



**SAKARYA**  
ÜNİVERSİTESİ

# **SAKARYA ÜNİVERSİTESİ BİLGİSAYAR MÜHENDİSLİĞİ**

## **İŞLETİM SİSTEMLERİ PROJE ÖDEVİ**

**Proje Adı:** Temel Bir Linux Kabuk Uygulaması Geliştirme

**Grup Adı:** 12AB Grubu

**Grup Üyeleri:**

oÖmer Faruk Aydın - G221210370 - 1.Öğretim A Grubu

oTarık Kartal - Y245012016 - 1. Öğretim A Grubu

oHarun Mahmut Yeşilyurt - G211210089 - 2. Öğretim B Grubu

oBatuhan Bahayetmez - G211210043 - 2. Öğretim A Grubu

oTalha İmran Göneç - G221210032 - 2. Öğretim B Grubu

## Giriş

Bu proje, temel bir Linux kabuk (shell) uygulaması geliştirerek işletim sistemi konseptlerini öğrenmeyi amaçlamaktadır. Proje boyunca, proses yönetimi, I/O (Giriş/Çıkış) yönlendirme ve Linux sinyal kullanımı gibi temel işletim sistemi bileşenlerini uygulamalı olarak deneyimledik.

Kabuk uygulaması, kullanıcıdan gelen komutları okuyarak bu komutları yorumlamakta ve çeşitli sistem fonksiyonları aracılığıyla yürütmektedir. Bu proje, işletim sistemi kavramlarının gerçek dünyadaki bir uygulamasını anlamak ve bunları başarıyla uygulayabilmek için tasarlanmıştır.

## Genel Tasarım

Proje, modüler bir yaklaşımla tasarlanmıştır ve her bölüm belirli bir fonksiyonu yerine getirmek üzere ayrılmıştır. Kaynak dosyalar ve fonksiyonlar arasındaki bağlantıyı anlamak, projenin okunabilirliğini ve bakımını kolaylaştırmıştır:

- **main.c:** Programın başlangıç noktası olarak görev yapar ve kullanıcıdan gelen girdileri döngüye alır. Kullanıcı komutlarını ilgili modüllere iletir.
- **shell.c:** Komutları analiz eder, gerekirse alt süreçler oluşturur ve gerekli sistem çağrılarını yapar.
- **shell.h:** Prototip tanımları ve sabitleri barındırır.
- **Makefile:** Projenin derlenmesi ve çalıştırılması için gerekli talimatları içerir.
- **README.md:** Projenin genel tanıtımını, çalışma talimatlarını ve grup bilgilerimizi içerir.

## Projenin temel bölümleri şu şekilde tasarlanmıştır:

- 1.Komut İstemi (Prompt):** Kullanıcının komut girmesi için "SAU >" şeklinde bir arayüz sunar.
- 2.Komut Ayırıştırma:** Girilen komutlar noktalı virgül (;), boru (|) gibi operatörlere göre ayrıştırılır.
- 3.Proses Yönetimi:** Yeni prosesler oluşturulur ve bu prosesler Linux sistem çağrıları ile yürütülür.
- 4.I/O Yönlendirme:** Standart girdi/standart çıktı, kullanıcı tarafından belirtilen dosyalara yeniden yönlendirilir.
- 5.Arka Plan Çalışma:** Komutlar arka planda yürütülerek kullanıcının diğer komutları girmesine izin verilir.

# 1.Prompt

**Prompt Yazımı ve Kodun Detaylı Açıklaması** Kabuk programında her komutun tamamlanmasından sonra veya arka plan komutunun hemen ardından kullanıcıdan yeni bir komut almak için bir komut istemi ("prompt") yazdırılır. Kullanıcı deneyimini iyileştirmek için bu prompt, programın her zaman hazır olduğunu belirtir.

## 1. Prompt'un Oluşturulması

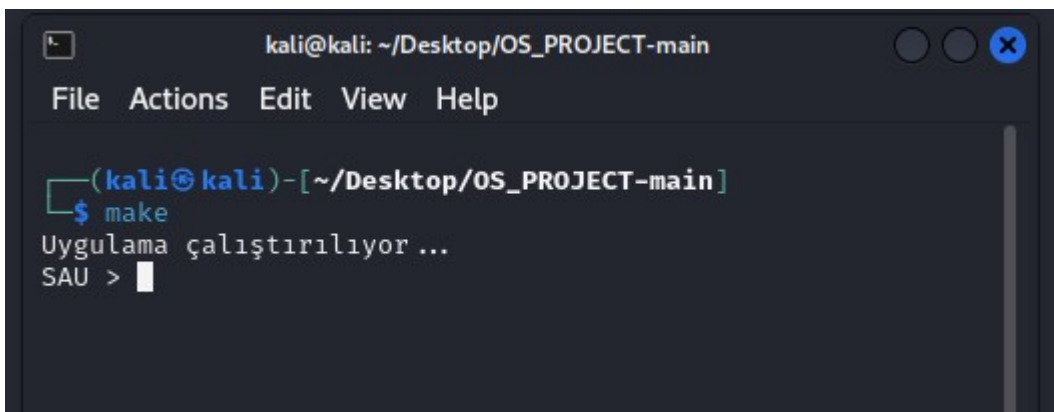
Kullanıcıya "SAU >" şeklinde bir komut istemi yazdırılır. Prompt'un her defasında doğru yazılması için tampon (buffer) içeriği temizlenir.

#### Kod Detayları:

```
// Kullanıcıya "SAU >" şeklinde bir komut istemi yazdırır
void print_prompt()
{
    printf("SAU > ");
    fflush(stdout);
}
```

#### Açıklama:

- **printf:** Kullanıcıya komut istemini ("prompt") yazdırır. **fflush:** Standart çıktı
- tamponunu boşaltır ve prompt'un hemen görünmesini sağlar. Bu, tamponun dolmasını beklemeden yazının ekrana çıkmasını garanti eder.



## 2.Quit

**Quit Komutunun İşlenmesi ve Kodun Detaylı Açıklaması** quit komutu, kabuk programını sonlandırmak için kullanılır. Kullanıcı bu komutu girdiğinde kabuk, aktif arka plan işlemlerini kontrol eder ve gerekirse bekler. Tüm arka plan işlemleri tamamlandıktan sonra kabuk sonlanır. **1. Komutun Tanımlanması**

Kullanıcı tarafından girilen komutlar ayrıştırılır ve "quit" anahtar kelimesi kontrol edilir. Eğer komut "quit" ise kabuk sonlandırılır.

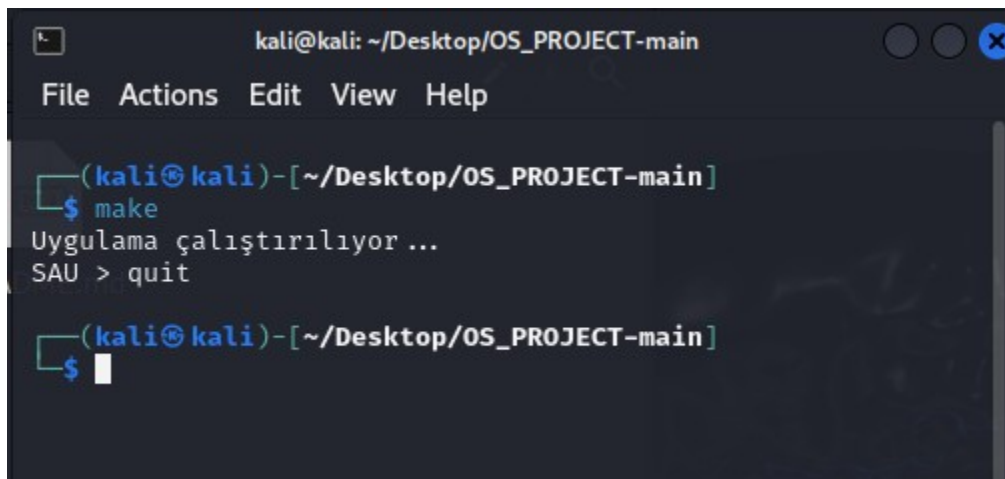
## Kod Detayları:

```
// quit komutu
if (args[0] != NULL && strcmp(args[0], "quit") == 0)
{
    exit(0);
}
```

## Açıklama:

- **strcmp fonksiyonu:** Girilen komutun "quit" olup olmadığını kontrol eder.
- **exit fonksiyonu:** Programı derhal sonlandırır.

