgestelde planning en het uiteindelijk tijdschema. Het was moeilijk al vanaf de start te weten hoe het project zou verlopen, aangezien we nog niet over de nodige kennis beschikten om te weten welke problemen we zouden tegenkomen bij het verwezenlijken van onze doelstellingen, zoals ook kort wordt besproken in Sectie 4. Hierdoor is de initiële planning weinig gedetailleerd.

5.1 Oorspronkelijke planning

De initiële planning van het project is zoals te zien op Figuur 2 zeer miniem. We onderscheiden hier twee zaken. Enerzijds hebben we de taak om de Go runtime draaiende te krijgen op Xen, wat de nodige vereiste is om tot een "Hello World"-unikernel te komen. Anderzijds is er ook het gedeelte van de extra's, hieronder vallen de zaken die extra functionaliteit bieden aan de unikernel. Een werkende IP stack, is bijvoorbeeld iets wat we in onze doelstellingen hebben opgenomen en iets wat extra functionaliteit voorziet aan onze unikernel. Ook andere zaken die de mogelijkheden van onze unikernel kunnen uitbreiden, behoren tot dit blok van de planning.

De oorspronkelijke planning is zeer minimaal omdat we op voorhand moeilijk konden voorspellen wat er allemaal nodig zou zijn om tot een werkende "Hello World"-unikernel te komen. Ook de hoeveelheid tijd nodig om dit te realiseren was niet gemakkelijk in te schatten. Een werkende unikernel maken is zowat het hoofddoel van dit project en neemt daarom ook het grootste deel van de oorspronkelijke planning in beslag. Wanneer dit bereikt was, kon de functionaliteit uitgebreid worden. Als schatting voor de "Hello World"-unikernel werd het einde van de paasvakantie genomen.



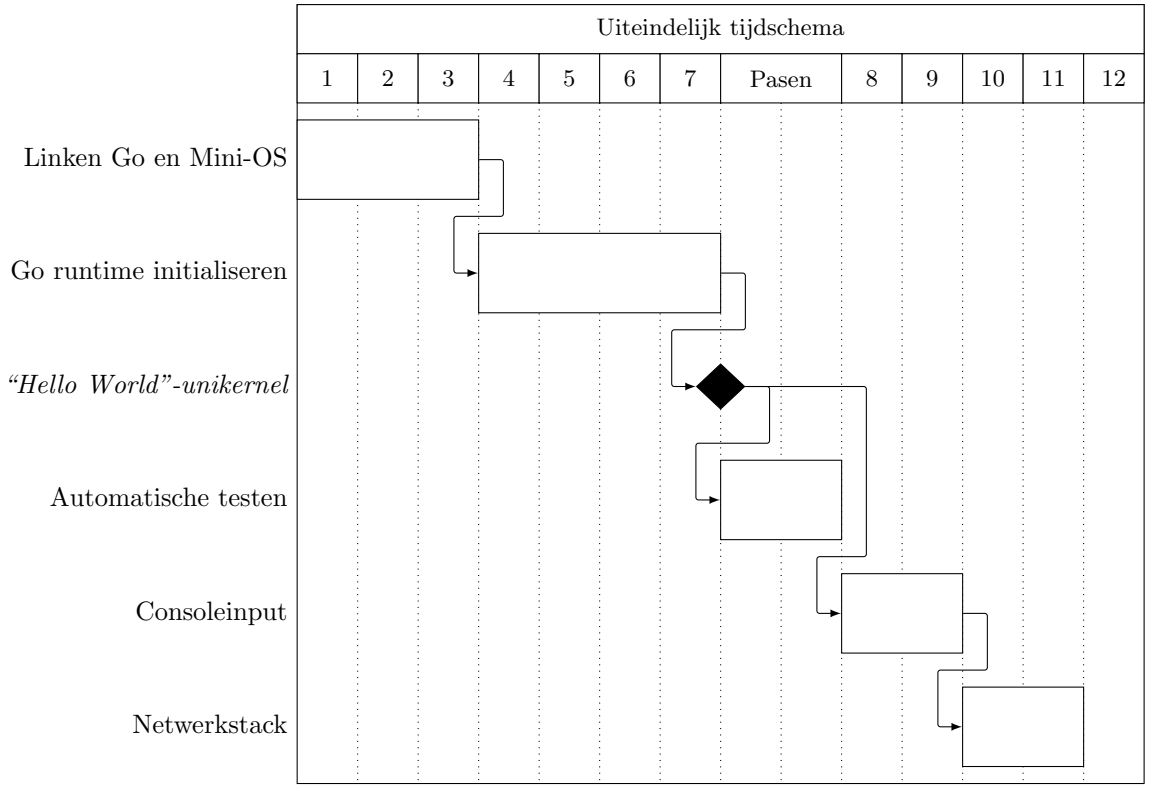
Figuur 2: Oorspronkelijke planning.

