Practicum handleiding Besturingssystemen 2012-2013

Instituut voor Informatica
Faculteit NWI
Universiteit van Amsterdam

Dick van Albada Roy Bakker Kamil Iskra Zeger Hendrikse
Dennis Kaarsemaker Joppe Bos Stefan Post

22 maart 2013

Hoofdstuk 1

Inleiding

Deze handleiding is bestemd voor het practicum Besturingssystemen[1] voor Informatica (voorjaar 2013), een eerstejaars vak in het Bachelors curriculum Informatica aan de Faculteit NWI van de Universiteit van Amsterdam.

1.1 Doelstellingen

Dit practicum beoogt de student inzicht te verschaffen in een aantal praktische problemen bij het implementeren en gebruik van besturingssystemen (operating systems). Daarbij kunnen overigens lang niet alle aspecten van zo'n systeem aan de orde komen. Dit jaar worden er vier (voor eerstejaars informatica studenten) of vijf (voor alle andere studenten) opgaven gegeven:

- In de eerste opgave wordt gekeken naar het opstarten van nieuwe processen.
- De tweede opgave zal betrekking hebben op processor scheduling.
- In de derde opgave wordt gekeken naar de toewijzing van geheugenruimte.
- In de vierde opgave komt de interne structuur van een eenvoudig file-systeem aan bod.
- In de vijfde opgave wordt gekeken naar racecondities, deadlock en semaforen.

Verder wordt nog een aantal andere doelstellingen nagestreefd. We noemen:

- Het verkrijgen van inzicht in het gedrag van programma's. Vaak betekent dit het doen en interpreteren van metingen. Systeemontwerpers, -bouwers en -beheerders zijn bij hun werk in hoge mate afhankelijk van metingen aan het (eventueel gesimuleerde) systeem en de correcte interpretatie daarvan.
- De programmeertaal C verder leren beheersen. De moeilijkheid van C zit hem vooral in het veelvuldig gebruik van pointers, gecombineerd met een niet zo erg waterdichte type-checking. Leer gebruik te maken van de hulpmiddelen die C ook biedt om je programma overzichtelijk en robuust te maken.

[1] `http://staff.science.uva.nl/~dick/education/BS.html`

- Met alle andere practica heeft dit practicum tot doel een goede en nette programmeerstijl bij te brengen. Hierbij hoort, vanzelfsprekend, ook het aanleren van het schrijven van een duidelijke documentatie en van een duidelijke en overzichtelijke verslaglegging.

1.2 Literatuur

N.B. De benodigde literatuur voor het college wordt bij de webpagina's over het college beschreven.

Naast deze handleiding moet er nog andere literatuur geraadpleegd worden over enerzijds theorie van Operating Systemen en anderzijds over de programmeertaal C en de programmeeromgeving Linux waarmee moet worden gewerkt in het practicum.

Allereerst moet natuurlijk het boek "Operating Systems in Depth" van Doeppner worden geraadpleegd.

Op het web zijn verschillende handleidingen over UNIX, LINUX en C te vinden.

Verder is het LINUX systeem ruim voorzien van on-line handleidingen, toegankelijk via het `man` commando. De handleiding daarvan is beschikbaar via `man man`. Er is natuurlijk via Google en via sites als deze[2] enorm veel informatie te vinden op het internet.

Voor meer informatie over make kan je b.v. ook terecht bij de pagina over de gnu make utility.[3]

1.3 Omgeving

Het ingeleverde materiaal moet in elk geval compileren en draaien op de Linux werkstations die gebruikt worden tijdens het practicum, en de server `deze.science.uva.nl`. Linux is dan ook veruit het meest geschikt om thuis mee te werken.

Als informaticastudent moet je eigenlijk ook wel je eigen Linux systeem kunnen installeren en onderhouden.

Het is echter mogelijk dat de opgaven ook onder Windows geprogrammeerd kunnen worden. Voor diegenen die dit echt *persé* willen zijn er wel mogelijkheden.

Het beste is dan om in je Windows-omgeving een virtuele Linux omgeving te installeren. Denk dan b.v. aan Virtualbox[4] of VMware player, maar er zijn ook een heleboel andere.

Een alternatief is ook de oplossing die Cygnus biedt, een complete Unix shell met GNU ontwikkel tools, zie de Cygwin home page[5]. Als "last resort" is DJGPP[6] van Delorie software geschikt.

Al zijn Cygwin en DJGPP in hoofdzaak 'POSIX compliant' leveren fork en threads vaak problemen op in deze omgevingen. De meest voor de hand liggende en ongecompliceerde keuze blijft dan ook Linux.

Mac OS X is in feite ook een UNIX systeem en zal i.h.a. bruikbaar zijn voor het ontwikkelen van je software. Toch is het in alle gevallen verstandig *je uiteindelijke tests en metingen op de*

[2] `http://www.cs.swarthmore.edu/~newhall/unixlinks.html`
[3] `http://www.gnu.org/software/make/`
[4] `http://www.virtualbox.org/`
[5] `http://www.cygwin.com/`
[6] `http://www.delorie.com/djgpp/`

practicumsystemen te doen. Dan zijn b.v. ook meetresultaten het best onderling vergelijkbaar en weet je zeker dat je werk aan de eisen voldoet.

Je kan vanuit huis op de practicummachine "access" inloggen met SSH, al is dat vrij ingewikkeld. Je moet wel eerst een VPN[7] tunnel opzetten wil je hier gebruik van kunnen maken en naast je UvA netID ook een science account hebben. Maak een SSH verbinding met `sremote.science.uva.nl` (science account, de enige machine die van buiten bereikbaar is) en daarvandaan met de Linux-server van het practicum `access.fnwi.uva.nl` (UvA netID). SSH is beschikbaar of desgewenst eenvoudig te installeren binnen Linux. Voor Windows zijn er ook meerdere opties, b.v. de combinatie van PuTTY en WinSCP. Openssh vanuit Cygwin wordt om security-redenen i.h.a. niet aanbevolen.

1.4 Samenwerking

Bij de alle opgaven mag worden samengewerkt door twee en niet meer dan twee personen. Samenwerken betekent ook dat beiden aanwezig moeten zijn bij het nakijken, en beiden vragen over het gemaakte werk moeten kunnen beantwoorden. Het spreekt vanzelf dat de ingeleverde code – afgezien van eventueel door de practicumleiding aangeleverde brontekst – eigen werk moet zijn.

1.5 Assistentie en nakijken

Gedurende de voor het practicum op het rooster gereserveerde uren zal, uitzonderingen daargelaten, steeds minstens één assistent aanwezig zijn. Buiten deze uren kunnen vragen eventueel via een mailtje aan de assistent worden gericht, *maar alleen als je ook steeds netjes op de practicumuren aanwezig bent*. Je mag er echter niet op rekenen dat deze dan ook meteen worden beantwoord.

Bij het practicum wordt geassisteerd door:

- Dick van Albada (G.D.vanAlbada@uva.nl)
- Mikołaj Baranowski (M.P.Baranowski@uva.nl)
- Koen Koning (Alleen tijdens practica)
- Bas Weelinck (Alleen tijdens practica)
- Ilja Kamps (Alleen tijdens practica)
- Robin de Vries (Alleen tijdens practica)

De eerste drie opgaven zullen tijdens het practicum worden nagekeken en worden becijferd als onvoldoende, voldoende of goed. Voor een onvoldoende opgave krijg je één herkansing. Je moet voor deze opgaven wel de geproduceerde code via Blackboard inleveren, vóór de aangegeven deadline. Nakijken kan op zijn laatst op de practicumsessie direct volgend op de deadline.

[7] `http:www.uva.nl/vpn`

Let op, je krijgt voor de eerste drie opgaven alleen een cijfer als je je werk op het practicum hebt laten nakijken en je werk via Blackboard inlevert. Alleen het één of het ander telt niet.

