SAKARYA ÜNİVERSİTESİ BİLGİSAYAR VE BİLİŞİM BİLİMLERİ FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

İşletim Sistemleri

GRP24OGR1SUBA

B221210103 SÜLEYMAN SAMET KAYA

B221210052 İBRAHİM GÜLDEMİR

G221210046 İSMAİL KONAK

B221210064 TUĞRA YAVUZ

B221210065 İSMAİL ALPER KARADENİZ

GitHub: https://github.com/ibrhmgldmr3/isletim_sistemleri_odev.git

1. GİRİŞ

Bu proje, İşletim Sistemleri dersi kapsamında temel proses yönetimi, giriş/çıkış (I/O) işlemleri ve sinyal yönetimi (özellikle SIGCHLD sinyalinin işlenmesi) konularını uygulamalı olarak öğrenmek ve pekiştirmek amacıyla geliştirilmiştir. Projenin odak noktası, bir kabuk (shell) uygulamasının temel işlevlerini hayata geçirerek işletim sistemlerinin terminal veya kabuk tabanlı programlarının temel mekanizmalarını derinlemesine anlamaktır.

Proje kapsamında ele alınan başlıca konular şunlardır:

- **Komut Satırından Girdi Alma:** Kullanıcıdan veri alarak işlem başlatma.
- **Girdi Ayrıştırma (Parsing):** Komutların ve argümanların ayrıştırılması.
- **Çocuk Süreçlerin Oluşturulması:** Fork-exec mekanizmasını kullanarak yeni süreçler başlatma.
- **Giriş/Çıkış Yönlendirmesi:** Dosyaya yazma, dosyadan okuma gibi I/O işlemlerinin yönetimi.
- **Arka Plan Süreç Yönetimi:** Arka planda çalışan süreçlerin başlatılması ve izlenmesi.
- **Pipelin Kullanımı:** Boru (pipe) mekanizmasını kullanarak süreçler arasında veri akışını sağlama.
- **Yerleşik Komutların Yönetimi:** quit, jobs, increment gibi temel yerleşik (built-in) komutların uygulanması ve yönetimi.

Bu çalışma, kullanıcı seviyesindeki bir kabuk uygulamasının nasıl çalıştığını uygulamalı bir şekilde anlamaya yönelik olarak tasarlanmış ve süreçlerin detaylı yönetimi üzerine yoğunlaşılmıştır. Böylece, işletim sistemlerinin temel işlevlerini somut bir uygulama üzerinden keşfetmek ve teorik bilgiyi pratiğe dönüştürmek mümkün olmuştur.

2. Geliştirilen Kabuk Programı: Teknik Özellikler ve Fonksiyonellik

Proje kapsamında geliştirilen kabuk programı, kullanıcıdan komut almak, bu komutları ayrıştırmak ve yürütmek üzere tasarlanmıştır. Kabuk programı, aşağıdaki temel fonksiyonları ve özellikleri sağlamaktadır:

1. **Komut İstemci (Prompt):**
   Kullanıcıya bir komut istemi sunar ve komut girdisini bekler.

2. **Komut Ayrıştırma ve Çalıştırma:**
   Girilen komutları satır bazında ayrıştırarak çeşitli durumları ele alır:

   - **Standart Komut Yürütme:** Girilen komutların temel olarak çalıştırılması.

   - **Pipe (|) Kullanımı:** Komutlar arasında veri akışı sağlanması.

   - **Giriş ve Çıkış Yönlendirmesi (<, >, >>):** Dosyalardan okuma ve dosyalara yazma işlemleri.

   - **Arka Planda Çalışma (&):** Komutların arka planda çalıştırılması.

   - **Birden Fazla Komut Çalıştırma (;):** Noktalı virgül ile birden fazla komutun sıralı yürütülmesi.

3. **Altyapı ve Sistem Çağrıları:**

   - **Yeni Süreç Oluşturma:** fork() sistemi çağrısı kullanılarak yeni süreçler oluşturulur.

   - **Komut Yürütme:** exec() ailesine ait bir fonksiyon ile yeni süreç, ilgili komutun çalıştırılabilir hale getirilir.

   - **Süreç Yönetimi:**

     - Ön planda çalışan süreçlerin tamamlanması için wait() veya waitpid() çağrıları kullanılır.

     - Arka planda tamamlanan süreçlerin durumu, SIGCHLD sinyali yakalanarak kullanıcıya bildirilir.