5.2 Uiteindelijk tijdschema

De uiteindelijke uitvoering is te zien op Figuur 3. De Go runtime draaiende te krijgen op Xen is hier opgesplitst in twee delen, namelijk Go en Mini-OS linken en de Go runtime initialiseren. Dit zijn de twee belangrijkste zaken die nodig waren om de Go runtime draaiende te krijgen op Xen en zo tot een werkende unikernel te komen.

Concrete extra's die we geïmplementeerd hebben, zijn het lezen van consoleinput en een gedeeltelijke netwerkstack. Consoleinput was initieel geen onderdeel van de doelstellingen, maar werd door de begeleiders gesuggereerd als eerste uitbreidingen. Het verwerken van consoleinput is namelijk een eerste stap richting interactieve programma's. Verder vertoont de implementatie sterke gelijkenissen met wat nodig zou zijn om de netwerkstack te implementeren. Het was dus ook een nuttige voorbereiding voor de implementatie van de netwerkstack.

Verder hebben we na het bereiken van een werkende unikernel ook automatische testen opgezet. Dit hebben we gedaan om er zeker van te zijn dat we bij het implementeren van uitbreiding geen functionaliteit zouden verliezen die we reeds hadden gerealiseerd. De "Hello World"-unikernel werd in week 7 bereikt, iets vroeger dan initieel geschat.



Figuur 3: Uiteindelijk tijdschema.

6 Technische uitvoering

In dit onderdeel beschrijven we uitvoerig hoe we onze doelstellingen bereikt hebben. We geven een chronologisch overzicht van de stappen die we ondernomen hebben om zo duidelijk te illustreren hoe ons project is verlopen.

6.1 Go code linken met Mini-OS

Zoals reeds aangehaald in Sectie 2.3 gebruiken we een minimaal besturingssysteem geschreven in C om daarboven onze applicatie te draaien. We krijgen dus twee afzonderlijke stukken code, namelijk onze Go applicatie en de C code van onze kernel die we moeten samenvoegen tot één geheel zodat ze met elkaar kunnen interageren.

6.1.1 Go code oproepen vanuit C

In een eerste stap moeten we de Go code compileren naar een vorm die kan gebruikt worden vanuit C. Gelukkig bestaat er cgo [25]. Cgo is een ingebouwd mechanisme in de Go programmeertaal dat toelaat om C code te mengen met Go code. Met dit mechanisme is het onder meer mogelijk om zogenaamde C-archives te maken.

Een C-archive is een statische bibliotheek die de gecompileerde Go code bevat samen met de Go runtime. Verder genereert de Go compiler ook een C-headerbestand. De combinatie van de statische bibliotheek en het headerbestand laat ons toe om Go functies vanuit Mini-OS op te roepen [26].

Fragment 1 toont een een applicatie geschreven in Go met, naast een lege `main`-functie die sowieso vereist is in Go, de functie `Main` die enkel iets naar het scherm print. Door het commando `go build -buildmode=c-archive` uit te voeren krijgen we naast een statische bibliotheek ook het door Go gegenereerde headerbestand uit Fragment 2. De C-applicatie uit Fragment 3 wordt door `gcc` gelinkt met de statische bibliotheek. Zo kunnen we de Go-functie effectief aanroepen.

Fragment 1 Voorbeeldprogramma geschreven in Go. De functie `Main` print een string naar het scherm door gebruik te maken van `fmt.Println`. Het `export`-commando zorgt ervoor dat cgo de functie beschikbaar maakt in de statische bibliotheek en het headerbestand.

```
package main

import "C"
import "fmt"

func main() {}

//export Main
func Main(unused int) {
    fmt.Println("Hello World!")
}
```

Fragment 2 Een deel van het gegenereerde headerbestand. Delen die niet relevant zijn, werden weggelaten met de tekens `[...]`.

```
/* Created by "go tool cgo" - DO NOT EDIT. */

[...]

/* Start of boilerplate cgo prologue.  */

#ifndef GO_CGO_PROLOGUE_H
#define GO_CGO_PROLOGUE_H
[...]
typedef long long GoInt64;
typedef GoInt64 GoInt;
[...]
#endif

/* End of boilerplate cgo prologue.  */

#ifdef __cplusplus
extern "C" {
#endif

extern void Main(GoInt p0);

#ifdef __cplusplus
}
#endif
```

Fragment 3 De functie `Main` is in Go geschreven en wordt opgeroepen vanuit Mini-OS, dat geschreven is in C.

```
#include "main.h"

void app_main() {
    Main(0);
}
```

6.1.2 C-archives linken met Mini-OS

Jammer genoeg is het onmogelijk om de methode die hierboven werd toegelicht rechtstreeks toe te passen op de Mini-OS code. De statische bibliotheek die door Go gegenereerd wordt, bevat vele referenties naar functies die op een normaal UNIX-systeem aanwezig zijn, maar die niet geïmplementeerd zijn in Mini-OS. Voorbeelden zijn de functies `abort`, `fprintf`, `malloc` en `free`.