Ook dit jaar zijn er WWW-pagina's[8] en natuurlijk Blackboard voor het college. Hierop komen berichten van algemeen belang voor college en practicum. Mededelingen van de practicumbegeleiding zullen daar worden geplaatst.

De vierde en eventuele vijfde opgave worden niet tijdens het practicum nagekeken. Hiervoor moet een verslag worden geschreven dat tijdig samen met de geschreven code via Blackboard wordt ingeleverd.

1.6 Inschrijven

Om een geldig cijfer voor dit vak te kunnen halen moet je je tijdig inschrijven via SIS. Als je dat nog niet hebt gedaan, doe dat dan zo snel mogelijk in overleg met de docent en de studieadviseur.

Voordat je überhaupt met het practicum kan beginnen moet je, als je die nog niet hebt, bij ESC een 'UvA netID' aanvragen voor de voor het onderwijs gebruikte werkstations.

Als je dat hebt gedaan, kan je op een van de LINUX werkstations aan het werk.

1.7 Eisen aan de in te leveren opgaven

Zoals uit de doelstellingen al duidelijk zal zijn, moet voor iedere opgave je code worden ingeleverd. Voor de laatste opgave(n) moet daar ook een verslag bij[9]. Pas ook hier toe wat je bij je projecten over vaardigheden hebt geleerd!

Geef bij elk van de in te leveren onderdelen (in ieder bestand) duidelijk naam en collegekaartnummer (van beide practicanten) aan.

Lever voor de laatste opdracht de volgende zaken in:

Administratie Naam en collegekaartnummer van de (beide) practicanten; opdrachtnummer en datum. *Op te nemen in ieder in te leveren bestand.*

Programmadocumentatie Een beschrijving van de ingeleverde programmatuur. Deze moet een duidelijk plan van aanpak voor het gestelde probleem bevatten, met eventueel daarbij gemaakte afwegingen. Het is niet de bedoeling dat hierin allerlei kleine implementatiedetails worden besproken. Verder moeten het gebruik en de te verwachten resultaten kort worden beschreven. Deze documentatie moet bij de vierde en vijfde opdracht worden ingeleverd, al kan hij soms ook kort blijven. Op te nemen in het verslag.

C-Code De (aangepaste) C-sources en hulpbestanden, zoals Makefiles. Lever steeds de hele .c of .h file in. Geef daarbij zonodig duidelijk aan welke functies (procedures of routines) gewijzigd of toegevoegd zijn. Als voor een opgave verschillende versies van een functie moeten worden gemaakt, lever die dan ook allemaal in. Maak steeds duidelijk bij welk onderdeel van de opgave een bepaalde versie hoort.

[8] `http://staff.science.uva.nl/~dick/education/BS.html`

[9] `http://staff.science.uva.nl/~dick/education/Computing.skills/Writing-a-report/index.html`

Zorg er steeds voor dat de interne documentatie (het commentaar) binnen de C-sources in overeenstemming is met de C-statements. Zorg er vooral voor dat uit je commentaar de samenhang tussen de statements duidelijk wordt. Commentaar dient er niet voor om in natuurlijke taal nogmaals de werking van een bepaald statement te vermelden: de lezer wordt geacht op de hoogte te zijn van de C-syntax en semantiek. Het commentaar dient een niveau hoger te staan, het kan de lezer bijvoorbeeld attenderen op een subtiele programmeertruc, of de interpretatie verduidelijken.

Experiment en/of tests Een beschrijving van je experiment of de tests uitgevoerd op je programma. Hierin geef je aan hoe je metingen of testresultaten zijn verkregen (o.a. de begintoestand en de volgorde van de diverse opdrachten) en een overzicht van de verkregen resultaten. Als dat naar jouw idee van belang is, kan je daarin een stukje programma-uitvoer opnemen, maar niet te veel. Zorg er voor dat je resultaten overzichtelijk gerangschikt zijn. Grafieken of tabellen kunnen een geweldig goed hulpmiddel zijn. Op te nemen in het verslag.

Meetresultaten (indien van toepassing) Een bespreking van je test- of meetresultaten. Probeer te verklaren waarom je resultaten op een bepaalde manier van je invoer of je algoritme afhangen. Verlies daarbij niet uit het oog dat je metingen over het algemeen maar zeer beperkt zijn en dat daardoor toevallige effecten ook een belangrijke rol kunnen spelen. Het is ook geen schande als je een waarneming niet kan verklaren, het is ook belangrijk onverwachte verschijnselen te kunnen signaleren. Op te nemen in het verslag.

Kijk op ook op Blackboard Op Blackboard kunnen aanvullende eisen en hints worden gegeven door de practicumassistent. Kijk geregeld onder het kopje Practicum Informatie of er nog iets nieuws bij staat.

1.8 Inleveren

Als er bij de opgave wordt aangegeven hoe de files moeten heten, moet je ook echt uitsluitend die namen gebruiken. Anders wordt het nakijken te bewerkelijk en kan je opgave niet worden beoordeeld.

Je ingeleverde code *moet* op zijn minst compileren. Test elke verandering, hoe onschuldig ook, voor je de code inlevert!

Overigens is er natuurlijk niet maar een enkele versie van C[10]. Je hebt op zijn minst te maken met K&R C, de oorspronkelijke versie. Dan heb je ANSI C[11], een veel nettere, officieel gestandaardiseerde versie, en de nieuwere standaard, C99[12]. Daarnaast heb je te maken met wat de gebruikte compiler feitelijk accepteert. De veelgebruikte GNU compiler[13] accepteert naar keuze alledrie. De ingeleverde programma's moeten in ieder geval aan de ANSI C standaard of de C99 standaard voldoen. Geef in je makefile de juiste compileropties mee.

[10] `http://en.wikipedia.org/wiki/C_(programming.language)`
[11] `http://en.wikipedia.org/wiki/ANSI.C`
[12] `http://en.wikipedia.org/wiki/C99`
[13] `http://gcc.gnu.org/`

Bij alle in te leveren opgaven is een deadline aangegeven.

Als inleveren via Blackboard niet werkt, *maar ook alleen dan* mag je de opgave (zip file, tar file of ge-gzipte tarfile) per e-mail inleveren. Stuur de mail dan aan de assistent en de docent. De meest voorkomende reden dat inleveren via Blackboard niet werkt, is overigens dat de deadline is verstreken, en dan ben je gewoon te laat.

Inleveren moet via Blackboard als assignment. Voor iedere opgave is een onbeperkt aantal inlevermogelijkheden beschikbaar, zodat je een eventuele herziene versie kan inleveren zonder dat er ge-reset hoeft te worden.

Het verslag wordt bij voorkeur gemaakt met behulp van LaTeX, en moet in ieder geval in pdf worden ingeleverd.

Lever alle gevraagde bestanden (inclusief het verslag) samen in als een (gecomprimeerde) tar file of zip file. Wees er bij het gebruik van de tar functie op verdacht dat tar de eerst opgegeven file als uitvoerfile gebruikt!

Dus:

```
> tar cvzf opgave1.tgz file1.c dir2
```

overschrijft `opgave1.tgz` en stopt daar de inhoud van `file1.c` en `dir2` in.

Let op!

