Obligatorisk opgave # 1 i BOSC

Ooperativsystemer og C

Author:
Tom Mørk Christensen
Jonas Elbækgaard Jørgensen

ITU-mail:
tmch@itu.dk
jelb@itu.dk

26. september 2014

Indhold

1 Indledning **2**

2 Implementationen, overordnet set **2**

3 De specifikke løsninger **3**
3.1 Systemkald - $K1$ . . . 3
3.2 Hostname - $K2$ . . . 3
3.3 Simple eksekveringer - $K3$ . . . 4
3.4 Ukendte funktioner - $K4$ . . . 4
3.5 Baggrundsprocesser - $K5$ . . . 4
3.6 Standard I/O - $K6$ . . . 5
3.7 Kommandosekvenser - $K7$ . . . 5
3.8 Quit on exit - $K8$ . . . 5
3.9 Interrupt - $K9$ . . . 5

4 Fejl og mangler **6**

5 Test **6**

1 Indledning

I denne rapport beskrives vor løsning af Obligatorisk Opgave 1 i kurset 'Operativsystemer og C' på IT-Universitet, efteråret 2014. Målet med opgaven er at konstruere en simpel kommandoliniefortolker (eller shell), i stil med f.eks. Unix Bash, og i den sammenhæng demonstrere vores forståelse af systemkald og proceshåndtering, samt lavniveau C programmering. Kommandoliniefortolkeren vil blive omtalt under navnet bosh.

I opgaven er stillet en række formelle krav til Bosh's funktionalitet. For at kunne referere til disse krav i rapporten, er de opsummeret her:

K1: Bosh skal kunne virke uafhængigt. Du må ikke bruge andre eksisterende shells, f.eks. er det ikke tilladt at anvende et systemkald *system*() til at starte bash.

K2: Kommando-prompt'en skal vise navnet på den host den kører på.

K3 En bruger skal kunne indtaste almindelige enkeltstående kommandoer, så som *ls*, *cat* og *wc*.

K4: Hvis en kommando ikke findes i operativ systemet skal der udskrives en *"Command not found"* meddelelse.

K5: Kommandoer skal kunne eksekvere som baggrundsprocesser (ved brug af &) sådan at mange programmer kan køres på samme tid.

K6: Der skal være indbygget funktionalitet som gør det muligt at lave redirection af *stdin* og *stdout* til filer.

K7: Det skal være muligt at anvende pipes til at sammenkæde kommandoer.

K8: Funktionen exit skal være indbygget til at afslutte shell'en.

K9: Tryk på Ctrl-C skal afslutte de(t) program, der kører i bosh shell'en, men ikke shell'en selv.

Vi vil i denne rapport først beskrive den overordnede implementation, og derefter gå ind i en nærmere beskrivelse af hvordan de enkelte krav er tilfredsstillet. Til sidst vil vi beskrive de dele af programmet som ikke møder kravene helt som vi ønskede det, samt en beskrivelse af hvilke tests vi har foretaget for at bekræfte at vores implementation fungerer efter hensigten.

2 Implementationen, overordnet set

Til opgaven er udleveret 5 programfiler samt en `MAKE` fil, så vi hurtigt kunne komme i gang med den egentlige opgave. Disse inkluderede en parser der håndterer kommandolinieinput, 2 structs som parseren benytter til at håndtere den fortolkede inputdata, 2 printfiler som lader programmet printe det parsede input til terminalen, samt hovedprogramfilen `bosh.c` som indeholder programmets main function, main loop mv. I vores implementation har vi kun rettet i og tilføjet til denne fil.

De fleste ændringer er foretaget i og omkring den eksisterende metode `executeshellcmd`, med et par undtagelser i form af nye hjælpefunktioner samt små ændringer i `main` metode til at akkommodere disse. Både hjælpemetoder og vil blive beskrevet i detaljer når vi nedenfor går i detaljer med de enkelte krav.

Som nævnt benytter programmet et main loop, som tager input fra brugeren og parser dette vha. `parser.c`. Denne gemmer en struct i hukommelsen, som indeholder det parsede input i et format som kan bruges under eksekvering. Metoden `initializeExecution` kaldes med en reference til den føromtalte struct (det parsede kommandoinput), denne deler den igangværende proces (ved hjælp af `fork`), og i barneprocessen redirectes standardinput til structens indhold, hvorefter metode executeshellcmd kaldes.

