Obligatorisk uppgift 1: mexec

Namn: **Abdiaziz Ibrahim Adam**
CS-användare: **dv23aam**
Kurs: **Systemnära programmering**

December 3, 2024

1 Introduktion

Rapporten beskriver skapandet av programmet `mexec`, som är en exekverare av en pipeline skrivet i C. Programmet implementerar grundläggande funktioner inom processhantering och systemanrop för att skapa och exekvera flera kommandon som körs parallellt där utdata från ett program blir indata till nästa. Målet med denna uppgift är att förstå och implementera en pipeline, som är en teknik som ofta används i Unix-baserade system för att länka flera kommandon tillsammans.

2 Beskrivning av programmet

Programmet `mexec` tar antingen en fil som indata eller läser direkt från terminalen (som standard input). Filen eller terminalens indata består av en sekvens av kommandon där varje kommando körs parallellt. Varje kommando exekveras i en separat process med hjälp av `fork()`, och deras in- och utdata kopplas ihop med hjälp av `pipe()`. Exekveringen av kommandona utförs genom `execvp()`, som ersätter den nuvarande barnprocessens kod med det angivna kommandot.

2.1 Programmets uppbyggnad

Programmet `mexec` har följande huvudkomponenter:

1. **Läsning av indata:** Programmet inleder med att läsa in kommandon, antingen från en fil eller standard input. Varje rad läses in som ett kommando och lagras i en dynamiskt allokerad array.

2. **Skapande av Pipes:** För varje par av intilliggande kommandon skapas en pipe, vilket möjliggör kommunikation mellan processerna. Varje pipe fungerar som en länk som överför utdata från ett kommando som indata till nästa.

3. **Forkning och Exekvering:** När alla kommandon har lästs in, skapar programmet en ny process för varje kommando med hjälp av `fork()`. I varje barnprocess omdirigeras indata och utdata genom pipes, vilket skapar en kedja av kommandon. Varje barnprocess körs med `execvp()`, som ersätter den nuvarande processen med det angivna kommandot.

4. **Avslutning och Minneshantering:** Föräldraprocessen stänger alla oanvända pipes och väntar på att alla barnprocesser ska avslutas. Programmet frigör allt allokerat minne för kommandon och pipes innan det avslutas.

Nedan visas ett anropsdiagram som förklarar hur funktionerna i programmet är strukturerade och hur de anropar varandra:

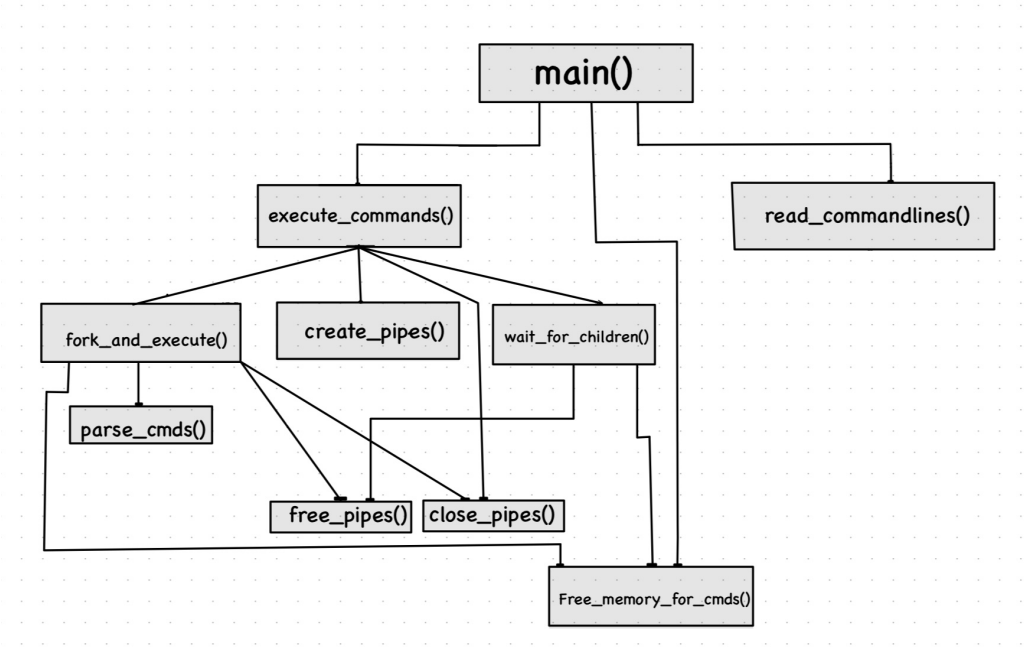

Figure 1: Anropsdiagram för programmet `mexec`.

Diagrammet 1 visar hur programmet `mexec` inleder sin körning i funktionen `main()`. Först anropas `read_commandlines()` för att hämta kommandon från användaren. Kontroll övergår sedan till `execute_commands()`, där `create_pipes()` skapar pipor för kommunikation mellan kommandona.