- De deadlines zijn echte harde deadlines.
- Zorg dat je op tijd inlevert - als het om 5 voor 12 mis gaat, had je maar eerder een bijna definitieve versie moeten inleveren.
- Zorg dat compileren lukt met enkel het commando `make`. Los ook *warnings* die de compiler geeft (met minimaal de optie `-Wall`) op.
- Zet alle in te leveren files eerst samen in één directory en lever ze vanuit die directory in. Deze directory heeft de naam `OS_opgaveX_achternaam1_achternaam2`, waarbij achternaam2 natuurlijk alleen van toepassing is bij de opgaven die in tweetallen worden gemaakt. Dan is er bij het uitpakken de minste kans op ongelukken, en kan er veel sneller worden nagekeken.
- Gebruik nooit absolute padnamen voor je files, want `tar` zou dan bij het uitpakken de files weer op de oorspronkelijke plaats proberen terug te zetten (dus in jouw directory!). Opgaven die hieraan niet voldoen kunnen niet worden nagekeken.
- Als is aangegeven dat je verschillende subdirectories moet gebruiken, lever ze dan in vanuit een gemeenschappelijke parent directory, zodat in ieder geval nooit verwijzingen omhoog (`../dit_of_dat`) nodig zijn. Opgaven die hieraan niet voldoen kunnen niet worden nagekeken.
- Controleer of je echt alles hebt ingeleverd.
- Vóór de deadline opnieuw inleveren van een verbeterde versie kan en mag, en wordt zelfs sterk aangeraden.
- Bij het nakijken van ingeleverde programma's moeten we je programma ook opnieuw kunnen compileren en linken. De C-compiler wil b.v. per sé dat source files de suffix `.c` hebben. Let op dat soort zaken bij het inleveren!

Opg.	onderwerp	deadline	gew.	bijzonderheden
1	Command shell	10/04	1	Teams van 2
2	CPU scheduling	21/04	1	Teams van 2
3	Geheugenbeheer	12/05	1	Teams van 2
4	Filesystemen	26/05	2	Teams van 2
5	Filosofen	26/05	2	Teams van 2, niet als je ook tutoraat moet volgen

1.9 Beoordeling

De eerste drie opgaven worden op het practicum nagekeken en becijferd met goed (9), voldoende (6.5) of slecht (4). Niet ingeleverde en besproken opgaven scoren helemaal geen punten.

Bij de beoordeling zal op een groot aantal aspecten van de ingeleverde opgave worden gelet. Vanzelfsprekend zullen daarbij een correcte implementatie en een juiste uitvoering en interpretatie van de gevraagde metingen zwaar meetellen. Leesbaarheid van de ingeleverde code is echter ook van groot belang: voor degenen die het programma moeten nakijken, en zeker ook voor degenen die het programma ontwikkelen. Er zal daarom ook zwaar worden getild aan de *duidelijkheid, leesbaarheid en kwaliteit van programmadocumentatie en -layout.* Let daarbij b.v. op het netjes *inspringen*[14] van statements binnen blokken, het gestructureerd tussenvoegen van lege regels, de plaatsing van het commentaar. Vergeet ook niet het stukje commentaar aan het begin van je programma met je naam, de datum en het doel van de code. Stel je maar voor dat je een commerciëel product moet afleveren. Daarbij wordt het commentaar bijna uitsluitend in het Engels geschreven, mede in verband met de huidige ontwikkelingen rondom het zogenaamde 'open source gaan' van software pakketten (denk bijvoorbeeld maar aan Mozilla en natuurlijk GNU/Linux). Natuurlijk is op dit practicum ook Nederlands toegestaan, maar let er dan wel op dat het geen raar mengseltje wordt van beide talen.

Let er ook op dat foutmeldingen voor de systeemprogrammeur duidelijk moeten aangeven wat er fout is, maar er ook voor moeten zorgen dat een minder deskundige gebruiker de nodige informatie krijgt. Vermijd dus zoveel mogelijk verwijzingen naar voor de gebruiker irrelevante details uit het programma. Overigens is het aan te bevelen zo min mogelijk foutmeldingen door de systeemprogrammatuur zelf te laten afdrukken - indien het geen fatale fout betreft kan beter een foutwaarde via vlaggen en de return-waarde van je functies worden teruggegeven.

[14]... **inspringen**. Wanneer men `vim` gebruikt, zorgen de volgende instellingen (die in de `.vimrc` gezet kunnen worden), voor een handige en intelligente hulp bij het inspringen:

```
set autoindentset smartindentset expandtabset softtabstop=8
set shiftwidth=4
```

Ook met emacs en xemacs kan vrij eenvoudig netjes geïndenteerde broncode worden geschreven. Achteraf herstellen van een slechte indentatie kan in geval van nood ook met een programma als `indent`.

Hoofdstuk 2

Opgave 1 - Fork & pipe en de command shell

Leerdoelen:

1. Het opstarten van nieuwe processen onder UNIX/LINUX,
2. Het opstarten van programma's onder UNIX/LINUX,
3. Het gebruik van pipes,
4. Inzicht in de werking van command shells,
5. Het gebruik van `sigaction`.

Let op: Dit is een redelijk ingewikkelde opgave. Je moet er veel voor uitzoeken. Begin er tijdig aan, anders krijg je hem niet op tijd af.

Het is bij deze opgave de bedoeling dat je een eenvoudige shell schrijft. Zo'n shell moet via een fork andere processen kunnen opstarten. Soms moeten er ook meerdere processen achterelkaar kunnen worden opgestart, die onderling verbonden zijn door een pipe. De beste procedure in dat geval blijkt te zijn, eerst de pipe door de shell te laten aanmaken en op de gewenste manier met stdout te verbinden. Daarna wordt het eerste kind afgesplitst en wordt de pipe op de juiste manier met stdin verbonden. Daarna wordt het tweede kind afgevorkt. Als er meer dan twee processen in de pijplijn zitten, dan moet bij de middelste processen netjes een verbinding met de voorganger en met de opvolger worden gemaakt.

2.1 Het ouderproces

De acties van deze shell zijn de volgende:

1. Druk een prompt af en lees een opdrachtregel van de terminal. Bij een EOF stopt de shell.

2. Haal het eerste "woord" op uit de opdrachtregel. Controleer of dit een "builtin" opdracht betreft. Zo ja, voer die uit en ga terug naar 1.

3. Lees de eerste opdracht. Opdrachten bestaan uit een programmanaam, gevolgd door nul of meer parameters, net als bij iedere andere shell. Ze eindigen met een "new-line" of een "pipe-symbool" '`|`'.

4. Eindigt de opdracht met een '`|`', dan moet je eerst een pipe aanmaken. Ga anders naar 6. Nu zijn er meerdere mogelijkheden:

   - De ouder doet het nodige loodgieterswerk met stdin/stdout en de aangemaakte pipes alvorens een nieuw proces af te splitsen. Deze benadering heeft de voorkeur.
   - De ouder vorkt een kindproces af en laat het loodgieterswerk daar aan over.

5. Doe de nodige administratie zodat het volgende proces zijn invoer uit de pijp kan lezen en lees de volgende opdracht na de '`|`'. Ga naar 4.

6. Fork een nieuwe shell af voor het laatste kind in de rij; de parent shell wacht op het eindigen van alle kinderen[1] en gaat daarna terug naar 1.

2.2 De kindprocessen

De kindprocessen doen het volgende:

1. Ze zorgen dat alle verbindingen met de nul, één of twee pipes correct zijn verbonden en sluiten alle niet-gebruikte verbindingen (functies dup2 en close).

2. Het kind voert vervolgens een "execvp" uit, waarbij het eerste woord van de opdracht wordt opgevat als een programma uit het gewone 'pad' in de `PATH` omegevingsvariabele, en de rest van de opdracht als de parameters. Bestaat het gevraagde programma niet, of geeft `execvp` een andere fout terug, dan eindigt het kindproces met een toepasselijke foutboodschap, b.v. middels "perror".

Afgezien van de '`|`' kent deze shell geen redirection en geen achtergrondprocessen.

2.3 Builtin opdrachten

De shell moet de volgende "builtin" opdrachten direct zelf uitvoeren:

- Verplicht:
  - exit - beëindig de shell.
  - cd - change directory (kan alleen als een builtin). De opdracht cd heeft precies één parameter nodig.
  - . of source - lees opdrachten uit een file (een parameter). Splits daarvoor geen nieuwe shell af. Keer na het einde van de file terug naar de vorige input-file.
    Dit soort opdrachten kunnen genest zijn.