Test 1: Programda herhangi bir işlem yapılmazken quit komutunu girin.

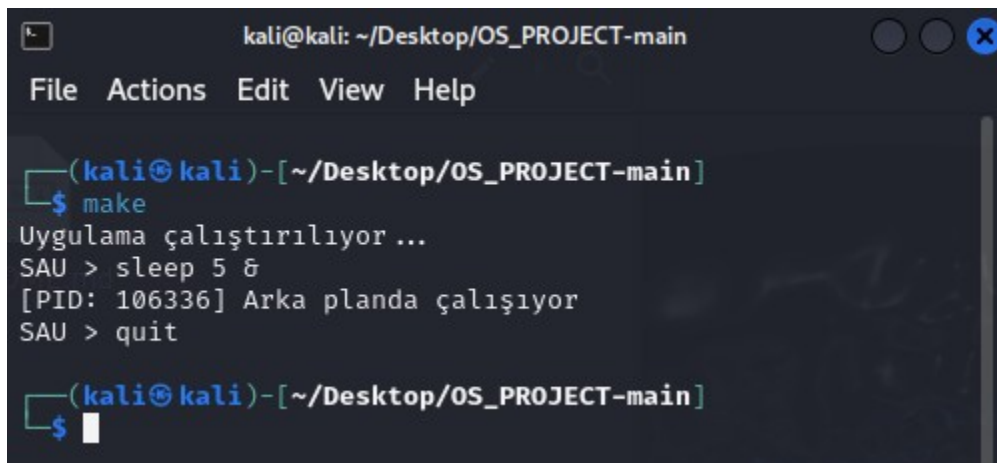


```
kali@kali: ~/Desktop/OS_PROJECT-main
File Actions Edit View Help

(kali@kali)-[~/Desktop/OS_PROJECT-main]
$ make
Uygulama çalıştırılıyor ...
SAU > quit

(kali@kali)-[~/Desktop/OS_PROJECT-main]
$
```

Test 2: Arka planda çalışan bir işlem varken quit komutunu girin.



```
kali@kali: ~/Desktop/OS_PROJECT-main
File Actions Edit View Help

(kali@kali)-[~/Desktop/OS_PROJECT-main]
$ make
Uygulama çalıştırılıyor ...
SAU > sleep 5 &
[PID: 106336] Arka planda çalışıyor
SAU > quit

(kali@kali)-[~/Desktop/OS_PROJECT-main]
$
```

# 3.Tekli Komutlar

## Tekli Komutların İşlenmesi ve Kodun Detaylı Açıklaması

Girilen bir komut, kabuk tarafından alt bir süreçte çalıştırılır ve tamamlanana kadar beklenir. Kabuk, bu işlem için fork, execvp ve waitpid gibi Linux sistem çağrılarını kullanır. Bu, komutun işlenmesi sırasında ana sürecin engellenmesini ve komutun tamamlandıktan sonra yeni bir komut almak için hazır olmasını sağlar.

## 1. Komutun Ayırıştırılması

Girilen komut, bağımsız değişkenlere ayrıştırılır. Bu işlem sırasında, arka plan çalışma için & sembolü de kontrol edilir.

### Kod Detayları:

```
// Komutları ayırıştır
args[i] = strtok(command, " ");
while (args[i] != NULL && i < MAX_ARGS - 1)
{
    if (strcmp(args[i], "&") == 0)
    {
        background = 1;
        args[i] = NULL;
        break;
    }
    i++;
    args[i] = strtok(NULL, " ");
}
```

### Açıklama:

- Komut, boşluk karakterine göre parçalanır ve her bir parça args dizisine kaydedilir.
- & sembolü algılandığında, arka plan çalışma modu etkinleştirilir.

## 2. Alt Süreç Oluşturulması

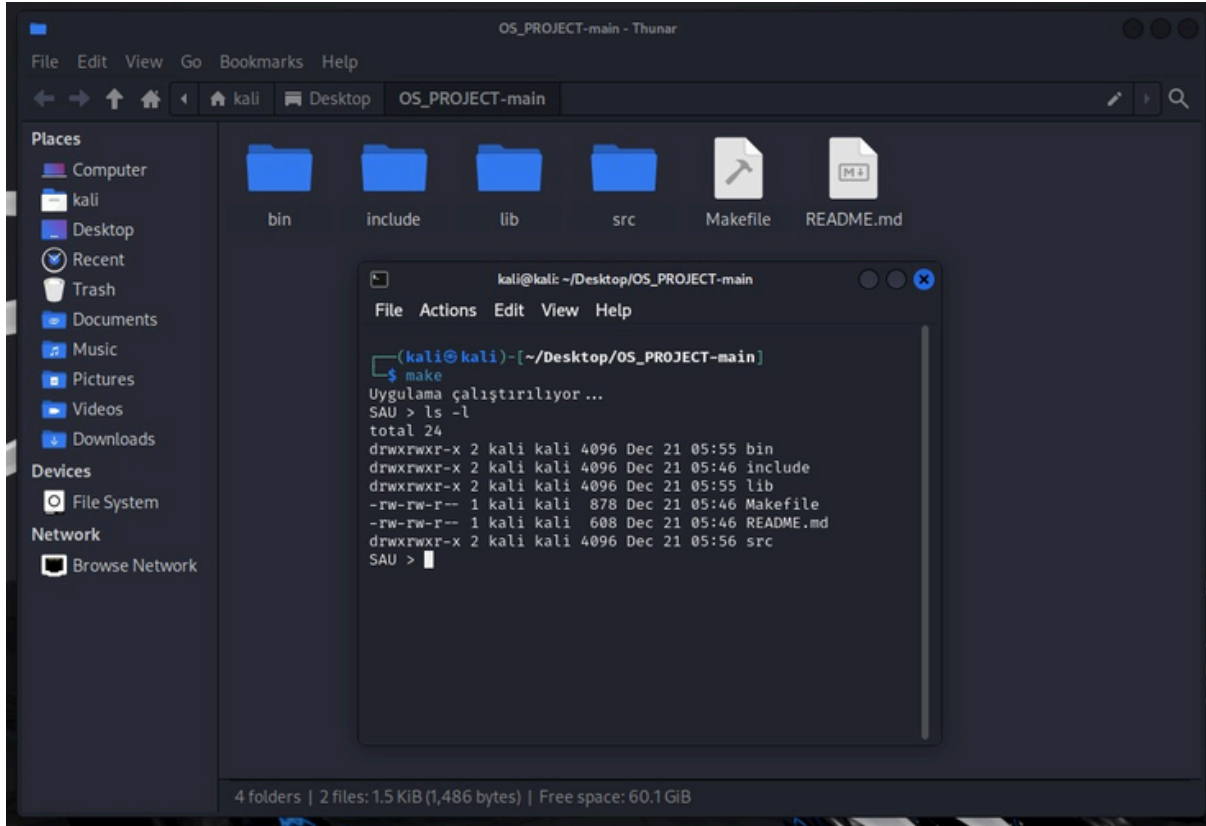
Yeni bir alt süreç oluşturmak için fork sistem çağrısı kullanılır. Alt süreç, girilen komutu yürütürken ana süreç bu işlem tamamlanana kadar bekler.

### Kod Detayları:

```
// Alt süreç oluştur ve çalıştır
pid_t pid = fork();
if (pid == 0)
{
    // Çocuk süreç
    handle_redirection(args);
    if (execvp(args[0], args) == -1)
    {
        perror("Komut çalıştırılamadı");
    }
    exit(EXIT_FAILURE);
}
else if (pid > 0)
{
    // Ana süreç
    if (!background)
    {
        int status;
        waitpid(pid, &status, 0);
    }
    else
    {
        printf("[PID: %d] Arka planda çalışıyor\n", pid);
    }
}
else
{
    perror("Fork başarısız oldu");
}
```

## Açıklama:

- **fork:** Yeni bir alt süreç oluşturur. Alt süreç (PID = 0) ve ana süreç ayrı şekilde devam eder.
- **execvp:** Alt süreç, girilen komutu yürütmek için bu sistem çağrısını kullanır. Komut bulunamazsa uygun bir hata mesajı yazdırılır.
- **waitpid:** Ana süreç, alt sürecin tamamlanmasını bekler.