Hoewel we in Sectie 1.1.2 hebben vermeld dat een uitvoerbaar programma geschreven in Go niet afhankelijk is van gedeelde bibliotheken zoals `libc`, ligt dat bij een C-archive anders. Om een goede

samenwerking tussen de C code en de Go runtime te garanderen hebben de Go-auteurs ervoor gekozen toch gebruik te maken van courante C-functies om sommige zaken te implementeren.

Gelukkig is de volledige lijst van functies voldoende klein. We hebben er dus voor gekozen deze functies ongeïmplementeerd te declareren in Mini-OS, zodat de linker niet meer klaagt bij het samenvoegen van onze statische Go bibliotheek en de Mini-OS objectbestanden.

In dit stadium beschikken we aldus over één binair bestand dat alle nodige code bevat. Dit bestand kunnen we laten uitvoeren op de Xen hypervisor, maar het programma zal uiteraard crashen. De `Main`-functie zal wel degelijk betreden worden, maar aangezien de Go runtime nog niet is geïnitialiseerd zal de unikernel snel crashen. Een volgende stap is de Go runtime aan de praat krijgen.

6.2 De Go runtime initialisatiefunctie

Zoals eerder besproken is het onvoldoende om enkel de hoofdfunctie van het Go programma op te roepen omdat de Go runtime eerst moet geïnitialiseerd worden. Deze staat in voor geheugenbeheer, heeft een ingebouwde scheduler en zorgt voor de link tussen het onderliggend besturingssysteem en de applicatie.

De functie die verantwoordelijk is voor het initialiseren van de runtime is `rt0_go` [27], dewelke wordt opgeroepen in een platformafhankelijke wrapper. In ons geval heet die wrapper `_rt0_amd64_unigornel_lib_go`, waarvan de implementatie is te vinden in Fragment 4.

Fragment 4 Een wrapper rond de platformonafhankelijke functie `rt0_go` die de Go runtime initialiseert. De functie geeft de argumenten en onmgevingsvariablen mee die gedefinieerd zijn in `sys_argc` en `sys_argv`. Deze symbolen zijn gedefinieerd in Mini-OS. Het fragment werd geschreven in Go assembly, wat door Go bijna rechtstreeks compileerd wordt naar machinecode voor het doelplatform. De registers `DI`, `SI` en `AX` worden bij een x86-64 doelplatform automatisch omgezet naar `rdi`, `rsi` en `rax`. `SB` wordt gebruikt om naar globale symbolen te verwijzen. `TEXT` wordt gebruikt om nieuwe functies te declareren.

```
TEXT _rt0_amd64_unigornel_lib_go(SB),NOSPLIT,$0
        MOVQ    sys_argc(SB), DI
        LEAQ    sys_argv(SB), SI
        MOVQ    $runtime.rt0_go(SB), AX
        CALL    AX
        RET
```

Wanneer men een C-archive bouwt, wordt deze functie in een aparte sectie geplaatst van de statische bibliotheek. In deze sectie vindt men de zogeheten `libc`-constructors terug. Wanneer een applicatie op een normaal besturingssysteem wordt uitgevoerd, zal de `libc`-bibliotheek deze functies automatisch aanroepen. Op die manier wordt de Go runtime correct geïnitialiseerd alvorens de klassieke `main`-functie van het C-bestand door `libc` wordt opgeroepen.

Uiteraard beschikken wij niet over een `libc`-implementatie in Mini-OS. Wij moeten deze functie dus manueel oproepen. Met behulp van het hulpprogramma `objcopy` zorgen we ervoor dat het symbool in de statische bibliotheek toegankelijk wordt vanuit andere C code. Zo kunnen we, naast de eerder besproken `Main` functie, ook deze functie vanuit Mini-OS oproepen.

6.3 De weg naar “Hello World!”

In deze fase wordt de functie die Go moet initialiseren correct opgeroepen. Uiteraard zal deze functie crashen aangezien het diensten van het onderliggend besturingssysteem verwacht die niet aanwezig zijn. In het verloop van dit onderdeel bespreken we de verschillende problemen die we tegenkwamen en hoe we deze hebben opgelost. We bespreken hoe we TLS, geheugenallocatie, draden, synchronisatieprimitieven en verscheidene systeemoproepen hebben geïmplementeerd om zo tot een werkende “Hello World”-unikernel te komen.

6.3.1 Thread-local storage

Wanneer de initialisatiefunctie van de Go runtime wordt opgeroepen, probeert Go het `fs`-register te gebruiken. Go verwacht dat dit register verwijst naar een adres in het geheugen dat kan gebruikt worden om variabelen die lokaal zijn aan een draad op te slaan. Dit heet dan de thread-local storage of TLS [28]. Doorgaans wordt dit register automatisch opgezet wanneer een besturingssysteem zoals Linux de applicatie opstart.

