

Erasmus Hogeschool Brussel
Toegepaste Informatica

Bachelor Toegepaste Informatica

Grafische simulatie van een proces-scheduler

Geschreven door

An Homan

Promotor: P. Van Laethem

Brussel, 2008

Contents

Contents	i
1 Inleiding	3
1.1 De Motivering	3
1.2 De Problemen	3
1.3 Wijze van het onderzoek	4
1.4 De Begrenzingsen	4
2 Situering	5
3 Onderzoek	7
3.1 Wat is het doel van cpu-scheduling?	7
3.2 Algoritme	8
3.3 First-come-first-served (FCFS)	8
3.4 Shortest-Job-First (SJF)	8
3.5 Prioriteit-scheduling	9
3.6 Round-Robin-Scheduling	9
3.7 Multilevel-queue-scheduling	9
3.8 Multilevel-feedback-queue-scheduling	10
3.9 Preemptive scheduling VS niet-preemptive scheduling	10
3.10 Dispatcher	10
3.11 Multiprocessor-scheduling	11
3.12 Thread-scheduling	11
4 Klassediagram en usecases	13
4.1 Proces	13
4.2 Queue	14
4.3 Algoritme	14
4.4 Burst	14
4.5 CPU	15

4.6	Timer - Controller	15
4.7	Scheduler	15
4.8	Usecases	15
5	Panels Aanmaken	17
5.1	Panels Aanmaken	17
5.2	Splitpanel	17
5.3	Proces Aanmaken	17
5.4	Burst Aanmaken	18
5.5	Overzicht - Algoritme kiezen	18
5.6	Simulatie	18
6	Problemen en hun oplossingen	21
6.1	Klasse Proces	21
6.2	Klasse Burst	21
6.3	Klasse Queue	22
6.4	Klasse Controller (Timer)	22
6.5	De form (FrmEindwerk)	22
6.6	DataSource	22
6.7	Procesnummer	23
6.8	Combobox op geen enkele keuze zetten	23
6.9	Onpaint	23
6.10	Preemptief	23
6.11	Property	24
7	Mogelijke Uitbreidingen	25
7.1	Algoritmes	25
7.2	Simulatie	25
8	Besluit	27

Dankwoord

De realisatie van dit werk heeft veel inspanning gevraagd. En dit niet enkel van mij, maar van heel mijn omgeving. Als iets niet lukte, ik begreep iets niet, het ging niet zoals het hoorde te gaan, kon ik bij deze mensen terecht. Daarom lijkt het me de normaalste zaak dat ik deze mensen dan ook bedank om samen met mij deze weg te nemen en me te steunen waar ik het nodig had. Dan heb ik het vooral over mijn promotor, mijnheer P. Van Laethem, hij heeft me gedurende die zeven maand elke veertien dagen geholpen met het oplossen van problemen, het me doen inzien dat wat ik deed, niet de juiste methode was. Soms zocht ik het zelfs te ver om de oplossing tegen te komen. Hiervoor bedank ik u, mijnheer P. Van Laethem. Eveneens had ik het gevoel dat ik de taal (CSHARP) niet al te best onder de knie had. Daarom had ik aan mevrouw G. Vermeiren gevraagd of ze me wat bijles kon geven. Zo wist ik welke functionaliteit deze taal allemaal machtig is. Het was een kleine verlenging op de lessen die we in het vorige academiejaar gezien hadden. Ook bij mijn medestudenten kon ik terecht voor meer informatie, om te begrijpen dat de code die ik schreef niet de juiste syntax was maar dat ik het anders bedoelde. Ook zij zijn een grote hulp geweest bij het realiseren van mijn werk. Last but not least , had ik nog graag het hele docenten-team willen bedanken. Dit omdat ik voor elk probleem terecht kon bij hen, omdat ik steeds op hun steun en hun weldoordachte steun kon rekenen.

Chapter 1

Inleiding

1.1 De Motivering

Zoals altijd begon dit werk bij een keuze die gemaakt moest worden. Die keuze dateert al van november van vorige jaar (2007). Er werd een hele lijst voorgeschoteld met mogelijke onderwerpen voor het eindwerk. Hoewel men toen midden in de stage waren, werd er toch grondig bestudeerd welk onderwerp het beste zou liggen, welke doelstellingen er werden bepaald en vooral in welke taal het zou moeten gemaakt worden. De keuze was niet echt vanzelfsprekend wegens de grote mogelijkheden die tot onze beschikking stond. Ook werd er ons op het hart gedrukt dat we niet verplicht waren om een onderwerp uit de lijst te kiezen maar dat er gerust voorstellen konden gedaan worden.

De verschillende onderwerpen hadden een samenvatting over wat de bedoeling was. Hieruit konden de studenten merken of het iets voor hen was om deze opdracht tot een goed eind te brengen. Bij mij was er een goed gevoel bij het volgende onderwerp: "Een grafische simulatie van een proces-scheduler". Dit omdat er al lessen over de scheduler gegeven was. Het leek me wel iets om visueel voor te stellen wat een proces-scheduler deed. De kennis had ik niet om iets visueel voor te stellen maar mijn enthousiasme zorgde ervoor dat er bijgeleerd werd.

1.2 De Problemen

Het grootste probleem was dat er snel in tijdsnood geraakt werd. Dit kwam omdat er meer tijd besteed werd aan de klassendiagram dan voorzien. Ook moest er een duidelijke en logische werking van scheduler begrepen worden. Eens dit verstaan was, kon er gezocht worden achter een oplossing voor het volgende geval. Namelijk het "preemptief"-probleem.

Een ander moeilijkheid was, dat er niet eenvoudig van de ene klasse in de andere klasse aan de datamembers kon geraakt worden. Hoewel er wel setters en getters aanwezig waren, was het onmogelijk om aan de datamembers te geraken.