## 4.Giriş Yönlendirme

**Giriş Yönlendirme Mantığı ve Kodun Detaylı Açıklaması** Komutun "komut <

GirişDosyası" şeklinde verilmesi durumunda, program alt prosesin standart girdisini belirtilen dosyadan alacak şekilde yönlendirir. Linux sistem çağrılarında dup2 bu işlemin temelini oluşturur. Eğer dosya bulunamazsa uygun bir hata mesajı döndürülür.

### 1. Komutun Tespiti

Komutun içinde "<" sembolü olup olmadığı kontrol edilir. Eğer "<" sembolü algılanırsa, sembolden sonra belirtilen dosya okuma modunda açılır ve standart girdi (STDIN) bu dosyaya yönlendirilir.

### Kod Detayları:

```
if (strcmp(args[i], "<") == 0)
{
    // Giriş yönlendirmesi
    redirected = 1;
    FILE *file = fopen(args[i + 1], "r");
    if (file == NULL)
    {
        perror("Giriş dosyası açılmadı");
        exit(EXIT_FAILURE);
    }
    if (fscanf(file, "%d", &num) != 1)
    {
        fprintf(stderr, "Hata: Dosyadan geçerli bir sayı okunamadı.\n");
        fclose(file);
        exit(EXIT_FAILURE);
    }
    fclose(file);
    break;
}
```

## Açıklama:

- ❑ **open Fonksiyonu:** Dosya yalnızca okuma modunda açılır. O\_RDONLY parametresi, dosyanın salt okunur modda açılmasını sağlar.
- ❑ **dup2 Sistem Çağrısı:** Standart girdi (STDIN\_FILENO) belirtilen dosya tanımlayıcısına yönlendirilir.
- ❑ **close Fonksiyonu:** Kullanılan dosya tanımlayıcısı kapatılır.

## 2. Dosya Açma Hataları

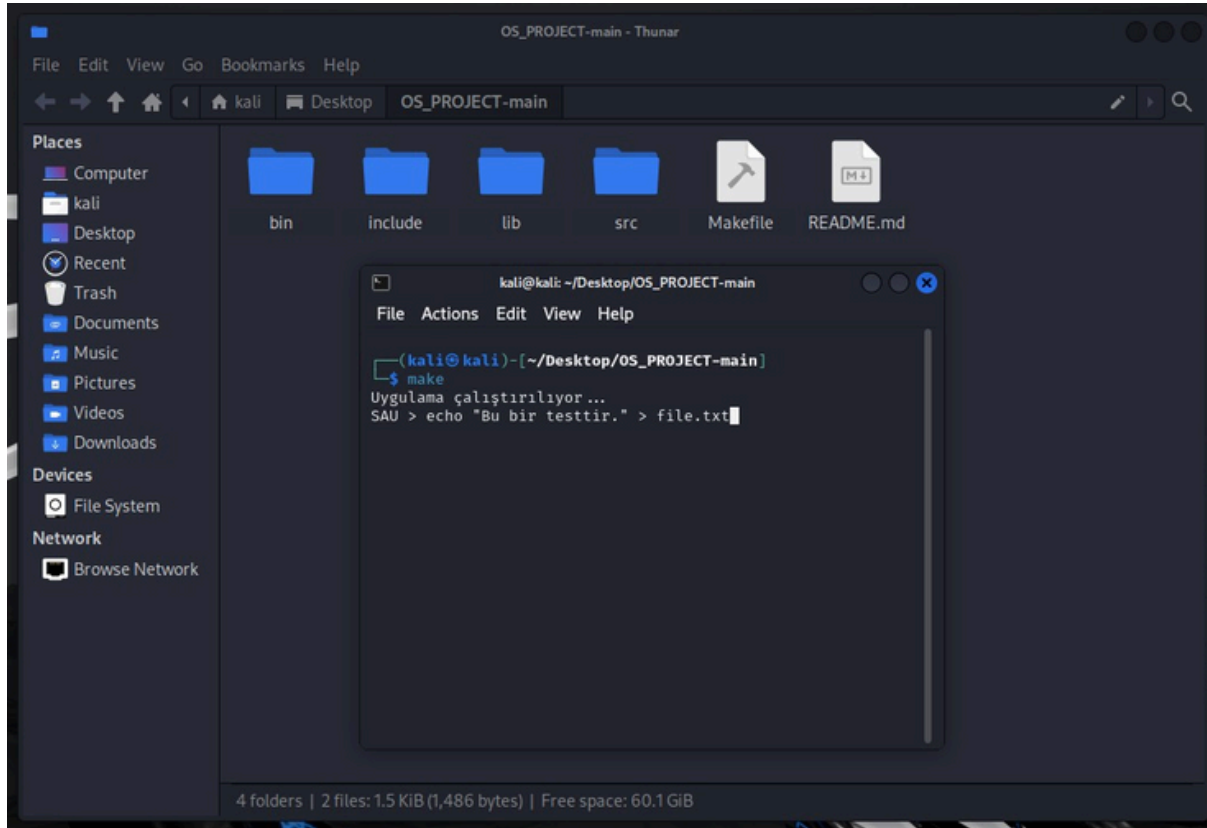
Dosyanın açılmaması durumunda, program bir hata mesajı yazdırır ve uygun bir çıkış kodu ile sonlanır.

### Kod Detayları:

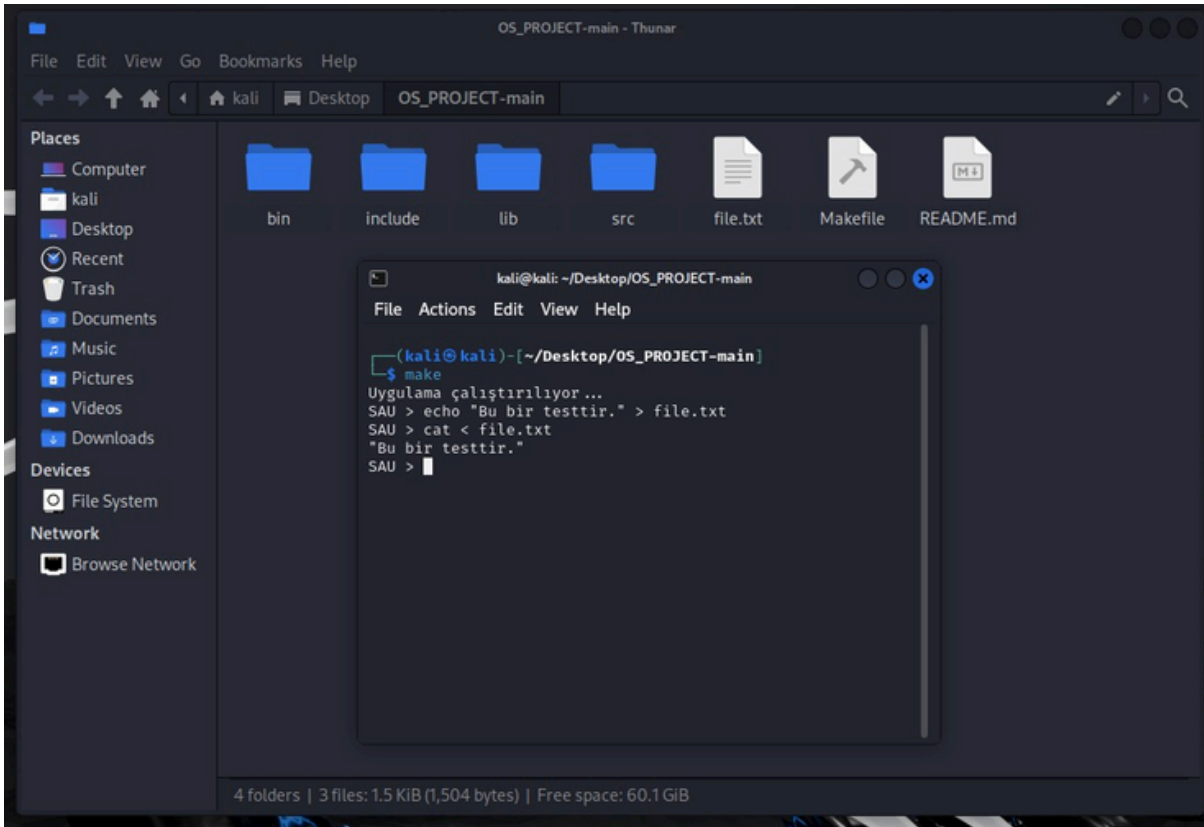
```
if (file == NULL)
{
    perror("Giriş dosyası açılmadı");
    exit(EXIT_FAILURE);
}
```

## Açıklama:

- ❑ Dosya açma işleminde hata oluşur ve dosya tanımlayıcısı negatif bir değer dönerse, perror ile hata mesajı yazdırılır.
- ❑ Program EXIT\_FAILURE kodu ile sonlandırılır.

