Bachelor
in de Industriële Wetenschappen: Informatica

Opleiding Industrieel Ingenieur
Campus Schoonmeersen

BESTURINGSSYSTEMEN II: LABO

D. Pareit
W. Van den Breen

Uitgave: 2015-2016

Deel I: Inleiding

Lokaal inloggen op een Linux toestel kan op twee manieren gebeuren. Je kan inloggen op de grafische console of je kan inloggen op een van de vijf a zes tekstuele consoles. Veranderen van console kan via de toetsencombinaties CTRL-ALT-F1, CTRL-ALT-F2, … , CTRL-ALT-F7. Wanneer je Linux draait m.b.v. VirtualBox, dan moet je mogelijks gebruikmaken van de toetsencombinaties HOSTKEY-F1, HOSTKEY-F2, …, HOSTKEY-F7 waarbij de HOSTKEY standaard is ingesteld op de rechter CTRL-toets. Na het opstarten van Linux wordt aan iedere console een tty (terminal type) gekoppeld of in het geval van een grafische console een login manager zoals bv. gdm (GNOME Display Manager).

Bij Fedora Linux en aanverwante distributies zal de eerste console ingenomen worden door de grafische console. De overige tekstuele consoles vind je door te switchen met CTRL-ALT-Fx. Voor Debian, Ubuntu, … zal je merken dat de grafische console terug te vinden is op de zevende terminal.

Inloggen op meerdere terminals is mogelijk. Wanneer je correct inlogt op een tekstuele console kom je terecht in een BASH-shell. Bij het inloggen op de grafische console wordt naar alle waarschijnlijkheid een desktopomgeving zoals bv. GNOME, KDE, XFCE, … gestart. In die desktopomgeving kan je dan allerhande GUI-gebaseerde programma's starten naar analogie met MS Windows.

Voor deze labo's is het gebruik van een BASH-shell noodzakelijk. Op een tekstuele console beschik je onmiddellijk na het inloggen over een BASH-shell en op een grafische console zal je een terminal-emulator moeten starten.

1. De terminal

Log in als gebruiker *root* met als wachtwoord *e=mc**2* en start wanneer van toepassing in GNOME de terminal-emulator. Tik in het venster de opdracht "**stty -a"** in.

1. Welke toetsencombinatie heb je nodig om een runnend programma te onderbreken?
2. Hoe kan je een runnend programma pauzeren? (het hervatten van een programma zullen we later zien)
3. Wanneer een programma vraagt om gegevens in te typen, met welke toetsencombinatie kan je dan aangeven dat de invoer stopt?

   Je kan vraag 1 en vraag 3 het best uitproberen m.b.v. het commando "**cat**". Deze opdracht kopieert alle invoer van het toetsenbord (standaard invoer) naar het scherm (standaard uitvoer).

Probeer dit nu ook uit met de opdracht "**mail root@localhost**". Hier zal eerst gevraagd worden om het onderwerp van het bericht in te geven, gevolgd door het bericht zelf. Wanneer je het einde van de invoer aangeeft, zal een mail verstuurd worden naar de gebruiker root op het lokale toestel. De gebruiker root kan zijn mail lezen door de opdracht **mail** uit te voeren zonder parameters. Hierdoor kom je terecht in een interactieve omgeving waar je een overzicht krijgt van alle berichten. Bij het intikken van een nummer dat naast een bericht staat, kan je de mail lezen. Het bericht verwijderen kan door het ingeven van de letter d gevolgd door het berichtnummer. De interactieve omgeving verlaten kan door het ingeven van de letter **q**.

4. Wanneer je op de commandolijn een aantal woorden intikt, hoe kan je dan het laatste woord verwijderen?

5. In een shell kan je aan "auto completion" doen door gebruik te maken van de tab-toets. Tik de letters "les" in, en tik vervolgens op de tab-toets. Je merkt dat er automatisch less verschijnt. Nu kan het zijn dat er nog opdrachten zijn die met less beginnen. Tik nogmaals op de tab-toets om te kijken welke opdrachten er met less beginnen. Een ander handigheidje is "reverse search" om eerder ingetypte commando's te suggereren tijdens het typen. Gebruik hiervoor "CTRL+r".

6. Linux telt enkele duizenden opdrachten met elk nog een variabel aantal opties. De opdracht zelf wordt doorgaans beperkt tot enkele letters die de afkorting vormen van een woord dat de functie van de opdracht deels omschrijft. Zo staat het commando "**ls"** voor "list**"**, "**mkdir**" voor "make directory", het commando "**wc"** voor "word count**"**, het commando "**cc"** voor "C-compiler**"**, het commando "**dd**" voor "convert and copy**"**. Bij het laatste voorbeeld is er een conflict omdat de afkorting cc reeds gebruikt wordt voor de C-compiler. Als oplossing heeft men de volgende letter van het alfabet gebruikt.

   "Auto completion" voor commando's werd reeds in het vorige punt naar voor gebracht maar geldt ook voor de bijhorende opties. Geef op de commandolijn het commando "**ls -**" in en gebruik vervolgens de tab-toets. Je merkt dat er een pak opties verschijnen die voorafgegaan worden door twee mintekens. Onder Linux kan je veelal opties op de commandolijn meegeven in twee formaten. Het korte formaat bestaat uit één letter en wordt voorafgegaan door één minteken. Dezelfde optie kan je ook meegeven d.m.v. het lange formaat. In dat geval zal de optie voorafgegaan worden door twee opeenvolgende mintekens. Wanneer je de keuze hebt tussen de twee formaten zal je merken dat de korte variant de voorkeur geniet.

Zo kan het commando "**wc --words --lines /etc/passwd**" ook geschreven worden als "**wc -w -l /etc/passwd**" en zelfs als "**wc -wl /etc/passwd**". De opdracht toont het aantal lijnen en het aantal woorden uit het bestand /etc/passwd.

2. Gebruik van enkele eenvoudige opdrachten

    1. Met de opdracht man kan je alle info over een welbepaalde opdracht achterhalen. Zo zal de opdracht "**man ls**" de info tonen van de ls-opdracht.