I executeshellcmd checkes om structen indeholder flere på hinanden følgende kommandoer. Hvis dette er tilfældet dele der på ny, i den nye barne(barns)proces sætte standard in/out op så output fra den 'yngste' proces kan føres som input til den 'ældste'. Den yngste proces checker igen om der er flere kommandoer mens de ældre afventer de yngre (ved hjælp af `wait`), og denne rekursion fortsættes indtil alle kommandoer har fået en proces. Når dette mål nås, eksekveres den yngste kommando først, sender sit output videre til sin forælder og således indtil rekursionen er trævlet op.

Herefter returnerer program eksekveringen til main loopet. Såfrem den givne kommando var exit eller quit vil programmet terminere, hvis ikke vil det kunne tage imod en ny kommandolinie fra brugeren.

3 De specifikke løsninger

3.1 Systemkald - $K1$

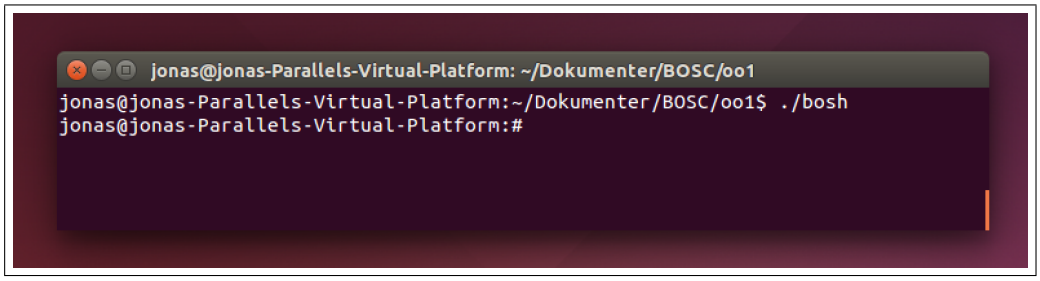
Som beskrevet ovenfor udføres en eller flere procesopdelinger ved hjælp af `fork` systemkaldet, indtil alle kommandoer har fået deres egen proces at køre i. Navnet på metode som skal køres læses ud af _shellcmd structen, og reflekterer direkte det input, som brugeren har afgivet. Programmet benytter nu `execvp` systemkaldet til at afvikle de pågældende kommandoer en for en.

Metoden execvp adskiller sig fra andre 'slægtninge' i `exec` familien ved at søge i operativsystemets `PATH` environment variabel efter et match, når det forsøger at eksekvere en kommando uden `'/'` præfix. Såfrem det pågældende program ikke findes der, forsøger execvp at køre programmet fra dets nuværende position, og derefter listen returneret af `confstr(_CS_PATH)`.

3.2 Hostname - $K2$

For at understøtte at bosh skulle vise værtsnavnet på den maskine hvorpå det eksekveres har vi udvidet den oprindelige `gethostname` funktion.[1] Funktionen, som findes på linje 32 - 49 i filen `bosh.c`, bruger kommandoen `'popen'` til at eksekvere kommandoen *hostname*, som er placeret i `'/bin'` mappen, og returnere en I/O stream, som indeholder resultatet af kommandoen. "hostname" returnere maskinens værtsnavn, og ved at skrive dette til en en variabel ved hjælp af fgets, kan vi vise værtsnavnet når bosh er klar til at modtage en ny kommando.
I tillæg til at kunne vise værtsnavnet har vi valgt også at vise brugernavnet på den bruger der er logget ind på maskinen. Dette er implementeret i metoden `"getusernamecmd"`, linje 41 - 48, som virker på samme måde som funktion til værtsnavnet bortset fra at den kommando der eksekveres er *id*, som er placeret i `'/usr/bin'` mappen, med parameteret $-un$. I figuren herunder ses hvordan bruger- og værtsnavn vises i bosh når det venter på input fra brugeren.

[1] Omdøbt til `"gethostnamecmd"`.



Figur 1: bosh skriver brugernavn og værtsnavn til terminalen.

