T.C.
SAKARYA ÜNİVERSİTESİ
BİLGİSAYAR VE BİLİŞİM FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

İŞLETİM SİSTEMLERİ PROJE ÖDEVİ

2024 GÜZ

HAZIRLAYANLAR

Osman Oğuzhan Kırık b231210372 1c

Mehmet Ali Tüfekçi b221210383 1b

Sıla Çanga G231210372 2c

Aleyna Çakır G231210370 2a

Halil İbrahim Sarıtemur B221210352 1a

Projenin Tanımı ve Özeti

Bu proje, Linux ortamında temel bir kabuk programı (shell) geliştirmeyi amaçlamaktadır. Kullanıcı, terminal üzerinden komutlar girerek işletim sistemi işlemlerini kontrol edebilir. Program, komutları analiz eder, alt süreçleri oluşturur ve sonuçları kullanıcıya döndürür.

Temel Özellikler:

- **Komut İstemi:** Kullanıcı, komutları girdikçe program > işaretiyle yeni komut girmeyi bekler.
- **Built-in Komutlar:** Örneğin, quit komutu ile program sonlanır.
- **Tekli Komut İcrası:** Komutlar ve argümanlar girilerek işlem yapılır (örneğin, ls -l).
- **Giriş/Çıkış Yönlendirme:** Komutların giriş ve çıkışı dosyalar üzerinden yönlendirilir (örneğin, cat < file.txt).
- **Arka Plan Çalıştırma:** Komutlar & işaretiyle arka planda çalıştırılabilir (örneğin, sleep 5 &).
- **Boru (Pipe) Kullanımı:** Komutlar arasında boru (|) kullanılarak veri işleme yapılır (örneğin, find /etc | grep ssh).

Çıktılar:

- **Komut Sonuçları:** Her komutun çıktısı terminalde gösterilir.
- **Hata Mesajları:** Dosya bulunamadığında hata mesajı verilir.
- **Arka Plan Bilgisi:** Arka planda çalışan komutların durumu, işlem kimliği ve bitiş kodu ile birlikte bildirilir.

Bu projede, Linux sistem çağrıları (fork, exec, dup2, waitpid) kullanılarak komutların doğru bir şekilde işlenmesi sağlanmıştır. Program, kullanıcıya verimli ve etkili bir etkileşim deneyimi sunar.

Tasarım ve Yöntemler

Kabuk programı, kullanıcı komutlarını işleyip sistem çağrılarını kullanarak işlem yönetimi sağlar. Programın temel işleyişi, komutları almak, ayrıştırmak, uygun sistem çağrılarını gerçekleştirmek ve çıktıyı sağlamak üzerine kuruludur.

1. **Komut Ayrıştırma ve Giriş İşleme:** Kullanıcıdan alınan komutlar, boşluklara göre ayrıştırılır ve komutlar diziler halinde depolanır. Dosya yönlendirme veya arka plan çalıştırma gibi özel işlemler de doğru şekilde işlenir.
2. **fork() Kullanımı:** Her komut için yeni bir çocuk süreç oluşturulur. fork() ile ana süreçten bağımsız bir çocuk süreç türetilir, böylece işlemler paralel şekilde yürütülür.
3. **execvp() Kullanımı:** execvp() fonksiyonu, komutları çalıştırmak için kullanılır. Komut ve argümanlar ile sistemdeki uygun program çalıştırılır. Çocuk süreç, execvp() ile işlem değişir ve komut çalıştırılmaya başlanır.
4. **Çıktı Yönlendirme (dup2()):** Dosyaya yönlendirme için dup2() fonksiyonu kullanılır. Çıktı, ekran yerine belirtilen dosyaya yazılır.
5. **Arka Plan Çalıştırma:** Komut sonuna & eklenmişse, komut arka planda çalıştırılır. fork() ile komut hemen başlar, ancak ana süreç yeni komut girmeyi bekler.
6. **waitpid() Kullanımı:** Çocuk süreçlerin tamamlanmasını beklemek için waitpid() fonksiyonu kullanılır. Bu fonksiyon, çocuk süreçlerin bitiş kodlarıyla durumlarını takip eder.
7. **pipe() Kullanımı:** Komutlar arasındaki veri iletimi için pipe() kullanılır. Komutlar birbirine bağlandığında, pipe() ile veriler bir komuttan diğerine iletilir.
8. **Hata Yönetimi:** Komut çalıştırılamazsa veya dosya bulunamazsa, anlamlı hata mesajları gösterilir. Ayrıca, sistem çağrılarına ilişkin hata kontrolü yapılır ve başarısız işlemler uygun şekilde ele alınır.