[1] Dat is het eenvoudigste als de shell zelf de ouder is van alle aangemaakte processen.

- Optioneel:
  - !! - herhaal vorige opdracht (is eigenlijk geen builtin, maar een soort editor commando).
  - !n - herhaal de n-de opdracht. (b.v. !3).

Na de '|' wordt niet naar builtins gezocht.

2.4 En verder

Vang de `^C` [CTRL-C] af in de parent shell middels een geschikte aanroep van `sigaction()`[2], maar niet in de nakomelingen. Dit kan betekenen dat je bij een kindproces de oorspronkelijke handler juist moet terugzetten.[3]

Om redenen van veiligheid mag je shell alleen programma's uit het standaard 'pad' uitvoeren; andere paden zijn taboe. Het pad naar het programma mag dus geen '/' bevatten, omdat je b.v. via "/bin/../" overal kan komen.

Zorg ervoor dat je eerst een duidelijk schema hebt van hoe de in- en uitvoer doorverbonden moet worden bij opdrachten als:

```
ls -alit | grep opsys | sed "s/^.*199[0-9]//" | cat
```

Relevante (manual) pages / hyperlinks:
`ps`[4], `open`[5], `fork`[6], `exec`[7], `mkfifo`[8], `close`[9], `write`[10], `read`[11], `wait`[12], `pipe`[13], `socketpair`[14], `string.h` (o.a. `strcat, strchr, strcpy, strtok`)[15]

Er staat een skelet van een mogelijke implementatie in `scanner.c` op Blackboard.

[2] `http://linux.die.net/man/2/sigaction`

[3] Let op: er zijn twee functies beschikbaar met vergelijkbare werking: `sigaction()` en `signal()`. `Sigaction()` conformeert aan de Posix standaard en heeft daarom de voorkeur, maar geeft problemen bij vertaling met `-ansi` of `-std=c99`. Een programma dat `signal()` gebruikt geeft die problemen niet, maar `signal` is kennelijk slecht gestandaardiseerd en dus niet portable. `Sigaction` compileert wel als je op de commandoregel bij het vertalen ook nog `-D_POSIX_SOURCE toevoegt`. Dit definiëert een macro voor de preprocessor die zorgt dat Posix routines beschikbaar worden gemaakt.

[4] `http://www.die.net/doc/linux/man/man1/ps.1.html`

[5] `http://www.die.net/doc/linux/man/man2/open.2.html`

[6] `http://www.die.net/doc/linux/man/man2/fork.2.html`

[7] `http://www.die.net/doc/linux/man/man3/exec.3.html`

[8] `http://www.die.net/doc/linux/man/man3/mkfifo.3.html`

[9] `http://www.die.net/doc/linux/man/man2/close.2.html`

[10] `http://www.die.net/doc/linux/man/man2/write.2.html`

[11] `http://www.die.net/doc/linux/man/man2/read.2.html`

[12] `http://www.die.net/doc/linux/man/man2/wait.2.html`

[13] `http://www.die.net/doc/linux/man/man2/pipe.2.html`

[14] `http://www.die.net/doc/linux/man/man2/socketpair.2.html`

[15] B.v. `http://linux.die.net/man/3/strtok`

Hoofdstuk 3

Opgave 2 – Scheduling

3.1 Doelstellingen

1. Het verkrijgen van inzicht in het gedrag van scheduling algoritmen.
2. Het gebruiken van een eenvoudig simulatieprogramma.
3. Verdere oefening in C programmeren.
4. Oefening in het doen van metingen.

Deze opgave met groepjes van twee maken.
Kijk ook regelmatig op de opmerkingen van de assistent op Blackboard!

3.2 Inleiding

Zoals op college behandeled is, heeft scheduling te maken met het toewijzen van hulpbronnen, of resources aan een proces. Sommige hulpbronnen moeten eenmalig bij het aanmaken van het proces worden toegewezen (je zou kunnen denken aan swapruimte op schijf, en in elk geval een process control block), sommige tijdens de uitvoering van het proces. Daarbij zijn dan weer resources die door het OS tijdelijk teruggenomen kunnen worden (zoals de processor) en andere die na toewijzing in gebruik moeten blijven tot het proces er voorlopig weer meer klaar is (zoals I/O devices).
Een goede scheduling heeft een enorme invloed op de prestaties van het systeem. Maar het hangt af van zowel de gebruikte criteria voor een goede prestatie, als van het feitelijke werkaanbod, welke scheduling-algoritme de beste prestaties levert. In deze opgaaf moet je voor een gegeven systeem proberen zo goed mogelijke prestaties te bereiken. Dat gebeurt in simulatie.
Simulatie vormt een prima methode om de kwaliteit van een scheduling methode te beoordelen, met name door zijn grote flexibiliteit en relatief lage kosten. Voor het bestuderen van scheduling wordt meestal gebruik gemaakt van de een of andere vorm van "discrete event simulation". Zo ook in deze opgaaf. De practicumleiding heeft een simulatieprogramma geschreven, waar alleen de scheduler nog aan ontbreekt. Hoewel het programma gebruik maakt van een random-generator, is het wel zo opgezet dat bij gelijke invoer, steeds dezelfde uitvoer wordt gegeven.

De simulatie kent drie soorten resources waarop een proces moet kunnen wachten:

1. Geheugen - eenmalig aan het begin. Totaal zijn er `MEM_SIZE` (gedefiniëerd in `mem_alloc.h`) eenheden geheugen beschikbaar (misschien stellen dat wel 8KB page frames voor). Een proces moet voordat het kan draaien geheugen toegewezen krijgen. Dit vormt een onderdeel van de opgave. Er is een aantal speciale functies die je hiervoor moet gebruiken. Het gebruik hiervan wordt in een skelet voor de uitwerking en in de header files geïllustreerd.
2. CPU - herhaald. Er wordt één processor gesimuleerd. Elk proces heeft afwisselend de processor en een I/O device nodig. Het schedulen van de CPU vormt een onderdeel van de opgave.
3. Drie I/O devices - herhaald. Laat de scheduling hiervan aan de aangeboden simulator over.

Het te schrijven programma

Voor dit onderdeel is door de practicumleiding een simulatieprogramma geschreven. Dit simulatieprogramma simuleert het draaien van processen. Het kan zowel worden gebruikt bij het testen van alleen de hoog-niveau scheduler (toewijzing van eenmalige resources; in dit geval, geheugen), als voor het gezamenlijk testen van een hoog-niveau en een CPU scheduler. De opzet van de wachtrij voor de CPU in het programma is zo dat zonder verdere maatregelen een FCFS scheduling wordt gebruikt, wat voor een batch systeem niet zo onlogisch is (maar RR met een niet te korte time slice is nog veel logischer).

Het programma moet uiteindelijk bestaan uit vier modules:
De simulator, aangeleverd als `simul.o`. Deze bevat o.a. de main routine.
Een gesimuleerde geheugentoewijzingsroutine, aangeleverd als `mem_alloc.o`.
Een state-of-the-art random number generator, de "Mersenne twister" van Matsumoto aangeleverd als `mt19937ar.o`.
Jullie eigen scheduler. Een skelet wordt aangeleverd als `scheduler-skeleton.c`.

De benodigde sources en headerfiles kunnen jullie in ieder geval vinden in de volgende directory `http://staff.science.uva.nl/~dick/education/src`.
Er zijn object files beschikbaar voor Linux, Linux-64, Solaris en Cygwin.
Voor allevier is er ook een volledig werkende versie `schedsim` beschikbaar als voorbeeld. De werking ervan is niet per se optimaal.

De functie `schedule(event_type event)` is nog niet uitgewerkt. Deze moeten jullie schrijven.

De schedule.h file

De benodigde wachtrijen en typedeclaraties, zoals `event_type`, worden beschreven in de file `schedule.h`.