3.3 Simple eksekveringer - $K3$

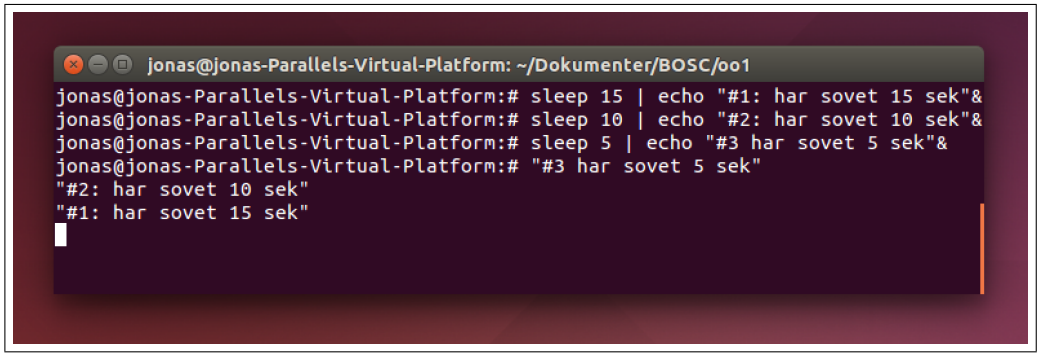
Vi bruger `execvp` funktionen til at køre kommandoer indtastet af brugern. `Execvp` leder efter den angivne kommando i `$PATH` miljøet på maskinen. Ved at bruge denne kommando er vi i stand til at eksekvere simple kommandoer.

3.4 Ukendte funktioner - $K4$

Hvis `execvp` ikke finder den kommando den får som parameter i `$PATH` miljøet bliver kontrollen returneret til den process hvorfra `execvp` blev kaldt. Denne adfærd har vi udnyttet til fortæller brugeren hvis en given kommando ikke blev fundet. Ved at lade fejlmeddelelsen være det eneste kode, som er placeret efter kaldet til `execvp` vil dette kun bliver eksekveret hvis kommandoen ikke blev fundet. Efter at fejlmeddelelsen er skrevet til terminalen benytter vi `exit()` funktionen til at forhindre at barneprocessen forlader `executionshellcmd` funktionen og begynder at lytte efter nye brugerinput.

3.5 Baggrundsprocesser - $K5$

For at understøtte kravet om at kommandoer skulle kunne køres som baggrundsprocesser har vi brugt en simpel if sætning til at styre hvorvidt forælder processen skal vente eller ej. For at afgøre om forældre processen skal vente tjekker vi om *background* feltet på shellcmd objektet er sat til sand(1). Hvis dette er tilfældet venter forældreprocessen på at barne processen returnere ellers fortsætter forælderprocessen og gør klar til at modtage en ny kommando. Dette gør det muligt at eksekvere flere kommandoer samtidig. I figur 2, side 4, ses et eksempel hvor flere kommandoer eksekveres som baggrundsprocesser samtidig.



Figur 2: Flere kommandoer eksekveres parallelt.

3.6 Standard I/O - $K6$

Krav nr. 6 specificere at det skal være muligt at omdirigere kommandoens standard input og output således at det er muligt at læse input fra, samt at skriver output til filer. Til at opnå dette har vi gjort brug af `dup2` og `freopen`. `Dub2` gør det muligt at udskifte en standard I/O stream med en anden, mens `freopen` gør det muligt at genåbne stdin/stdout som en anden fil.
Når vores første barneproces bliver oprette tjekker vi først om brugeren har specificeret en input fil. Er dette tilfældet genåbner vi den specificerede fil som stdin. Her efter tjekker vi om der er angivet en output fil. Er dette også tilfældet åbner også denne fil og ændre processen stdout til filen, ved hjælp af `dub2`. Når dette er gjort fortsætter eksekveringen af bruger inputtet.

3.7 Kommandosekvenser - $K7$

Som det fremgår af krav 7 skal det være muligt at sammenkæde kommandoer ved hjælp af *pipe* operatoren "|". Betydningen af denne operator er at resultatet af kommandoen på operatorens venstre side skal bruges input til kommandoen på operatorens højre side. Et eksempel på en sådan kommandosekvens kan ses i figur 2, side 4.