1.3 Wijze van het onderzoek

Zoals eerder verteld, zijn er al lessen geweest in verband met de proces-scheduler. In het boek dat er gebruikt werd, in het tweede, stond er ook een hoofdstuk in van de proces-scheduler ("Operating Systems met Java"). Het was dan ook vanzelfsprekend dat er in dit boek werd gelezen om meer te weten te komen van de werking van deze scheduler.

Het zoeken naar oplossingen in verband met code ging vooral op het internet voort. Het bekijken naar anderen hun oplossingen, het testen in andere projecten (niet meteen in het eindwerk) en het begrijpen wat de andere mensen doen, hebben me ook ver vooruit geholpen.

1.4 De Begrenzungen

De vijand van dit werk was vooral de tijd. Het duurde steeds een tijd alvorens er oplossing zich aanbood. Meestal was de gedachte wel juist maar is de afwerking niet correct of een verkeerde methode aanroepen.

Hierdoor kwam het werk steeds in tijdsnood waardoor het extra hard werken was om een probleem tijdig te kunnen oplossen. Het werd zelfs zenuwslopend dat er niet meteen een resultaat uit de bus kwam.

Chapter 2

Situering

Het eindwerk had verschillende doelstellingen. Er moesten dus verscheidene zaken gebeuren om dit eindwerk tot een goed einde te kunnen brengen. Allereerst werd er opzoeken gedaan om de vele scheduling-technieken te begrijpen. Nadien was het de bedoeling om een schema te maken waarin alles duidelijk werd. Het moest dus uiteindelijk een object georiënteerd ontwerp gemaakt worden. Dit gebeurde door middel van de klassendiagram en usecases. Als laatste fase is het natuurlijk om dit allemaal om te zetten in code die uiteindelijk tot een werkende applicatie eindigd.

Het eindresultaat zou een grafische simulatietool zijn die in de les van Toegepaste Informatica, 2de jaar, Besturingssystemen, gebruikt kan worden om de studenten de verschillende scheduling technieken visueel te laten zien.

Chapter 3

Onderzoek

3.1 Wat is het doel van cpu-scheduling?

Met deze methode wilt men het besturingsysteem van de computer productiever maken door de CPU te laten omschakelen tussen processen. Men moet een onderscheid kunnen maken tussen éénprocessorsystemen en multiprogramming. Het eerste is de primairste, hiermee kan maar één enkel proces tegelijkertijd actief zijn. Alle anderen moeten wachten tot de CPU vrij is en opnieuw een proces voor actie kan selecteren. Meteen ontdekt men dat dit niet productief en relatief traag is. Om dit tegen te gaan heeft men dus multiprogramming ontwikkeld. Hier is het dus mogelijk om meerdere processen tegelijkertijd actief te laten zijn. Het gevolg hiervan is dat de CPU-gebruik gemaximaliseerd kan worden. Dit gaat als volgt te werk: Een proces wordt uitgevoerd totdat deze interactie nodig heeft, meestal is dit van IO-type. Normaal gezien zou bij een simpel systeem, de cpu hier moeten wachten totdat de interactie klaar is, alvorens verder te gaan met het proces. De tijd dat de CPU wacht is puur verspilte tijd. In multiprogramming-systeem, is deze tijd er niet. Meteen als men merkt dat het proces interactie nodig heeft, wordt deze weggestuurd, uit de cpu, naar de IO-apparaat. Op dit moment is de CPU opnieuw vrij. Hij kan dus een ander proces behandelen. Dit geeft natuurlijk een veel snellere activiteit en wordt de inactiviteit van de CPU tot het minimale behouden. Dit zorgt ervoor dat het besturingsysteem productiever is. Indien het proces dat interactie nodig had van een IO-apparaat klaar is, zal deze opnieuw ter beschikking worden gesteld van de CPU en zal de processor deze behandelen indien er aan bepaalde voorwaarden worden voldaan (zie verder). Het CPU-schedulen bestaat uit een cyclus die steeds opnieuw wordt uitgevoerd. Het is een cirkel van CPU-activiteit en het wachten van bepaalde IO. Elke uitvoering van elk proces begint met een CPU-burst, nadien IO-burst; zo wisselen ze elkaar af. Uiteindelijk eindigt het proces waarmee het begonnen is, een CPU-burst met

als aanvraag om de uitvoering te mogen stoppen. Maar eigenlijk is er nog niet besproken wat een CPU-scheduler is. Dit zal ik nu even kort beschrijven. Telkens als de CPU inactief is, moet het besturingssysteem een ander proces kiezen uit de readyqueue. Eens hij hier een proces gekozen heeft, geeft hij die aan de CPU zodat die toch actief zou zijn. Er bestaat ook nog een synoniem voor de CPU-Scheduler, namelijk het kortertermijn-scheduler.

Ik had het daarnet over een readyqueue, dit is een rij met daarin processen. Zij (de processen) hebben de status waarmee ze kunnen aangeven wat ze nu van plan zijn. De processen in de readyqueue hebben de status ready. Met andere woorden, ze zijn klaar om de CPU in te gaan. Natuurlijk kan er maar één proces tegelijkertijd in de CPU, vandaar dat er moet gekozen worden uit de hele rij processen om te worden uitgevoerd.

3.2 Algoritme

De scheduler moet beslissen welk proces in readyqueue naar de CPU mag gaan. Natuurlijk gaat dit niet random. Hier zijn verschillende algoritme's voor uitgewerkt.