       Man toont de informatie a.d.h.v. een pager, een programma dat de informatie niet alleen op het scherm toont maar dat ook gebruikersinteractie toelaat. Zo kan je scrollen, zoeken naar bepaalde woorden, etc. Standaard is de pager ingesteld op het commando **less**. Dit betekent dat wanneer je wil weten hoe je voorwaarts moet zoeken in een manpagina, je dit moet gaan zoeken bij het commando less en niet bij het commando man. Vraag de manpagina op van het commando less en ga na hoe je een tekst die met less wordt getoond kan afsluiten. Hoe kan je voorwaarts zoeken in een manpagina?
    2. Hoe kan je zorgen dat er bij het zoeken in een manpagina geen rekening wordt gehouden met het verschil tussen hoofd- en kleine letters?
    3. Bekijk de manpagina van het commando man en ga na hoeveel secties er gekend zijn.
    4. Wanneer je de opdracht "**man read**" intikt krijg je de info te zien van het commando read. Er is echter ook een systeemaanroep read aanwezig. Hoe kan je aan man meegeven dat je niet de info wenst te zien van het commando read maar wel van de systeemaanroep read?
    5. Met de opdracht "**ls"** krijg je van een directory een overzicht van alle bestanden en subdirectories te zien. Wanneer je geen directory opgeeft, wordt de huidige werkdirectory genomen. Wat doen de opties **-l** en **-h**?
    6. Bekijk met "**ls /"** de inhoud van de hoofddirectory.
    7. In punt 5 en 6 heb je gemerkt dat de uitvoer voorzien wordt van kleuren. Zo worden bv. directories in het donkerblauw gekleurd en symbolische links in het lichtblauw. Dit gedrag is te wijten aan het feit dat er voor ls een alias gedefinieerd is die telkens ls vervangt door "**ls --color=auto**". Ga met de opdracht "**alias**" na welke andere aliassen er bestaan.
    8. Met de opdracht "**cd**" kan je van werkdirectory veranderen. Wanneer je geen directorynaam opgeeft wordt je home-directory de nieuwe werkdirectory. Voer "**cd /tmp**" uit en keer daarna terug naar je home-directory.
    9. Een directory aanmaken kan via de opdracht "**mkdir**". Maak in je home-directory een directory aan met als naam 'c' waar je toekomstige C-programma's naartoe kan kopiëren.
    10. Een bestand kopiëren gebeurt via de opdracht "**cp**". De eerste parameter is de bron, de tweede de bestemming. Een bestand verplaatsen doe je via de opdracht "**mv**".

Deel II: Compileren in de Shell

In Linux is het vrij eenvoudig om een C of C++ programma te compileren zonder gebruik te maken van een IDE. Het enige wat je nodig hebt is een eenvoudige tekstverwerker. In een grafische omgeving kun je bvb. **gedit** gebruiken. In tekstuele consoles is **nano** eenvoudig te gebruiken, terwijl **vi**, **vim**, **emacs**, … al wat geavanceerder zijn.

Open in BASH een tekstverwerker naar keuze en tik daarin onderstaand codefragment:

```
#include <stdio.h>

int
main(){
        printf("Hello world!");
        return 0;
}
```

Sla dit bestand vervolgens op als hello.c (let op de extensie). Om nu dit bestand te compileren voer je op de commandolijn "**cc hello.c -o hello**" of "**make hello**" uit. Na het succesvol compileren kan je bestand uitvoeren door op de commandolijn "**./hello**" in te tikken. Bemerk dat je voor het uitvoerbaar bestand de "./" niet mag vergeten!

Wanneer je met de opdracht "**file**" informatie over het uitvoerbaar bestand opvraagt, zal je merken dat er gebruikgemaakt wordt van **shared libraries** en dat het uitvoerbaar bestand dus niet alle code bevat om de string "Hello world!" naar het scherm te schrijven. Is dit niet de bedoeling, dan kan je bij het compileren de compileroptie "**-static**" opgeven. Hierdoor wordt alles statisch gelinkt en krijg je dus een uitvoerbaar bestand dat alle code bevat om de gegeven tekst naar het scherm te schrijven.

Opdracht

Compileer bovenstaand bestand dynamisch en statisch en bekijk het verschil in grootte. De grootte van een bestand kan je het gemakkelijkst achterhalen met de opdracht "**du**" of met de opdracht "**ls**". Maak in beide gevallen handig gebruik van de optie "-h" om de bestandsgrootte in bytes, KB, MB, … te krijgen.

Deel III: Profiler programma's

Bij het schrijven van programma's zijn er een aantal zaken waar je toch moet op letten. Zo mag bv. de geschreven code geen memory leakage vertonen, mag de uitvoeringstijd niet te groot zijn en moet het aantal systeemaanroepen tot een minimum worden beperkt. Onder Linux zijn er een aantal profiler programma's waarmee je uitvoerbare bestanden kan onderzoeken.

Valgrind

Valgrind is een tool die in zijn meest eenvoudige vorm onderzoekt of de geschreven software memory leakage vertoond.

Opdracht

Schrijf een C-programma dat via malloc een tabel van 2000 gehele getallen aanmaakt en deze getallen daarna gewoon uitschrijft. Je hoeft bewust het in beslag genomen geheugen niet vrij te geven. Compileer het programma en voer het daarna uit d.m.v. "**valgrind ./prog**". Herschrijf nu het programma waar je het gealloceerde geheugen vrijgeeft en controleer opnieuw met valgrind of het programma nog geheugenlekken vertoond.

Time

Om na te gaan hoeveel CPU-tijd een bepaald programma in beslag heeft genomen kan je de opdracht "**time**" gebruiken. Het commando time geeft standaard drie timestamps terug. Het betreft, de totale tijd tussen het starten en het stoppen van het proces, de CPU-tijd voor het uitvoeren van code in USER-mode en tot slot de CPU-tijd voor het uitvoeren van code in de KERNEL-mode. De allereerste tijdsaanduiding is eigenlijk de minst interessante omdat een proces doorgaans niet in één keer wordt uitgevoerd maar door het besturingssysteem verschillende keren wordt onderbroken.

Opdracht

Herneem het programma dat je daarnet hebt geschreven en voer het nu uit d.m.v. "**time ./prog**".

Strace

Soms kan het interessant zijn om te weten welke systeemaanroepen een programma gebruikt. Bekijk de uitvoer van "**strace cat /etc/passwd**". Dit geeft een sequentieel overzicht van alles wat bij het uitvoeren van "cat /etc/passwd" gebeurt. Wanneer je een overzicht wil krijgen van welke systeemaanroepen er gebruikt worden en hoeveel keer ze werden opgeroepen, dan kan je aan strace de optie "**-c**" meegeven.

Deel IV: Een eerste programmeeropdracht

Er zijn een aantal commando's die "ls" als prefix hebben en die allemaal wel de inhoud van "iets" naar het scherm schrijven. Net zoals **ls** de inhoud van een directory toont, toont **lspci** de aanwezige PCI(e)-apparaten, toont **lsusb** de aanwezige USB-apparaten en toont **lsmod** de modules, of drivers, die momenteel door het systeem gebruikt worden.

Opdracht

De bedoeling is een eigen versie van het commando "**lspci -n**" te programmeren in de programmeertaal C. Dit commando overloopt alle mogelijke PCI-adressen en gaat na of er zich een apparaat bevindt. Bevindt er zich een apparaat, dan wordt het adres (busnummer, devicenummer en functienummer) naar het scherm geschreven, samen met het vendorID en het deviceID.

Info en werkwijze

Om een register van een bepaald apparaat, aangesloten op de PCI-bus, aan te spreken moet je te beschikken over het busnummer, devicenummer, functienummer en tot slot het registernummer.

Het aantal bussen binnen het PCI-systeem kan oplopen tot 256. Om te zoeken naar hardware is het dus aangewezen om alle mogelijke bussen af te lopen.

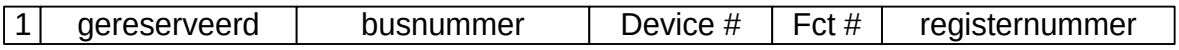
Een device, of apparaat is een fysiek aangesloten “ding” op het PCI-systeem. Voorbeelden hiervan zijn videokaarten, geluidskaarten, onderdelen van de chipset, netwerkadapters, etc. Iedere bus kan theoretisch 32 apparaten bevatten.

Ieder apparaat heeft ten minste één functie en ieder aangesloten apparaat op de PCI-bus kan maximum 8 functies hebben, genummerd van 0 tot 7.