Deze functionaliteit is niet aanwezig in Mini-OS. Bijgevolg moesten we er zelf voor zorgen dat het `fs`-register correct geïnitialiseerd wordt, zodat Go dit kan gebruiken voor de TLS. Hiervoor moesten we op correcte wijze segmentatie voorzien in Mini-OS. We moesten op correcte wijze de x86-64 GDT opvullen met geldige segment descriptors door gebruik te maken van hypercalls. Zo konden we het `fs`-register vullen met de juiste segment selector elke keer we van draad wisselden. Voor de daadwerkelijke implementatie verwijzen we naar de code.

6.3.2 Geheugenallocatie

De volgende plaats waar de Go runtime er niet in slaagt zich te initializeren is bij het opzetten van het geheugenbeheer en de garbage collector. In deze sectie gaan we in op de oplossing van het probleem rond geheugenallocatie. Kort samengevat zal enerzijds de Go runtime bij het opstarten van een applicatie 512 GB aan virtuele adresruimte willen reserveren. Anderzijds is de beschikbare virtuele adresruimte in Mini-OS gelijk aan de totale hoeveelheid fysiek geheugen dat het krijgt van Xen. Deze twee voorwaarden zijn niet te combineren.

We botsen hier dus op een beperking van Mini-OS tegenover een volwaardig besturingssysteem. De beste oplossing is om Mini-OS te voorzien van een omvangrijker geheugenallocatiemechanisme zoals ook terug te vinden is in hedendaagse besturingssystemen. Dit zou betekenen dat virtuele en pseudo-fysieke adressen niet langer één op één op elkaar worden afgebeeld in de paginatabellen. Zo kan tijdens de uitvoering van het systeem op dynamische wijze geheugen worden gealloceerd of vrijgegeven door Go.

Het implementeren van een volwaardige pagina-allocator in Mini-OS is een tijdrovende taak. Wij hebben ervoor gekozen de geheugenvereisten van Go sterk te verminderen. Zo zal Go slechts 64 MB proberen alloceren liever dan 512 GB. Deze 64 MB wordt onmiddellijk gealloceerd in Mini-OS en volledig gereserveerd voor Go. Deze oplossing neemt niet veel tijd in beslag en laat toch toe eenvoudige programma's, zoals een "Hello World"-applicatie, uit te voeren. Na onze kleine aanpassing aan de Go runtime, restte ons alleen nog een minimale `mmap`-implementatie te voorzien in Mini-OS.

Onze oplossing is echter een kortetermijnoplossing. De langetermijnvisie is om niet Go maar Mini-OS zodanig aan te passen dat er volwaardige geheugenallocatie plaatsvindt, zoals ook wordt besproken in Sectie 10.2.

6.3.3 Draden en synchronisatie

Mini-OS bevat een coöperatieve scheduler die verschillende draden kan beheren en uitvoeren. Deze scheduler voldoet aan de vereisten van Go. We moesten enkel nog de nodige `pthread`-functies implementeren die gebruikt werden door Go om draden te beheren.

Wat wel van de grond af aan moest geïmplementeerd worden, waren de synchronisatieprimitieven. De Go runtime bevat een eigen semafoorimplementatie gebaseerd op het futexmechanisme [29] van Linux. Dit futexmechanisme hebben wij nagebootst in Mini-OS. Na de implementatie van dit mechanisme moesten we opnieuw enkel de Go runtime aanpassen opdat die de juiste Mini-OS functies zou aanroepen.

6.3.4 Hello World!

Na de implementatie van bovenstaande subsystemen bleven nog een aantal kleinere, desalniettemin onontbeerlijke systeemoproepen over die we op eenvoudige wijze konden implementeren. De belangrijkste worden hieronder opgesomd. Voor een volledige lijst verwijzen we naar de code.

Ten eerste werden alle functies die verbonden zijn met UNIX-signalen niet geïmplementeerd. Deze worden in een normale omgeving gebruikt om primitieve communicatie tussen processen te voorzien.

Aangezienr slechts één proces op onze unikernel wordt uitgevoerd, is dit subsysteem gewoonweg overbodig.

Verder werden de systeemoproepen `usleep` en `now` geïmplementeerd, die respectievelijk een draad minstens een bepaald aantal microseconden doet slapen en de huidige systeemtijd teruggeeft. Uiteraard werd als laatste ook de `write`-functie geïmplementeerd om naar de console te printen.

Op dit ogenblik waren we in staat de gewenste "Hello World"-unikernel succesvol uit te voeren op de Xen hypervisor. De Go code is te vinden in Fragment 5. De consoleoutput die we verkrijgen bij het uitvoeren van de unikernel op Xen is te vinden in Fragment 7 in Appendix A en wordt daar uitvoering besproken.

Fragment 5 De code van de Go applicatie, die gebruikt werd om de "Hello World"-unikernel te bouwen. De tekst wordt succesvol geprint naar het scherm.

```
package main

import "C"
import "fmt"

func main() {}

//export Main
func Main(unused int) {
        fmt.Printf("Hello World!\n")
}
```

6.4 Consoleinput

Nu we een minimaal werkend unikernelsysteem hadden ontwikkeld, wouden we zo snel mogelijk een netwerkstack toevoegen. Onze begeleiders raadden ons echter aan eerst te proberen lezen van de console. Dit zou hoogstwaarschijnlijk gemakkelijker zijn, maar zou ons wel verder op weg helpen het I/O-mechanisme in Xen-gasten te leren.