4. **Güvenli Kapanış:**
   Kullanıcı quit komutunu girdiğinde, kabuk programı arka planda çalışan tüm süreçlerin tamamlanmasını bekler ve ardından güvenli bir şekilde sonlanır.

Bu temel özellikler, proje gereksinimlerini karşılayarak kullanıcı dostu ve güvenilir bir çalışma ortamı sunmayı amaçlamaktadır.

3. PROJENİN AMACI VE KAPSAMI

- Bu proje, modüler ve derli toplu bir yapıda birkaç temel fonksiyon/modüle ayrılarak tasarlanmıştır. Aşağıda, projenin ana bileşenleri ve bu bileşenlerin görevleri detaylı şekilde açıklanmıştır.

3.1 Kabuk Döngüsü (Main Loop)

Program, sürekli çalışan bir döngü içerisinde tasarlanmıştır. Döngünün temel işlevleri şunlardır:

1. **Prompt Gösterimi:** Kullanıcıdan komut beklemek amacıyla ekrana bir > işareti yazılır.
2. **Komut Okuma:** Kullanıcı girdisi fgets() gibi işlevlerle okunur.
3. **Girdi Analizi:** Aşağıdaki unsurlar kontrol edilir:
    - **Noktalı virgül (;):** Çoklu komut ayracı.
    - **Boru (|):** Pipe ile komut zincirleme.
    - **Giriş yönlendirmesi (<):** Dosyadan veri okuma.
    - **Çıkış yönlendirmesi (>, >>):** Dosyaya veri yazma.
    - **Arka plan simgesi (&):** Arka planda çalışma işareti.
4. **Fonksiyon Çağrısı:** Analiz edilen girdiye uygun fonksiyon çağrılır ve komut işlenir.
5. **Program Sonlandırma:** Kullanıcı quit komutunu girdiğinde:
    - Eğer arka planda çalışan süreç yoksa program sonlanır.
    - Arka planda çalışan süreçler varsa, bunların tamamlanması beklenir.
6. **Döngü Yeniden Başlatma:** Yeni bir komut beklemek için döngü başa döner.

3.2 Tekli Komut ve Yerleşik (Built-in) Komutlar

Tekli Komut İşleme:

- **Çocuk Süreç Oluşturma:** Komut, fork() ile yeni bir süreçte çalıştırılır.
- **Program Çalıştırma:** Çocuk süreçte exec() ile ilgili program çağrılır.
- **Ebeveyn Sürecin Beklemesi:**
    - Komut ön planda çalışıyorsa ebeveyn süreç wait() çağrısı ile çocuğun tamamlanmasını bekler.
    - Arka planda çalışıyorsa ebeveyn süreç beklemeden yeni komutları işleyebilir.

Yerleşik Komutlar:

- **quit:** Programı sonlandırır. Arka planda süreçler varsa bunların tamamlanmasını bekler.
- **increment:** Standart girdiden bir tamsayı alır, 1 artırır ve sonucu ekrana yazdırır.
- **jobs:** (İsteğe bağlı) O anda arka planda çalışan süreçleri listelemek için kullanılır.

3.3 Giriş ve Çıkış Yönlendirmesi (Redirection)

- **Giriş Yönlendirmesi (<dosya.txt):**
    1. Çocuk süreç oluşturulur.
    2. Standart giriş (STDIN), belirtilen dosyaya yönlendirilir (dup2() ile).
    3. Dosya bulunamazsa "Giriş dosyası bulunamadı." mesajı verilir.
- **Çıkış Yönlendirmesi (>dosya.txt, >>dosya.txt):**
    1. Çocuk süreç oluşturulur.
    2. Standart çıktı (STDOUT), belirtilen dosyaya yönlendirilir.
        - > dosyayı sıfırdan başlatır.

        - >> dosyaya ekleme yapar.
    3. Komut çıktısı belirtilen dosyaya yazılır.