3.3 First-come-first-served (FCFS)

Dit is een van de gemakkelijkste algoritme's. De methode dat hier wordt toegepast is dat het proces dat als eerste om de CPU vraagt, ook de eerste is die wordt aangewezen om de CPU te mogen gebruiken. Als er een proces de readyqueue binnenkomt, wordt hij automatisch aan het einde van de wachtrij gekoppeld. Als de CPU vrij komt, dan wordt het eerste proces in die wachtrij, los gemaakt en gestuurd naar de CPU, zodat hij zijn ding kan doen. Het voordeel is dat dit eenvoudig te begrijpen is. De keerzijde is dat de wachttijd dikwijls vrij lang is. Er kan ook een fille-effect ontstaan, dit is dat vele kleine processen erg lang moeten wachten totdat een groot proces de CPU weer vrijgeeft. Dit is niet echt efficiënt, aangezien de IO-apparaten niet tot hun maximale kracht gebruikt worden.

3.4 Shortest-Job-First (SJF)

Bij dit algoritme wordt er gekeken naar de lengte van de volgende CPU-burst van dat proces. Hier wordt dus gekozen aan de hand van de volgende CPU-burst. Indien de CPU vrijkomt, bekijkt de scheduler in de readyqueue welk proces de kortste volgende CPU-burst heeft. Wanneer meerdere processen toevallig dezelfde lengte hebben van CPU-burst dat volgt, wordt de FCFS-algoritme tussen deze processen uitgevoerd. Dit om uiteindelijk toch maar een één proces te hebben die de CPU binnen kan. Niet vergissen, dit algoritme gaat om de volgende CPU-

burst. Niet om de totale lengte. Dit algoritme is integenstelling tot de eerste erg optimaal, zo is de gemiddelde wachttijd tot het minimale gehouden. Het is onderhand wel duidelijk dat er niet alleen voordelen bestaan aan zaken. Het nadeel bij deze methode is dat het vrijwel moeilijk is om te schatten wat de lengte is van de volgende CPU-burst. Een benadering tot deze lengte is wel mogelijk als men voor ogen neemt dat de volgende CPU-burst vergelijkbaar is met de vorige.

3.5 Prioriteit-scheduling

Zoals de naam doet vermoeden zal dit gaan over prioriteiten, voorrangsregels. Processen krijgen prioriteiten en de degene met de hoogste prioriteit (dringendste), wordt als eerste naar de CPU gestuurd. Meestal is er een vaste reeks van getallen die worden gebruikt als prioriteit, 0 - 7 en 0 - 4095. Er is geen algemene overeenkomst wat nu de hoge prioriteit voorstelt (De 0 of het hoogste getal). Een nadeel van deze methode is dat men aan starvation kan doen. Hiermee wordt bedoeld dat er op een bepaald moment zoveel processen komen met hoge prioriteit (zij worden dus als een van de eerste naar de CPU geleid) en dat heeft als gevolg dat processen met lagere prioriteit niet kunnen uitgevoerd worden. Uiteraard is hier een oplossing op gevonden. Men noemt het 'Aging' (veroudering). Hiermee wordt ook de prioriteit als rode draad genomen maar telkens als een proces van een lagere prioriteit moet wachten, wordt een prioriteit verhoogt. Waardoor het proces wel uitgevoerd kan worden.

3.6 Round-Robin-Scheduling

Dit is een algoritme vooral ten voordele van time-sharing-systemen. Het is vergelijkbaar met de FCFS-algoritme maar er wordt een kleine tijdshoeveelheid bepaald dat processen in de CPU mogen uitvoeren. Is deze tijd voorbij, worden ze onherroepelijk uit de CPU gestuurd en mag een ander proces beginnen met uitvoeren. Deze tijdseenheid is meestal tussen de 10 en 100 miliseconden. De readyqueue wordt in dit opzicht bekeken als een FCFS-rij waarbij de nieuwe processen aan het einde van de rij worden toegevoegd.

3.7 Multilevel-queue-scheduling

Dit is een methode waar gebruikt gemaakt kan worden van groepen. Bijvoorbeeld de processen die op de voorgrond gedraaid worden en processen die op de achtergrond gedraaid worden. Het splitst in feite de readyqueue in verschillende wachtrijen. Elke wachtrij heeft zo zijn eigen gedefinieerde algoritme. En een proces blijft steeds in zijn wachtrij. Een andere mogelijkheid is dat de tijd verdeeld

kan worden tussen de verschillende wachtrijen. Zo kan elke wachtrij een proces sturen dat uitgevoerd kan worden

3.8 Multilevel-feedback-queue-scheduling

Dit is ongeveer hetzelfde als hierboven maar hier kan wel gewisseld worden van wachtrijen. Het doel achter dit algoritme is om processen met verschillende CPU-burst-eigenschappen te onderscheiden. Indien een proces teveel CPU-tijd gebruikt, wordt dit proces in een wachtrij gestoken met een lagere prioriteit. Andersom kan ook, als een proces te lang moet wachten voor hij uitgevoerd kan worden, wordt hij in een wachtrij gezet met een hogere prioriteit. Dit algoritme is de meest algemene vorm van schedulen. Het geeft de mogelijkheid om meerdere algoritme's in één methode te steken, het kan eigenlijk op maat worden ingesteld van de processen. Door die flexibiliteit is het dan ook de moeilijkste algoritme. De reden hiervan is, dat men alle parameters van de processen zo nauwkeurig mogelijk moeten zijn.

3.9 Preemptive scheduling VS niet-preemptive scheduling

Het eerste is een wijze waardoor het onderbreekbaar is door middel van prioriteit. Daar staat tegenover dat de niet-preemptive scheduling staat voor het niet te kunnen onderbreken van processen die aan het uitvoeren zijn in de CPU. De flexibiliteit zorgt ook hier voor een nadeel, processen moeten gegevens delen. Bij twee processen die gegevens moeten delen, kan het zijn dat de ene de gegevens bezig is aan het wijzigen en wordt onderbroken. De andere moet dan verder kunnen werken met dezelfde gegevens. Dit moet, zoals u merkt, goed gecoördineerd worden.

