UMEÅ UNIVERSITET
Institutionen för Datavetenskap

7 oktober 2022

Systemnära programmering

mexec

Name Vincent Johansson
Username dv14vjn

Handledare
Abdulsalam Aldahir, Oscar Kamf, William Sandström

Innehåll

1 Systembeskrivning

Programmet är en egen implementation av pipe funktionen i bash, tex om input i en bash terminal skulle varit ”cat -n mexec.c | grep -B4 -A2 return | less” så ska detta programmet kunna utföra samma uppgift. Input av kommandon till programmet sker antingen via en fil där varje kommando är på en ny rad, det vill säga ”cat -n mexec.c” på en rad och de andra på en egen rad, eller genom direkt inläsning ifrån terminalen. Även vid inläsning direkt i terminalen läser programmet varje rad som ett kommando.

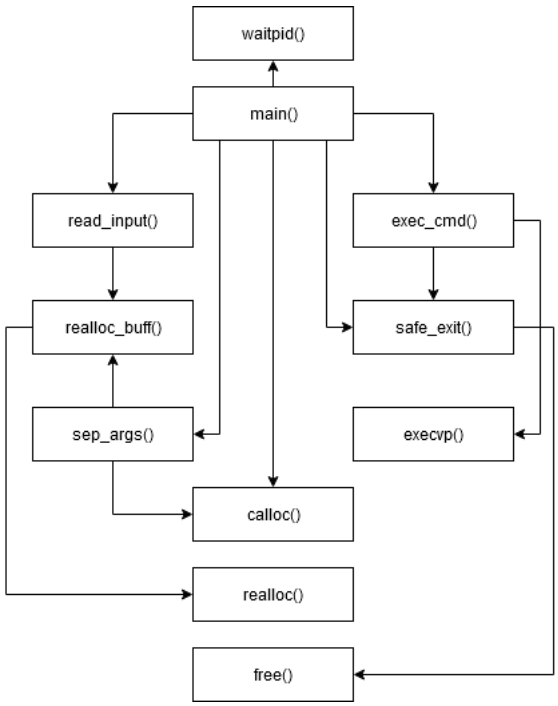
1.1 Funktionalitet

All input blir omdirigerad till stdin, oavsett om input ges som en fil eller via terminal. Innan programmet läser av input allokeras minne via safe_calloc() i main(). I figur 1 visas hur de olika funktionerna interagerar med varandra, först anropar main read_input() som läser av input från stdin och lagrar varje kommando för sig i en string array. Funktionen read_input() räknar även hur många kommandon som lästs, för att kunna skapa rätt antal pipes till barnprocesserna senare.

När input har blivit läst, skapas lika många pipes som angivna kommandon och en for-loop körs det antalet gånger. I for-loopen anropar main() sep_args() som delar upp en den första strängen med kommandon som första barnprocessen ska exekvera. De uppdelade kommandona sparas i en array av string arrayer, detta behövs göras för att kunna skicka kommando och argument till execvp() funktionen. Om otillräckligt med minne blivit allokerats till att spara input eller de separerade kommandona så anropas realloc_buff() som allokerar om en större plats i minnet. De funktioner som kan behöva realloc_buff() funktionen går att se i figur 1.

När första raden med kommando och argument delats anropas exec_cmd(), i denna funktionen skapas första pipen som ska skicka data från första barnprocessen till nästa. När pipen har initialiserats görs ett fork() anrop som skapar den första barnprocessen. I barnprocessen omdirigeras stdout till skrivänden av pipen och läsänden stängs. I föräldern stängs skrivänden, och ifall det inte är den första barnprocessen som precis skapats stängs läsänden i föregående pipe. Förälraprocessen lämnar läsänden till första pipen orörd tills nästa barnprocess har skapats för att undvika att nästa barnprocess inte ska kunna omdirigera stdin till den pipen.

Efter att första barnprocessen skapats frigörs minnet för de separerade kommandona och argumenten i main() och for-loopen börjar om på nytt. Detta fortsätter tills for-loopen körts så många gånger som antal kommandon som lästs från input. När for-loopen körts klart väntar föräldern in alla barnprocesser och om någon av barnprocesserna avlutats felaktigt så avslutas föräldraprocessen med felkoden från barnprocessen. Om inget problem upptäcks anropas safe_exit() som frigör allt använt minne innan programmet avlslutas. I varje barnprocess anropas safe_exit() ifall något problem uppstår när execvp anropas. För en tydligare förklaring till hur programmet använder fork() och barnprocesser finns figurer och text i algoritmbeskrivning för pipes sektionen.



Figur 1: Anropsdiagram
Funktionerna till vänster är funktioner som förbereder indata för användning till de funktioner som är på högersidan av anropsdiagrammet.