Ieder proces wordt beschreven door een record dat deels toegankelijk is voor het studentprogramma en deels (via het gebruik van een `void` pointer) ontoegankelijk.

Deze records staan steeds in een van de volgende vier lijsten:

Nieuwe processen Deze wachten nog op de toewijzing van geheugen.
In het record staat de benodigde hoeveelheid geheugen.
De simulator genereert nieuwe processen en plaatst deze in de lijst van nieuwe processen. De proces-parameters zoals geheugengebruik, CPU en IO gebruik worden hierbij ook vastgelegd in het ontoegankelijke deel van het record. Alleen de voor de scheduler kenbare informatie wordt ook in het toegankelijke deel geplaatst.
De hoog-niveau scheduler kan processen uit deze rij selecteren, geheugen toewijzen en daarna in de ready queue `ready_proc` plaatsen. Als de simulator een nieuw proces in deze rij heeft geplaatst, roept hij de scheduler aan met een `NewProcess_event`.

Ready processen Het eerste proces in deze lijst wordt na terugkeer uit de functie `schedule` uitgevoerd. Als het bij de uitvoering I/O gaat doen, zet de simulator het proces in de I/O queue `io_proc` en roept `schedule` aan met een `IO_event`. Loopt het proces in deze time slice af, dan wordt het in de `defunct_proc` queue geplaatst, in afwachting van het vrijgeven van het gebruikte geheugen. De scheduler wordt dan met een `Finish_event` aangeroepen.
De simulator gebruikt bij het simuleren van het CPU-gebruik een time slice, die om een FCFS gedrag te krijgen heel groot is gemaakt. Als het proces toch gewoon tot het eind van zijn time slice doorrekent, blijft het proces staan waar het was, en wordt alleen de gebruikte CPU-tijd verhoogd. De scheduler wordt dan met een `Time_event` aangeroepen, waarna hij b.v. het betreffende proces helemaal achterin de `ready-queue` zou kunnen plaatsen.

I/O De processen in deze lijst komen vooralsnog niet voor scheduling in aanmerking. De simulator plaatst ze naar verloop van tijd terug naar de ready queue en roept dan de scheduler met een `Ready_event` aan.

Defunct De processen in deze lijst zijn klaar, maar hebben nog geheugen bezet. Dit moet worden vrijgegeven, waarna de functie `rm_process(proces)` kan worden aangeroepen om het record te verwijderen en statistische gegeven te verzamelen.

Als er geen ready processen zijn, zal het `NULL`-proces gaat lopen tot er een significant event optreedt (een nieuw proces, of een proces dat zijn I/O afrondt).

Naast de boven beschreven zaken zijn er nog een functie en twee functie-pointers gedefiniëerd:

`finale` Finale is een functie variabele die wijst naar een parameter-loze void functie die door de simulator helemaal aan het eind wordt aangeroepen. Door deze functie-variabele naar een eigen functie te laten verwijzen kan je b.v. door jezelf verzamelde statistische informatie laten afdrukken.

`reset_stats` `Reset_stats` is een functie variabele die wijst naar een parameter-loze void functie die door de simulator wordt aangeroepen op het moment dat de eigenlijke meting start. Hij kan worden gebruikt om allerlei tellers op nul te zetten na de aanloopfase van de simulatie. Door deze functie-variabele naar een eigen functie te laten verwijzen kan je b.v. door jezelf verzamelde statistische informatie ook op dat moment resetten.

`set_slice(double slice)` Met deze functie kan je zorgen dat de simulator na `slice` gesimuleerde tijdseenheden een `Time_event` genereert. Hiermee kan je b.v. round-robin CPU scheduling realiseren. Een dergelijke scheduler kan een interessant effect op de performance hebben. Deze functie heb je nodig als je een preemptieve CPU-scheduler wil maken.
Let op - deze functie heeft alleen effect op de volgende slice. *Daarna wordt de lengte van de time-slice weer op een hele grote waarde teruggezet.*

De hoog-niveau scheduler

Elk proces in de wachtrij voor nieuwe processen vraagt een bepaalde hoeveelheid geheugen aan. Pas als die (eenmalig) is toegewezen, kan het proces beginnen met executeren. In het programma-skelet staat duidelijk aangegeven hoe die toewijzing in zijn werk gaat. Voor het implementeren van de hoog-niveau scheduler, hoeft je scheduler alleen iets met een '`NewProcess_event`' en met een '`Finish_event`' te doen. Bij de drie andere events kan je meteen teruggaan naar de simulator.

Een van de problemen waarmee de hoog-niveau scheduler te maken kan krijgen is starvation. Processen die veel geheugen-ruimte nodig hebben moeten wachten tot er voldoende aaneengesloten ruimte beschikbaar is.

Het kan nodig zijn met de scheduling van kleine processen te wachten om voldoende geheugen vrij te krijgen. Anderzijds, kan dit wachten aanleiding geven tot een slechte CPU bezetting en een slechte throughput.

De CPU scheduler

De CPU scheduler heeft in principe twee handvatten om de volgorde van executie van processen te sturen:

1. Het vooraan in de ready queue plaatsen van het te executeren proces.
2. Het zetten van een time-slice voor deze beurt op de CPU van dat proces.

Je kan in principe experimenteren met round-robin scheduling, maar ook met b.v. shortest job first, of multi-level queues of b.v. iets met prioriteiten. De gesimuleerde processen horen een redelijk consistent gedrag te vertonen, waarbij de lengte van de CPU burst per proces niet te veel varieert.

De `schedule.h` header file bevat de nodige declaraties en definities, plus een boel commentaar. Bekijk dit zorgvuldig, want het geeft je extra houvast en mogelijkheden bij het maken van de opgave.

De feitelijke opdracht

Het simulatieprogramma staat je toe apart een belasting van het geheugen, van de CPU en van het IO systeem in te stellen. Deze kloppen in praktijk ongeveer, maar hangen b.v. ook van de kwaliteit van je scheduler af.

Hoog niveau scheduler

Probeer een hoog niveau en een CPU scheduler te maken die bij een test-set van 1000 processen en belastingen van 0.7 voor geheugen, CPU en IO minimaal de volgende doelstellingen realiseren:

- Er worden zoveel mogelijk processen afgerond (gaat makkelijk als je kleine processen voorrang geeft)
- De maximale turn-around tijd is niet te hoog (maar ook weer niet te veel voorrang).

Bestudeer het gedrag van je scheduler ook voor belastingen van 0.5 en 0.8. Voor de hoog-niveau scheduler moet je minimaal een FCFS geheugen-toewijzing implementeren. Als je dat correct doet, en wat experimenten kan laten zien en het resultaat uitleggen, heb je al een voldoende verdiend.

CPU scheduler

Voor een goede beoordeling moet in elk geval naast de bovengoemde hoog-niveau scheduler ook een Round-Robin CPU scheduler zijn gebouwd en beproefd.
Een goede CPU scheduler zorgt dat zowel de CPU als de I/O devices efficient worden gebruikt en liefst ook dat andere resources, zoals geheugen niet te lang door een proces bezet worden gehouden. Een eerste stap naar zo'n scheduler kan het gebruik van een Round-Robin (RR) scheduler zijn. Maak een RR CPU scheduler met een instelbare time-slice.

Je kan ook verder gaan met je onderzoek aan de scheduler en kijken of je taken prioriteiten kan geven op basis van hun eigenschappen en daarmee de throughput verder verhogen zonder dat je bepaalde taken helemaal uitsluit.

Om dit te kunnen doen moet je mogelijk over elke job wat meer onthouden dan de "pcb" datastructuur je normaal toestaat. Gebruik daarvoor een zelf te definiëren structure waar je de `your_admin` pointer naar laat wijzen. Vergeet niet een en ander weer netjes op te ruimen bij het beëindigen van de job. Verbetert het gedrag van je systeem vergeleken met een systeem met een eenvoudige FCFS CPU scheduler? Hoeveel? Hoe is de verdeling van het zoet en het zuur (om eens het politieke lingo te gebruiken)?