3.10 Dispatcher

Dit is een bestandsdeel dat ook betrekking heeft tot de scheduling. Het is een deel die het beheer over de CPU geeft aan het proces dat is geselecteerd door de kortetermijn-scheduler. Hij heeft als eigenschap snel te zijn, dit omdat hij bij elke switch (ready->wait, wait->ready) wordt aangeroepen. Er bestaat dispatch latency, dit is de tijd die de dispatcher nodig heeft om steeds een proces te stoppen en een andere te starten. Dit is tijd voor interne zaken, met andere woorden, puur tijd verlies. Dit moet dus zo klein mogelijk gehouden worden en dat is de reden waarom een dispatcher snel zou moeten zijn.

3.11 Multiprocessor-scheduling

Tot nu toe hebben we systemen besproken met maar één processor. Het is ook mogelijk om meerdere processors te hebben. Tevens is het hier niet mogelijk (net als in systemen met maar één processor) om één gegarandeerd beste oplossing te hebben. We hebben systemen waarin twee processors homogeen zijn qua functionaliteit, elke beschikbare processor kan dan gebruikt worden voor alle processen in de wachtrij. Als er meerdere identieke processors aanwezig zijn, is er een mogelijkheid op 'load sharing'. We kunnen dan elke beschikbare processor toewijzen aan een aparte wachtrij. Het nadeel hiervan is dat de situatie wel kan bestaan dat de ene processor niets doet en de andere heel druk in de weer zijn. Een oplossing voor dit, is dat men dan een gemeenschappelijk readyqueue aanmaakt. Zo gaan alle processors naar dezelfde queue en worden de processen gepland op de beschikbare processors. In dit opzicht kunnen we via twee verschillende manier schedulen.

De eerste manier is: alle scheduling (IO en ander systeemactiviteit) gaat via één masterserver. De andere voeren enkel gebruikers-code uit. Dit is dan bij name een assymetrische multiprocessing. Dit is redelijk eenvoudig omdat er slechts één processor de datastructuren van het systeem benadert, dit heeft als gevolg dat er minder gegevens gedeeld moet worden.

De tweede aanpak is: elke processor is verantwoordelijk voor zijn eigen scheduling. Elke processor bekijkt de algemene readyqueue en kiest daar een proces uit. Het moeilijke hieraan is, dat de processors goed geconfigureerd moeten worden, want ze mogen niet dezelfde proces kiezen in de gemeenschappelijke wachtrij. Deze methode noemt men de symmetrische multiprocessing (SMP).

3.12 Thread-scheduling

Een thread is een besturingsstroom binnen een proces. Threads op gebruikersniveau zijn threads die zichtbaar zijn voor de programmeur en niet voor de kernel. Threadlibrary is degene die verantwoordelijk is voor threads op gebruikersniveau. De kernel van het besturingssysteem beheert de threads op kernelniveau. Process-contention scope (PCS) is de strijd om de CPU dat plaatsvindt tussen threads van hetzelfde proces. Voor de beslissing welke kernelthread gepland mag worden op de CPU gebruikt de kernel system-contention scope (SCS). Hier vindt de strijd zich tussen alle threads in het systeem.

Chapter 4

Klassediagram en usecases

Voor een duidelijk ontwerp te maken werd er gevraagd om de cursus van Software Engineering erbij te halen. Dit project werd vergeleken met het vorige multidisciplinair project dat het vorige academiejaar moest worden gemaakt.

Het eerste dat men moest doen, was informatie bijleren en het maken van een ontwerp. Het moest een uitstekende basis zijn om het volledige project op te laten steunen. Vandaar dat er voldoende tijd aan deze zaken besteed werd. Als einde van deze fase moest er een klassendiagram aangemaakt worden. Met deze schema moest de werking van de scheduler duidelijk worden. Tevens moest er een beeld gevormd worden over hoe de applicatie werkte.

Figuur 1 - Klassediagram

4.1 Proces

Het proces heeft verschillende eigenschappen. Elk proces heeft een uniek nummer, zo is het onmogelijk om tweemaal identieke processen aan te maken. Voor de gebruiksvriendelijkheid werd er aan elk proces eveneens een naam gegeven. Een proces komt op een bepaalde tijd aan, dit wordt dan ook bijgehouden. Het is ook mogelijk om verschillende statussen in te brengen aan de processen. Als er gekozen werd om het proces het status "New" te geven, heeft dit als betekenis dat er een nieuw proces werd aangemaakt en dat nieuw binnenkomt. Een tweede mogelijkheid is om het proces als "Ready" te beschouwen. Hiermee bedoeld men dat het proces klaar is om in de CPU te komen. Als laatste categorie is er de "Wait"-status. Hierbij geeft de gebruiker weer dat de proces aan een IO-burst (Input/Output) zal beginnen. Een proces bestaat uit minstens één burst. Naargelang de soort burst (IO of CPU) zal deze afgehandeld worden door de CPU of een IO-apparaat.

4.2 Queue

In dit ontwerp werden er twee soorten queue's onderscheiden, "Wait" en "Ready". De "Wait"-queue heeft een rij van processen die bij het ontstaan, de status "Wait" hebben gekregen. En er zijn ook processen die een IO-burst moeten uitvoeren. Deze gaan ook naar de "Wait"-queue. De "Ready"-queue is een wachtrij waar de processen wachten om de CPU binnen te treden. Hier komen dus processen in die de status "Ready" of "New" hebben gekregen. Alsook kunnen er processen van de "Wait"-queue naar deze rij komen om dan in de CPU te komen.

4.3 Algoritme

Dit is een belangrijk element dat duidelijk in het middelpunt van het schema staat. In deze klasse worden de verschillende algoritme's beschreven. De superklasse is "Algoritme" en is abstract. De andere algoritme's erven van deze klasse over. De klasse "Algoritme" zou niet veel mogen implementeren omdat het een basis is voor de andere soorten algoritme's.