Tijdens het implementeren van deze functionaliteit leerden we over event channels, ring buffers en grant tables [30]. Dit zijn allen mechanismen in Xen die het mogelijk maken data uit te wisselen met de hypervisor en andere gasten. De details besparen we u.

Uiteindelijk bleek het dat reeds een groot deel van de leesfunctionaliteit in Mini-OS beschikbaar was. Na enkele aanpassingen in Mini-OS en een implementatie van de `read`-systeemoproep konden we op eenvoudige wijze consoleinput aanvaarden van de gebruiker. De Go code van deze toepassing wordt getoond in Fragment 6.

Fragment 6 Een Go programma dat tekst leest van de console. Daarna wordt de string opnieuw uitgeprint naar de console. Deze code kan succesvol worden uitgevoerd bovenop ons minimaal besturingssysteem.

```
package main

import "C"
import (
        "bufio"
        "fmt"
        "os"
        "strings"
)

func main() {
        Main(0)
}

//export Main
func Main(unused int) {
        reader := bufio.NewReader(os.Stdin)

        fmt.Print("Hello, what's your name? ")
        name, _ := reader.ReadString('\n')
        fmt.Printf("Hello, %s\n", strings.TrimSpace(name))
}
```

6.5 Netwerkstack

Als kers op de taart besloten we een primitieve netwerkstack gedeeltelijk te implementeren. Uiteindelijk zijn we erin geslaagd met goed gevolg te pingen naar onze unikernel. Dit wil zeggen dat een volledige ethernet-, ARP-, en IPv4-laag werd geïmplementeerd, samen met een onvolledige ICMP-laag.

Wanneer ethernetpakketten binnenkomen in Mini-OS worden deze doorgegeven aan `go-tcpip`. Dit is onze eigen bibliotheek, geschreven in Go, waarin alle bovenliggende lagen op modulaire wijze werden geïmplementeerd. Op gelijkaardige wijze worden ethernetframes vanuit `go-tcpip` doorgegeven naar Mini-OS, waar ze worden doorgegeven aan de Hypervisor.

Het zenden en versturen van ethernetframes en de communicatie met Xen die hiermee gepaard gaat, is volledig geschreven in C en was al deels aanwezig in Mini-OS. Voor een grondige analyse van onze oplossing verwijzen we naar de code in Mini-OS en `go-tcpip`.

7 Resultaten

Het doel van ons project was om een unikernel te maken die Go applicaties ondersteunt. Ons eerste resultaat was een werkende "Hello World"-unikernel die correct kon uitgevoerd worden. Nadat we ons eerste resultaat hadden bereikt, leek dit een goed moment om testen te schrijven die garanderen dat dit gedrag behouden blijft voor de rest van het project. Meer uitleg over deze testen kan u lezen in Sectie 8.

Aangezien we met deze doelstelling de basisfunctionaliteit van de unikernel hadden geïmplementeerd, konden we de resterende tijd nog gebruiken om extra functionaliteit toe te voegen. Uiteindelijk zijn we erin geslaagd ook het lezen van de console mogelijk te maken. Verder hebben we de netwerklaag gedeeltelijk geïmplementeerd. We kunnen ping requests naar onze unikernel sturen, waarna deze op gepaste wijze antwoordt. Dit wil zeggen dat we op correcte wijze de ethernetlaag, de ARP-laag, de IPv4-laag en een gedeeltelijke ICMP-laag hebben geïmplementeerd.

In het algemeen kunnen we zeggen dat we onze doelstelling hebben bereikt. We hebben al in Sectie 3 aangegeven dat de extra functionaliteit zoals het ontwikkelen van een TCP/IP stack, het lezen van de console en het beschikbaar stellen van een minimaal read-only bestandssysteem, enkel zou verwezenlijkt worden indien we tijd over hadden. Zoals we hierboven hebben aangegeven zijn we er uiteindelijk in geslaagd om een deel van deze extra functionaliteit te realiseren. Hieruit kunnen we dus concluderen dat we ons project tot een goed einde hebben gebracht.

8 Testen en testmogelijkheden

Zoals we reeds besproken hebben in Sectie 5.2, hebben we automatische testen opgezet van zodra de "Hello World"-unikernel werkte. Zo konden we verzekeren dat correct functionerende componenten van ons project geen regressie zouden vertonen bij het implementeren van de extra's. Hiervoor hebben we enkele eenvoudige integratietesten gemaakt die automatisch worden uitgevoerd door een Jenkinssetup.