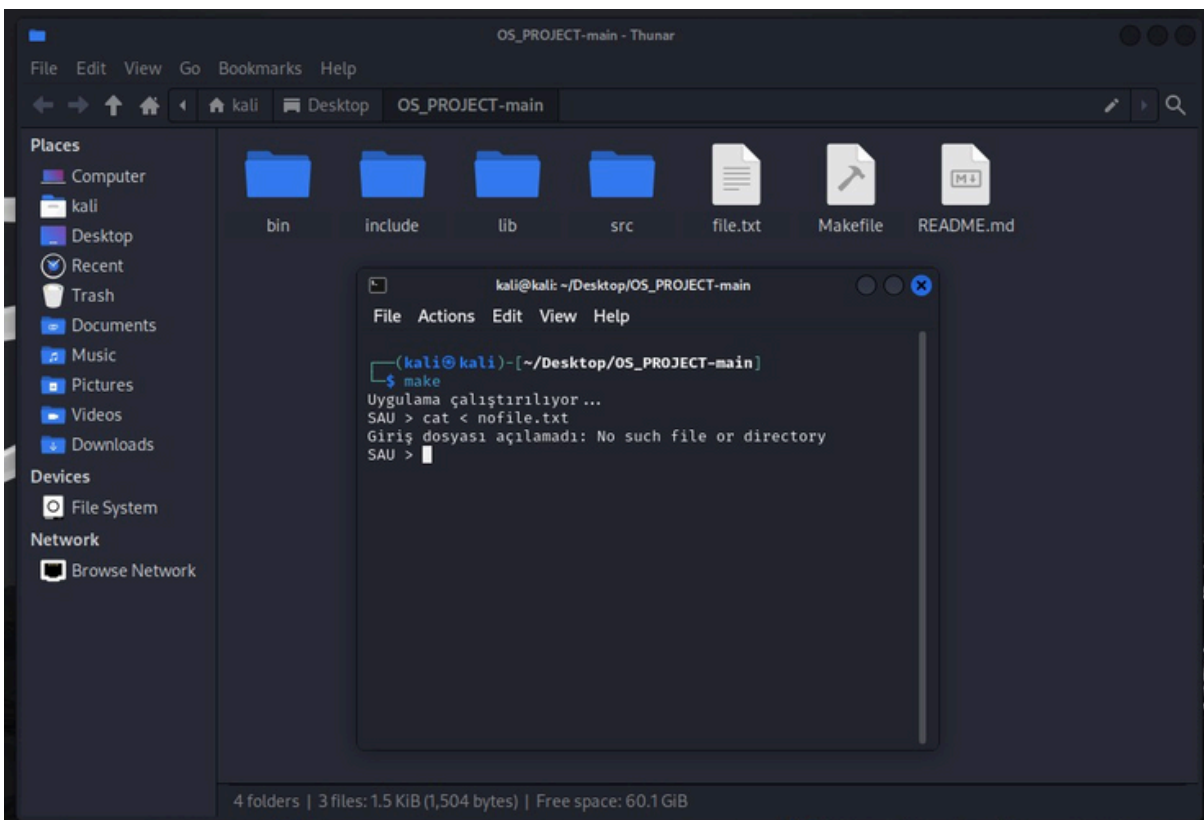


- Dosyayı giriş olarak kullanarak bir komut çalıştıralım:
- SAU > cat < file.txt



## Mevcut Olmayan Bir Dosya ile Test:

- Var olmayan bir dosya ile giriş yönlendirme deneyelim:
- SAU > cat < nofile.txt





# 5.Çıkış Yönlendirme

## Çıkış Yönlendirme Mantığı ve Kodun Detaylı Açıklaması

Komutun "komut > çıkışDosyası" şeklinde verilmesi durumunda, program alt prosesin standart çıktısını belirtilen dosyaya yönlendirir. Linux sistem çağrılarından dup2 bu işlemin temelini oluşturur.

### 1. Komutun Tespiti

Kullanıcı tarafından verilen komutun içinde ">" sembolü olup olmadığı kontrol edilir. Bu sembol bulunduğunda, belirtilen dosya yazma modunda açılır ve standart çıktı (STDOUT) bu dosyaya yönlendirilir.

#### Kod Detayları:

```
else if (strcmp(args[i], ">") == 0)
{
    // Çıkış yönlendirme
    int fd = open(args[i + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0)
    {
        perror("Çıkış dosyası açılmadı");
        exit(EXIT_FAILURE);
    }
    dup2(fd, STDOUT_FILENO);
    close(fd);
    args[i] = NULL;
}
```

#### Açıklama:

- **open Fonksiyonu:** Dosya yazma modunda açılır. O\_WRONLY, dosyanın yalnızca yazma işlemine izin verir; O\_CREAT dosya yoksa oluşturur; O\_TRUNC mevcut dosyanın içeriğini temizler.
- **dup2 Sistem Çağrısı:** Standart çıktı (STDOUT\_FILENO) belirtilen dosya tanımlayıcısına yönlendirilir.
- **close Fonksiyonu:** Kullanılan dosya tanımlayıcısı kapatılır.

### 2. Dosya Açma Hataları

Dosyanın açılmaması durumunda, program bir hata mesajı yazdırır ve uygun bir çıkış kodu ile sonlanır.

#### Kod Detayları:

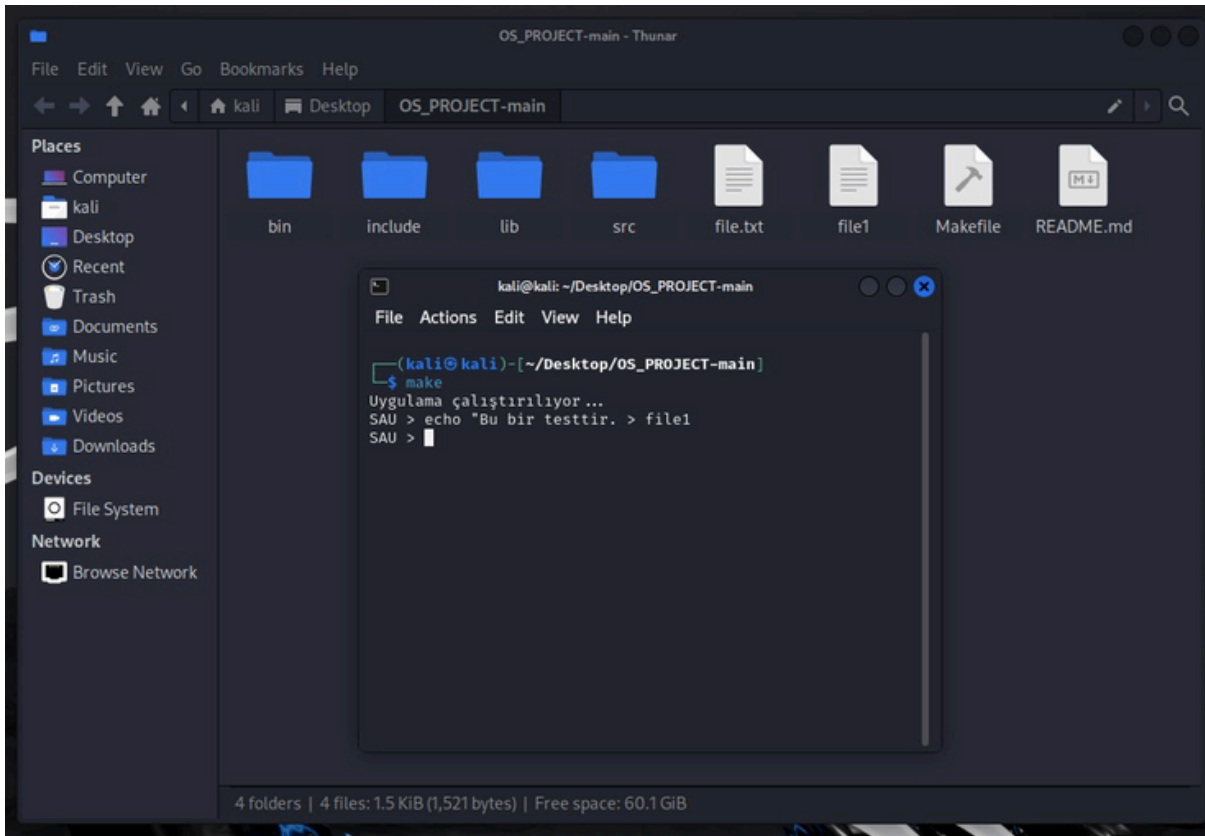
```
if (fd < 0)
{
    perror("Çıkış dosyası açılmadı");
    exit(EXIT_FAILURE);
}
```