4.4 Burst

Een burst is een deel van een proces. Het zijn de zogenaamde taken die een proces zou moeten uitvoeren. Een taak kan uit twee verschillende categorieën bestaan: IO-burst of CPU-burst.

De IO-burst is een burst die interactie nodig heeft van iemand of iets anders dan de CPU. De CPU-burst daarentegen is een taak van een proces die de CPU moet afhandelen. Deze behoudt het proces dan in zijn midden tot de tijd van de burst om is. In dit ontwerp werden er twee soorten queue's onderscheiden, "Wait" en "Ready".

De "Wait"-queue heeft een rij van processen die bij het ontstaan, de status "Wait" hebben gekregen. En er zijn ook processen die een IO-burst moeten uitvoeren. Deze gaan ook naar de "Wait"-queue. De "Ready"-queue is een wachtrij waar de processen wachten om de CPU binnen te treden. Hier komen dus processen in die de status "Ready" of "New" hebben gekregen. Alsook kunnen er processen van de "Wait"-queue naar deze rij komen om dan in de CPU te komen.

Dan wordt er gekeken naar de volgende burst, (dit is natuurlijk afhankelijk van algoritme!) en naargelang de soort van burst wordt het ofwel bijgehouden in de CPU of wordt het afgevoerd naar de "Wait"-queue waar zijn burst zal worden afgehandeld.

4.5 CPU

Dit is de klasse die de proces zal binnenkrijgen waar de CPU-burst zal beginnen. Nadat de burst(s) zijn werk gedaan heeft in de CPU, wordt het dan ook verwijderd van de CPU. Er zijn verschillende wegen die het dan kan volgen:

- Van de CPU naar de "Wait"-queue.
- Van de CPU naar de "Ready"-queue.
- Van de CPU verdwijnt het van het systeem

4.6 Timer - Controller

Deze klasse is degene die alle andere klassen aanspreekt en controleert. Hij gaat na of er iets moet gebeuren in verschillende klassen. Een voorbeeld is, dat hij nakijkt of er een proces naar de "Wait"- of de "Ready"-queue moet gaan. Het is eigenlijk een boodschapper die de overige klassen laat weten wanneer ze wat moeten doen.

4.7 Scheduler

Deze klasse heeft ongeveer hetzelfde doel als de Controller, alleen is het toch net ietsjes anders. De Controller zegt namelijk wat de scheduler moet doen, zegt hem welk proces hij moet nemen. Maar het is de scheduler die het in feite uitvoert.

4.8 Usecases

Figuur 2 - Usecase: Nieuwe Processen

Wanneer moet een proces dat nieuw is (het wordt net aangemaakt) in de readyqueue komen? Dit maakt bovenstaande usecase duidelijk.

De Timer (Controller) kijkt elke seconde of de proces zijn aankomsttijd gelijk is aan de teller van de timer. Deze teller wordt elke seconde verhoogt. Dus elke seconde controleert de Timer of er een aankomsttijd in één van de processen gelijk is aan het huidige getal dat in de teller zit. Indien er eentje is laat men dit weten aan de scheduler zodat hij weet dat er iets gedaan moet worden met een bepaald proces. De klasse Proces laat ondertussen ook weten aan de Timer dat het proces preemptief is. De scheduler stuurt het proces wiens aankomsttijd gelijk is aan de teller naar de readyqueue. Want enkel daar kan hij kans maken om naar de CPU gestuurd te worden en te worden afgehandeld.

Chapter 5

Panels Aanmaken

5.1 Panels Aanmaken

Nu het ontwerp helemaal klaar is, de logische denkwijze begrepen is, is het de tijd om te beginnen aan het ontwerpen van de formulieren die door de gebruiker zouden moeten worden ingevuld. Hier kon de creativiteit helemaal zijn gang gaan. Toch moest het geen abstracte kunst worden met knopjes en labels en menu's. Het moest toch nog wat overzichtelijk blijven. Van mijn eerst ontwerp is er dus niet veel meer overgebleven.

Met wat tips van mijn promotor, is het ontwerp omgevormd tot wat het nu is.

5.2 Splitpanel

Aan de linkerkant van de applicatie ziet men verschillende linklabels. Dit is een navigatiemethode waardoor de gebruiker zich doorheen de applicatie kan vertoeven. Dit geeft een meerwaarde aan de gebruiksvriendelijkheid van dit werk. Dit menu zal ten allen tijde aanwezig zijn, steeds op dezelfde plaats, enkel de rechterkant van de applicatie veranderd naar gelang men vordert in de applicatie.

5.3 Proces Aanmaken

Dit paneel heeft als doel een nieuw proces aan te maken. De gevraagde informatie die men nodig heeft om een proces aan te maken zijn: nummer, aankomsttijd, status en een naam. Maar een proces bevat meer dan dat. Het heeft eveneens een lijst met minstens een burst. Dit wordt aangemaakt in een andere panel.

Figure 5.1:

5.4 Burst Aanmaken

Als men op dit panel terecht komt, wilt dit zeggen dat de eerste stap tot het maken van een nieuw proces al achter de rug is. Het maken van een burst kan door een soort aan te duiden en een lengte (positief numeriek getal) mee te geven. Men kan nadien op de knop drukken, door hierop te drukken wordt die ene burst aangemaakt en in de lijst van burst van het proces gezet. Men kan zoveel bursts aanmaken als men maar wilt. Er is ook een mogelijkheid om de proces die aangemaakt is te wijzigen. Indien men alles liever zou wissen, is er de knop "Proces Verwijderen", die de klus voor zich neemt. Alsook is er een knop om een volgende proces aan te maken of indien men genoeg processen heeft, kan men het algoritme kiezen. Dit is de volgende panel.