Vi har implementeret sammenkædningsfunktionaliteten ved hjælp af *pipe* strukturen fra *unistd* bibliotektet. Som nævnt i den overordnede beskrivelse af implementationen bliver kommandoerne parset i omvendt rækkefølge således at den kommando der ligger forrest i shellcmd objektet er den kommando som brugeren indtastede sidst. For at kæde kommandoerne sammen deler vi processen rekursivt så længe at der er flere kommandoer tilbage i shellcmd, på den måde ender vi med lige så mange barneprocesser som der er kommandoer i inputet. Hver gang en proces deles opretter vi en 'pipe', ændre standard inputet på forældre process til at være læse-enden af pipen, og ændre barneprocessens output til at være skrive-enden af pipen. Her efter venter forældreprocessen på at barneprocessen terminere mens barneprocessen arbejder videre med de resterende kommandoer. På den måde sikre vi os at kommandoerne eksekveres i den rigtige rækkefølge.

3.8 Quit on exit - $K8$

For dette krav har vi valgt en simpel og direkte fremgangsmåde. Når metoden `initialExecution` køres, checkes om indholdet af den sidst indtastede kommando matcher strengene $'exit'$ og $'quit'$. Gør den det, returnerer metoden med value 0, hvilket afbryder `while` løkken og programmet standser ved at returnere `EXIT_SUCCESS`.
Se afsnit 4 'Fejl og mangler', side 6, for yderligere oplysninger.

3.9 Interrupt - $K9$

For at undgå at bosh terminere når brugeren benytter tastekombinationen "CTRL + C", men at bosh i stedet terminere alle sine barneprocesser har vi tilføjet to signal handlers. Den første handler, parent_int, linje 27 - 29, bliver brugt at forældre processen når den modtager et signal af typen `SIGINT`. Handleren skriver en besked til skærmen om at børne processer bliver termineret, hvor efter den returnere kontrollen til brugeren. Den anden handler, child_int, linje 23 - 25, bliver benyttet af alle børeneprocesser. Handleren lukker den aktuelle process og på den måde bliver alle processer med undtagelse af forældreprocessen lukket. For at sikre at forældreprocessen ikke anvender den forkerte handler bliver child_int først sat efter at forældreprocessen bliver delt.

4 Fejl og mangler

På grund af den måde exit-funktionaliteten er implementeret ($K8$), er der begrænsninger på hvilke situationer exit kommandoen fungerer. Vi foretager et check i `initialExecution` metoden, hvilket betyder at vi reelt kun undersøger den første kommando i _shellcmd structen, dvs. den sidst indtastede kommando i en given kommandolinie. Samtidigt betyder det, at programmet terminerer uden at eksekvere nogen anden kommando.

Det kan argumenteres at dette ikke er den optimale løsning, i det der kunne være situationer hvor en ønsker at sætte en serie af lange operationer til at køre, for til sidst at lukke bosh ned. Vores løsning understøtter på nuværende tidspunkt ikke denne funktionalitet, selvom det formelle krav om at konstruere en exit-funktion til at lukke programmet er opfyldt.

5 Test

Vi har forsøgt at teste `bosh` via 2 forskellige fremgangsmåder: simpel afprøvning med standard metodekald og avanceret afprøvning med et specialkonstrueret testprogram.

Vi har forsøgt denne opdeling af forskellige årsager. Til dels har vi ikke erfaring med sytematisk test af software i C, og ingen kendskab til frameworks som kunne tillade opsætning af automatiserede tests. Til dels er en stor del af den definerede funktionalitet ganske banal, og kan således testes til fulde med kommandolinieinput.

Her følger en beskrivelse af de tests og afprøvninger vi har foretaget til hver af de funktionelle krav.

K1: Som det vil fremgå af tests af de følgende systemkrav, er der ikke nogen tvivl om at `bosh` er i stand til at foretage systemkald.

K2: Ved start af `bosh` på forskellige maskiner og med forskellige brugere logget ind, skifter kommandolinie prompten titel alt efter navn på den givne host computer og bruger. Dette vil ligeledes fremgå for den enkelte bruger når programmet køres.

K3: Det er muligt at teste den simple funktion `ls` ved først at køre den udenfor `bosh` (for at etablere det korrekte svar), og derefter starte `bosh` og køre `ls` igen. Dette producerer det samme resultat. Hvis man herefter kører kommandoen `mkdir test`, vil et nyt kald til `ls` demonstrere at der rigtigt nok er oprettet et dir ved navn test.