Kullanılan Sistem Çağrıları

Kabuk programınızda, işlem yönetimi ve I/O yönlendirmeleri için çeşitli sistem çağrıları kullanılmıştır. Bu çağrılar, programın doğru şekilde çalışabilmesi ve kullanıcı komutlarını doğru şekilde işlemesi için temel araçlardır.

fork()
fork() sistem çağrısı, ana süreçten bir çocuk süreç oluşturur. Bu çağrı, kabuk programının temel işleyişini sağlar çünkü her komut çalıştırılmadan önce yeni bir çocuk süreç oluşturulması gerekir. fork() çağrısı, iki süreçten her birine farklı bir döngüde devam etme imkanı sunar:
Ana süreç (parent): fork() çağrısının dönüş değeri 0'dan farklıdır ve ana süreç, çocuk sürecin tamamlanmasını bekler.
Çocuk süreç (child): fork() çağrısının dönüş değeri 0'dır ve çocuk süreç, belirli komutları çalıştırmak için execvp() gibi sistem çağrılarını kullanır.
Bu, programın birden fazla işlemi paralel bir şekilde yönetmesini sağlar.

execvp()
execvp() fonksiyonu, belirli bir komutu çalıştırmak için kullanılır. Bu fonksiyon, verilen komut ve argümanlarla sistemdeki uygun programı başlatır. execvp() çağrısı, bir çocuk süreçte çağrıldığında mevcut işlemi yeni bir programla değiştirir. Bu, komut satırında verilen komutun yürütülmesini sağlar.
Komutun adı ve argümanları, execvp() fonksiyonuna geçilir.
Eğer komut geçerli bir dosya yoluna sahipse, program bu komutu çalıştırır ve mevcut süreç (çocuk süreç) çalışmaya başlar.
Eğer komut geçerli değilse, execvp() hata döndürür.
Bu çağrı, bir komutun çalıştırılması için gerekli tüm işlemleri gerçekleştiren ana araçtır**.**

dup2()
dup2() fonksiyonu, dosya tanımlayıcılarını yeniden yönlendirmek için kullanılır. Bu, özellikle dosya yönlendirmesi yaparken önemlidir. Bir komutun çıktısının (stdout) bir dosyaya yönlendirilmesi veya bir komutun girdisinin (stdin) bir dosyadan alınması gerektiğinde, dup2() kullanılır.
Çıktıyı bir dosyaya yönlendirmek için, önce dosya tanımlayıcısı (file descriptor) açılır, ardından dup2() ile bu tanımlayıcı stdout'a kopyalanır.
Komutun çıktısı normalde ekrana yazılmak yerine, belirtilen dosyaya yazılır.
Bu fonksiyon, komutların I/O işlemleri üzerinde tam kontrol sağlamanızı mümkün kılar ve programın çıktılarını doğru bir şekilde yönlendirebilirsiniz**.**

waitpid()
waitpid() fonksiyonu, bir çocuk sürecin tamamlanmasını beklemek için kullanılır. Ana süreç, çocuk süreci tamamlandıktan sonra çalışmaya devam eder. Bu çağrı, ana sürecin çocuk sürecin bitiş durumunu ve kodunu almasına imkan tanır. Çocuk süreç tamamlandıktan sonra ana süreç, durumu alır ve buna göre işlem yapar.
Ana süreç, çocuk sürecin sonlanmasını bekler.
Çocuk süreç tamamlandığında, waitpid() fonksiyonu çocuk sürecin bitiş kodunu geri döndürür.
Ana süreç, çocuk süreçle ilgili bilgileri alarak işlemi sonlandırabilir.

pipe()
pipe() fonksiyonu, iki komut arasındaki veri iletimini sağlamak için kullanılır. Pipe, verilerin bir komuttan diğerine doğru bir şekilde aktarılmasını sağlar. Eğer bir komutun çıktısı diğer bir komutun girdisi olarak kullanılacaksa, pipe() ile bir boru (pipe) oluşturulur. Bu boru, veri akışını yönetir.
Bir yazma uç noktası ve bir okuma uç noktası oluşturulur.
İlk komutun çıktısı, pipe'a yazılırken, ikinci komut bu çıktıyı okur.
Bu sistem, birbirine bağlı komutlar arasında veri iletimini sağlar ve komutlar arasında işlevsel bağı oluşturur.