5.5 Overzicht - Algoritme kiezen

Op deze panel is het mogelijk om het geheel opnieuw te bekijken van wat men gemaakt heeft. Welke processen zijn er nu weer aangemaakt? Welke eigenschappen was nu voor die ene proces? Dit is dan ook voor de gebruiker gemakkelijk, aangezien de lijst al gesorteerd is op aankomsttijd. Zo kan de gebruiker perfect weten welk proces het eerste in de animatie te voorschijn zal komen. Op dit formulier wordt ook een keuze gevraagd in verband met de soorten algoritme's. Dit is eigenlijk een cruciale vraag. De simulatie is (logischerwijs) afhankelijk van het algoritme dat wordt gekozen. In het uitklapbare menu is het enkel mogelijk om de voorgeschreven algoritmes te gebruiken. De gebruiker heeft geen recht om een uitgevonden algoritme hierin te schrijven. Dit zal overigens ook niet gaan. De gebruiker staat nog voor een keuze. Hier zijn twee categorieën in verband met de methode van het simuleren van de bevraagde algoritme, met bijhorende processen. Men kan ervoor kiezen om op het volgende formulier terecht te komen waar er beknopte informatie te bezichtigen is, gedurende de simulatie. Anderzijds is er ook een mogelijkheid om een uitgebreide vorm te hebben wat de uitleg betreft van de simulatie.

5.6 Simulatie

Wat de gebruiker ook kiest van de voorgaande mogelijkheden, de formulieren zijn identiek, enkel de informatie zal uitgebreider zijn dan bij de andere. Deze panel voorziet aan de linkerkant een lijst van procesnamen en hun respectievelijke bursts. Dit geordend volgens aankomsttijd. Men heeft een lege lijst met daarboven uitleg, hier komt de uitleg tevoorschijn. Hieronder staat de knop "Start". Dit knopje

brengt alles ingang, dankzij deze knop zal de gantt-diagram getekend worden volgende de ingevoerde gegevens (processen, burst, algoritme). Er zal tevens een timer tevoorschijn komen zodat men dan alles goed kan zien. Uiteraard is het mogelijk om de simulatie vroegtijdig te stoppen door op de knop "Annuleren " te drukken. Hierdoor gaat men terug naar het vorige formulier waardoor men een andere simulatie kan opbouwen (veranderen van het algoritme). Indien de gebruiker de applicatie onmiddellijk wilt stoppen, bestaat er de knop "Exit All".

Chapter 6

Problemen en hun oplossingen

Het project had zijn moeilijkheden over heel de periode. Het aanmaken van bepaalde panels hebben dus ook hun tijd nodig gehad.

6.1 Klasse Proces

Het maken van een lege lijst was de eerste moeilijkheid die men tegen kwam. Het creëren van een proces moest al gebeuren als er op de knop van het eerste formulier werd gedrukt. Hoewel het ingewikkeld uitzag, was de oplossing om in de klasse proces een `private List<Burst> lijst = new List<Burst>();` bij te schrijven zodat er een lijst van bursts is.

6.2 Klasse Burst

Er moest overgeërfd worden van een superklasse (Burst). Zoals eerder gezegd heeft een burst mogelijke soorten: een IO-burst of een CPU-burst. Hoewel dit redelijk eenvoudig lijkt, heeft dit wel lang geduurd eer er een oplossing gevonden werd. De moeilijkheid was dat de klasse burst ook een datamember had, namelijk `lengte`. Maar die moest dus ook bereikbaar zijn in de subklasse. Zoals geweten mag een datamember nooit public zijn. Dan was een tussentijdse oplossing, de datamember als `protected` zetten. Dit zorgt ervoor dat de `lengte` wel bereikbaar is in de overervende klasse maar dit moest in het algemene 'form' ook gebruikt worden. De oplossing waarmee het probleem geheel van de baan is geveegd, is door bij de constructor van de overervende classes de `int lengte` als argument toe te voegen. Bijvoorbeeld: `public IOBurst(int lengte) this.Lengte = lengte;`

6.3 Klasse Queue

Deze klasse maakt een lijst van processen. Deze lijst moet overgeërfd worden door de onderliggende klassen. Tevens zijn er andere functies in deze klasse die door de onderliggende klassen moeten worden geïmplementeerd. Het probleem was dat er van de overervende klassen geen singleton gemaakt werd. Eveneens werd er geen nieuwe Wait/Ready-Queue aangemaakt in de klasse maar telkens een nieuwe queue, van de klasse Queue. Dit is verholpen door de volgende regel code: `private static ReadyQueue instance = new ReadyQueue("Ready-queue");` Dit voor de respectievelijke ReadyQueue, hetzelfde geldt natuurlijk voor de WaitQueue.

6.4 Klasse Controller (Timer)

Deze klasse heeft een teller, hij wordt verhoogt telkens als de 'form' de functie Tick() aanroept. Eveneens is de Controller een singleton. Het grootste gedeelte van de Controller is de functie Tick(). Hier wordt gezegd wat er elke seconde moet gebeuren. Hier moet beschreven worden wat er met een bepaalde proces op een bepaalde tijdstip moet gebeuren. Er zijn verschillende overgangen mogelijk: `new -> ready; ready -> running; running -> wait` en `running -> stop`.

Het probleem dat hier ontstond was dat er gekeken moest worden naar de teller en de aankomsttijd van een proces. Alsook moest men weten wanneer een proces uit de CPU moest gehaald worden. Dit kon enkel als men wist hoelang de burst duurde. Maar er moest ook rekening gehouden worden met de algoritme's en de volgende burst.

6.5 De form (FrmEindwerk)

In deze klasse gebeurt alles wat maar verband houdt met de formulier de gebruiker voor zich krijgt.

6.6 DataSource

Het eerste probleem dat zich opdroeg was in het eerste paneel. Hier was er een listbox geplaatst om een lijst weer te geven van de aangemaakte processen. Het was moeilijk om te weten hoe de informatie zou worden weergegeven in een listbox. Na wat opzoekingswerk is er een oplossing gevonden, namelijk DataSource. Het gebruik van DataSource is een manier om informatie in een listbox in te vullen.