Iedere functie van een apparaat heeft 256 registers. Registers 0 tot 3F bevatten een zee aan informatie over een bepaalde functie van een bepaald apparaat. Registers 40 tot FF bevatten dan weer merkspecifieke informatie.

Registers 0 en 1 bevatten het vendorID en registers 2 en 3 bevatten het deviceID. Wanneer bijvoorbeeld het vendorID 8086 is en het deviceID 2829 kan je vrij snel achterhalen dat het om een SATA AHCI controller gaat van Intel. Deze informatie kan je terugvinden in de PCI vendor database (http://www.pcidatabase.com).

Om de PCI-bus af te scannen moet je telkens een 32-bit getal naar het indexregister (adres = 0xcf8) van het PCI-systeem sturen. Dit 32-bit getal heeft steeds de volgende structuur:



31 0

Je laat het busnummer variëren van 0 tot 255 (8 bit), het devicenummer telkens van 0 tot 32 (5 bit) en ten slotte het functienummer van 0 tot 7 (3 bit). Het registernummer (8 bit) stel je in op 0.

Het antwoord krijg je door vervolgens een 32-bit getal te lezen van het dataregister (adres = 0xcfc). Indien het antwoord uitsluitend 1-bits telt (0xffffffff), is er op die plaats geen apparaat aanwezig. In het andere geval ontvang je de inhoud van de eerste vier registers (registers 0 tot 3) van de geadresseerde functie. De meest significante 16-bits bevatten het deviceID en de minst significante 16-bits het vendorID.

Opmerkingen:

- Om rechtstreeks de hardware te kunnen benaderen, moet je in je code daar expliciet toestemming voor vragen en ook voor krijgen. Bekijk de man-pagina's van de systeemaanroepen **iopl** en **ioperm**.
- Net zoals de meeste systeemaanroepen geven deze functies -1 terug wanneer er iets foutloopt en wordt de gehele variabele **errno** ingesteld. Om de foutboodschap die bij **errno** hoort naar het scherm te schrijven gebruik je de functie **perror(…)** die naast de string die als parameter wordt meegegeven ook de stringversie van errno op het scherm toont. Gebruik dus voor het tonen van foutboodschappen van systeemaanroepen steeds de perror-functie en maak dus geen gebruik van printf.
- Om een 32-bit getal naar een adres te schrijven kan je gebruikmaken van de systeemaanroep "**outl**". Een 32-bit getal van een adres lezen kan via "**inl**". Bekijk de manpagina van outl/inl voor de juiste syntax. Opgelet, inl en outl zijn ook gewone Linux-commando's. Zorg er dus voor dat je de juiste man-pagina oproept.

Deel V: I/O-Systeemaanroepen

Alle systeemaanroepen die een verwijzing nodig hebben naar een open bestand, maken gebruik van een **file descriptor**. Een file descriptor is doorgaans een klein positief geheel getal dat kan verwijzen naar gewone bestanden maar ook naar directories, sockets, pipes, …. Alles wat eigenlijk op het bestandssysteem bestaat kan worden geopend en kan aan een file descriptor worden gekoppeld. Bij het starten van een programma zullen drie file descriptors van de shell worden overgeërfd. De drie file descriptors verwijzen respectievelijk naar het **standaard invoerkanaal**, het **standaard uitvoerkanaal** en het **standaard foutenkanaal**. Onderstaande tabel geeft een overzicht.

File descriptor	Doel	POSIX-naam	stdio-stream
0	Standaard invoerkanaal	STDIN_FILENO	stdin
1	Standaard uitvoerkanaal	STDOUT_FILENO	stdout
2	Standaard foutenkanaal	STDERR_FILENO	stderr

Een bestand openen gebeurt via de systeemaanroep **open(pathname, flags, mode)** die een geheel getal, de file descriptor dus, teruggeeft of -1 wanneer er een fout optreedt. De eerste paramater die je moet opgeven is het pad naar het bestand. De tweede parameter bevat een aantal statusvlaggen die bij het openen van het bestand worden geraadpleegd. Wanneer je een bestand wil openen om er uitsluitend van te lezen, geef je als flags *"O_RDONLY"* op. Wanneer je naar een bestand wil schrijven en er ook van wil lezen is de flags-parameter gelijk aan *"O_RDWR"*. Combineren van statusvlaggen gebeurt via de logische |-operator. Zo kan je een bestand openen om er naar te schrijven en tevens opgeven dat het moet worden aangemaakt wanneer het niet zou bestaan. De flags-parameters is in dit geval "*O_WRONLY | O_CREAT*". De combinatie *"O_WRONLY | O_CREAT | O_APPEND"* betekent dat het bestand zal worden aangemaakt wanneer het niet bestaat, dat er enkel naar geschreven zal worden en dat alle data die er naartoe wordt geschreven achteraan zal worden toegevoegd. De O_APPEND vlag verzekert dus dat indien het bestand reeds zou bestaan, de bestaande inhoud niet zal worden overschreven.

De laatste parameter van de systeemaanroep open is optioneel en is enkel van belang wanneer de O_CREAT vlag is opgegeven.

Om van een file descriptor te lezen gebruik je de systeemaanroep **read(fd,buffer,count)**. Buffer is een void-pointer naar een geheugenlocatie waar de gelezen bytes zullen worden weggeschreven. Count is het aantal bytes die je van de opgegeven file descriptor wil lezen. De read-systeemaanroep geeft het aantal gelezen bytes terug en -1 wanneer er bij het lezen een fout optreedt.

Met de **write(fd,buffer,count)** systeemaanroep kan je bytes van een geheugenlocatie naar een filedescriptor schrijven. De parameters zijn gelijkaardig aan die van de read-systeemaanroep en ook de return-waarde is hetzelfde, i.e. het aantal weggeschreven bytes of -1.

Tot slot kan je met de systeemaanroep **close(fd)** een filedescriptor sluiten.

Opdrachten

1. Schrijf een C-programma dat een bestand van ongeveer 10MB aanmaakt met willekeurige lettertekens gelegen in het gesloten interval [a..z].
2. Tot nog toe werd er niets gezegd over de optimale buffergrootte. De bedoeling is een C-programma te ontwikkelen dat het hierboven aangemaakte bestand verschillende keren inleest en dit met buffergroottes van 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 en tot slot 8192 bytes. De uitvoer van het programma moet er ongeveer zo uitzien:

```
BUF_SIZ=   1    Time=3.60
BUF_SIZ=   2    Time=1.79
BUF_SIZ=   4    Time=0.98
BUF_SIZ=   8    Time=0.48
BUF_SIZ=  16    Time=0.24
BUF_SIZ=  32    Time=0.12
BUF_SIZ=  64    Time=0.06
BUF_SIZ= 128    Time=0.03
BUF_SIZ= 256    Time=0.02
BUF_SIZ= 512    Time=0.01
BUF_SIZ=1024    Time=0.00
BUF_SIZ=2048    Time=0.00
BUF_SIZ=4096    Time=0.00
BUF_SIZ=8192    Time=0.00
```

   Om de tijd tussen het starten het stoppen met lezen op te meten kan je gebruikmaken van onderstaande code:

```
double start=clock();
…
double time=(clock()-start)/CLOCKS_PER_SEC;
```

   De clock-functie en de macro CLOCKS_PER_SEC vind je in de bibliotheek pthread.h.
3. Doe nu hetzelfde maar voor de write-systeemaanroep. Maak voor iedere verschillende buffergrootte een bestand aan van ongeveer 10MB (cfr. vraag 1). Nadat je de tijd voor

een gegeven buffergrootte hebt opgemeten moet je vanzelfsprekend het aangemaakte bestand terug verwijderen. Een bestand verwijderen kan je doen m.b.v. de **unlink**-systeemaanroep.