Je kan je programma b.v. vertalen met:

```
> gcc -Wall -ansi -O2 -o schedule schedule.c mem_alloc.o simul2.o mt19937ar.o -lm
```

maar zorg voor een makefile.

Beoordeling

Voor een voldoende moet je de hoog-niveau scheduler correct en netjes hebben geïmplementeerd (minimaal FCFS hoog niveau scheduling, liefst een wat slimmere hoog niveau scheduler); voor een 'goed' moet je ook een CPU scheduler hebben gemaakt en onderzocht.

Nakijken en Inleveren

Hoewel je geen uitgebreid verslag hoeft in te leveren, is het verstandig voor jezelf je bevindingen in een kort verslag te beschrijven. Geef daarin ook de overwegingen voor het gekozen ontwerp voor je scheduler. Gebruik dit verslag als een handleiding bij het presenteren van je resultaten aan de assistent. Lever je scheduler code plus het verslag (plain text, postscript of pdf) in als aangegeven op de inlever pagina.

Hoofdstuk 4

Opgave 3 - Geheugenbeheer

4.1 Doelstellingen

1. Het verkrijgen van inzicht in de administratie die onder functies als `malloc()` en `free()` ligt.
2. Inzicht in het effect van verschillende geheugentoewijzingsstrategiën.
3. Verdere oefening in C programmeren.
4. Het gebruiken van een eenvoudig simulatieprogramma.
5. Oefening in het doen van metingen.

4.2 Inleiding

In deze opgave wordt geheugen-allocatie bekeken. Om dat te kunnen doen is door de practicumleiding een programma geschreven dat aanroepen naar een memory-management routine simuleert alsof er door steeds nieuwe processen werkgeheugen wordt aangevraagd, en indien verkregen, een tijdje wordt gebruikt en daarna weer wordt teruggegeven. Het programma houdt allerlei gegevens bij over aanvragen, afwijzingen en toewijzingen en controleert b.v. ook of een stuk geheugen maar aan een proces tegelijk wordt toegewezen.

Er bestaat een groot aantal verschillende algoritmen waarmee geheugen kan worden toegewezen. We noemen First Fit, Best Fit, Worst Fit, Circular First Fit ofwel Next Fit, Buddy. In deze opgave gaan we twee zulke algoritmen implementeren:

- **Best Fit**. Hierbij wordt het kleinste gat opgezocht waar de aanvraag (plus benodigde administratie) in past. Als je een gat vindt waarin de aanvraag precies past, kan je meteen toewijzen.

- **Next Fit**. Zoek het eerste gat waarin de aanvraag past vanaf het einde van het meest recent toegewezen blok. Als je met zoeken aan het einde van het te beheren geheugen komt, begin je weer vooraan tot het put waar je bent begonnen.

De geschreven routines moeten weer worden getest met de door de practicumleiding geschreven simulator.

4.3 De opgave

Voor deze opgave moeten service routines worden geschreven die stukken geheugen toewijzen en vrijgeven. Bovendien moeten er weer een paar monitor routines worden geschreven.

De practicumleiding heeft een programma geschreven dat herhaaldelijk stukken geheugen aanvraagt en op een gegeven moment de toegewezen stukken weer vrijgeeft. De aangevraagde stukken geheugen zijn van willekeurige grootte en volgen een bepaalde verdeling. Het aanvragen en vrijgeven gebeurt op willekeurige (gesimuleerde) tijdstippen, zoals van echt op een computer draaiende processen verwacht mag worden. Wel is er voor gezorgd dat gemiddeld teveel geheugen wordt aangevraagd, zodat ook regelmatig aanvragen moeten worden geweigerd.

Er worden achtereenvolgens drie verschillende verdelingen van de grootte van de aangevraagde ruimte gebruikt, te weten, een uniforme verdeling, een (negatief) exponentiele verdeling en een verdeling waarbij de kans op een bepaalde aanvraag ongeveer evenredig is met 1/N, waarbij N de grootte van de aanvraag is. Deze laatste verdeling geeft aanleiding tot relatief meer hele kleine en hele grote aanvragen dan de negatief-exponentiële.

Voor iedere verdeling wordt eerst een behoorlijk aantal aanvragen gesimuleerd om te zorgen dat eventuele 'aanloop verschijnselen' voorbij zijn. Daarna wordt voor een op te geven aantal aanvragen een aantal statistische gegevens verzameld en aan het eind afgedrukt.

Het aanvragen van geheugen doet het programma door middel van het aanroepen van `mem_get`, het vrijgeven gebeurt door het aanroepen van `mem_free`. Bovendien roept het programma geregeld de functies `mem_internal` en `mem_available` aan, die dan karakteristieken moeten teruggeven. Deze waarden worden tezamen met de tot dan toe gebruikte cpu tijd afgedrukt zodat het geheugen gebruik en de performance gemeten en kunnen worden vergeleken. Ook roept het programma helemaal aan het begin `mem_init` aan, die voor de eventuele initialisaties zorgt.

Voordat het programma termineert, wordt '`mem_exit`' aangeroepen, die voor eventuele afsluitingen zorgt.

Voor jullie is alleen de gecompileerde versie van het programma beschikbaar. Je kan deze vinden op Blackboard.

Voor huisgebruik zijn ook andere gecompileerde versies beschikbaar. Daarnaast zijn er complete programma's beschikbaar (o.a. Linux en Cygwin) om te kunnen zien wat voor testuitvoer je idealiter zou kunnen verwachten. Hier is overigens weer een andere strategie dan de gevraagde gebruikt.

4.4 Algemeen

- door middel van '`mem_init`' krijg je een array van `MEM_SIZE` (gedefiniëerd in `mem_alloc.h`) long integers ('woorden') in beheer.
- de aangevraagde stukken geheugen hebben de grootte van een geheel aantal integers.
- er mag door de ingeleverde functies niets worden weggeschreven of afgedrukt; zet dus al je '`printf`'-statements - die je bijvoorbeeld gebruikt hebt bij het debuggen - tussen commentaarhaken of gebruik het `#ifdef` preprocessor commando. Vb.:

```
#define DEBUG
    ...
#ifdef DEBUG        fprintf(stderr,"mem_get: geen voldoende groot gat\n");
#endif
```

4.5 De routines

`void mem_init(long mem[])`

parameters: `mem`: het geheugen van `MEM_SIZE` long integers waarover het beheer moet worden gevoerd.

werking: Verricht de nodige initialisaties; dient dan ook te worden aangeroepen voordat enig van onderstaande functies wordt aangeroepen.

`long mem_get(long size)`

parameters: `size`: het aantal aangevraagde long integers (>0)

werking: Wijs een stuk geheugen toe ter grootte van size. De te schrijven routine moet werken volgens een van de gevraagde algoritmen.

return value: de index van de eerste long integer van het toegewezen stuk geheugen; -1 als het toewijzen niet lukt.

`void mem_free(long index)`

parameters : `index`: de index van de eerste long integer van een eerder toegewezen stuk geheugen.

werking: Geef het stuk geheugen dat begint op index weer vrij en probeer het eventueel samen te voegen met eventuele naburige gaten.
De routine probeert na te gaan of een correcte index is meegegeven. Bij een foute index doet hij verder niks.

`double mem_internal()`

parameters: geen

werking: Bereken de interne fragmentatie die gedefiniëerd wordt door: internal = (aantal loze woorden)/(totaal aantal toegewezen woorden) Loze woorden zijn woorden die niet zijn aangevraagd maar wel specifiek aan een proces zijn toegewezen, b.v. voor administratie of om een 'page' te vullen. Je administratie van de vrije ruimte hoort daar dus niet bij.

return value: De interne fragmentatie.