3.4 Boru (Pipe) Yönetimi

- **Amaç:** Birden fazla komutun birbirine veri aktararak çalışmasını sağlamak.
    - Örnek: echo 12 | increment
        - echo 12 çıktısı, increment girdisi olarak kullanılır ve sonuç 13 ekrana yazılır.
- **Uygulama:**

1. pipe() ile boru dosya tanımlayıcıları oluşturulur.
2. Her komut için bir çocuk süreç oluşturulur.
3. dup2() ile borunun yazma ucu, sonraki komutun okuma ucuna bağlanır.
4. Ebeveyn süreç, tüm çocuk süreçlerin tamamlanmasını bekler

3.5 Arka Plan Süreç Yönetimi ve SIGCHLD

- **Arka Plan Komutları:** Komutun sonunda & bulunması, komutun arka planda çalışacağını belirtir.
    - Ebeveyn süreç, çocuk süreci beklemeden yeni komutları işleyebilir.
- **Sinyal Yönetimi (SIGCHLD):**
    - Arka plan süreci tamamlandığında sistem tarafından SIGCHLD sinyali gönderilir.
    - Bu sinyal, özel bir işleyici (sigaction()) ile yakalanır.
    - Süreç hakkında aşağıdaki bilgiler ekrana yazdırılır:
        - Süreç kimlik numarası (PID).
        - Çıkış kodu (Exit Code).

- **Program Kapanışı:**
    - Kullanıcı quit komutunu girdiğinde, arka planda süreç varsa bunların tamamlanması beklenir.
    - Tüm süreçler tamamlandıktan sonra program sonlanır.

4. KULLANILAN SİSTEM ÇAĞRILARI VE KÜTÜPHANELER

Projede, sistem düzeyinde işlevsellik sağlamak ve uygulamanın gereksinimlerini karşılamak amacıyla aşağıdaki sistem çağrıları ve kütüphaneler etkin bir şekilde kullanılmıştır:

- **fork()**: Yeni bir çocuk süreç oluşturmak için kullanılmıştır. Bu çağrı, ebeveyn sürecin bir kopyasını oluşturur ve süreçlerin paralel çalışmasına olanak tanır.
- **execvp()**: Mevcut süreci, belirtilen bir yürütülebilir dosya (örneğin, /bin/ls veya /usr/bin/cat) ile değiştirir. Bu, dış komutların çağrılmasında temel bir rol oynar.
- **wait() ve waitpid()**: Çocuk süreçlerin tamamlanmasını beklemek için kullanılmıştır. Ön plan süreçlerinde wait() veya waitpid() çağrısı ile bloklayıcı bir bekleme uygulanırken, arka plan süreçleri için waitpid(-1, &status, WNOHANG) kullanılarak engelleyici olmayan (non-blocking) bir bekleme mekanizması sağlanmıştır.
- **pipe()**: İki süreç arasında veri iletişimini sağlamak için bir boru (pipe) mekanizması kurulmuştur. Bu, süreçler arası veri aktarımı için kullanılmıştır.
- **dup2()**: Standart giriş ve çıkış (stdin/stdout) yönlendirme işlemleri gerçekleştirilirken dosya tanımlayıcılarını yeniden yönlendirmek için kullanılmıştır.
- **sigaction()**: Sinyallerin yönetimi için kullanılmıştır. Özellikle, çocuk süreçlerin bitişini yakalamak amacıyla SIGCHLD sinyalinin ele alınmasında kritik bir işlev üstlenmiştir.
- **open() ve close()**: Dosyaların açılması ve kapatılması gibi düşük seviyeli dosya işlemlerinde kullanılmıştır. Bu çağrılar, esnek ve yüksek performanslı dosya yönetimi sağlar.
- **strtok(), strstr() ve diğer string fonksiyonları**: Komut satırı girdilerinin ayrıştırılması ve işlenmesinde rol oynamıştır. Bu fonksiyonlar, metin tabanlı girdilerin ayrıştırılmasını ve gerekli verilerin elde edilmesini sağlamıştır.

Proje kapsamında yukarıdaki sistem çağrıları ve fonksiyonların kullanımı için ilgili başlık dosyaları (unistd.h, sys/wait.h, signal.h, vb.) dahil edilmiştir. Bu sayede, süreç yönetimi, dosya işlemleri ve sinyal yönetimi gibi temel işletim sistemi işlevleri etkin bir şekilde gerçekleştirilmiştir.

5. ADIM ADIM UYGULAMA DETAYLARI

Bu bölümde, bir kabuğun (shell) tipik komutları nasıl işlediği, örnek senaryolar üzerinden adım adım açıklanmaktadır. Her bir senaryo, sürecin temel işleyiş mekanizmasını detaylandırır:

Tek Komut İşleme

Örnek: ls -l

1. Kullanıcı, ls -l komutunu prompt'a yazar.
2. Kabuk, fork() sistem çağrısını kullanarak bir çocuk süreç oluşturur.
3. Çocuk süreç, execvp("ls", ["ls", "-l", NULL]) çağrısı ile ls komutunu çalıştırır.
4. Ebeveyn süreç, wait() fonksiyonu ile çocuk sürecin tamamlanmasını bekler.
5. Komut tamamlandıktan sonra yeni bir prompt kullanıcıya gösterilir.