4. Herneem vraag 3 maar voeg aan de flags-parameter de *O_SYNC* vlag toe. Dit laatste zorgt ervoor dat er noch in USER-mode noch in KERNEL-mode zal worden gebufferd en dat de bytes rechtstreeks naar de schijf zullen worden geschreven.

   Ter info, de vaste schijf beschikt om performantieredenen over een bepaalde hoeveelheid write-back-cache-geheugen waar gegevens van I/O- schrijfopdrachten tijdelijk worden bewaard. Dit betekent dat er bij een eventuele spanningsonderbreking wel degelijk gegevens kunnen verloren gaan. Bij programma's, zoals gegevensbanken, waar gegevensverlies nefast is, wordt er aangeraden om "write-caching" uit te schakelen. Dit kan je doen door in een shell de opdracht "**hdparm -W0 /dev/sda**" uit te voeren.
5. Bestudeer gronding de werking van de shell-opdracht **cat** en schrijf in C een eigen versie van cat. Bekijk bv. wat er gebeurt wanneer je de opdracht "**cat /etc/passwd - /etc**" opgeeft of wanneer cat geen argumenten meekrijgt. Wanneer je een directory opgeeft als argument, geeft cat een foutmelding. Dit gedrag hoef je niet na te bootsen.
6. Schrijf een eigen versie van de opdracht **cp** waar twee argumenten op de opdrachtlijn worden verwacht. Het eerste argument is het bronbestand en het tweede het doelbestand. Wanneer de eerste parameter een directory is, wordt een foutboodschap naar het scherm geschreven en stopt het programma met exit-status 1. Wanneer het tweede argument een directory is, wordt opnieuw gestopt met een passende foutboodschap en met exit-status 1.

   Om na te gaan of een argument een directory is, kan je gebruikmaken van de systeemaanroep **stat**. Om een programma te stoppen met exit-status 1 maak je best gebruik van de bibliotheekfunctie **exit** die dan achter de schermen de systeemaanroep **_exit** aanroept.
7. Schrijf een C-programma met als naam *watchfile.c* dat één argument, een bestand, op de opdrachtlijn verwacht. Het programma loopt in een oneindige lus en schrijft telkens een boodschap naar het scherm wanneer het bestand dat op de opdrachtlijn werd meegegeven werd gewijzigd. Wanneer er geen argument werd opgegeven of het argument is geen gewoon bestand wordt een foutboodschap getoond en wordt het programma afgesloten met exit-status 1.

Deel VI: Processen en POSIX-threads

Een nieuw proces aanmaken gebeurt met de systeemaanroep **fork()**. De systeemaanroep maakt een nieuw proces aan, het *kind*, dat bijna een exacte kopie is van het proces dat de systeemaanroep heeft gebruikt, het *ouderproces*. Na het beëindigen van de systeemaanroep beschik je over twee processen, het kind en het ouderproces, die *beide* de uitvoering verderzetten bij het terugkeren van de fork()-systeemaanroep. Beide processen beschikken over dezelfde programmatekst maar hebben elk hun eigen kopie van de stapel, de heap, en het datasegment. Initieel bevat de stapel, de heap, en het datasegment van het kindproces dezelfde informatie als de overeenkomstige segmenten van het ouderproces. Om in een programma het onderscheid te kunnen maken tussen het kind en het ouderproces moet je weten dat de systeemaanroep fork() twee verschillende return-waardes heeft. In het kindproces geeft fork() de waarde 0 terug, terwijl in het ouderproces het proces-ID van het nieuwe aangemaakte kindproces wordt teruggegeven.

Opdrachten

1. Gegeven onderstaande code:

```
int main(int argc, char **argv){
    …
    fork();
    fork();
    fork();
    …
}
```

   Voorspel hoeveel kindprocessen er zullen worden aangemaakt zonder de code uit te voeren.

2. Schrijf een programma dat drie kindprocessen aanmaakt en zorg ervoor dat ieder kindproces zijn proces-ID naar het scherm schrijft en daarna stopt. Het proces-ID kan je m.b.v. de systeemaanroep **getpid()** opvragen en een proces kan je beëindigen met de functie **exit(int exitstatus)**. De exitstatus van een correct beëindigd proces is steeds 0 terwijl een waarde verschillend van 0 duidt op een fout.

In bovenstaande opdracht kan je nog met een eenvoudige if/else-structuur het onderscheid maken tussen wat het ouderproces en wat het kindproces moet uitvoeren. Het kindproces moet immers alleen maar zijn proces-ID naar het scherm schrijven en zichzelf beëindigen. Wanneer het gaat om meer dan een paar lijnen code, is het interessanter om een nieuw programma te ontwikkelen, eventueel zelfs in een andere programmeertaal, dat je onmiddellijk na de fork()-systeemaanroep uitvoert. Hetzelfde geldt wanneer het kindproces een programma moet uitvoeren dat reeds bestaat en dat dus niet door jou nog moet worden ontwikkeld.

Een nieuw programma uitvoeren gebeurt met de systeemaanroep **execve(const char* filename, char *const argv[], char *const envp[])**. Execve zal het tekstsegment, het datasegment, de stapel en de heap van het huidige proces overschrijven met de gegevens van het uitvoerbaar bestand *filename*. Argumenten voor het nieuwe programma worden als tweede parameter *argv* opgegeven. Als laatste parameter *envp* kan je een tabel van strings, in de vorm van naam-waardeparen, doorgeven aan het nieuwe programma.

Bemerk dat bij de oplijsting van systeemaanroepen met **strace** execve op de eerste lijn staat. Concreet wil dit zeggen dat van het shellproces een kopie gemaakt wordt en dat onmiddellijk daarna het proces wordt overschreven met de inhoud van de opdracht die je op de opdrachtlijn hebt ingetikt.

Naast het rechtstreeks gebruik van de systeemaanroep execve, bestaan er een aantal *bibliotheek-functies* die allemaal na een aantal tussenstappen execve zullen aanroepen. Deze functies beginnen allemaal met het woord exec gevolgd door één of twee letters. Het verschil met execve zit hem in het aantal parameters en de manier waarop de argumenten aan het nieuwe programma worden doorgegeven. Hieronder vind je een overzicht.