`void mem_available(long *tot_space, long *big_hole, long *nr_holes)`

parameters: `tot_space`: (pointer naar) een long integer die het totaal aantal aantal vrije long integers in het geheugen moet gaan bevatten.

`big_hole`: (pointer naar) een long integer die de grootte van het grootste gat moet weergeven (in long integers).

`nr_holes`: (pointer naar) een long integer die het aantal gaten moet gaan bevatten.

werking: Bereken bovenstaande drie karakteristieken. Het is zinvol om de waarde van `big_hole` te corrigeren voor de bij toewijzing eventueel benodigde administratie. Bij `tot_space` ligt dat een beetje anders. Je moet in ieder geval niet de grootte van ieder gat op deze wijze bijstellen bij de berekening van `tot_space`.

`void mem_exit(void)`

parameters: Geen

werking: `mem_exit` wordt alleen als laatste routine aangeroepen en beëindigt het gebruik van de memory manager. Deze routine zorgt zonodig voor opruimen en afronden, maar mag ook verder leeg zijn.

4.6 Toelichting en hints

Je routines zullen geheugenruimte nodig hebben voor hun eigen administratie. Het zal duidelijk zijn dat memory management routines slecht van hun eigen diensten gebruik kunnen maken. Dit heeft b.v. het gevolg dat voor de benodigde administratie geen geheugen kan worden aangevraagd door aanroepen naar b.v. `malloc`.

Er wordt van je verwacht dat je dat deel van de administratie dat met een toegewezen of vrij stuk geheugen te maken heeft ook in het te beheren geheugen zelf onder brengt. De meest vanzelfsprekende plaats hiervoor is vlak voor het uitgegeven stuk geheugen. Dit maakt ook de implementatie van `mem_free` eenvoudiger.

Je mag wel de gewone lokale stack variabelen in je routines gebruiken en b.v. ook een enkele statische variabele, b.v. om het adres van het toegewezen geheugen te onthouden en de locatie van de laatste toewijzing voor Next Fit.

Maak eerst een goed ontwerp voor je administratie; waarschijnlijk kan je dat voor beide algoritmen gebruiken. Hou er ook rekening mee dat het kleinste gat dat je bij een toewijzing mag overhouden, ook de administratie voor dat gat moet kunnen bevatten (misschien moet je daarom soms een paar woorden extra toewijzen).

Reken die en de voor de administratie gebruikte geheugenplaatsen mee als 'loos' bij de berekening van de interne fragmentatie.

Jullie moeten twee files maken: `worst-fit.c` waarin de service routines komen te staan die met Worst Fit werken, en `first-fit.c` voor de First Fit routines. In beide files moeten dan (minstens) alle vijf gespecificeerde functies staan. Je kunt dan als volgt compileren:

```
> gcc -O2 -Wall -ansi -o next-fit mem_test2.o next-fit.c -lm
```

Maak een geschikte `Makefile` voor `worst-fit` en `first-fit`

Ons testprogramma zorgt ervoor dat je een overzicht krijgt van een aantal maten voor de system performance (b.v. de fractie mislukte aanvragen als functie van de grootte van het gevraagde blok, en de gemiddelde bezetting van het geheugen.) Deze gegevens krijg je b.v. door:

```
> script besttest
>> best-fit... invoer ...
>> exit
> vi besttest
```

Let op - na het gebruik van script staat aan het einde van elke regel nog een `^M` teken dat je zal willen verwijderen. Dat kan b.v. met de volgende vi opdracht:

```
:1,$s/^V^M$//
```

Het `^` teken staat hier voor de control toets, type dus eerst een `^V`, direct gevolgd door een `^M`. Je kan natuurlijk ook het programma `dos2unix` gebruiken.

De uitvoer van het programma voor elk van de drie verdelingen ziet er ongeveer zó uit:

```
Gemiddelden over 1000 aanvragen:Toegewezen: 439 ; Geweigerd: 561
De gemiddelde grootte van de toegewezen aanvragen is 5120.1
De gemiddelde grootte van de geweigerde aanvragen is 10574.5
Het experiment liep over 336.475783 gesimuleerde tijdseenheden
Gemiddeld over deze tijd waren er -
    8741.7 woorden geheugen in gebruik
    7642.3 woorden geheugen vrij
       0.0 woorden geheugen in fragmenten
Er waren gemiddeld 1.74 stukken geheugen toegewezen (max 5, min 0)
Er waren gemiddeld 1.49 gaten, met een maximum van 4 en een minimum van 1
De gemiddelde interne fragmentatie was  0.0000
```

Verder worden er nog twee histogrammen geproduceerd. De eerste laat als functie van de grootte van het aangevraagde stuk geheugen zien hoeveel aanvragen er zijn gehonoreerd en hoeveel er zijn afgewezen.

De tweede laat zien wat de relatieve kans was een gat van een bepaalde grootte aan te treffen als grootste gat. Bij kleine aantallen aanvragen zien de histogrammen er soms wat wonderlijk uit, maar laat je daardoor niet van de wijs brengen.

De betekenis van de meeste gegevens in de uitvoer zal zonder meer duidelijk zijn. De 'tijd' die wordt gebruikt is een gesimuleerde tijd, en heeft dus b.v. niets met de efficiency van jouw implementatie van het algoritme uit te staan. Het gemiddelde van de produkten van de groottes van de aanvraag en de duur levert een maat voor de belasting; per slot zal een grote aanvraag die maar kort nodig is het systeem minder zwaar belasten dan een die het gevraagde geheugen ook nog lang vasthoudt.

Wees er op bedacht dat het simulatie-programma de aanvragen op basis van een random-generator doet en dat deze zo is opgezet dat de startwaarde normaliter uit de klok wordt afgeleid. Je mag dan niet verwachten dat een programma twee keer hetzelfde resultaat zal opleveren. Als je dat wel wil, kan je een random seed waarde als eerst parameter op de commandoregel meegeven. Zo kan je beide algoritmen b.v. nauwkeuriger vergelijken. Doe dan wel een aantal experimenten.

4.7 Nakijken en Inleveren

Voor data, zie pagina over inleveren.
Hoewel een verslag niet strikt vereist is, is het net als bij de opgave over scheduling verstandig een log van je experimenten bij te houden als leidraad bij het nakijken.
Wat moet worden ingeleverd:

- Een makefile waarmee de programma's 'best-fit' en 'next-fit' kunnen worden gemaakt.
- De source files 'best-fit.c' en 'next-fit.c'
- Waar nodig een korte toelichting bij je implementaties.
- Indien beschikbaar, je log van je experimenten.

Beoordeling

- Correcte werking programma.
- Nette en overzichtelijke implementatie.
- Correcte afhandeling van bijzondere gevallen, zoals zeer kleine gaten.
- Een duidelijke uitleg van hoe je programma werkt.
- Een duidelijke uitleg aan de hand van je meetresultaten van hoe beide algoritmen presteren.

Hoofdstuk 5

Opgave 4 – Filesystemen - File Systems

One of the principal functions of an operating system is to support the storage and retrieval of data in a reliable, robust and efficient manner. With the steady increase in capacity of storage systems over the years (from tens of MB for a home system around 1987, to several TB in 2013), the increase in demand on these storage systems (from simple text and small images to many high-resolution videos and more), the requirements on the software managing these storage devices have changed drastically. In this assignment, we will look at a version of the FAT-based file management system. These systems are still the standard for e.g. memory sticks, memory cards in cameras etc. The version that we will be looking at is the FAT-12 system as used for floppy disks. It has sufficient complexity to demonstrate many of the principles behind file managent systems and is simple enough that one can readily understand how it works. Read about FAT file systems e.g. on Wikipedia[1] and, of course, in the sheets on file systems.