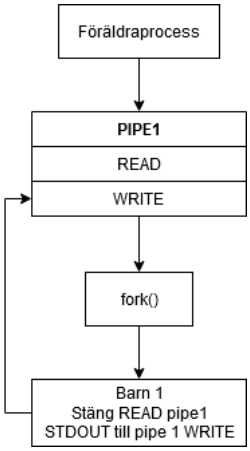
2 Algoritmbeskrivning för pipes

2.1 Algoritm

Om det inte är sista kommandot som ska exekveras
    Skapa pipe
Skapa barnprocess via fork()
Om barnprocess
    Om input bara innehöll ett kommando
        Exekvera kommandot
    Annars om det är första barnprocessen
        Omdirigera stdout till pipe
    Annars om det inte är första barnprocessen
        Omdirigera stdout till senast skapade pipe
        Omdirigera stdin till förra pipe som skapades
    Annars om det är sista processen
        Omdirigera stin till förra pipe som skapades
    Stäng alla oanvända pipes
    Exekvera kommando
Om föräldraprocess
    Stäng skrivänden på senast skapade pipe
    Stäng läsänden på förra pipe som skapades

2.2 Detaljerad beskrivning

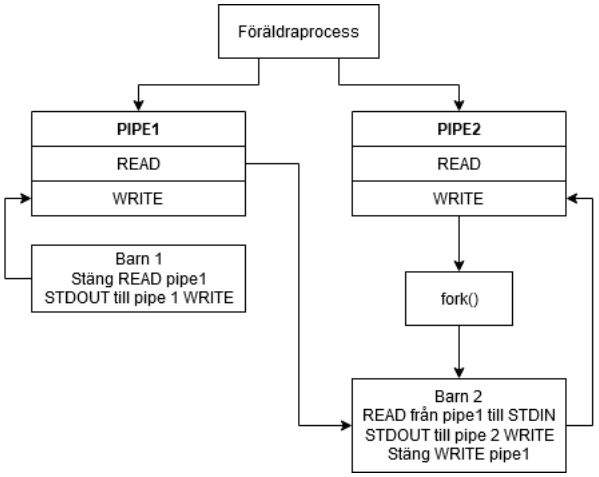
Beskrivningen kommer att utgå ifrån algoritmen i sektion 2.1. Föräldraprocessen kommer alltid att stänga både READ och WRITE i pipes efter att barnprocesserna omdirigerat till pipes, detta visas inte i figurerna så därför specificeras det nu.



Figur 2: fork 1

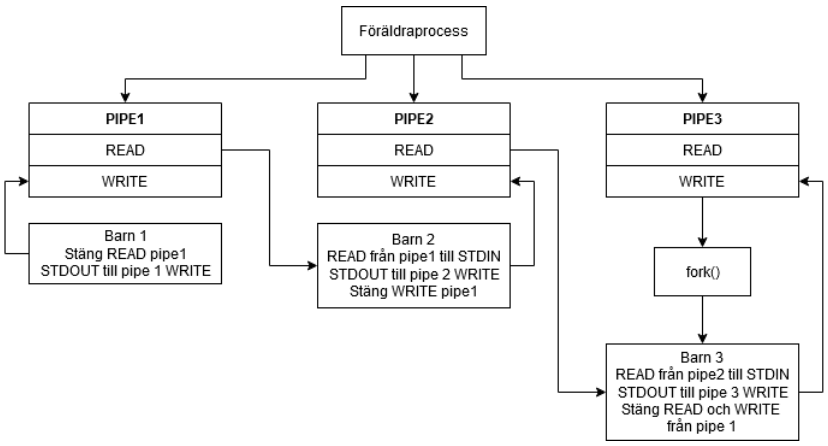
När första barnprocessen ska skapas initierar föräldraprocessen en ny pipe och därefter anropas fork() funktionen som skapar barnprocessen. I barnprocessen omdirigeras STDOUT till WRITE änden i pipe och därefter stängs READ änden eftersom den inte används i första barnprocessen. När omdirigeringar gjorts exekveras första kommandot. Se figur 2 för att se hur barnprocessen är kopplad till pipen.

I nästa steg skapar föräldraprocessen en ny pipe innan fork() anropas igen för att skapa andra barnprocessen. I den nya barnprocessen omdirigeras STDOUT till den nya pipens WRITE ände och därefter omdirigeras READ från första pipen till STDIN. WRITE änden i första pipen stängs då andra barnet inte kommer använda den och sen exekveras andra kommandot. Indata till kommandot kommer att läsas in ifrån READ på första pipe. Se figur 3 för en visuell representation hur barnen är kopplade till pipes.



Figur 3: fork 2

Samma process som tidigare utförs i föräldraprocessen görs igen, ny pipe skapas innan fork() anropas för att skapa en ny barnprocess. I den nya och tredje barnprocessen omdirigeras READ från andra pipe till STDIN, om fler kommandon ska utföras hade inte STDOUT blivit omdirigerat men i figur 4 antas att ett till kommando ska exekveras och därför omdirigeras STDOUT till WRITE i trejde pipe. Både READ och WRITE i första pipe stängs innan kommandot exekveras och indata till kommandot kommer att läsas in ifrån READ på andra pipe. Se figur 4 för en visuell representation av hur barnen är kopplade till pipes.



Figur 4: fork 3

3 Diskussion och reflektion

När jag påbörjade uppgiften försökte jag först göra en rekursivimplementation, något som jag senare blev avråd från. Skrev då om funktionen från grunden, och istället räknade hur många kommandon som programmet tagit emot och skapade pipes och barnprocesser utifrån det antalet. Hade svårt i början att förstå hur jag skulle få pipes att fungera mellan ett godtyckligt antal barnprocesser, men när jag väl listat ut att varje pipe måste stängas korrekt i alla barnprocesserna var det inga större problem. Fortfarande lite nyfiken på hur jag hade kunnat göra en rekursiv lösning, där alla barnprocesser fortfarande skapas i föräldraprocessen så det är något jag kommer att testa på min fritid. Jag skulle också vilja lära mig hur jag skulle kunna strukturera en funktion som hanterar all allokering och frigörning av minne, något jag känner vi inte fått speciellt mycket information om hittills under studierna.