#### Açıklama:

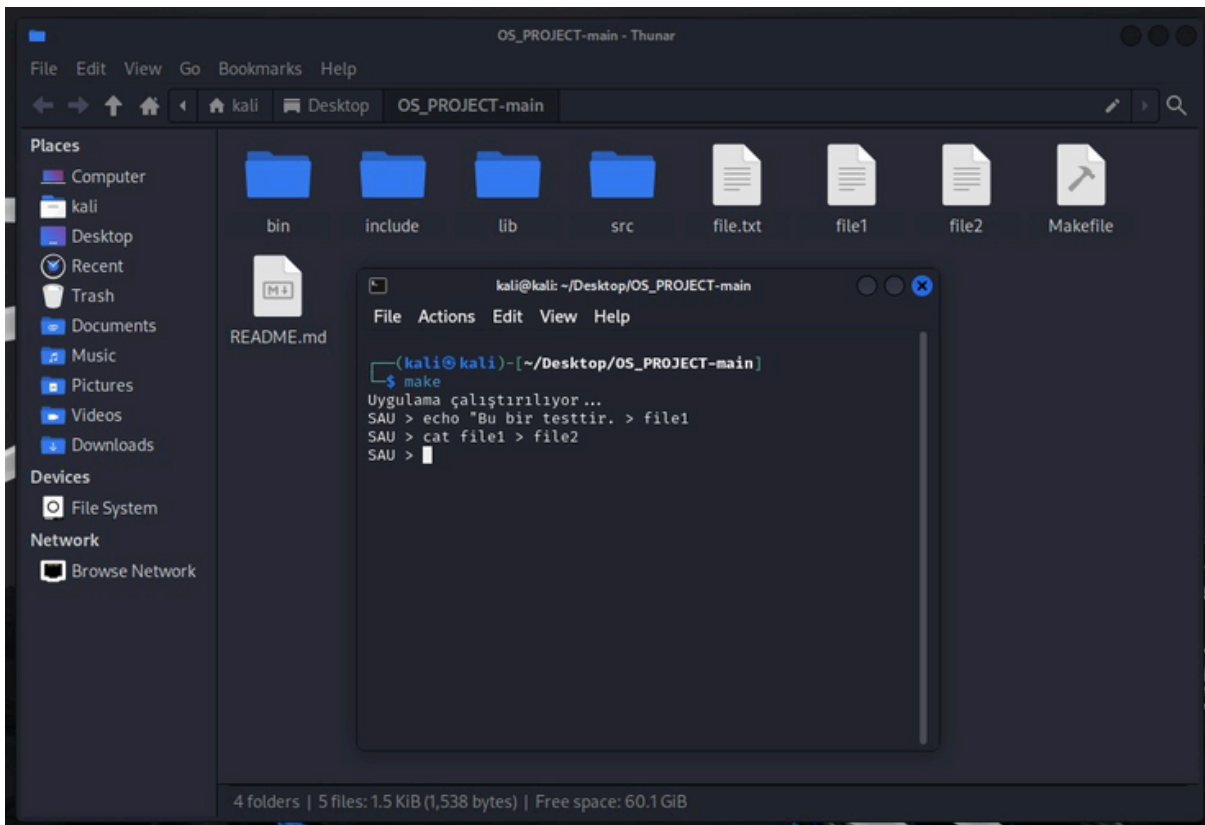
- Dosya açma işleminde hata oluşur ve dosya tanımlayıcısı negatif bir değer dönerse, perror ile hata mesajı yazdırılır.
- Program EXIT\_FAILURE kodu ile sonlandırılır.

## Dosya İçeriğini Başka Bir Dosyaya Yönlendirme:

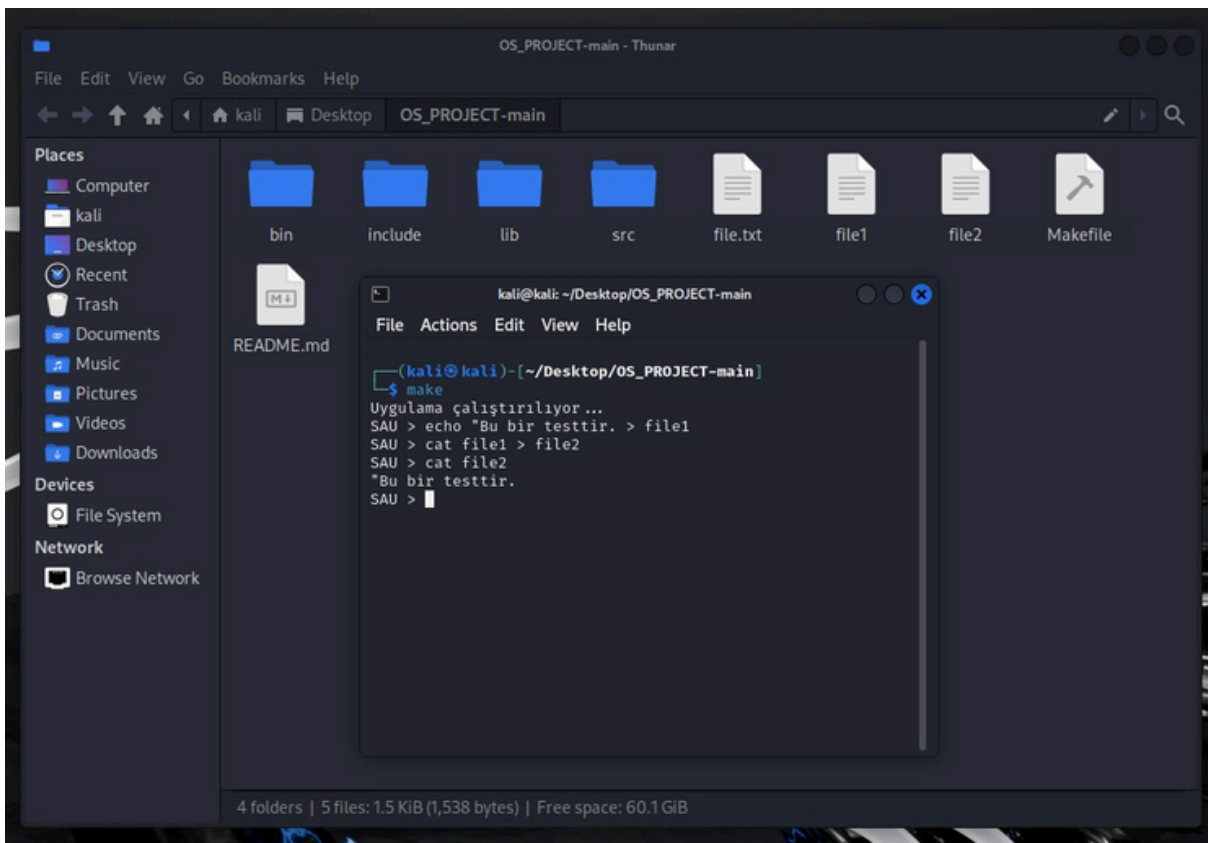
- İlk olarak bir dosya oluşturalım ve içine veri yazalım:
- SAU > echo "Bu bir testtir" > file1



- Daha sonra bu dosyanın içeriğini başka bir dosyaya (file2) yönlendirelim:
- SAU > cat file1 > file2