Çalıştırılabilir Komutlar ve Kullanıcı Etkileşimi

Kabuk programınız, kullanıcı komutlarını alıp işlem yapan bir sistem olarak çalışır. Kullanıcı etkileşimi terminal üzerinden sağlanır ve kullanıcıya geri bildirim verilir.

Kullanıcı Komutlarının Alınması
Program, terminalden gelen komutları fgets() fonksiyonu ile alır. Komut, terminalde görünen prompt mesajıyla (örneğin, $) kullanıcıya gösterilir ve kullanıcı Enter tuşuna bastığında komut alınır.

Komutların Ayrıştırılması
Alınan komut, strtok() fonksiyonu ile parçalanır. İlk parça komut adı, geri kalan parçalar ise komutun argümanlarıdır. Bu aşama, komutun doğru bir şekilde işlenebilmesi için gereklidir.

Komutların Yürütülmesi
Komut, fork() ile çocuk süreçte çalıştırılır. Eğer dosya yönlendirme veya pipe varsa, dup2() ve pipe() ile işlenir. Çocuk süreçte execvp() ile komut çalıştırılır. Çıktılar ya da hata mesajları terminalde kullanıcıya iletilir.

Çıkış Durumu ve Geri Bildirim

- **Tekli Komutlar**: Komut ve argümanlar alındığında fork() ile bir çocuk süreç oluşturulur, ardından execvp() ile komut çalıştırılır.
- **Dosya Yönlendirme**: dup2() ile dosya tanımlayıcıları yönlendirilir.
- **Arka Plan Çalışma**: Komut sonunda & eklenirse, komut arka planda çalışır ve yeni komut için prompt gösterilir.
- **Pipe Kullanımı**: Komutlar arasında pipe() ile veri akışı sağlanır; bir komutun çıktısı diğerinin girdisi olur.

Kullanıcıya Geri Bildirim
Komut başarılıysa çıktı terminalde görüntülenir, hata varsa kullanıcıya açıklayıcı bir mesaj gösterilir. Kullanıcı doğru komut formatını girmesi için yönlendirilir.

Test ve Sonuçlar

Proje, işlevselliği doğrulamak ve hataları tespit etmek amacıyla detaylı bir şekilde test edilmiştir. Test edilen komutlar ve sonuçlar şu şekildedir:

- **ls**: Dizin içeriği başarıyla listelendi.
- **pwd**: Çalışma dizini doğru şekilde gösterildi.
- **cat <dosya_adı>**: Dosya içeriği başarıyla görüntülendi.
- **echo**: Kullanıcıya metin mesajları doğru şekilde iletildi.

Yanlış Komutlar: Geçersiz komutlar girildiğinde, uygun "Komut bulunamadı" hata mesajı verildi.

Karşılaşılan Hatalar ve Çözümler:

- **Yanlış Komutlar**: Başlangıçta yanlış komutlar boş çıktı veriyordu. execvp() fonksiyonu hata döndürdüğünde kullanıcıya açıklayıcı hata mesajları gösterildi.
- **Dosya Bulunamama**: cat komutuyla dosya bulunamadığında hata mesajı gösterilmedi. Dosya varlığı kontrol edilip, "Dosya bulunamadı" mesajı eklendi.

Karşılaşılan Sorunlar ve Çözümler

1. **Sistem Kurulumu**: Geliştirme ortamını doğru şekilde yapılandırmak zaman alıcıydı. Çözüm olarak, gerekli kütüphaneler kuruldu ve sistem çağrılarının nasıl çalıştığı üzerine araştırmalar yapıldı.
2. **Komut Ayrıştırma**: Komutlar bazen hatalı parametreler içeriyordu. strtok() fonksiyonu ile komutlar doğru şekilde ayrıştırıldı ve gereksiz boşluklar temizlendi.
3. **fork() ve execvp() Sorunları**: Çocuk süreçler bazen hatalı çalışıyordu. fork() ve execvp() doğru şekilde kullanıldı, süreçler doğru şekilde yönetildi ve waitpid() ile süreç tamamlanması sağlandı.
4. **Çıktı Yönlendirme ve Pipe İşlemleri**: dup2() fonksiyonu ile dosya tanıtıcıları doğru yönlendirildi, pipe işlemleri düzgün şekilde çalıştırıldı.
5. **Hata Mesajları ve Kullanıcı Geri Bildirimi**: Başlangıçta eksik olan hata mesajları eklendi, kullanıcıya açık geri bildirimler sağlandı.

Sonuç olarak, proje başarıyla tamamlandı ve tüm işlevler beklendiği şekilde çalıştı.