SAKARYA ÜNİVERSİTESİ
BİLGİSAYAR VE BİLİŞİM BİLİMLERİ FAKÜLTESİ
İŞLETİM SİSTEMLERİ
ÖDEV

GRUP ÜYELERİ:
Sude Dönmez B221210581 – 1B
Zeynep Gizem Fırat B221210074 – 1B
Esra İclal Akyol B221210032 – 1B
Sara Mohamed B221210566 – 1B
Zeynep Kebapcı B221210091 – 1A

Dersi Veren: Ahmet ZENGİN/Abdullah SEVİN

ÖDEVİN HAZIRLANIŞI

Ödevde amacımız, Linux işletim sistemi üzerinde C dilini kullanarak bir kabuk (Shell) uygulaması geliştirmekti. Bizi yönlendirmesi için elimizde bir ödev yönergesi bulunuyordu, bu yönergeye göre ödevde yapmamız gereken işlemler şunlardı:

1. **Prompt : Kabuğun her komutun ardından “>” karakterini ekrana basması.**
2. **Quit (Built-in Komut) : Kabuğun ‘Quit’ komutu ile sonlanması.**
3. **Tekli Komutlar : Kabuğun tek bir komut okuduğunda komutu çalıştırması, komut tamamlanıncaya kadar da girişi engellemesi ve ardından isteme geri dönmesi.**
4. **Giriş Yönlendirme : Programın verilen komutu çalıştırmadan önce alt prosesin standart girdisini verilen giriş dosyasına yönlendirmesi.**
5. **Çıkış Yönlendirme : Programın verilen alt prosesin standart çıktısını çıkış dosyasına yönlendirmesi.**
6. **Arkaplan Çalışma : Bir komut ‘&’ ile bittiğinde o komutun arkaplanda çalıştırılabilmesi, bu süreçte kabuğun işlemler için engellenmesi ve arkaplan işlemi sona erdiğinde kabuğun çıkış durumunu proses kimliği ile birlikte kullanıcıya bildirmesi.**
7. **Boru (Pipe) : Birden fazla komutun birbirine borular aracılığıyla bağlanıp çalıştırılabilmesi.**

Bu işlemleri nasıl yaptığımızı sıra sıra açıklayalım:

1. PROMPT

Kullanıcı kabuğa bir prompt girdiğinde her seferinde ‘>’ karakterini yazdırmak için print_prompt adını verdiğimiz bir fonksiyon oluşturduk. Bu fonksiyon basitçe ekrana ‘>’ yazdıran bir fonkisyon. Gerekli prompt işlevinin yerine getirilmesi için de main fonksiyonunda bir döngü içerisinde bu fonksiyonu kullanarak yapıyı tamamladık. Kullanıcı girdisinin hemen görüntülenmesi için de bizden istenen fflush(stdout) komutundan yararlandık.

2. QUIT

Kullanıcının kabuğa ‘Quit’ yazıp yazmadığını tespit etmek amacıyla, komutları sırasıyla yürütmekle görevli olan execute_sequential_commands fonksiyonumuzda bir şart bloğu tanımladık. Bu şart bloğu eğer kullanıcıtarafından girilen prompt ‘Quit’ ise ilk olarak arkaplanda bir işlemin çalışıp çalışmadığına bakıyor ve çalışan bir arkaplan

komutu varsa önce bu arkaplan komutunun bitmesini bekliyor. Sonrasında ise exit() komutuyla programı yani kabuğu sonlandırıyor.

3. TEKLİ KOMUTLAR

Tekli komutların icrası için sequential_execution adını verdiğimiz fonksiyonumuzdan yararlandık. Bu fonksiyonu girilen komutların nasıl bir komut olduğunu tespit etmek amacıyla çeşitli işlemler yapacak (; ibaresi var mı, bunun kontrolünü yapmak, boru hattı içerip içermediğine bakmak gibi) şekilde tanımladık. Bu fonksiyonun özelliklerinden bir diğeri de tekli komutları doğru şekilde yürütmek. Girilen tekli komutu ayrıştırarak parse_command adlı bir başka fonksiyona gönderiyor ve bu sırada giriş/çıkış dosyaları ve arkaplan işlem bayraklarının durumunu da göz önünde bulunduruyor.

4. GİRİŞ YÖNLENDİRME

Bu işlemi yapabilmek için üç farklı fonksiyonumuzdan yararlandık. command_parsing fonksiyonu komutu ayrıştırarak gerekli giriş dosyasının adını gerekli fonksiyonlara iletme görevini yaparken execute_sequential_commands fonksiyonu ise giriş dosyasını açıp bu giriş dosyasının standart girişe yönlendirilmesini sağlar. Böylece standart girişe yönlendirilen dosyayla beraber ilgili komut yürütülerek işlem sağlanmış olur.

5. ÇIKIŞ YÖNLENDİRME

Bu işlem, giriş yönlendirme ile oldukça benzer bir işlem olup yine aynı mantıkla çalışır. command_parsing fonksiyonu komutu ayrıştırarak gerekli çıkış dosyasını ilgili fonksiyona iletir. execute_sequential_commands fonksiyonu ise çıkış dosyasını açıp bu çıkış dosyasının standart çıkışa yönlendirilmesini sağlar. Böylece standart çıkışa yönlendirilen dosyayla beraber ilgili komut yürütülerek işlem sağlanmış olur.

6. ARKAPLAN ÇALIŞMA

Bu işlemi yerine getiren background_processes adında bir fonksiyon yazdık. Bu fonksiyon içinde arkaplan proseslerinin bir listesini ve bu liste içindeki her bir proses için kimlik bilgisini tutmakla görevli. Arkaplan fonksiyonları yürütülürken bu komutlarının tamamlanıp tamamlanmadığını kontrol etmek de yine bu fonksiyonun sorumluluğundadır. Bir komut tamamlandığında o komutu arkaplan listesinden çıkararak komutların sırayla icra edilmesini sağlar. Arkaplanda çalışan komutlar bittiğinde ise biten komutun kimlik bilgisini, tamamlanma durumuyla beraber kullanıcıya terminal ekranında gösterir.

7. BORU (PIPE)

Bu işlemi yerine getirebilmek için pipeline_execution isminde bir fonksiyon tanımladık. Bu fonksiyon, komutları ‘|’ sembolüne göre ayırarak her bir komutu bir listede tutar. Bu komutlar için gerekli tanımlayıcıları tanımlayarak boru hattını oluşturur ve her bir komut için bir fork çağrısı yapar. Fork çağrılarını yaparken komutları işleyiş şekli, önceki borudan giriş alma ve sonraki boruya çıkış gönderme şeklindedir. Borunun okuma ve yazma uçlarını gerekli durumlara göre kapatıp açarak işlemlerin tamamlanmasını sağlar ve en sonunda da bütün işlemler tamamlanana kadar bekler. Böylece basit bir kabuk (shell) uygulaması yapmak için oluşturmamız gereken bütün fonksiyonları oluşturmuş olduk. Bu yapı sayesinde tekli komutları icra edebiliyor, arkaplanda komut çalıştırabiliyor, boru yapısını verimli bir şekilde kullanabiliyor ve giriş/çıkış yönlendirme işlemlerini kullanabiliyoruz.

GİTHUB LİNKİ

https://github.com/gizemfirat/osgrup29