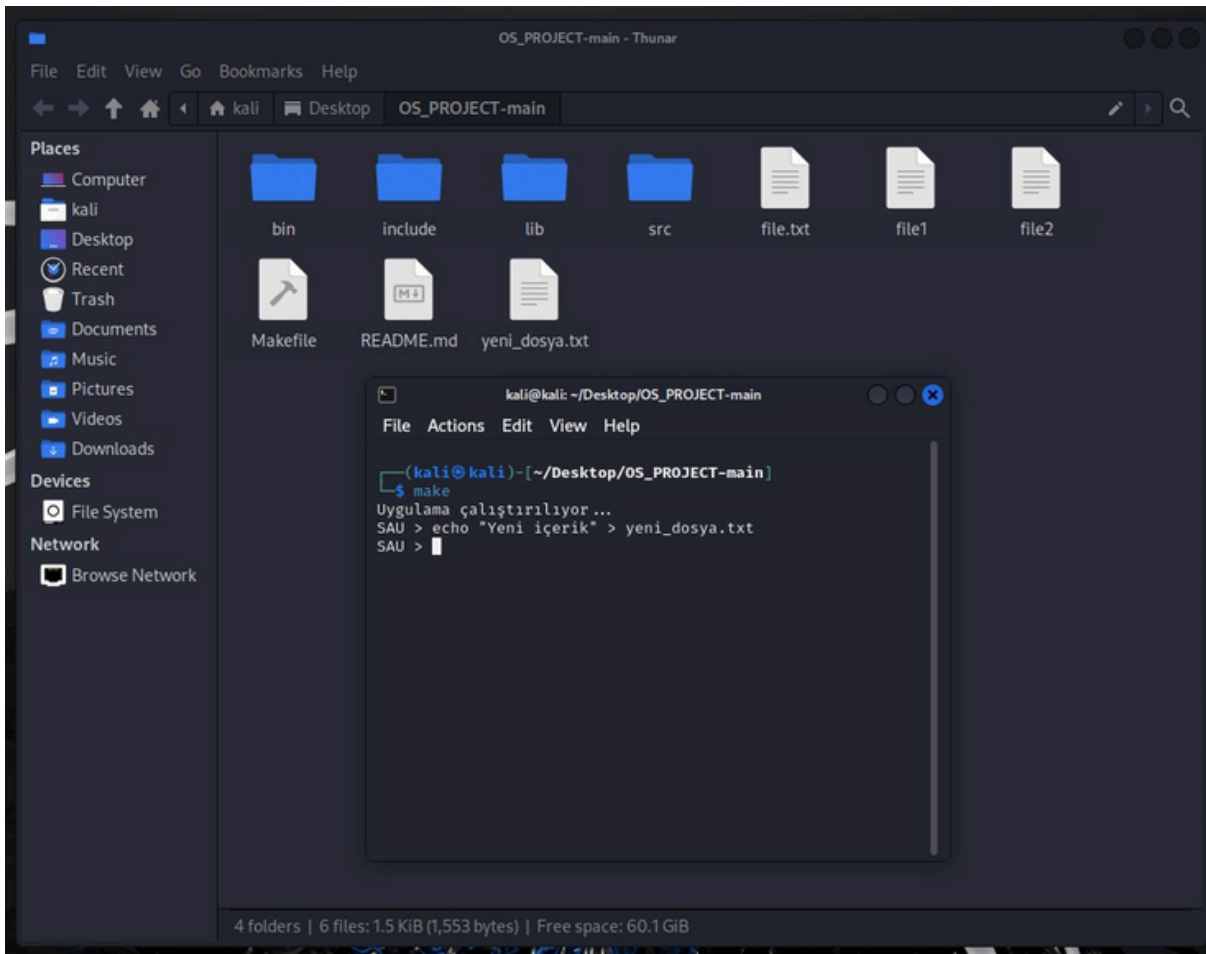


- Son olarak file2 dosyasını kontrol edelim:
- SAU > cat file2

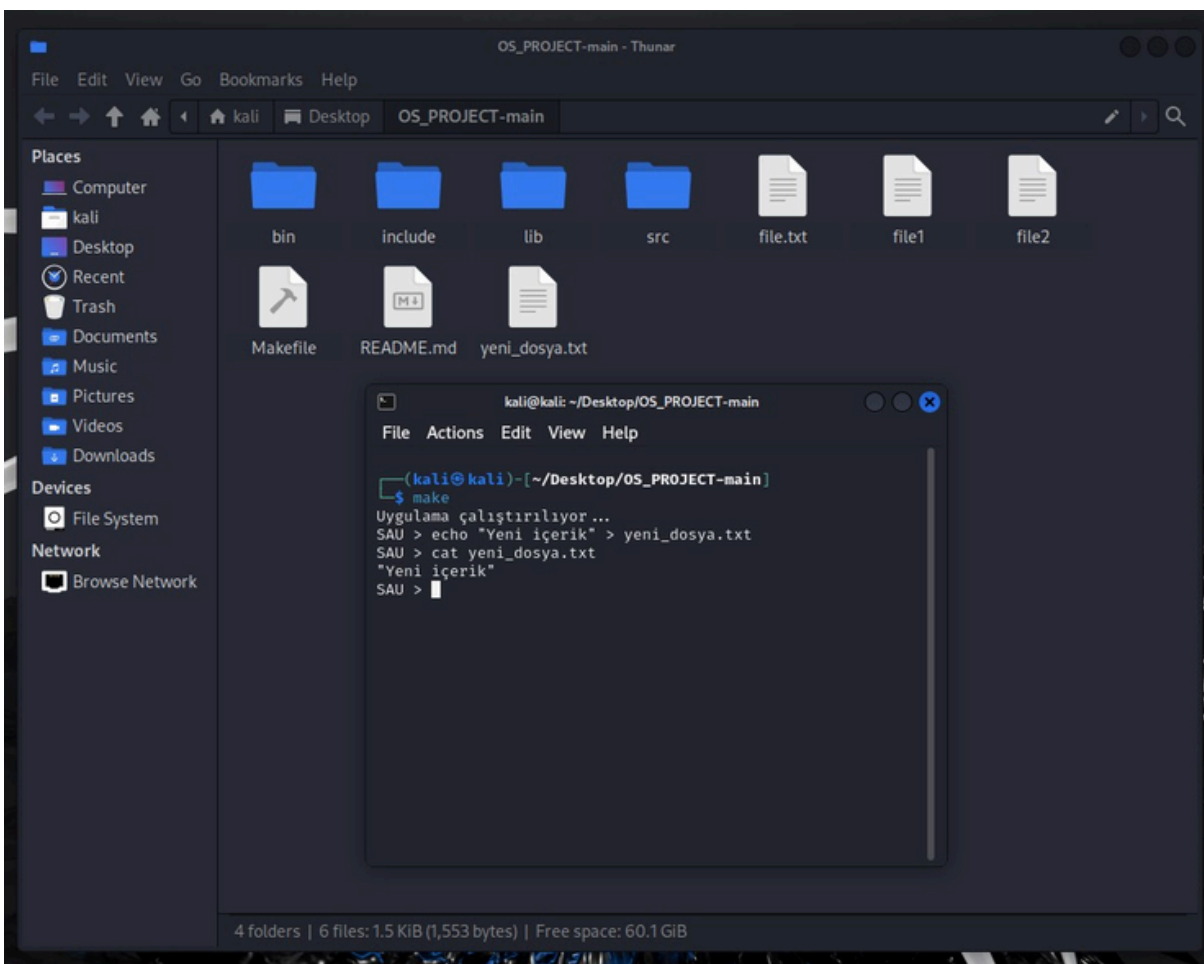


## Yeni Bir Dosya Oluşturma ve Yönlendirme:

- Çıkışı yeni bir dosyaya yönlendirelim:
- SAU > echo "Yeni içerik" > yeni\_dosya.txt