In this assignment, you will be required to do the following:

1. Extract files from a floppy disk image provided by the course management.
2. Analyse in what ways a FAT-12 system could be damaged and thus become inconsistent. Describe your conclusions in your report.
3. The course management has created a number of damaged versions of the floppy disk image. Find out what is wrong with each of those images.
4. Write code to automatically detect as many of such inconsistencies as you can.
5. Analyse which of the inconsistencies could be fixed automatically and how (you do not need to write code for this) and for which inconsistencies no easy fix is possible. Describe your conclusions in your report.
6. Write a clear report on you findings and the capabilities of your code.
7. Submit code and report by the deadline (May 26, 2013).

The course management provides the basic code to read a FAT-12 disk image and several such disk images at `http://staff.science.uva.nl/~dick/education/FAT-12/` and on Blackboard. The good disk-image is not necessarily the first one!

[1] `http://en.wikipedia.org/wiki/File_Allocation_Table`

Please note, that this assignment, being a final assignment, will not be corrected during the practical work sessions. It will be graded on a scale of zero to ten. If the grades for your other assigments are "good" (an 9), this assignment, plus assignment nr. 5 will determine your final grade if they are graded with higher marks.

Hoofdstuk 6

Opgave 5 – Filosofen en threads

N.B. Deze opgave is bedoeld voor alle studenten die niet met het tutoraat voor eerstejaars informatici 2012-2013 hoeven mee te doen, zoals zij in 2010 of eerder met hun studie zijn begonnen en die het oude vak "Operating Systemen"moeten afronden, bijvak studenten, e.d.

6.1 Doelstellingen

1. Het verkrijgen van inzicht in problemen rond racecondities en deadlock
2. Het programmeren met POSIX threads
3. Verdere oefening in C programmeren
4. Verdere oefening in het doen van metingen
5. Verdere oefening in het schrijven van verslagen

Deze opgave met groepjes van twee maken.
Kijk ook regelmatig op de opmerkingen van de assistent op Blackboard!

6.2 Filosofen en threads

Het gebruik van threads maakt het o.a. mogelijk om

- Bepaalde problemen helderder te programmeren doordat diverse onderdelen van een programma die afwisselend moeten worden uitgevoerd in hoge mate onafhankelijk van elkaar kunnen worden geformuleerd,
- Deeltaken, zoals het afhandelen van service-aanvragen in een server programma eenvoudig kunnen worden afgesplitst,
- De mogelijkheden van multi-processorsystemen beter kunnen worden benut.

Anderzijds kan het bestaan van meerdere onafhankelijke "threads of execution" binnen een programma aanleiding geven tot race-conditities, deadlock en starvation. Zoals bekend, is het

"dining philosophers[1]" probleem een standaardprobleem voor het illustreren van deadlock en starvation.
De opdracht is dan ook om het dining philosophers probleem uit te programmeren in C, gebruik makend van POSIX threads (pthreads).
De vorken / eetstokjes dienen te worden gemodelleerd als mutexen (pthread_mutex).
De opdracht moet aan de volgende eisen te voldoen:

1. De oorspronkelijke thread, die we de "host" zullen noemen voert de volgende taken uit:
   (a) Hij bepaalt hoeveel filosofen er gesimuleerd moeten worden (minimaal 2, maar er mag geen ingeprogrammeerde bovengrens zijn)
   (b) Hij initialiseert de diverse benodigde variabelen, zoals de mutexen voor de eetstokjes
   (c) Hij maakt de threads waarmee de filosofen zullen worden gesimuleerd, en kiest het gedrag van elke individuele filosoof (daarover later meer).
   (d) Daarna controleert hij minimaal 100 maal per seconde of er misschien deadlock is ontstaan. Als dat mocht zijn gebeurd, zorgt hij ervoor dat de simulatie netjes wordt beëindigd. Tussen de controles door mag de host geen CPU-tijd gebruiken.
   (e) Na 10 seconden signaleert hij alle filosofen dat het feest over is en dat ze moeten gaan afrekenen.
   (f) Hij voert een reeks `phread_join`-s uit voor de simulatie-threads.
2. Elke filosoof voert de volgende acties uit:
   (a) Zij trekt een nummertje, waarmee de plaats aan tafel wordt bepaald en dus ook de nummers van de te gebruiken eetstokjes (nb een apart af te schermen kritieke sectie).
   (b) Zij meldt de volgende informatie (naar `stdout`, via `printf`; dit is weer een aparte kritieke sectie): stoelnummer, soort gedrag.
   (c) Zij probeert vervolgens twee eetstokjes te pakken (afhankelijk van het gekozen gedrag kan dat mislukken, zie onder punt 4).
   (d) Als het gelukt is die twee stokjes te pakken gaat ze eten en legt vervolgens de twee stokjes weer netjes terug.
   (e) Als er geen signaal is dat er nu moet worden afgerekend, probeert ze weer opnieuw twee eetstokjes te pakken (terug naar stap 3).
   (f) Afrekenen bestaat er uit te melden hoe vaak deze filosoof heeft geprobeerd stokjes en pakken, en hoe vaak ze feitelijk heeft gegeten.
   (g) Na het afrekenen verlaat ze de zaak middels een `pthread_exit`.
3. Beëindigen deadlock: de host zet een vlag aan die aangeeft dat de simulatie moet worden beëindigd, pakt vervolgens alle geclaimde eetstokjes weer van de filosofen af en maakt ze weer beschikbaar. De filosofen komen op die manier uit de lock, en rapporteren netjes hun resultaten tot zover.

[1] `http://en.wikipedia.org/wiki/Dining.philosophers`

4. Mogelijke gedragingen van de filosofen. NB. Bij geen van deze gedragingen hoort gebruik te worden gemaakt van `sleep` of `usleep` of dergelijke functies om eten of filosoferen te simuleren. Stokjes worden direct na het verkrijgen weer teruggelegd, en direct na het terugleggen wordt opnieuw geprobeerd. Wel is het soms zinvol een `sched_yield`[2] te gebruiken (waarom?). Minimaal de vier onderstaande gedragingen moeten worden geprogrammeerd; je zou daarnaast b.v. ook nog een wispelturige filosoof kunnen maken die steeds van gedrag wisselt:

   (a) Rechtshandig: wacht tot een rechterstokje vrij is, pak dat, wacht dan op het linkerstokje, pak dat en ga eten.

   (b) Linkshandig: net zo, maar dan eerst het linkerstokje.

   (c) Opportunistisch: probeert b.v. eerst rechts (`trylock`), indien niet beschikbaar meteen links (ook met een `trylock`). Als er een stokje is verkregen, doe je een lock op de andere. Als er geen stokje is verkregen doet de filosoof een `sched_yield`, waardoor een andere thread aan de beurt komt, een probeert daarna meteen weer opnieuw.

   (d) Verlegen: de enige van deze vier gedragingen die tot een mislukte pak-poging kan leiden. De filosoof probeert b.v. eerst het rechterstokje te pakken. Als dat niet lukt geeft ze het op (mislukking) met een `sched_yield`, maar probeert het natuurlijk direct daarna opnieuw. Lukt het pakken wel, dan wordt het andere stokje geprobeerd. Is dat er niet, dan wordt het eerste stokje ook weer teruggelegd en wordt een mislukte poging geregistreerd. Na een `sched_yield` wordt wel meteen opnieuw geprobeerd.

5. Voer experimenten met verschillende aantallen filosofen en verschillende samenstellingen van de groep. Bij welke combinaties treedt deadlock op, en hoe lang duurt het gemiddeld totdat dat gebeurt? Treedt er starvation op en zo ja, in welke situaties. Is die starvation absoluut (dat een bepaalde filosoof echt nooit aan de beurt komt), of relatief (een filosoof komt veel minder aan de beurt dan de anderen).

Inleveren

Deze opgave heeft dezelfde deadline als opgave 4.
Als gebruikelijk: Lever sources, Makefile en een kort verslag in volgens de aanwijzingen over inleveren.

[2] `http://www.opengroup.org/onlinepubs/007908799/xsh/sched.yield.html`

Inhoudsopgave