SAKARYA ÜNİVERSİTESİ BİLGİSAYAR VE BİLİŞİM BİLİMLERİ FAKÜLTESİ BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

İşletim Sistemi

B221210103 SÜLEYMAN SAMET KAYA

B221210052 İBRAHİM GÜLDEMİR

G221210046 İSMAİL KONAK

B221210064 TUĞRA YAVUZ

B221210065 İSMAİL ALPER KARADENİZ

İşletim Sistemleri Ödev Raporu

1. GİRİŞ

Bu proje, İşletim Sistemleri dersi kapsamında temel proses yönetimi, I/O (giriş/çıkış) işlemleri ve sinyal kullanımı (özellikle SIGCHLD işleme) konularını uygulamalı olarak pekiştirmek amacıyla hazırlanmıştır. Proje süresince, bir kabuk (shell) uygulamasının en temel fonksiyonları olan:

-Komut satırından girdi alma,

-Girdi ayrıştırma (parsing),

- Çocuk süreç (fork-exec) oluşturma,

- I/O yönlendirmesi (dosyaya yazma, dosyadan okuma),

- Arka plan süreç yönetimi,

- Boru (pipe) kullanımı,

- Basit yerleşik komutların (built-in) yönetilmesi (örneğin quit, jobs, increment)

- konuları ele alınmıştır. Bu sayede bir işletim sisteminin terminal veya kabuk benzeri programlarının temelinde yatan mekanizmalar somut bir örnek üzerinden incelenmiştir.

2. PROJENİN AMACI VE KAPSAMI

- Proje kapsamında geliştirilen kabuk programı:Komut istemi (prompt) göstererek kullanıcının komut girmesini bekler. Kullanıcı komut girdikten sonra, bu komutu satır bazında ayrıştırır; Standart komut yürütme, Pipe (|) ayrıştırma, Giriş yönlendirmesi <, Çıkış yönlendirmesi > veya >>, Arka planda çalışma (&), Noktalı virgül ; ile birden fazla komut yürütme gibi durumları ele alır. Altyapı olarak: fork() sistemi çağrısı ile yeni süreçler oluşturulur, exec() ailesinden bir fonksiyon ile yeni süreç, ilgili komutu çalıştıracak hale getirilir, wait() / waitpid() ile ön planda çalışan çocuk süreçler tamamlanıncaya kadar beklenir, SIGCHLD sinyali yakalanarak arka planda tamamlanan süreçlerin durumu kullanıcıya bildirilir. Kabuk, quit komutu girildiğinde, arka planda çalışan süreçler bitene kadar bekler ve ardından güvenli şekilde sonlanır. Bu temel fonksiyonlar projede istenen tüm gereksinimleri karşılamayı hedeflemektedir.

3. PROJENİN AMACI VE KAPSAMI

- Proje, derli toplu bir yapıda birkaç temel modüle/fonksiyona bölünerek tasarlanmıştır. Aşağıda, tasarımın ana bileşenleri ve bu bileşenlerin görevleri açıklanmaktadır.

3.1 Kabuk Döngüsü (Main Loop)

Program, sonsuz bir döngü içinde çalışır. Döngüde: Prompt (>) ekrana yazılır. Kullanıcıdan komut satırı (fgets() vb. ile) okunur. Girdi satırı içerisinde: Noktalı virgül (;) var mı,

Boru (|) var mı, Giriş yönlendirmesi (<) var mı, Çıkış yönlendirmesi (>) veya (>>) var mı, Sonunda & var mı gibi kontroller yapılır. İlgili duruma göre (öncelik sırası veya tespit edilen sembole göre) doğru fonksiyon çağrılarak komut çalıştırılır. Eğer komut quit ise ve arka planda süreç yoksa program sonlanır. Eğer arka planda çalışan süreçler varsa, bunların bitmesi beklenir. Döngü en başa döner ve yeni bir komut bekler.