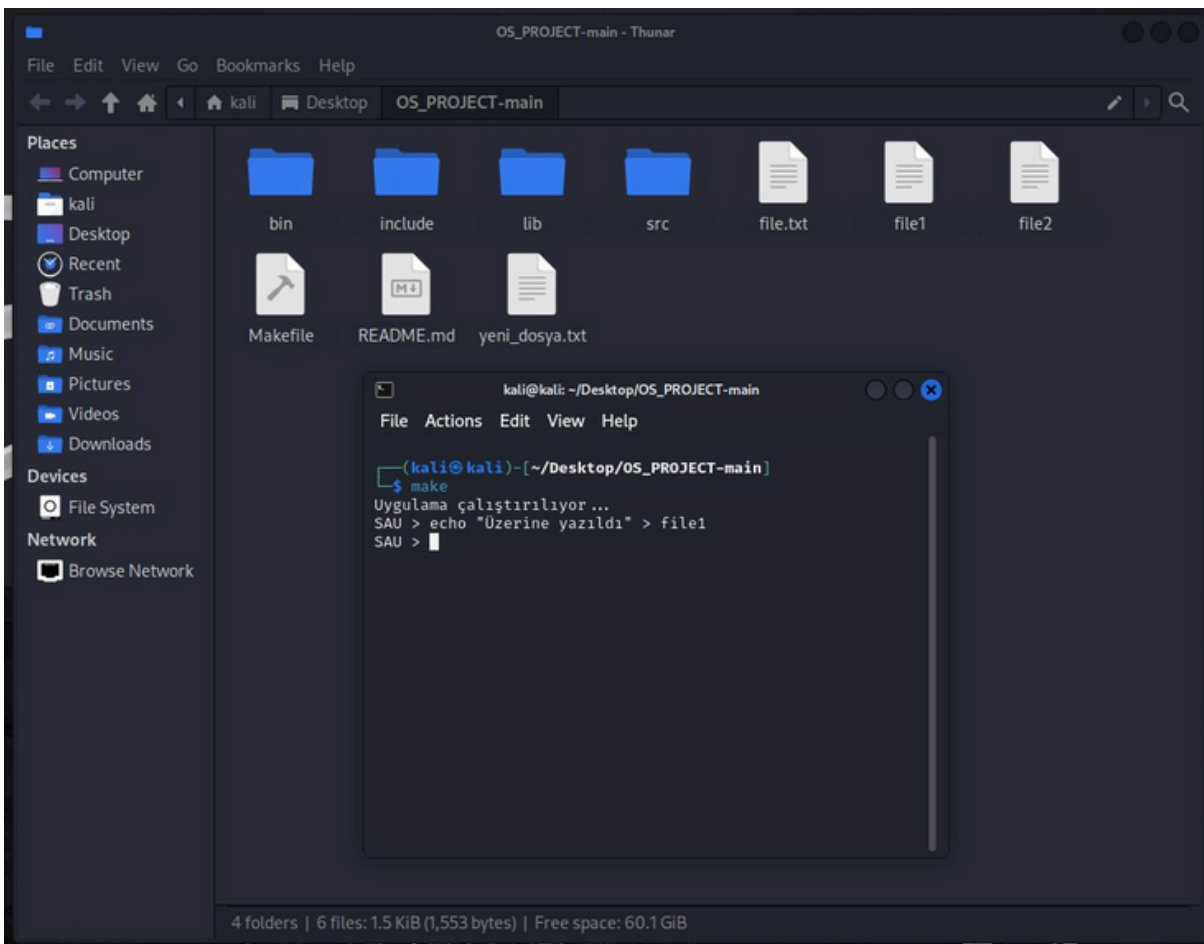


- Sonrasında dosyanın içeriğini kontrol edelim:
- SAU > cat yeni\_dosya.txt

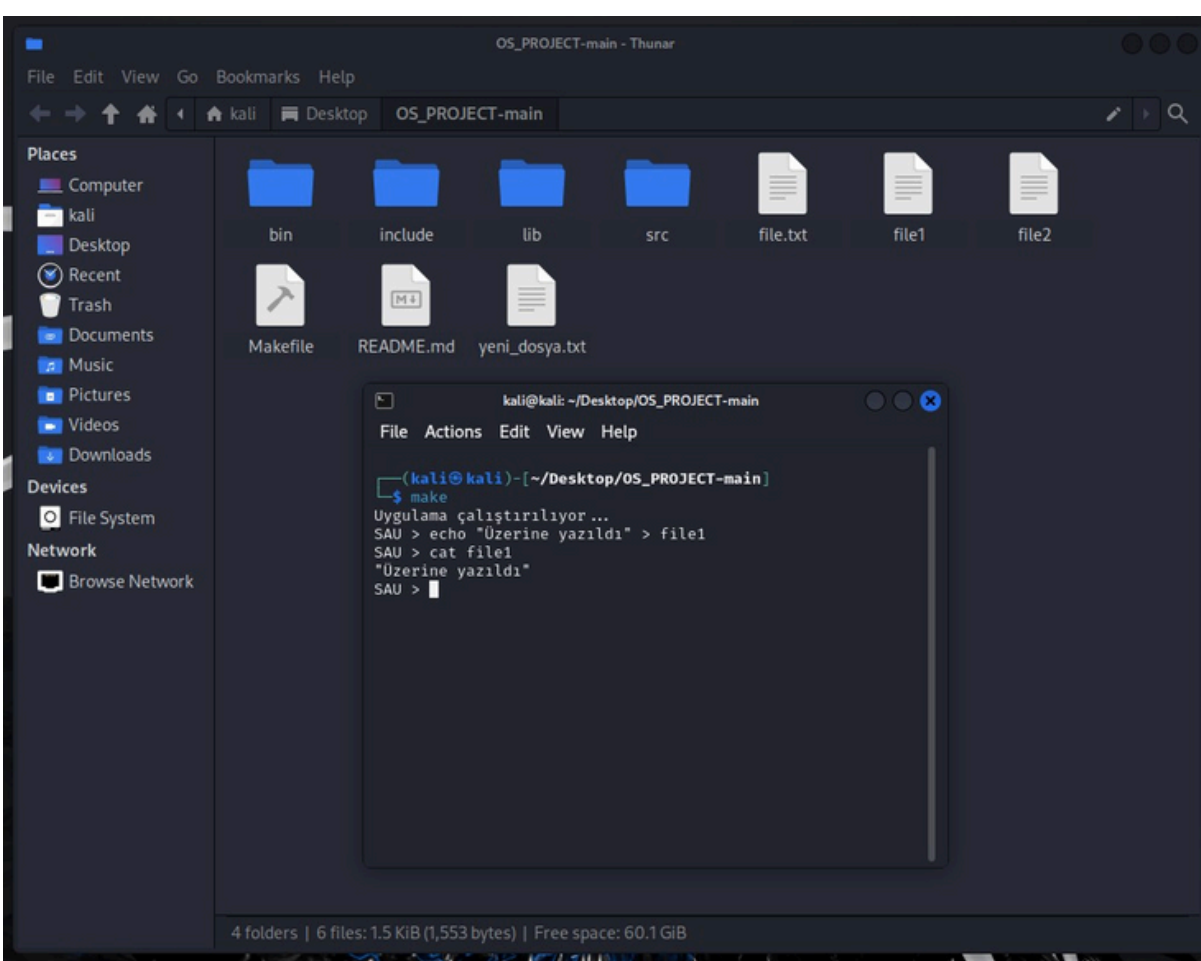


## Mevcut Bir Dosyanın Üzerine Yazma:

- Mevcut bir dosyanın içeriğini değiştirelim:
- SAU > echo "Üzerine yazıldı" > file1