Çıkış Yönlendirmeli Komut İşleme

Örnek: cat file1 > file2

1. Kabuk, komut sonunda > operatörünü tespit eder.
2. Yeni bir çocuk süreç oluşturulur.
3. Çocuk süreçte, **STDOUT** dosya tanımlayıcısı, file2 dosyasına yönlendirilir (varsa üzerine yazılır).
4. cat file1 komutu çalıştırılır; çıktısı file2 dosyasına kaydedilir.
5. İşlem tamamlandığında kabuk, kullanıcıya yeni bir prompt gösterir.

Giriş Yönlendirmeli Komut İşleme

Örnek: cat < file.txt

1. Kabuk, komut satırında < operatörünü tespit eder.
2. Yeni bir çocuk süreç oluşturulur.
3. Çocuk süreçte, **STDIN** dosya tanımlayıcısı, file.txt dosyası ile değiştirilir.
4. cat programı, standart girişini terminal yerine file.txt dosyasından alır.
5. İşlem tamamlanınca kabuk, yeni bir prompt ile kullanıcıya döner.

Arka Planda Çalışan Komut

Örnek: sleep 5 &

1. Kabuk, komutun sonunda & operatörünü tespit eder.
2. Yeni bir çocuk süreç oluşturulur ve sleep 5 komutu arka planda çalıştırılır.
3. Kabuk, ön planda beklemeden kullanıcıya yeni bir prompt gösterir.
4. Arka plandaki süreç tamamlandığında, kabuk bir **SIGCHLD** sinyali alır.
5. Kabuk, bu sinyali yakalar ve [pid X] retval: 0 benzeri bir mesaj gösterir.

Boru Operatörü Kullanımı

Örnek: echo 12 | increment

1. Kabuk, iki komutun | karakteri ile ayrıldığını algılar.
2. Bir boru (pipe) oluşturulur:
    - Yazma ucu echo 12 komutuna,

    - Okuma ucu increment komutuna bağlanır.
3. echo 12 komutunun çıktısı, borunun yazma ucuna aktarılır.
4. increment komutu, giriş verisini borunun okuma ucundan alır.
5. İşlem tamamlandığında, sonuç olan 13 ekranda görüntülenir.

Noktalı Virgülle Ayrılmış Sıralı Komutlar

Örnek: echo 12; sleep 2; echo 13

1. Kabuk, komutları ; karakteri ile ayrılmış bağımsız parçalar olarak algılar.
2. Her komut için sırasıyla:
    - Yeni bir çocuk süreç oluşturulur.
    - İlgili komut çalıştırılır.
    - Çıktısı tamamlandığında bir sonraki komuta geçilir.
3. İşleyiş sırası:
    - echo 12 hemen çalıştırılır ve 12 ekrana yazılır.
    - Ardından sleep 2 çalıştırılarak 2 saniye beklenir.
    - Son olarak echo 13 çalıştırılarak 13 ekrana yazılır.

Bu açıklamalar, bir kabuğun işlevselliğini ve komutların nasıl işlendiğini detaylandırarak kullanıcıya net bir rehber sunar.

6. HATA YÖNETİMİ VE GÜVENLİK

Kabuk, kullanıcı deneyimini iyileştirmek ve sistem güvenilirliğini artırmak amacıyla kapsamlı bir hata yönetimi ve güvenlik altyapısına sahiptir:

1. **Hata Mesajları ve Doğrulama:** Yanlış dosya adı girişi, geçersiz semboller veya eksik parametreler gibi durumlarda, kullanıcıyı bilgilendirmek için anlamlı ve açıklayıcı hata mesajları üretilir. Örneğin, "Giriş dosyası bulunamadı." veya "Geçersiz komut girdisi." şeklinde geri bildirim sağlanır.

2. **Arka Plan Süreç Yönetimi:** Arka plan süreçlerinin sayısı, sistem kaynaklarının etkin kullanımı ve performansın korunması için belirli bir üst sınırla (örneğin, 10 işlem) sınırlandırılmıştır. Bu sınırın aşılması durumunda yeni süreç başlatılmasına izin verilmez ve kullanıcıya durumu bildiren bir hata mesajı sunulur.