Functie	Specificatie van de programmanaam	Specificatie van de argumenten	Environment-argumenten
Systeemaanroep			
execve	Volledige padnaam	Tabel van C-strings	envp-argument
Bibliotheekfuncties			
execle	Volledige padnaam	Opsomming van C-strings	envp-argument
execlp	Bestandsnaam(PATH wordt geraadpleegd)	Opsomming van C-strings	Niet van toepassing
execvp	Bestandsnaam(PATH wordt geraadpleegd)	Tabel van C-strings	Niet van toepassing
execv	Volledige padnaam	Tabel van C-strings	Niet van toepassing
execl	Volledige padnaam	Opsomming van C-strings	Niet van toepassing

3. Schrijf een C-programma **writestring.c** dat het proces-ID naar het scherm schrijft gevolgd door de string die als enige parameter wordt meegegeven en vervolgens 10 seconden wacht vooraleer te eindigen.
    1. Herschrijf nu opdracht 2 waarbij het kindproces de systeemaanroep **execv** gebruikt om **“writestring hello”** uit te voeren.
    Opgelet: In het ouderproces moet je wachten tot wanneer het kindproces klaar is, waarna je nog een boodschap naar het scherm schrijft. Dit doe je door gebruik te maken van de systeemaanroep **waitid**.

Voeg in het ouderproces de lijn **"waitid(P_ALL, pid, NULL, WEXITED);"** toe om te wachten op alle kindprocessen.

2. Idem als deel 1 maar maak nu gebruik van de systeemaanroep **execl**. De laatste C-string-parameter moet **(char *)0** zijn om het einde van de opsomming aan te geven. Bemerk dat gewoon 0 schrijven niet voldoende is en zelfs fout is. Wanneer de grootte van een int verschillend is van de grootte van een char *, zal het aantal argumenten dat doorgegeven wordt aan execl verkeerd zijn.

4. Schrijf een *C++*-programma met als naam *watchfiled.cc* dat alle bestanden in de gaten houdt die opgesomd zijn in het tekstbestand *watchfile.txt*. Telkens wanneer een opgesomd bestand in het tekstbestand wordt gewijzigd, wordt een boodschap naar het scherm geschreven. Het C++-programma loopt in een oneindige lus die telkens het bestand watchfile.txt lijn per lijn inleest en nagaat of het bestand dat zich op een gegeven lijn bevindt reeds in de gaten wordt gehouden. Wanneer dit niet zo is wordt een kindproces aangemaakt dat *watchfile* (zie Sectie V, opdracht 7) uitvoert op het "nieuwe" bestand.
5. Pas *watchfiled.cc* aan zodat nu ook rekening wordt gehouden met kindprocessen die beëindigd worden. Een kindproces voert watchfile uit en zal beëindigd worden wanneer het bestand bv. verwijderd of verplaatst wordt. De bedoeling is ook dat de bestandsnaam van het verwijderde/verplaatste bestand in watchfile.txt wordt verwijderd. Controleren of een proces gestopt is, kan met de systeemaanroep **waitpid**. Bekijk hoe je waitpid kan gebruiken in non-blocking mode ttz. zonder dat de uitvoering wordt gepauzeerd.

   Wanneer een gebruiker een bestand in watchfile.txt verwijdert zou je eigenlijk het bijhorend kindproces moeten stoppen. Dit hoef je niet te implementeren.

Interprocescommunicatie, afgekort IPC, in Unix en dus ook in Linux gebeurt veelal door gebruik te maken van pipes. Een **unamed pipe** of **anonymous pipe** wordt gebruikt wanneer twee processen rechtstreeks van elkaar afstammen. Processen die geen gemeenschappelijke band hebben, moeten gebruikmaken van **named pipes**. Een pipe is een **unidirectionele communicatielijn** tussen twee processen. Wanneer twee processen gegevens willen uitwisselen in beide richtingen heb je twee pipes nodig!

Om een anonieme pipe aan te maken kan je gebruikmaken van de systeemaanroep **pipe(int fd[2])** die -1 teruggeeft wanneer de pipe niet kan worden aangemaakt. In het andere geval zal fd twee filedescriptors bevatten, nl. fd[0] om van de pipe te lezen en fd[1] om naar de pipe te schrijven. Niettegenstaande dat zowel het ouderproces als het kindproces beide kunnen lezen en schrijven van/naar de pipe, wordt aangeraden om in beide processen de filedescriptor die niet gebruikt wordt te sluiten. Zoals reeds eerder aangegeven is de informatiestroom doorheen een pipe slechts in één richting, ttz. van het ouderproces naar

het kindproces of omgekeerd. Is communicatie in beide richtingen nodig dan moet je twee pipes voorzien.

Onderstaande code verwacht op de commandolijn één parameter, nl. het aantal aan te maken kindprocessen. Ieder kindproces genereert een willekeurig getal en stuurt het gegeneerde getal op naar het ouderproces. Het ouderproces wacht tot wanneer het kindproces zijn getal naar de pipe heeft geschreven om vervolgens het getal van de pipe te lezen en het getal naar het scherm te schrijven.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv){
	pid_t pid;
	if (argc!=2){
		printf("One argument expected!\n");
		exit(1);
	}
	else {
		int fd[2];
		int i;
		for(i=0;i<atoi(argv[1]);i++){
			if (pipe(fd)<0){
				perror(argv[0]);
				exit(1);
			}
			if ((pid=fork())<0){
				perror(argv[0]);
			}
			if (pid==0){
				close(fd[0]);
				srand(getpid());
				int number=rand();
				if ((write(fd[1],&number,sizeof(int))!=sizeof(int))<0){
					perror(argv[0]);
					exit(1);
				}
				exit(0);
			}
```

```
                else {
                    close(fd[1]);
                    if (wait(NULL)==pid){
                        int number;
                        read(fd[0],&number,sizeof(int));
                        printf("pid=%d value=%d!\n",pid,number);
                    }
                }
            }
        }
        return 0;
}
```

6. Pas de gegeven code aan zodat het ouderproces het grootste van de gegeneerde getallen bepaalt en vervolgens ieder kindproces op de hoogte brengt wie de winnaar is, ttz. welk proces het grootste getal heeft gegenereerd. De uitvoer met zes kindprocessen ziet er als volgt uit:

```
Process 1819 is the winner
I'm the winner!
Process 1819 is the winner
Process 1819 is the winner
Process 1819 is the winner
Process 1819 is the winner
```

POSIX-threads

De POSIX Pthreads standaard is de de-facto standaard voor het programmeren van multithreaded programma's in een POSIX-compatibele omgeving. Deze standaard specificeert de interface om threads te creëren en te manipuleren. Als bibliotheek zijn pthreads op de meeste platformen beschikbaar. De voornaamste primitieven zijn het opstarten van een thread(**pthread_create**) en het wachten op een thread(**pthread_join**). De standaard is vanzelfsprekend veel ruimer dan bovenvermelde functies.

Voor onderstaande opdrachten zijn de functies **pthread_create** en **pthread_join** van groot belang. De syntax van beide functies kan je hieronder vinden:

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

int pthread_join(pthread_t thread, void **retval);

Opdrachten

1. Schrijf een programma dat gebruikmaakt van vier threads die elk een verschillend cijfer naar het scherm schrijven. Wanneer een thread het bijhorend cijfer 100 keer naar het scherm heeft geschreven, stopt de thread.

2. Genereer 1.000.000.000 willekeurige reële getallen die je bijhoudt in een tabel. Schijf nu twee functies die zoeken naar respectievelijk het kleinste getal en het grootste getal en deze getallen als return-waarde teruggeven. Schrijf nu een hoofdprogramma dat gelijktijdig zoekt naar het grootste en het kleinste getal in een tabel van 1.000.000.000 reële getallen. Schrijf beide getallen naar het scherm.

3. Multithreading kan ook leiden tot snelheidswinst. Een mooi voorbeeld hiervan is bv. een matrixvermenigvuldiging. Om de snelheidswinst op te merken maak je best gebruik van twee vierkante matrices met 1000 rijen en 1000 kolommen. Ook gebruik je best vier tot acht threads om de het resultaat te berekenen. Gebruik voor de dimensie en ook voor het aantal threads constanten.

   Wanneer er bijvoorbeeld acht threads worden gebruikt, kan je het resultaat als volgt berekenen. De eerste thread laat je de 0^{de}, de 8^{ste}, de 16^{de}, … rij van het resultaat bepalen. De tweede thread ontfermt zich over de 1^{ste}, 9^{de}, 17^{de}, … rij van het resultaat. Iedere thread berekent dus DIM/8 rijen van het eindresultaat waarbij de rijen op een afstand van het aantal threads van elkaar liggen.

   Om het opvullen van een matrix vlot te laten verlopen, geef je het element op i^{de} rij en op de j^{de} kolom de waarde i+j. Dit kan eenvoudig worden geprogrammeerd a.d.h.v. een dubbele for-lus. Doe dit voor beide matrices en merk op dat je dus identieke matrices met elkaar vermenigvuldigt.

   Schrijf ook een programma dat geen Pthreads gebruikt om duidelijk het verschil in snelheid te zien.

4. Omdat iedere aangemaakte thread beschikt over zijn eigen stapel, kunnen ook recursieve functies parallel worden uitgevoerd. Om dit te illustreren maken we gebruik van de onderstaande recursieve sorteerfunctie:

```
void sort (int *tab, int n){
    if (n>1){
        /* sorteer linkerhelft */
        sort (tab,n/2);
        /*sorteer rechterhelft*/
        sort (tab+n/2,n-n/2);
```

```
            merge(tab,n);
        }
}
```

Schrijf nu zelf de methode merge(tab,n) die twee gesorteerde tabellen samenvoegt. De eerste gesorteerde tabel is de linkerhelft van tab en de tweede tabel is de rechterhelft van tab.

Schrijf een hoofdprogramma waarin je twee tabellen aanmaakt met elk 100.000.000 gehele getallen. Om het algoritme veel werk te geven, zorg je ervoor dat beide tabellen omgekeerd geordend zijn. Genereer dus geen random-waarden! Maak nu gebruik van twee threads die elk één tabel zullen sorteren.

Net zoals bij de vorige opdracht maak je ook een versie waarbij je geen gebruikmaakt van Pthreads om eventuele snelheidswinst aan te tonen.

Thread synchronisatie

Bovenstaande opdrachten hebben één ding gemeenschappelijk, nl. iedere thread loopt onafhankelijk van de andere threads. Sommige toepassingen vergen echter dat threads onderling moeten kunnen samenwerken en eventueel gebruik moeten maken van dezelfde gedeelde bronnen. Soms kan het ook gebeuren dat een bepaald stuk code maar door één thread tegelijkertijd mag uitgevoerd worden, een zogenaamde kristische sectie. Om dit te kunnen doen bestaan er **Mutexen** en **Semaforen**. Het verschil tussen beide is dat mutexen een gedeelde bron beschermen en dus fungeren als vergrendelingsmechanisme daar waar semaforen dienst doen als signaleringsmechanisme. Doordat semaforen gebruikmaken van berichten/signalen, is het gebruik ervan dus veel algemener dan dat van mutexen.

Gebruik van mutexen

Om een mutex te kunnen gebruiken, moet deze in eerste instantie geïnitialiseerd worden. Onderstaande code toont hoe je een statische gealloceerde mutex moet initialiseren.

```
pthread_mutex_t mtx= PTHREAD_MUTEX_INITIALIZER;
```

Om een mutex te vergrendelen en een mutex te ontgrendelen worden onderstaande functies gebruikt:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Wanneer een thread de lock-functie uitvoert en de mutex is niet vergrendeld, wordt de mutex vergrendeld. Wanneer de mutex reeds vergrendeld is, wordt er gewacht tot wanneer dat de thread die hem vergrendeld heeft hem terug ontgrendeld. Het spreekt voor zich dat

deadlocks eenvoudig kunnen worden bekomen door de bronnen niet correct vrij te geven. Denk hierbij aan een situatie waar er twee mutexen gebruikt worden om twee gedeelde bronnen te beschermen. Het is niet ondenkbaar dat thread1 mutex1 vergrendeld heeft, en wacht op het ontgrendelen van mutex2 daar waar thread2 net op het omgekeerde aan het wachten is. Dit ziet er dan zo uit:

Thread A	Thread B
pthread_mutex_lock(&mtx1); pthread_mutex_lock(&mtx2);	pthread_mutex_lock(&mtx2); pthread_mutex_lock(&mtx1);

Gebruik van POSIX semaforen

Er worden twee soorten semaforen onderscheiden, nl. unnamed en named semaforen. Net zoals named pipes worden named semaforen gebruikt voor IPC tussen niet gerelateerde processen. Voor communicatie tussen threads en gerelateerde processen volstaan unnamed semaforen. Wanneer semaforen worden gebruikt voor IPC tussen gerelateerde processen, moeten ze worden bijgehouden in een gedeelde virtuele geheugenpagina die kan worden aangevraagd via de systeemaanroep **mmap**. Tussen threads volstaat het wanneer unnamed semaforen globaal worden gedeclareerd of op de heap worden geplaatst.

Een semafoor initialiseren gebeurt via:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

pshared is 0 wanneer de semafoor gedeeld wordt onder threads en is verschillend van nul wanneer het gaat om een semafoor die gedeeld wordt onder gerelateerde processen. Dit argument geeft onrechtstreeks ook aan waar de semafoor *sem resideert. Bij threads volstaat het om een semafoor globaal te declareren of via malloc aan te maken op de heap. Voor processen daarentegen moet er gebruikgemaakt worden van een pagina in de virtuele adresruimte(zie later).

De werking van een semafoor is vrij eenvoudig. Bij initialisatie wordt aan de semafoor een waarde toegekend die niet negatief mag worden. Wanneer een proces of een thread een semafoor tegenkomt die een negatieve waarde heeft, wordt de uitvoering stopgezet tot wanneer de waarde terug positief wordt. De onderstaande operaties zullen de waarde van de semafoor **sem* respectievelijk decrementeren en incrementeren.

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

Wanneer een semafoor een waarde heeft die gelijk is aan 0, zal de functie sem_wait() de uitvoering van de actieve thread of het actieve proces blokkeren tot wanneer de waarde

strikt positief wordt. In dat geval wordt de waarde van de semafoor door de functie sem_wait() gedrecrementeerd.

Bij een aanroep van de sem_post() functie wordt de waarde van de semafoor geincrementeerd en wordt het proces of de thread die momenteel aan het wachten is geactiveerd. Indien er meerdere processen of threads wachtende zijn, bepaalt de scheduler welk proces of welke thread de uitvoering mag verderzetten.

Een unnamed semafoor wordt vernietigd door de onderstaande functie:

```
int sem_destroy(sem_t *sem);
```

Een semafoor moet worden vernietigd vooraleer dat het geheugen waarin hij resideert wordt vrijgegeven. Dit gebeurt bv. wanneer een functie waarin de semafoor werd aangemaakt terugkeert naar de oproepende instantie. Bij het gebruik van een gedeelde virtuele geheugenpagina tussen processen, moet de semafoor vernietigd worden wanneer er geen enkel proces nog gebruik van maakt en vooraleer het geheugen via de systeemaanroep **munmap** terug wordt vrijgegeven.

Opdrachten

1. Schrijf een programma dat gebruikmaakt van 8 threads voor de verkoop van tickets. Het totaal aantal tickets wordt bijgehouden in een globale variabele waar iedere thread automatisch toegang tot heeft. Iedere thread voert dezelfde functie uit, nl. een functie die door een oneindige lus loopt waar telkens één ticket verkocht wordt. Van zodra alle tickets de deur uit zijn, rapporteert iedere thread het totale aantal tickets dat hij verkocht heeft. Om threadwissels te verzekeren plaats je in de oneindige lus de opdracht **sleep(rand()%3)** die een delay van 0, 1 of 2 seconden veroorzaakt. Maak gebruik van een mutex om de gedeelde bron, i.e. de globale variabele te beschermen. Bekijk ook wat er gebeurt wanneer de gedeelde bron niet wordt vergrendeld.
2. Herneem opdracht 1 maar maak nu gebruik van semaforen om de gedeelde bron te beschermen. Aangezien er maar twee toestanden mogelijk zijn spreekt men hier van een binaire semafoor.
3. Schrijf een C-programma dat het probleem van de etende filosofen oplost gebruikmakend van threads. Een uitgebreide beschrijving van het probleem vind je terug in de cursus besturingssystemen I: architectuur van dhr. Moreau.

Gebruik van shared memory en memory mapped I/O

IPC kan gebeuren d.m.v. pipes, semaforen of shared memory. Om in de virtuele adresruimte van een proces een geheugenmapping aan te maken wordt gebruikgemaakt van de systeemaanroep mmap.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