6.7 Procesnummer

Een volgend probleem was om een veld automatisch te laten optellen. Hiermee wordt bedoeld dat de procesnummer telkens moet worden opgehoogd. Dit lijkt gemakkelijk, maar hier zijn wat problemen mee. Er werd een private datamember voor de procesnummer gedefinieerd, deze werd alsook geïnitieert op 0 (nul). Telkens als dit paneel wordt opgeroepen is het de bedoeling dat dit procesnummer automatisch wordt opgehoogd. Dit gebeurt met `number++`; Deze wordt dan in het veldje geplaatst en gebruiker kan deze niet meer aanpassen.

6.8 Combobox op geen enkele keuze zetten

Als men op de knop drukt op de tweede panel om opnieuw een nieuw proces aan te maken, is het de bedoeling dat de gebruiker terug gestuurd wordt naar de eerste panel. Maar dit is niet enkel wat er gedaan moet worden. De gegevens moesten ook automatisch gewist worden als men opnieuw naar deze panel ging. De combobox op geen keuze zetten was een klein probleempje. De oplossing die gevonden werd was door de 'selecteditem' van de combobox op -1 te plaatsen.

6.9 Onpaint

Deze functie zorgt ervoor dat er getekent wordt in de picturebox. Elke seconde kijkt de Controller hoeveel tijdseenheden de burst nog moet doen alvorens hij gedaan heeft. Deze integer wordt doorgestuurd naar de Onpaint. Zo kan hij de lengte van de burst tekenen. Er werd allereerst een snelle kijk genomen over hoe men in CSHARP moesten tekenen. Dat werd gedaan met 'graphics'. Het testen van hoe dit werkte, ging door in een andere solution dat van het project. Eens dit redelijk begrepen werd, kon er getekent worden in het eindwerk.

6.10 Preemptief

Het fenomeen "preemptief" heeft als definitie dat een proces dat bezig is met uitvoeren kan worden onderbroken voor een ander proces dat dringend moet beginnen. Dit kan aan de hand van prioriteit bijvoorbeeld. Als de tweede proces (proces dat meteen moest actief worden) gedaan heeft, wordt de eerste proces terug in de CPU geladen en wordt verder behandeld. Daar staat tegenover dat er ook zo iets bestaat als "niet-preemptief". Dit is dus het omgekeerde als hiervoor werd gezegd. Een proces dat in actie is kan niet worden onderbroken om een ander proces voor te laten gaan. Vanuit het standpunt van dit eindwerk, werd enkel niet-preemptief geïmplementeerd. Het feit dat een bepaald proces zou kunnen gestopt worden in de CPU om een ander proces voor te laten is moeilijker te

implementeren dan de theorie. Er moet constant in de proceslijst gekeken worden of er een proces is die om één of andere reden voor moet gaan. Indien dit het geval is, moet er aan de CPU gemeld worden om onmiddellijk te stoppen met de proces waarmee hij bezig is, deze even opzij zetten. En vervolgens moet hij dit ene dringende proces uitvoeren. Maar stel nu voor dat er nadien (nog voor het beëindigen van het proces waar de CPU nu mee bezig is) nog een proces is die voorhang moet krijgen. Dan wordt opnieuw een proces aan de zijkant gezet en wordt de derde proces afgehandeld. Indien dit uiteindelijk gelukt is, moet men de andere twee processen uitvoeren (deze die aan de zijkant werden gezet). Wanneer dit achter de rug is, kan opnieuw de gewone gang van zaken verder gaan, tot er weer een proces wordt gevonden met hogere (dringendere) prioriteit. Het is hier dan ook uit duidelijk dat het erg ingewikkeld kan zijn. Toch is er hierover nagedacht. Er is een datamember "preemptief" dat een "bool" (true,false) in de klasse Controller in het leven roepen om hier toch rekening mee te houden. Alles hangt af van deze member. Dit beïnvloed het gehele systeem aangezien de Controller hierop moet reageren en de boodschap doorgeven aan CPU.

6.11 Property

Eerder werd het probleem al besproken dat er aan bepaalde private datamembers niet kon worden aangeraakt. Dit leek dan toch noodzakelijk te zijn voor de verdere werking van de applicatie. Normaal gezien heeft men hiervoor dan een "getter" en een "setter", die ervoor zorgt dat er toch iets gedaan kan worden met die datamember (ook buiten de betreffende klasse). Dit was dan ook automatisch geïmplementeerd in de klasse. Toch leek het onmogelijk om via een andere klasse deze datamember te gebruiken. Na wat opzoekingwerk is "property" tot mijn besef gekomen. Dit is een publieke methode dat in de betreffende klasse wordt toegevoegd. Hierin wordt dan de kernwoorden 'get' en 'set' geplaatst. Bij elk van deze kernwoorden worden er accolades naast gevoegd en hierin wordt dan één lijntje code geplaatst. Bij de 'get' zorgt ervoor dat de gegevens terug gestuurd worden. Dus een 'return this.***' is hier op zijn plaats. Een 'set', zet de waarde in de locale datamember. Tussen de accolades bij een 'set' is het volgende wat er hoort te zijn: 'this.*** = value'. Hoewel het erg identiek lijkt op de 'getter' en de 'setter' is hiermee toch het probleem opgelost.

Chapter 7

Mogelijke Uitbreidingen

In dit deel worden bepaalde voorstellen naar voren gebracht die kunnen worden aangebracht aan de applicatie. De opdracht was groot en is dus mogelijk tot uitbreiding. De verruiming van de applicatie is creatief bedoeld. Het zijn extra 'tools' waarvan gebruik gemaakt kan worden en kan dienen als een extra stimulans om de werking van een proces-scheduler te begrijpen. De voorstellen zijn dan ook eerder aan de creatieve kant gezocht, dit omdat het 'spelend-leren' te bevorderen van de gebruiker.