K4: Forsøger man derimod at køre den ikke-eksisterende kommando `qwerty` vil man få en fejlmeddelelse og afvikling af kommandolinieinputet vil terminere.

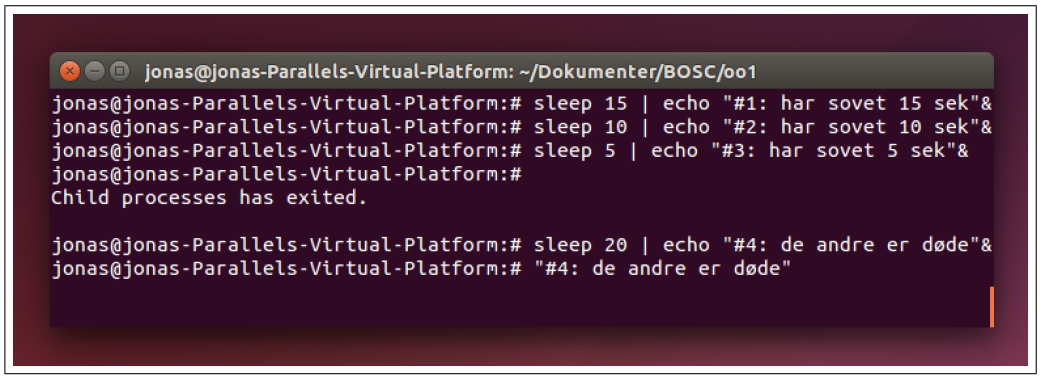
K5: Såfrem man benytter & tegnet i forbindelse med et metodekald, vil processen eksekvere i baggrunden, mens kontrol af bosh returneres til brugeren med det samme. Dette betyder, at brugeren kan indtaste nye kommandoer som igen kan få lov at eksekvere i baggrunden. Det er muligt at teste dette ved f.eks. at bruge funktionen `sleep` som demonstreret i figur 2 side 4. Her ses, at processen kører i baggrunden, men giver kontrol tilbage til brugeren så denne kan starte flere `sleep` processer, og at resultatet af disse bliver printet når processerne terminerer.

K6: Den umiddelbare test af dette krav er at køre det angivne eksempel. Gør man dette vil man se, at der rigtigt nok genereres en fil 'antalkontoer' og at denne indeholder et enkelt tal, nemlig antallet af kontoer på hostmaskinen. Dette viser, at det er muligt at køre kommandoen `wc -l` med input `/etc/passwd`, og redirecte resultatet fra ned i en nye fil med navn `antalkontoer`.

K7: Ligeledes tester vi dette krav ved at eksekvere det angivne eksempel. Vi ser, at kommandoen `ls` først genererer et output som essentielt er en liste af direktoriets indhold, og dette resultat pipes over i `wc -w` kommandoen, som udskriver det faktiske antal ord i listen.

K8: Ret let at teste: tast `exit` i kommandolinien og `bosh` terminerer. Hvis man taster flere kommandoer efter `exit` kommandoen sker der dog intet (se Fejl og mangler). Ligeledes ser vi, at `bosh` terminerer med det samme, hvis man piper flere kommandoer efter hinanden og afslutter med `exit`.

K9: Det er trivielt at demonstrere at `CTRL+C` ikke slår selve `bosh` ihjel. Det er derimod sværere at gennemskue om bagvedliggende barneprocesser kan afbrydes. For at afprøve dette har vi sat et et antal sleep processer igang (som tidligere), men før disse terminerer og printer en besked til terminalen interupter vi dem med `CTRL+C`. For at demonstrere at de faktisk ER termineret kan vi starte en længere sleep proces og afvente at den terminerer. Herved har vi bevist, at de tidligere processer er termineret. Se figur 3.



Figur 3: Kørende kommandoer afbrydes af `CTRL + C`.

Vi forsøgte som sagt at skrive et program, som skulle kunne bruges til at teste bosh, men kunne ikke få det til at fungere som ønsket. Vi har for en god ordens skyld inkluderet det i afleveringen, og kunne for øvelsens skyld godt tænke os at vores feedback inkluderer det.