3. **Sinyal Yönetimi:** Kabuk, beklenmeyen kesintilere karşı dayanıklılık sağlamak amacıyla sinyal yönetim mekanizmalarını etkili bir şekilde kullanır. Kritik işlemler sırasında sinyaller güvenli bir şekilde yakalanır ve işlenir. Örneğin, SIGINT veya SIGTERM sinyalleri, işlemlerin güvenli bir şekilde sonlandırılması için özel olarak ele alınır.

4. **Hata Kodları ve errno Kontrolleri:** Sistem çağrılarında oluşabilecek hatalar için errno kontrolleri gerçekleştirilir. Beklenmeyen durumlar için uygun düzeltici işlemler uygulanır ve hataların sistem geneline yayılmasını engellemek için gerekli önlemler alınır.

Bu yaklaşım, kabuğun kararlılığını artırırken kullanıcıların hataları hızlı bir şekilde teşhis edip çözmesine olanak tanır ve aynı zamanda sistemin güvenli bir şekilde çalışmasını sağlar.

7. SONUÇ VE DEĞERLENDİRME

Bu proje kapsamında, işletim sistemlerinin temel prensiplerinden biri olan çoklu süreç yönetimi; I/O yönlendirmesi, boru mekanizması ve sinyal işleme konularıyla birlikte uygulamalı olarak ele alınmıştır. Geliştirilen kabuk (shell), bir işletim sistemi kabuğunun temel işlevlerini başarıyla yerine getirebilecek aşağıdaki özelliklere sahiptir:

- **Prompt** görüntüleme ve kullanıcıdan komut alma
- **Tekli komut** icrası (ön planda çalıştırma)
- **Giriş (<)** ve **çıkış (>, >>)** yönlendirmesi

- **Arka planda çalıştırma (&)** ve tamamlanan süreçlerin durum bilgisini görüntüleme
- **Boru mekanizması (|)** kullanılarak çoklu komut zinciri oluşturma
- **Noktalı virgül (;)** kullanarak sıralı komut çalıştırma
- Yerleşik (built-in) komutlar:
    - **quit**: Kabuktan çıkış.
    - **jobs**: Arka planda çalışan süreçlerin listesini gösterme.
    - **increment**: Örnek bir yerleşik komut olarak süreçlere eklenmiştir.

Bu proje, işletim sistemi tasarımı ve süreç yönetimi ilkelerini uygulamalı olarak öğrenmek ve kavramları derinleştirmek açısından faydalı bir örnek teşkil etmektedir. Ayrıca, proje yapısının genişletilmeye uygun olması, çeşitli ek özelliklerin kolayca entegre edilebilmesine olanak tanımaktadır.

Potansiyel geliştirme alanları şunlardır:

- Komut geçmişi (history) tutma ve geçmiş komutlara erişim.
- Boru mekanizması ile I/O yönlendirme entegrasyonu.
- Komut tamamlama ve kullanıcı dostu hata mesajları.

Sonuç olarak, bu proje hem temel işletim sistemi prensiplerinin uygulanmasını hem de genişletilebilir bir kabuk tasarımının nasıl yapılabileceğini göstermiştir. Bu yönleriyle, ileri düzey çalışmalar ve özellik eklemeleri için sağlam bir temel sunmaktadır.

8. KAYNAKLAR

Operating System Concepts – Abraham Silberschatz, Peter B. Galvin, Greg Gagne
Bilgisayar bilimlerinde işletim sistemlerinin temel ilkelerini ve modern tasarım prensiplerini detaylı bir şekilde ele alan bu eser, işletim sistemlerine dair kapsamlı bir kaynak niteliğindedir.

Linux System Programming – Robert Love
Linux işletim sistemi üzerinde sistem programlama konularını derinlemesine inceleyen bu kitap, uygulamalı örneklerle birlikte düşük seviyeli programlama konularında rehberlik sağlar.

Teşekkür

Bu projeyi geliştirirken, işletim sistemleri dersinde edindiğimiz çoklu süreç yönetimi ve sinyal işleme konularını uygulama fırsatı bulduk. Proje süresince kazandığımız deneyim, gerçek bir kabuk uygulamasının temel yapı taşlarını anlamamıza büyük katkı sağladı.