Inom `execute_commands()` anropas `fork_and_execute()` för att skapa nya processer. Denna funktion använder `parse_cmds()` för att tolka kommandona. När kommandona har körts stänger programmet oanvända pipor via `close_pipes()` och frigör resurser med `free_pipes()` och `free_memory_for_cmds()`.

Funktionen `wait_for_children()` anropas för att vänta på att alla barnprocesser avslutas. Felhantering sker genom funktionen `handle_error()` som inte syns i diagrammet. Denna funktion anropas i flera funktioner.

Sammanfattningsvis visar denna struktur hur `mexec` hanterar inläsning, exekvering och resursförvaltning av kommandon i en pipeline.

3 Algoritm beskrivning

Programmet `mexec` använder en serie systemanrop för att skapa och hantera processer och pipor. Algoritmen följer följande steg:

1. För varje par av intilliggande kommandon:
   (a) Skapa ett rör (pipe) för kommunikation.

2. För varje kommando:
   (a) Forka en ny process.
   (b) I barnprocessen:
      i. Omdirigera in- och utdata med hjälp av rör (pipes).
      ii. Exekvera kommandot med `execvp()`.
   (c) I föräldraprocessen:
      i. Stäng oanvända rör (pipes).
      ii. Vänta på att alla barnprocesser ska avslutas.

Algoritmen som programmet använder för att starta processer och bygga upp pipor kan delas in i flera steg. Nedan visas en grafisk representation av hur processer och pipor skapas och kommunicerar steg för steg:

Först och främst, för varje par av intilliggande kommandon skapas ett rör (pipe) för kommunikation. Detta är avgörande för att möjliggöra dataflöde mellan kommandona. I Figur 2 visas hur programmet initierar pipor för varje par av intilliggande kommandon, vilket gör det möjligt för utdata från ett kommando att bli indata för nästa.

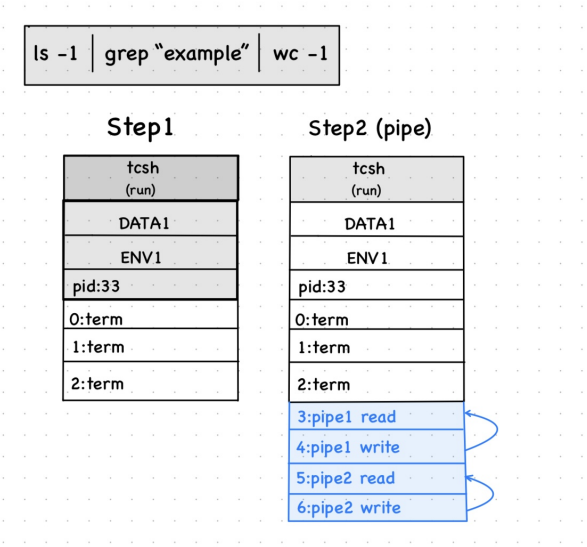

Figure 2: Skapande av pipor för kommunikation mellan kommandon.

Därefter, för varje kommando, forkas en ny process. I figur 3 illustreras hur programmet använder systemanropet `fork()` för att skapa en ny process för varje kommando. Varje process tilldelas en unik process-ID (PID), vilket möjliggör parallell körning av flera kommandon. Denna del av algoritmen är avgörande för att säkerställa att varje kommando kan exekveras parallellt.

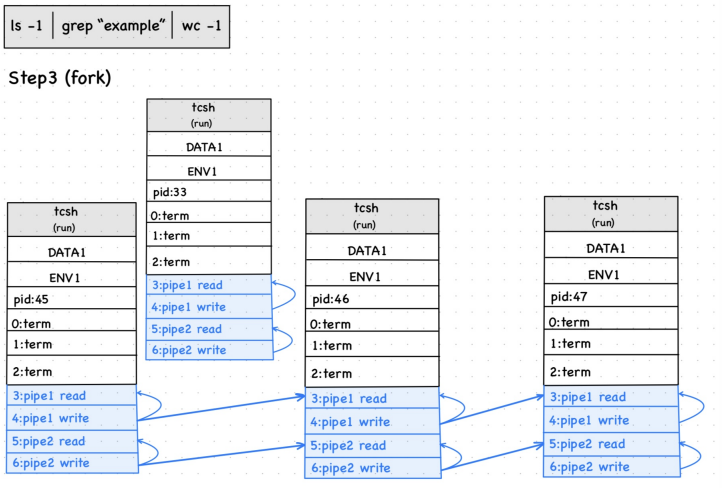

Figure 3: Forking av processer för varje kommando.

Inuti varje barnprocess kopplas in- och utdata med hjälp av rör (pipes). Detta åstadkoms genom funktionen `dup2()`, som säkerställer att varje barnprocess får rätt data från sin föregångare. I figur 4 kan vi se hur denna omdirigering sker. Detta steg är kritiskt för att flödet av data ska fungera som avsett i pipelinen.