8.1 Jenkins en GitHub

Jenkins is een buildserver, die software bouwt, testen uitvoert en resultaten op een overzichtelijke wijze voorstelt aan ontwikkelaars. Onze Jenkinsinstantie wordt gehost op een virtuele machine en is gelinkt aan een fysieke server met een werkende Xeninstallatie. Jenkins werd automatisch gelinkt met onze code op GitHub waardoor pull requests vanzelf worden getest, net als de verschillende masterbranches van ons project.

Bij iedere aanpassing bouwt Jenkins vooreerst de aangepaste Go compiler die ook de bijgewerkte runtime bevat. Vervolgens compileert het enkele integratietesten geschreven in Go, linkt ze met ons minimaal besturingssysteem tot een unikernel en voert ze uit op de Xeninstallatie. De consoleoutput wordt nagekeken om te controleren of alles nog werkt naar behoren. De testen worden gebouwd, uitgevoerd en gecontroleerd door een combinatie van shell- en pythonscripts.

8.2 Beschikbare integratietesten

In dit onderdeel bespreken we kort de beschikbare integratietesten. Elke integratietest keurt een of meerdere subsystemen van onze unikernel. Merk op dat er geen integratietest beschikbaar is om de nog onafgewerkte netwerkfunctionaliteit te testen.

Hello World Dit is de eerste test die we schreven voor de meest elementaire unikernel die we kunnen bedenken. De test probeert de tekst `Hello World!` uit te printen naar de console van de virtuele machine.

Sleep and time Deze unikernel test het tijd- en schedulersysteem van ons minimaal besturingssysteem. Het verzekert dat draden op correcte wijze inslapen en ontwaken. Verder controleert het ook of Go correcte toegang heeft tot de systeemtijd.

Read from console Lezen van de console is een van de extra functionaliteiten die we geïmplementeerd hebben. Het is dan ook aangeraden om te verzekeren dat dit subsysteem in de toekomst blijft werken. De test leest input van de console en print ze opnieuw uit naar het scherm.

Call C from Go Deze test verzekert dat we C-functies van ons besturingssysteem kunnen oproepen vanuit Go door gebruik te maken van `cgo`. Dit garandeert dat we op correcte wijze de link maken tussen Go en onze minimale kernel.

9 Samenwerking

In Sectie 4 hebben we reeds aangehaald dat er geen expliciete taakverdeling was en dat de taken hoofdzakelijk als groep werden uitgevoerd. In dit gedeelte bespreken we kort de samenwerking, organisatie en communicatie binnen de groep, die kortom uitstekend was.

Het verwezenlijken van de technische uitvoering gebeurde grotendeels tijdens bijeenkomsten in groep. Dit waren vaak sessies waarbij we fysiek samenkwamen, maar ook online sessies via Skype waren meer dan wekelijkse kost. Zo konden we gemakkelijk nieuwe ontdekkingen tonen maar ook moeiteloos problemen en mogelijke oplossingen bespreken. De samenwerking tijdens het project verliep bijzonder vlot omdat we al eerder samen aan projecten gewerkt hadden, waardoor we al heel vlot konden communiceren en coöpereren.

Uiteraard werd er ook thuis op individuele basis aan het project gewerkt. In dit geval werden kleine deeltaken op voorhand verdeeld en toegewezen. Nu en dan probeerden we zelfs allen hetzelfde probleem

individueel op te lossen. Zo konden we met drie tegelijkertijd nieuwe informatie vinden en verwerken om zo snel mogelijk een openstaand probleem op te lossen. Bovendien kregen we op die manier soms meerdere implementaties van een subsysteem, waaruit we de beste konden kiezen. Een mooi voorbeeld hiervan is het lezen van de console. Hiervoor waren er twee oplossingen beschikbaar, waarvan we er één kozen.

Verder kwamen we ook wekelijks samen met onze begeleiders waarbij we aan de hand van een korte presentatie of demo toonden wat gedurende die week gerealiseerd was. Tijdens deze bijeenkomsten werden de problemen die we tegengekomen waren aangekaart en werd over onze aangebrachte oplossingen geredeneerd. De hulp en hints die we van tijd tot tijd toegereikt kregen, waren onmisbaar voor het goed verloop van ons project.

Aangezien dit project ons gebood zeer veel nieuwe kennis te vergaren over zaken zoals Xen en virtualisatie, het compilatieproces van Go en de Go runtime, hebben we ervoor gezorgd dat deze informatie goed werd gedeeld. We hebben ervoor gekozen om met een wiki als kenniscentrum te werken waar iedereen informatie op kon plaatsen. Op deze manier werd alles op eenzelfde plaats bewaard en is belangrijke informatie herhaaldelijk en onmiddellijk toegankelijk.

10 Openstaande problemen en tekorten

Als we terugblikken op het project en de beslissingen die we hebben gemaakt, kunnen we stellen dat we geen fatale fouten hebben gemaakt waarvan we de gevolgen nog steeds kunnen zien. Er zijn wel enkele zaken die anders hadden gekund en het uiteindelijk resultaat ten goede zouden komen.