7.1 Algoritmes

Het lijkt vanzelfsprekend dat de verdere implementatie van de algoritmes zal moeten plaats vinden. Momenteel is er maar eentje geïmplementeerd maar die moet dus vergezeld worden door de andere mogelijke algoritmes. Dit zal dan ook meteen zorgen voor het vrijere gevoel van de gebruiker ten opzichte van zijn keuze. Momenteel is hij verplicht om de ene mogelijkheid die er is, te gebruiken. Dit zou dus naar de toekomst toe kunnen worden uitgebreid door bijvoorbeeld: Shortest-Job-First, Prioriteit, Roud-Robin, Multilevel-queue en uiteindelijk de Multilevel-feedback-queue. Dit kan wel als kans hebben dat de applicatie enkele miliseconden extra gaat nodig hebben om op te starten, maar indien er niet met de chronometer naast zit, zal dit nog steeds goed meevallen.

7.2 Simulatie

Het is niet onbegrijpelijk dat de meeste aanpassingen gedaan kunnen worden aan de simulatie zelf. Hier zijn namelijk een tal van factoren die kunnen worden toegevoegd aan het bestaande. Zo kan er eventueel in de listbox met procesnamen

op de panel van de simulatie, een kleurtje krijgen. Deze kleur dat ze hebben, is dan ook de kleur van tekening die door de simulatie wordt gebruikt. Dit omdat kleuren nog steeds eenvoudiger is te onderscheiden dan de namen.

Ook kan er bijvoorbeeld iets gedaan worden aan de simulatie zelf. Daarmee wordt er bedoeld, dat de tekening die er gemaakt wordt misschien ook kan gekozen worden. Met bijvoorbeeld een extra keuze aan de gebruiker om de type van tekening te kiezen. Hier wordt dan aan andere figuurtjes gedacht dan de momentele rechthoek. Het kan ook mogelijk zijn om bij de aanmaak van een proces een bepaalde afbeelding te kiezen die dan gebruikt zal worden tijdens de simulatie. Hoe persoonlijker de applicatie, hoe fijner voor de gebruiker. Alsook kan er, als de applicatie ten einde is, een overzicht gemaakt worden, noem het misschien beter een rapport. Waarin heel de uitleg staat en met alles gemaakte processen en bursts. Dit om het wat officiëler te hebben. Dit kan misschien een functie zijn die kan worden aangevinkt bij het begin van de applicatie. Zo kunnen testers zich laten uitleven zonder nadien een rapport te verkrijgen die ze dan toch weggooien of dichtklikken. Hier kan dan ook gevraagd worden of het rapport moet worden opgeslagen in een map, zodat deze later nog eens bezichtigd kan worden.

Er kan eventueel een pauze knop worden toegevoegd bij de applicatie. Dit is best wel handig als de docent tijdens het lopen van de simulatie verbaal nog informatie wilt geven. Zo hoeft de docent ook niet steeds alles van begin tot einde af te lopen, indien hij onderbrak.

Wie weet lijkt het wel iets om een record knop te hebben. Die zou dan als bedoeling hebben om de hele animatie op te nemen, of een deel, hangt af wanneer de gebruiker op deze knop heeft gedrukt. Als de simulatie ten einde is, kan er beslist worden waar dit bestand wordt opgeslagen. Deze kan dan eventueel worden doorgestuurd naar andere mensen om van via afstand toch visueel te kunnen uitleggen wat er bedoeld wordt.

Men kan nog een stapje verder gaan. Momenteel wordt de uitleg in tekstuele vorm weergegeven in een listbox. Deze kan al als mogelijke uitbreiding, opgeslagen worden maar wat dan met de mindergoedziende mensen? Het zou voor deze mensen toch handiger zijn moest de uitleg niet enkel geschreven worden maar ook worden uitgesproken door de boxen. Dit zorgt ervoor dat dit programma zijn doelgroep vergroot wat natuurlijk ideaal is voor de producent.

Chapter 8

Besluit

De komende maanden zijn maanden geweest van hard werken en de zenuwen niet de overhand te geven. Het was een periode van zoeken op het internet naar oplossingen en meer informatie. Het project verliep moeizaam en het was intensief werken maar uiteindelijk met kleine vorderingen liep het tot zijn eind met een werkende applicatie.

De doelstellingen zijn vervuld, er is informatie verworven over het te maken applicatie, er is een object geïntegreerd ontwerp gemaakt en uiteindelijk heeft men verschillende klassen met regels code.

Het was niet gemakkelijk om de problemen op te lossen, zeker als men niet weet waarom het niet gaat. De oplossing was dat de gedachten werden opgeschreven op een kladblad en dat er zo in de Nederlandse taal werd gezegd wat de code moest doen. Het enige wat er dan nog moest gebeuren, was het omzetten in code. En dit ging zonder al te veel problemen.

Er werd in dit project gekozen voor een .Net-omgeving en niet voor Java. Deze keuze is nog steeds het meest gewenst omdat het gemakkelijker werken is dan in Java.

In de laatste maanden is het project wat anders aangepakt dan voorheen. Er werd geen dagen verspild aan een probleem maar er werd verder gedacht en geprobeerd om andere problemen een oplossing te bieden. Het was een methode om niet zoveel tijd meer kwijt te raken dan voorheen.

Een conclusie die er getrokken kan worden is dat het project er meestal gemakkelijker uitziet dan dat het gemaakt wordt. Ook al is het volledige ontwerp in het hoofd, toch moet er voldoende kennis zijn om alles juist te implementeren.

Het project heeft een hele periode van het laatste jaar in beslag genomen. Er werd dag in, dag uit, over de projecten gepraat en iedereen gaf ideeën om het beter en gemakkelijker te maken. Het is een applicatie geworden waardat eraan gewoegd is maar die er uiteindelijk wel staat.