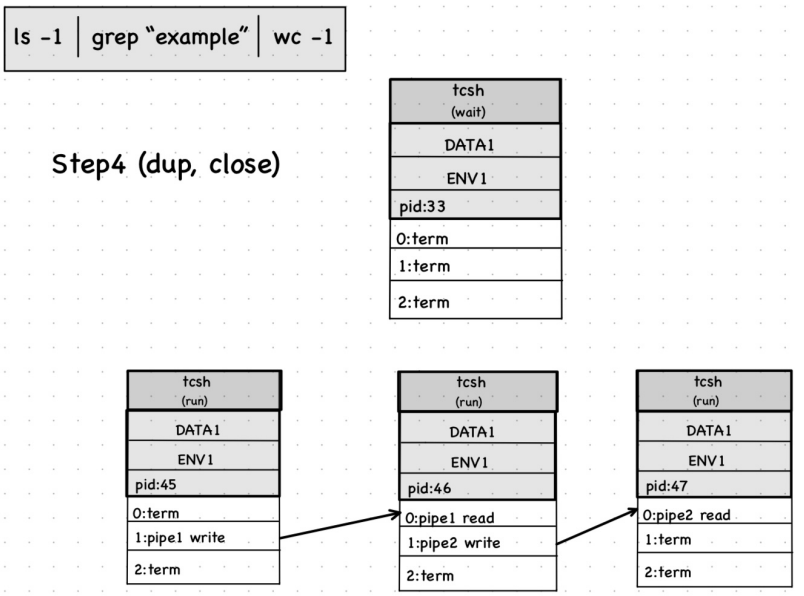

Figure 4: Omdirigering av in- och utdata med hjälp av rör (pipes).

Slutligen väntar föräldraprocessen på att alla barnprocesser ska avslutas. I figur 5 visas denna sista del av algoritmen, där föräldern ser till att programmet inte avslutas förrän alla kommandon har exekverats. Detta steg är viktigt för att garantera att alla resultat från de tidigare kommandona behandlas korrekt innan programmet avslutas.

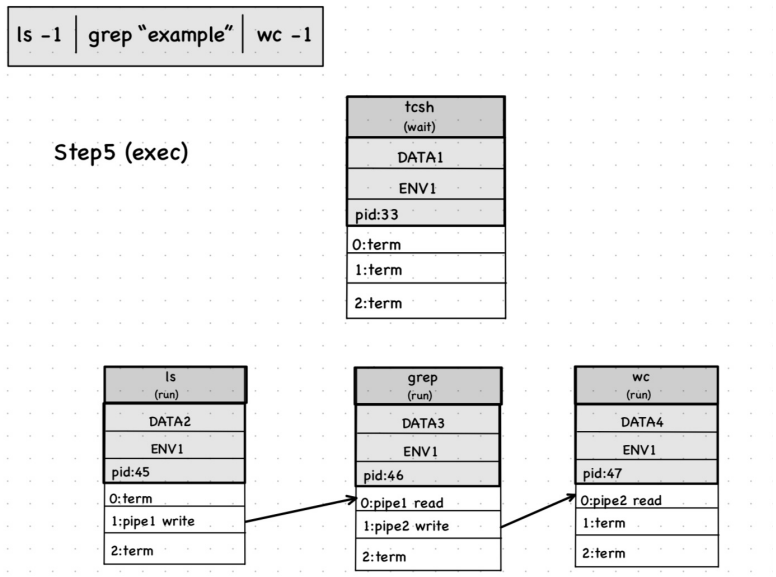

Figure 5: Vänta på att alla barnprocesser ska avslutas.

Den förväntade outputen kan se ut som följande:

```
2
```

Detta betyder att det finns två rader som matchar sökningen ”example” i de listade filerna. Dessa figurer illustrerar hur programmet effektivt hanterar och exekverar en kedja av kommandon med hjälp av processer och pipor. Genom att följa denna algoritm kan programmet säkerställa korrekt och effektiv kommunikation mellan de olika kommandona i en pipeline.

4 Diskussion och Reflektion

Under utvecklingen av programmet `mexec` stötte jag på flera utmaningar, främst relaterade till kommunikation mellan processerna via pipes. Jag hade inledningsvis svårt att koppla ihop in- och utdata effektivt, men genom att studera användningen av `dup2()` och säkerställa att alla rör stängdes korrekt lyckades jag lösa detta problem.

En annan utmaning var minneshantering. Korrekt allokering och frigöring av minne var avgörande, särskilt när programmet försökte köra ogiltiga kommandon som `nosuchcmd`. Jag behövde noggrant frigöra minne för varje resurs för att undvika läckor, vilket jag löste genom kodgranskning och tester.

Slutligen vill jag säga att denna uppgift har varit mycket lärorik och har fördjupat min förståelse för hur processer fungerar i Unix-liknande operativsystem samt hur pipes effektivt länkar samman kommandon. Jag ser fram emot att lära mig mer om systemprogrammering och att bygga vidare på den kunskap jag har fått genom detta projekt.