```
int munmap(void *addr, size_t length);
```

De parameter addr is het adres in de virtuele adresruimte waar je de mapping wil plaatsen. Het is aangewezen om als waarde van addr NULL te kiezen waardoor de kernel een geschikt adres zal zoeken voor de gevraagde mapping.

De length-parameter is de grootte van de mapping in bytes. Doorgaans zal de kernel dit afronden naar een veelvoud van de paginagrootte (cfr. de return-waarde van sysconf(_SC_PAGESIZE)).

Het prot-argument geeft aan welke toegangsrechten er aan de geheugenmapping gekoppeld zullen worden. Mogelijke waarden zijn PROT_NONE, PROT_READ, PROT_WRITE en PROT_EXEC. Bij PROT_NONE is er geen toegang tot de geheugenruimte, bij PROT_READ mag de inhoud gelezen worden, bij PROT_WRITE mag de inhoud gewijzigd worden en tot slot mag bij PROT_EXEC de inhoud uitgevoerd worden.

De flags-parameter is doorgaans één van onderstaande:

MAP_PRIVATE

> Maakt een private geheugenmapping aan waarbij eventuele aanpassingen onzichtbaar zullen zijn voor andere processen die van dezelfde mapping gebruikmaken. Bij een bestandsmapping zullen aanpassingen niet naar het onderliggend bestand geschreven worden.

MAP_SHARED

> Maakt een gedeelde mapping aan. Aanpassingen zullen zichtbaar zijn bij andere processen die van dezelfde mapping gebruikmaken. Bij een bestandsmapping zullen de aanpassingen naar het onderliggende bestand geschreven worden. Deze zullen echter niet onmiddellijk naar het bestandssysteen worden geschreven maar op een moment waarop de kernel dit het meest opportuun acht.

De fd-parameter is de file descriptor die aan de geheugenmapping gekoppeld moet worden.

Zoals de naam doet vermoeden is de offset-parameter het startpunt van de mapping in het bestand. Wanneer voor de offset de waarde 0 gekozen wordt en als length de grootte van het bestand, wordt een volledige mapping van het bestand bekomen.

Mmap geeft bij succes het startadres van de geheugenmapping als return-waarde.

Munmap doet het tegenovergestelde van mmap en verwijdert de geheugenmapping.

Bestandmappings

Om een bestandsmapping te bekomen moet met de systeemaanroep **open** een file descriptor gevraagd worden en moet deze vervolgens gebruikt worden bij de systeemaanroep mmap. Na het uitvoeren van mmap mag de file descriptor met **close** gesloten worden.

Het spreekt voor zich dat de toegangsparameters bij het openen van de descriptor en bij de mapping consistent moeten zijn. Het zou onlogisch zijn mocht je bij de mapping PROT_WRITE vermelden terwijl de fd-parameter bekomen werd met O_RDONLY.

Memory-mapped I/O

Aangezien een bestand in het geheugen kan worden geladen, kunnen een aantal I/O-operaties vervangen worden door geheugenoperaties. Dit heeft volgende voordelen:

1. De systeemaanroepen read en write veroorzaken achter de schermen twee dataoverdrachten. De eerste tussen het bestand en de kernel buffer cache, een tweede tussen de kernel buffer cache en een buffer in user-space. De tweede overdracht wordt door gebruik te maken van geheugenmapping overbodig. Bij input is de data onmiddellijk na de mapping voorhanden en bij output worden de wijzigingen door de kernel, weliswaar niet ogenblikkelijk, naar het bestand weggeschreven.
2. Zoals in punt 1 geschetst kan de er wat tijd bespaard worden door het aantal data-overdrachten te halveren maar het is ook zo dat er bij een read/write-operatie data zal bijgehouden worden in twee buffers, nl. één in de kernel en één in user-space. Bij een geheugenmapping is er enkel nog een buffer nodig in de kernel waardoor er dus ook op het geheugenverbruik kan bespaard worden.

Performantiewinst is het meest waarschijnlijk bij random I/O-operaties in een groot bestand. Bij sequentiële toegang zal er weinig tot geen performantiewinst bekomen worden omdat bij beide technieken het volledige bestand één keer in het geheugen zal worden geladen. Hierdoor zal het uitsparen van een dataoverdracht en een beetje geheugen niet opwegen tegen de tijd die verloren gaat aan disk I/O.

Let wel, het gebruik van virtueel geheugen heeft ook grote nadelen. Voor kleine I/O-operaties is de kost van memory-mapped I/O aanzienlijk (mapping, unmapping, paginafouten, aanpassen van de MMU TLB, …)

Opdrachten

1. Schrijf een eigen versie van cat waarbij er nu gebruikgemaakt wordt van memory-mapped I/O.
2. Interprocescommunicatie m.b.v. semaforen kwam tot nu toe nog niet aan bod. Schrijf een C-programma dat 200 kindprocessen aanmaakt die elk een uniek getal genereren. Het pid van het proces dat het grootste getal gegeneerd heeft, alsook het getal zelf, wordt door alle kindprocessen naar het scherm geschreven. Om een nieuw

shared memory object te maken kan je de systeemaanroep **shm_open** gebruiken. Deze aanroep geeft een getal terug dat je bij mmap kan gebruiken als file descriptor. De initiële grootte van het shared memory object is 0. Om het shared memory object een grootte te geven kan je de systeemaanroep **ftruncate** gebruiken.

Deel VII: Programmeren in Bash

Patterns, expansions en het opzoeken van hulp

Zoals reeds eerder vermeld bevat de shell enkele honderden commando's met elk nog tal van command line options. Het is dan ook logisch dat wanneer je meer informatie zoekt over een welbepaalde opdracht, je dit doet a.d.h.v. het commando man waarbij je de naam van de opdracht meegeeft als parameter. Als extra parameter kan je ook nog het sectienummer meegeven of de optie -a wanneer je alle man-pagina's waar de naam die je als parameter hebt opgegeven wil overlopen.

Gebruik voor de onderstaande opdrachten de ingebouwde documentatiemogelijkheden om op de vragen te antwoorden.

Opdrachten

1. Met welk van de commando's cp, dd, ln, mktemp, touch en cat kan je vlug een aantal (lege) bestanden aanmaken waarvan de namen als parameters van het commando worden opgegeven?
2. Wat verschijnt er op het scherm indien je de opdracht head /etc/passwd uitvoert? Zoek nu de verwante opdracht voor het tonen van de laatste lijnen van een bestand. Hoe kan je steeds de laatste lijnen van een bestand op het scherm laten verschijnen wanneer er een ander proces achteraan het bestand lijnen toevoegt?
3. Waarvoor dient de optie -rf bij de opdracht rm? Maak met een editor een bestand aan met als naam "-rf". Hoe kan je het bestand "-rf" verwijderen?
4. Welke opties moet je toevoegen aan het commando wc om enkel de grootte van een bestand te tonen zonder extra informatie?
5. In Bash zijn er ook Bash-builtin opdrachten zoals cd, set, pwd, exec, printf en : waarvoor er geen aparte man-pagina's beschikbaar zijn. Een overzicht kan je bekomen door **man builtin** of door de man-pagina van Bash op te vragen. Zoek informatie op over het gebruik van de opdrachten cd, set, pwd, exec, printf en :.
6. Wat doet het commando sync?
7. Hoe kan je met het commando dd een afbeelding maken van een USB-pen? Welke device-file heb je hiervoor nodig? Bekijk de uitvoer van de opdracht "fdisk -l".
8. Hoe kan je met dd een kopie maken van de eerste 512 bytes van de vaste schijf? Bekijk met het commando **strings** welke tekststrings in die 512 bytes verscholen zitten.
9. Gebruik het commando find om een lijst van bestanden te krijgen die de afgelopen 24u nog werden aangepast.
10. Met het commando wodim kan je van op de opdrachtlijn een CD/DVD-branden. Met het commando genisoimage kan je een ISO-bestand aanmaken. Hoe kan je van de inhoud van de /root directory een ISO-bestand maken?
11. Bij vraag 10 zal je merken dat de namen van de bestanden/directories werden gewijzigd. Je kunt dit vermijden door de image in Joliet-formaat weg te schrijven.

Naast reguliere expressies kent Unix ook patterns om een verzameling strings te beschrijven. De mogelijkheden van standaard patterns zijn veel beperkter dan bv. reguliere expressies en worden gebruikt zowel in opdrachten als in shellscripts. In recente Bash-versies werden deze patterns uitgebreid. Extended pattern matching kan worden aangezet met de opdracht "shopt –s extglob".

Bij het uitvoeren van een commando met een pattern wordt eerst een lijst met bestandsnamen gegenereerd die aan het opgegeven patroon voldoen. Dit wordt meestal "Pathname Expansion" genoemd. Meer uitleg kan je vinden in de man-pagina van Bash onder de rubriek "Pathname Expansion".

Maak voor onderstaande opdrachten de volgende lege bestanden aan: a, b, c, d, e, ab.c, a.b, b.a, b.c, c.d en d.e.

12. Voer de volgende opdrachten uit:
    - dir a*.*
    - dir a*
    - dir *a
    - dir a*

    Bemerk een belangrijk verschil met reguliere expressies. Bovendien wijkt de uitvoer af van de uitvoer van dezelfde commando's in Windows.
13. Voorspel en controleer de uitvoer van de opdrachten:
    - printf "%s\n"[abcd]
    - printf "%s\n" [!abcd]
    - printf "%s\n" [^abcd]
    - printf "%s\n" [a-d]
    - printf "%s\n" [abcd]*[abcd]
14. Voer de onderstaande commando's uit.
    - printf "%s\n" [a-e]
    - printf "%s\n" [a/-e]
    - printf "%s\n" [a\-e]
    - printf "%s\n" [!\!]*

    Wat is de bedoeling van het \-teken in deze opdrachten? Wat gebeurt er indien je het verkeerde /-teken gebruikt?
15. Hoe kan je een lijst met bestandsnamen bekomen die precies uit één enkel karakter bestaan?
16. Vraag een lijst met bestandsnamen die uit precies twee karakters bestaan. Vergelijk de uitvoer met die van de vorige opgave.
17. Voer volgende opdrachten uit:
    - ls *
    - dir *
    - printf "%s\n" *

- ls “*”
- printf “%s\n” “*”

Wat is het subtiele verschil in uitvoer tussen de eerste drie opdrachten? Verklaar dit verschil door nog een aantal bestanden aan te maken en beide commando’s opnieuw uit te voeren.

Verklaar het belangrijke verschil tussen de eerste en de vierde opdracht. Verklaar ook de verschillende uitvoer van de twee laatste opdrachten.

18. Zorg dat er geen bestanden in de werkdirectory staan waarvan de naam met abc begint. Verklaar dan het verschil in uitvoer tussen volgende opdrachten:
    - ls abc*
    - printf “%s\n” abc*
19. Voer de opdracht rm –f ?? uit. Verklaar daarna het verschil in uitvoer tussen volgende opdrachten:
    - printf “%s\n” ???
    - printf “%s\n” ??e
    - printf “%s\n” ??
20. Enigszins verwant aan pathname expansion is de mogelijkheid tot brace expansion. Dergelijke uitdrukkingen bestaan uit een eventuele prefix, gevolgd door een rij strings tussen accolades (gescheiden door komma’s) en afgesloten met een eventuele suffix. Elke string binnen de accolades wordt gecombineerd met een prefix en een suffix. In tegenstelling tot pathname expansions genereert een brace expansion een rij strings, zonder dat gecontroleerd wordt of deze strings met bestaande bestanden overeenkomen. Zowel in de prefix, de elementenstrings of de suffix van een brace expansion kunnen (recursief) andere brace expansions opgenomen worden. Met welke brace expansion kan je alle getallen tussen 0 en 29 (grenzen inbegrepen) als argumenten aan een opdracht (bv. printf) meegeven? Hoe kan je deze brace expansion beknopt schrijven, door bv. een range te gebruiken?
21. Je kan brace expansions nesten. Genereer de hexadecimale voorstellingen van alle even gehele getallen kleiner dan 256.
22. Wanneer het eerste karakter van een string een tilde(~) is, wordt er aan tilde expansion gedaan. Dit wil zeggen dat alle karakters tussen de tilde en de eerste slash beschouwd worden als een gebruikersnaam. Wat verschijnt er op het scherm wanneer je de volgende opdrachten uitvoert:
    - echo ~root/
    - echo ~mail/
    - echo ~{mail,root}/

    Wat wordt door de shell het eerst vervangen, de tilde of de accolades?