10.1 Functionaliteit overzetten van Mini-OS naar Go

Het grootste deel van de code geschreven tijdens het project, waren aanpassingen in Mini-OS. Toch was het onze bedoeling zoveel mogelijk te implementeren in het typeveilige Go. Er zijn delen die geïmplementeerd zijn in C, omdat dit nu eenmaal sneller was dan een implementatie te voorzien in Go. Deze delen zouden eigenlijk kunnen herschreven worden in Go. Een voorbeeld hiervan is het futex-mechanisme dat wordt besproken in Sectie 6.3.3.

10.2 Optimalisaties van het geheugenbeheer

Bij het geheugenbeheer hebben we al aangegeven dat Go een zeer grote hoeveelheid virtueel geheugen alloceert. De kortetermijnoplossing die we hiervoor hebben gekozen past de vereiste van Go aan zodat deze compatibel is met Mini-OS. Dit kan u lezen in Sectie 6.3.2. Een langetermijnoplossing zou zijn om het omgekeerd aan te pakken en Mini-OS aan te passen om te voldoen aan de vereisten van Go. Dit zouden we kunnen realiseren door een volwaardige pagina-allocator te implementeren in Mini-OS. Aangezien dit een taak is die wel wat extra tijd vereist, zijn we hier niet toe gekomen. Om tot een productieklaar unikernelsysteem te komen is dit echter zeker een conditio sine qua non.

10.3 Onafgewerkte netwerkstack

De netwerkstack die we geïmplementeerd hebben, wordt kort beschreven in Sectie 6.5. Deze netwerkstack bevat enkel een ethernet-, ARP-, IPv4- en ICMP-laag. De netwerkstack is dus verre van afgewerkt daar de belangrijkste protocolen, namelijk TCP en UDP, ontbreken. Verder is deze netwerkstack nog niet beschikbaar vanuit de Go runtime, wat wel een vereiste is voor een productieklaar unikernelsysteem.

10.4 Meer functionaliteit

Naast bovenstaande tekortkomingen gerelateerd aan functionaliteit die we zelf geïmplementeerd hebben, zijn er ook diensten die nog niet door Mini-OS worden aangeboden, maar wel handig zouden zijn voor ons project. Zo gebruikt Mini-OS momenteel een coöperatieve scheduler, die we zouden kunnen vervangen door een preëmptieve scheduler om zo meer controle te krijgen over het wisselen van draden. Een ander

voorbeeld is de ondersteuning voor SMP. Hierbij wordt ondersteuning geboden voor meerdere virtuele CPU's, waardoor draden tegelijkertijd op meerdere processoren kunnen worden uitgevoerd.

11 Besluit

We besluiten dat het mogelijk is een unikernel te maken die Go applicaties ondersteunt. Het begin van dit project was zeer moeilijk, aangezien de kennis vereist om de eerste stappen te zetten ons ontbrak. Uiteraard maakt dit het project ook erg interessant. We leerden immens veel bij en gaandeweg konden we deze nieuw verworven kennis gebruiken om vele praktische problemen op te lossen en een complex systeem te realiseren.

Wat we ook zeker niet over het hoofd mogen zien is dat wij een nuttige bijdrage hebben geleverd aan de unikernelgemeenschap. Daarom hebben we besloten op het einde van dit project de code publiek te maken in de hoop dat anderen inspiratie kunnen putten uit ons werk of kunnen bijdragen aan het vervolg van dit project.

12 Referenties

[1] Rob Pike. Go at Google: Language Design in the Service of Software Engineering, 2012. SPLASH 2012.

[2] John Graham-Cumming. Go at CloudFlare, 2012. URL `https://blog.cloudflare.com/go-at-cloudflare/`.

[3] Kelsey Hightower. Go at CoreOS, 2014. URL `https://blog.gopheracademy.com/birthday-bash-2014/go-at-coreos/`.

[4] Patrick Lee. Open Sourcing Our Go Libraries, 2014. URL `https://blogs.dropbox.com/tech/2014/07/open-sourcing-our-go-libraries/`. Dropbox.

[5] The Go Programming Language: Documentation, . URL `https://golang.org/doc/`.

[6] About the go command. URL `https://golang.org/doc/articles/go_command.html`.

[7] The Go Programming Langauge: Command go. URL `https://golang.org/cmd/go/`.

[8] The Go Programming Language: Build Constraints, . URL `https://golang.org/pkg/go/build/`.

[9] Daniel A Menascé. Virtualization: Concepts, Applications, and Performance Modeling. In *Int. CMG Conference*, pages 407–414, 2005.

[10] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[11] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. *HP Labs Tec. Report*, 2007.

[12] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating System Concepts*, volume 4. Addison-Wesley Reading, 1998.

[13] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[14] Anil Madhavapeddy and David J Scott. Unikernels: The Rise of the Virtual Library Operating System. *Communications of the ACM*, 57(1):61–69, January 2014.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[16] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud Computing: An Overview. In *Cloud Computing*, pages 626–631. Springer, 2009.

[17] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472. ACM, 2013.