3.3 Tekli Komut ve Yerleşik (Built-in) Komutlar

Tekli komut senaryosunda; fork() ile bir çocuk süreç oluşturulur, exec() ile ilgili program çağrılır, ebeveyn süreç wait() ile çocuğu bekler (eğer komut ön planda çalışıyorsa). Arka planda çalıştırılacak komutların sonunda & bulunur. Bu durumda ebeveyn süreç, wait() çağırmadan doğrudan yeni komutlara geçebilir. Arka planda tamamlanan süreçler ise SIGCHLD sinyaliyle yakalanır. Yerleşik komutlar (Built-in) olarak projede şu komutlar ele alınmıştır: quit: Kabuktan çıkış yapar. Arka plan süreç varsa bitmesi beklenir, aksi takdirde doğrudan sonlanır. increment: Standart girdi (stdin) üzerinden bir tamsayı okur, bir artırır ve çıktıyı ekrana basar. (Proje gereği özel bir fonksiyon) jobs: O anda arka planda çalışan süreçleri listelemek için kullanılır (isteğe bağlı ek özellik olarak da uygulanabilir).

3.4 Giriş ve Çıkış Yönlendirmesi (Redirection)

Giriş yönlendirmesi <dosya.txt>: Çocuk süreç oluşturulur, Çocuğun standart girdisi (STDIN) ilgili dosyaya yönlendirilir (dosya açılır, dup2() ile STDIN_FILENO yönlenir), Komut çalıştırılır. Dosya yoksa “Giriş dosyası bulunamadı.” mesajı verilir. Çıkış yönlendirmesi >dosya.txt veya >>dosya.txt: Çocuk süreç oluşturulur, Çocuğun standart çıktısı (STDOUT) ilgili dosyaya yönlendirilir (dup2() aracılığıyla). > durumu dosyayı sıfırdan (truncate) açar. >> durumu ise ekleme (append) modunda açar. Komutun çıktısı bu dosyaya yazılır.

3.5 Boru (Pipe) Yönetimi

Boru sembolü (|) kullanıldığında, birden çok komutun zincirleme şekilde birbirine bağlanması hedeflenir. Örnek: echo 12 | increment echo 12 çıktısı, increment komutunun girdisi olur ve sonuçta ekrana 13 basılır. Projede, pipe() sistem çağrısı ile gerekli boru dosya tanımlayıcıları oluşturulur. Peş peşe gelen komutlar için, her komut bir child süreç olarak fork() edilir. Bir önceki komutun yazma ucunun, sonraki komutun okuma ucuyla eşleştirilmesi dup2() ile sağlanır. En sonunda, tüm çocuk süreçler tamamlanana kadar ebeveyn süreç bekler.

3.6 Arka Plan Süreç Yönetimi ve SIGCHLD

Arka plan komutları '&' sembolü ile sonlanır. Ebeveyn süreç, bu çocuk süreci beklemeden yeni komutlara döner. Arka plan süreci bittiğinde, sistem SIGCHLD sinyali gönderir. Kabuk programı, bu sinyali özel bir işleyici fonksiyonla (sigaction()) yakalar ve biten süreç hakkında: Sürecin pid numarası, Çıkış kodu (exit code) gibi bilgileri ekrana yazar. Kullanıcı quit komutu

girdiğinde, eğer hâlâ arka planda çalışan süreçler varsa, kabuk bu süreçlerin bitmesini bekler. Ardından kabuk kapatılır.

4. KULLANILAN SİSTEM ÇAĞRILARI VE KÜTÜPHANELER

Projede başlıca şu sistem çağrıları ve fonksiyonlar kullanılmıştır: fork(): Yeni bir çocuk süreç oluşturur. execvp(): Mevcut süreci, belirtilen yürütülebilir (ör. /bin/ls, /usr/bin/cat) haline getirir. wait() / waitpid(): Çocuk sürecin bitmesini bekler (ön plan yürütme için). Arka plan süreçlerde waitpid(-1, &status, WNOHANG) şeklinde, biten süreci engellemeyen (non-blocking) bekleme kullanılır. pipe(): İki süreç arasında veri aktarımı için boru mekanizması. dup2(): Dosya tanımlayıcılarını yeniden yönlendirmeye (stdin/stdout) yarar. sigaction(): Sinyalleri ele almak için (ör. SIGCHLD sinyalini yakalama) kullanılır. open(), close(): Dosya açma/kapama operasyonları. strtok(), strstr() vb. string fonksiyonları: Komut ayrıştırma (parsing) amaçlı. Tüm bu çağrılar proje dosyalarında ilgili başlık dosyaları (unistd.h, sys/wait.h, signal.h vb.) eklenerek kullanılmıştır.