[18] A programming framework for building type-safe, modular, systems. URL https://mirage.io.

[19] Gorump on GitHub. URL https://github.com/deferpanic/gorump.

[20] Justin Cormack. The Rump Kernel: A Tool for Driver Development and a Toolkit for Applications.

[21] Rump Kernels on GitHub. URL https://github.com/rumpkernel.

[22] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, 2015.

[23] Mini-OS. URL http://wiki.xenproject.org/wiki/Mini-OS.

[24] Satya Popuri. A tour of the Mini-OS kernel. URL https://www.cs.uic.edu/~spopuri/minios.html.

[25] Andrew Gerrand. C? Go? Cgo!, 2011. URL https://blog.golang.org/c-go-cgo.

[26] Svett Ralchev. Sharing Golang packages to C and Go, 2015. URL http://blog.ralch.com/tutorial/golang-sharing-libraries/.

[27] Siarhei Matsiukevich. Golang Internals, Part 5: The Runtime Bootstrap Process, 2015. URL http://blog.altoros.com/golang-internals-part-5-runtime-bootstrap-process.html.

[28] Siarhei Matsiukevich. Golang Internals, Part 3: The Linker, Object Files, and Relocations, 2015. URL http://blog.altoros.com/golang-internals-part-3-the-linker-and-object-files.html.

[29] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *AUUG Conference Proceedings*, page 85. AUUG, Inc., 2002.

[30] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 1st edition, 2007.

A Output van de “Hello World”-unikernel

Fragment 7 toont de consoleoutput bij het uitvoeren van de “Hello World”-unikernel. We zien de opstartinformatie die Xen meegeeft aan Mini-OS. Daarna initialiseert het geheugenbeheer zich, alsook de timerinterface, de console, het Xen-grantmechanisme en als laatste de scheduler. Er worden verschillende draden gestart. De draad `main` voert de `Main`-funtie uit van het Go-programma zoals uitgelegd in Fragment 5. De draad `initialize_go` voert de functie uit die zorgt voor de initialisatie van de Go runtime uit Sectie 6.2. Er worden enkele `mmap`-operoepen gedaan om geheugen te alloceren, waaarna `Hello World!` uiteindelijk wordt afgedrukt. Daarna wordt de unikernel afgesloten.

Fragment 7 De consoleoutput bij het uitvoeren van de "Hello World"-unikernel.

```
Xen Minimal OS!
  start_info: 0000000000220000(VA)
    nr_pages: 0x10000
  shared_inf: 0xb6f25000(MA)
     pt_base: 0000000000223000(VA)
nr_pt_frames: 0x5
    mfn_list: 00000000001a0000(VA)
   mod_start: 0x0(VA)
     mod_len: 0
       flags: 0x0
    cmd_line:
       stack: 000000000015a8c0-000000000017a8c0
MM: Init
      _text: 0000000000000000(VA)
     _etext: 000000000008f5e2(VA)
   _erodata: 000000000012d000(VA)
     _edata: 000000000012eee0(VA)
stack start: 000000000015a8c0(VA)
       _end: 000000000019adf8(VA)
  start_pfn: 22b
    max_pfn: 10000
Mapping memory range 0x22b000 - 0x10000000
setting 0000000000000000-000000000012d000 readonly
skipped 1000
MM: Initialise page allocator for 2a9000(2a9000)-10000000(10000000)
MM: done
Demand map pfns at 10001000-0000002010001000.
Initialising timer interface
Initialising console ... done.
gnttab_table mapped at 0000000010001000.
Initialising scheduler
Thread "Idle": pointer: 0x000000000032d070, stack: 0x0000000000330000
Thread "xenstore": pointer: 0x000000000032d0d8, stack: 0x0000000000340000
xenbus initialised on irq 1 mfn 0x106b8f
Thread "shutdown": pointer: 0x000000000032d140, stack: 0x0000000000350000
go_main.c: app_main(000000000017a8c0)
Thread "main": pointer: 0x000000000032d220, stack: 0x0000000000360000
Thread "initialize_go": pointer: 0x000000000032d300, stack: 0x0000000000380000
mmap(addr=000000c000000000,len=0x4412000,prot=0x0, flags=0x1002)
mmap(addr=0000000000000000,len=0x40000,prot=0x3, flags=0x1002)
mmap(addr=0000000004410000,len=0x100000,prot=0x3, flags=0x1002)
mmap(addr=0000000004408000,len=0x8000,prot=0x3, flags=0x1002)
mmap(addr=0000000004000000,len=0x1000,prot=0x3, flags=0x1002)
mmap(addr=0000000000000000,len=0x10000,prot=0x3, flags=0x1002)
Thread "pthread-0": pointer: 0x000000000032d488, stack: 0x000000000003a0000
mmap(addr=0000000000000000,len=0x40000,prot=0x3, flags=0x1002)
Thread "pthread-1": pointer: 0x000000000032d5a0, stack: 0x000000000003b0000
Hello World!
```