5. ADIM ADIM UYGULAMA DETAYLARI

Bu bölümde, kabuğun aldığı tipik komutların nasıl işlendiği örnek senaryolarla özetlenmektedir: Tek Komut (ls -l) Prompt'ta ls -l yazılır. Kabuk, fork() ile bir çocuk süreç oluşturur. Çocuk süreç, execvp("ls", ["ls", "-l", NULL]) komutunu çalıştırır. Ebeveyn süreç, çocuk bitene kadar wait() ile bekler. Komut tamamlanınca yeni bir prompt gösterilir. Çıkış Yönlendirmeli Komut (cat file1 > file2)Komut sonundaki > tespit edilir. Çocuk süreç oluşturulur. Çocuk süreçte STDOUT dosya tanımlayıcısı file2 dosyasına yönlendirilir (sil-baştan yazma). cat file1 komutu çalıştırılır, çıktısı file2 dosyasında gözlemlenir. Giriş Yönlendirmeli Komut (cat < file.txt) Komut satırında < tespit edilir. Çocuk süreçte STDIN, file.txt dosyası ile değiştirilir. cat programı, standart girdisini terminal yerine file.txt dosyasından almış olur. Arka Planda Komut (sleep 5 &) Son karakter & tespit edildiğinde, kabuk ön planda beklemeden yeni prompt'a döner. sleep 5 arka planda çalışır. 5 saniye sonra süreç bittiğinde SIGCHLD sinyali kabuğa gelir, kabuk sinyali yakalayıp [pid X] retval: 0 benzeri bir mesaj gösterir. Boru Kullanımı (echo 12 | increment) Kabuk, iki komutun | ile ayrıldığını tespit eder. Boru oluşturur: yazma ucu echo 12, okuma ucu increment komutuna bağlanır. echo 12 çıktısı pipe'a, increment girişi pipe'dan alınır. Sonuç olarak ekranda 13 görüntülenir. Noktalı Virgülle Sıralı Komutlar (echo 12; sleep 2; echo 13)

; karakteri ile ayrılan her parça birer ayrı komut olarak algılanır. Kabuk sırasıyla her komut için yeni bir alt süreç oluşturur, ilgili komut çalıştırılır. Komutun çıktısı geldikten sonra bir sonrakine geçilir. echo 12 hemen çalışır, ardından sleep 2 ile 2 saniye beklenir, en son echo 13 basılır.

6. ADIM ADIM UYGULAMA DETAYLARI

Yanlış dosya adı girişi, kullanılmayan semboller gibi durumlarda kabuk uygun hata mesajları basar (ör. "Giriş dosyası bulunamadı."). Arka plan süreç sayısı belli bir üst sınırla (ör. 10) sınırlandırılmış, daha fazlasına izin verilmeyecek şekilde kontrol konmuştur.Sinyal yönetimi ve errno kontrolleri yapılarak, beklenmeyen kesintilerde kabuğun sağlam çalışması hedeflenmiştir.

7. SONUÇ VE DEĞERLENDİRME

Bu projede, işletim sistemlerinin temel prensiplerinden biri olan çoklu süreç yönetimi ile birlikte I/O yönlendirmesi, boru mekanizması ve sinyal işleme uygulamalı olarak gösterilmiştir. Geliştirilen kabuk (shell) aşağıdaki özellikleri yerine getirmektedir: Prompt görüntüleme ve komut alma Tekli komut icrası (ön planda) Giriş (<) ve çıkış (>, >>) yönlendirmesi Arka planda (&) komut çalıştırma ve biten süreçlerin bilgisini gösterme Boru (|) kullanımı ile çoklu komut zinciri Noktalı virgül (;) ile sıralı çoklu komut çalıştırma Yerleşik (built-in) komutlar: quit, jobs, increment (örnek) Proje; bir işletim sistemi kabuğunun çekirdek işlevlerini göstermesi, süreç yönetimi ve sinyal mekanizmalarını tanıtması açısından faydalı olmuştur. Genişletilmeye ve geliştirmeye de oldukça uygun bir yapıdadır. İstenirse, komut geçmişi (history), boru ile birlikte yönlendirme veya komut tamamlama gibi ek özellikler kolayca eklenebilir.

8. KAYNAKLAR

-Operating System Concepts – Abraham Silberschatz, Peter B. Galvin, Greg Gagne

- Linux System Programming – Robert Love

Teşekkür

Bu projeyi geliştirirken, işletim sistemleri dersinde öğrendiğimiz çoklu süreç yönetimi ve sinyal işleme konularını uyguladık. Proje boyunca elde ettiğimiz deneyim, gerçek bir kabuk uygulamasının temelini kavramamıza yardımcı oldu.