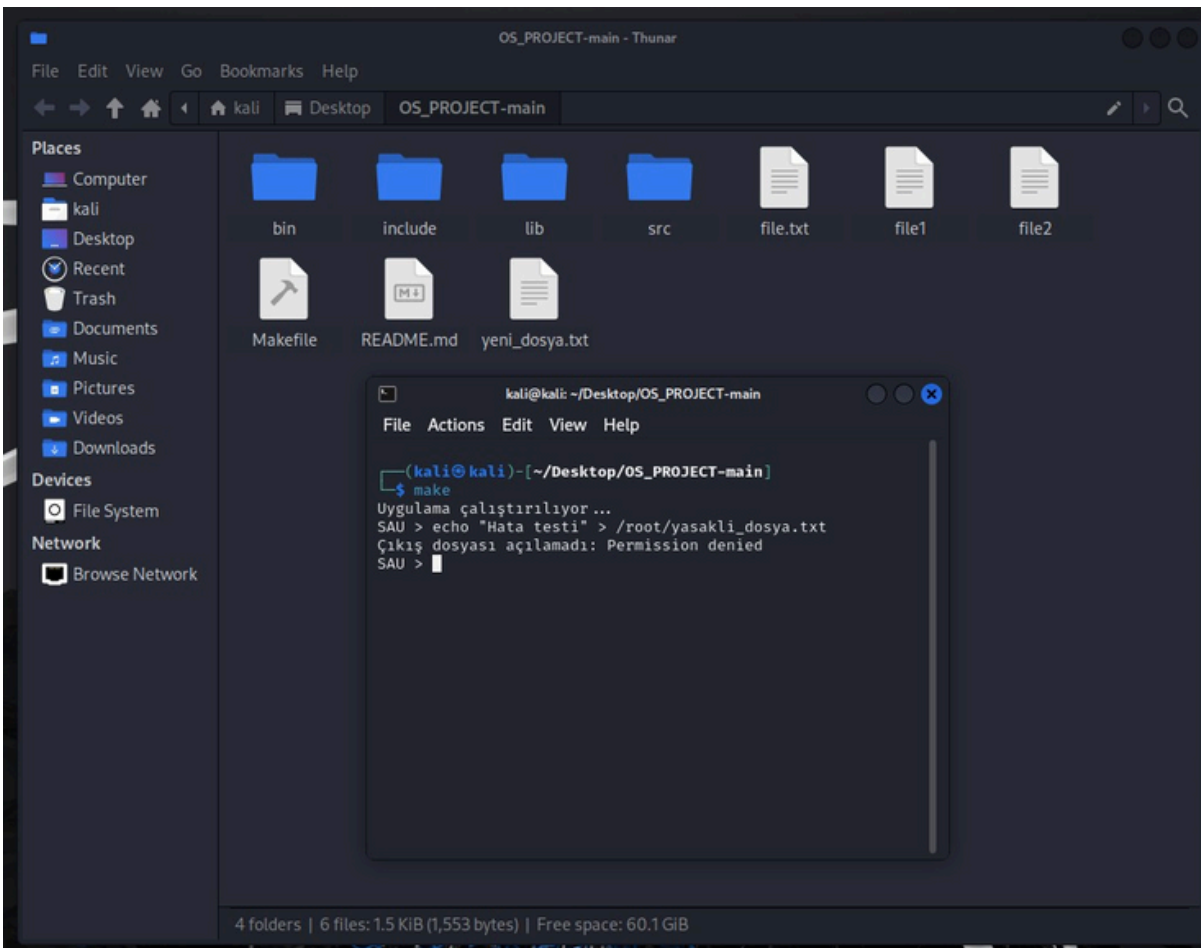


- Son olarak file1 dosyasını kontrol edelim:
- SAU > cat file1



## Hatalı Dosya Yolu ile Test:

- Yazma izni olmayan bir yere çıkış yönlendirmeyi deneyelim:
- SAU > echo "Hata testi" > /root/yasakli\_dosya.txt



# 6.Arka Plan Çalışma

## Arka Plan Çalışma Mantığı ve Kodun Detaylı Açıklaması

Komutun sonuna '&' eklenmesi durumunda, bu komut arka planda çalıştırılır ve kabuk yeni komutlar almayı sürdürür. Çalışan komutun PID'si (Process ID) ve dönüş değeri kabuk tarafından raporlanır.

### 1. Komutun Arka Plan Durumunun Tespiti

Komutlar "&" sembolü ile kontrol edilerek arka planda çalışacak bir komut olup olmadığı belirlenir. Bu işlem sırasında, "&" sembolü komut argümanlarından kaldırılır ve komutun geri kalanı yürütülmek üzere hazırlanır.

#### Kod Detayları:

```
// Komutları ayırıştır
args[i] = strtok(command, " ");
while (args[i] != NULL && i < MAX_ARGS - 1)
{
    if (strcmp(args[i], "&") == 0)
    {
        background = 1;
        args[i] = NULL;
        break;
    }
    i++;
    args[i] = strtok(NULL, " ");
}
```

Bu kod, komutun sonunda "&" olup olmadığını kontrol eder. Bulunursa, background değişkeni 1 olarak ayarlanır ve "&" argümanlarından temizlenir.

### 2. Arka Planda Çalışacak Komutun Oluşturulması

fork sistem çağrısı kullanılarak yeni bir alt süreç oluşturulur ve bu süreç arka planda yürütülmek üzere ayrıştırılır.



## Kod Detayları:

```
// Alt süreç oluştur ve çalıştır
pid_t pid = fork();
if (pid == 0)
{
    // Çocuk süreç
    handle_redirection(args);
    if (execvp(args[0], args) == -1)
    {
        perror("Komut çalıştırılamadı");
    }
    exit(EXIT_FAILURE);
}
else if (pid > 0)
{
    // Ana süreç
    if (!background)
    {
        int status;
        waitpid(pid, &status, 0);
    }
    else
    {
        printf("[PID: %d] Arka planda çalışıyor\n", pid);
    }
}
else
{
    perror("Fork başarısız oldu");
}
```

## Açıklama:

- ❑ **Alt Süreç (Çocuk Proses):** Yeni bir süreç oluşturulur ve bu süreç girilen komutu yürütür.
- ❑ **Ana Süreç:** Eğer background değişkeni 1 ise, kabuk yeni komutlar almayı sürdürür ve arka plandaki komutun PID'sini raporlar. Eğer background değişkeni 0 ise, kabuk alt sürecin tamamlanmasını bekler.

## 3. PID ve Dönüş Durumu Bilgilendirmesi

Arka planda çalışan bir komutun tamamlandığında kabuk, komutun PID'sini ve dönüş durumunu (exit code) bildirir. Bu bilgi, kullanıcının arka plandaki komutun durumu hakkında bilgi sahibi olmasını sağlar.

### Kod Detayları:

```
{  
    printf("[PID: %d] Arka planda çalışıyor\n", pid);  
}
```

**Açıklama:** Bu mesaj, arka planda çalışan komutun PID'sini ekrana yazdırır. Arka plandaki bir komut tamamlandıktan sonra bu bilgi ile takip sağlanabilir.

## 7) Boru (Pipe)

**Pipe'ların Mantığı ve Uygulama** Birden fazla komutun boru (|) operatörü kullanılarak

birbirine bağlanması durumunda,

bir komutun çıktısı bir sonraki komutun girdisi olarak kullanılır. Bu işlem, pipe ve dup2 sistem çağrıları ile sağlanır. **1. Komutları Ayırıştırma** Borular ile ayrılan komutlar

strtok ile ayrıştırılır ve her komut bir diziye kaydedilir. **Kod Detayları:**

```
// Komutları pipe ile ayır  
commands[i] = strtok(command, "|");  
while (commands[i] != NULL && i < MAX_ARGS - 1)  
{  
    i++;  
    commands[i] = strtok(NULL, "|");  
}
```

Bu kod, "|" operatörü ile ayrılan her komutu commands dizisine kaydeder.

### 2. Pipe Oluşturma

Komutlar arasında veri akışı sağlamak için gerekli sayıda pipe oluşturulur. Her iki komut arasında bir pipe tanımlanır.

### Kod Detayları:

```
// Tüm pipe'ları oluştur  
for (int j = 0; j < num_commands - 1; j++)  
{  
    if (pipe(pipefds + 2 * j) < 0)  
    {  
        perror("Pipe oluşturulamadı");  
        exit(EXIT_FAILURE);  
    }  
}
```

Bu kod, gerekli sayıda pipe oluşturur. pipefds dizisi, her bir pipe'ın dosya tanımlayıcılarını saklar.



### 3. Alt Süreçlerin Çalıştırılması

Her komut için bir alt süreç oluşturulur.

- İlk komut için sadece çıktı pipe'a yönlendirilir.
- Orta komutlar için hem girdiler hem de çıktılar pipe ile yönlendirilir.
- Son komut için sadece girdiler pipe'dan alınır.

#### Kod Detayları:

```
// İlk komut
if (j != 0)
{
    dup2(pipefds[2 * (j - 1)], STDIN_FILENO);
}
// Son komut değilse
if (j != num_commands - 1)
{
    dup2(pipefds[2 * j + 1], STDOUT_FILENO);
}
```

dup2, girdiyi veya çıktıyı ilgili pipe tanımlayıcısına yönlendirir.

### 4. Tüm Pipe'ların Kapatılması

Alt süreç çalıştıktan sonra tüm pipe'lar kapatılır.

#### Kod Detayları:

```
// Tüm pipe'ları kapat
for (int k = 0; k < 2 * (num_commands - 1); k++)
{
    close(pipefds[k]);
}
```

### 5. Alt Süreçlerin Beklenmesi

Ana süreç, tüm alt süreçlerin tamamlanmasını bekler.

#### Kod Detayları:

```
// Tüm çocuk süreçlerin bitmesini bekle
for (int j = 0; j < num_commands; j++)
{
    wait(NULL);
}
```

Bu kod, tüm alt süreçlerin bitmesini bekler.

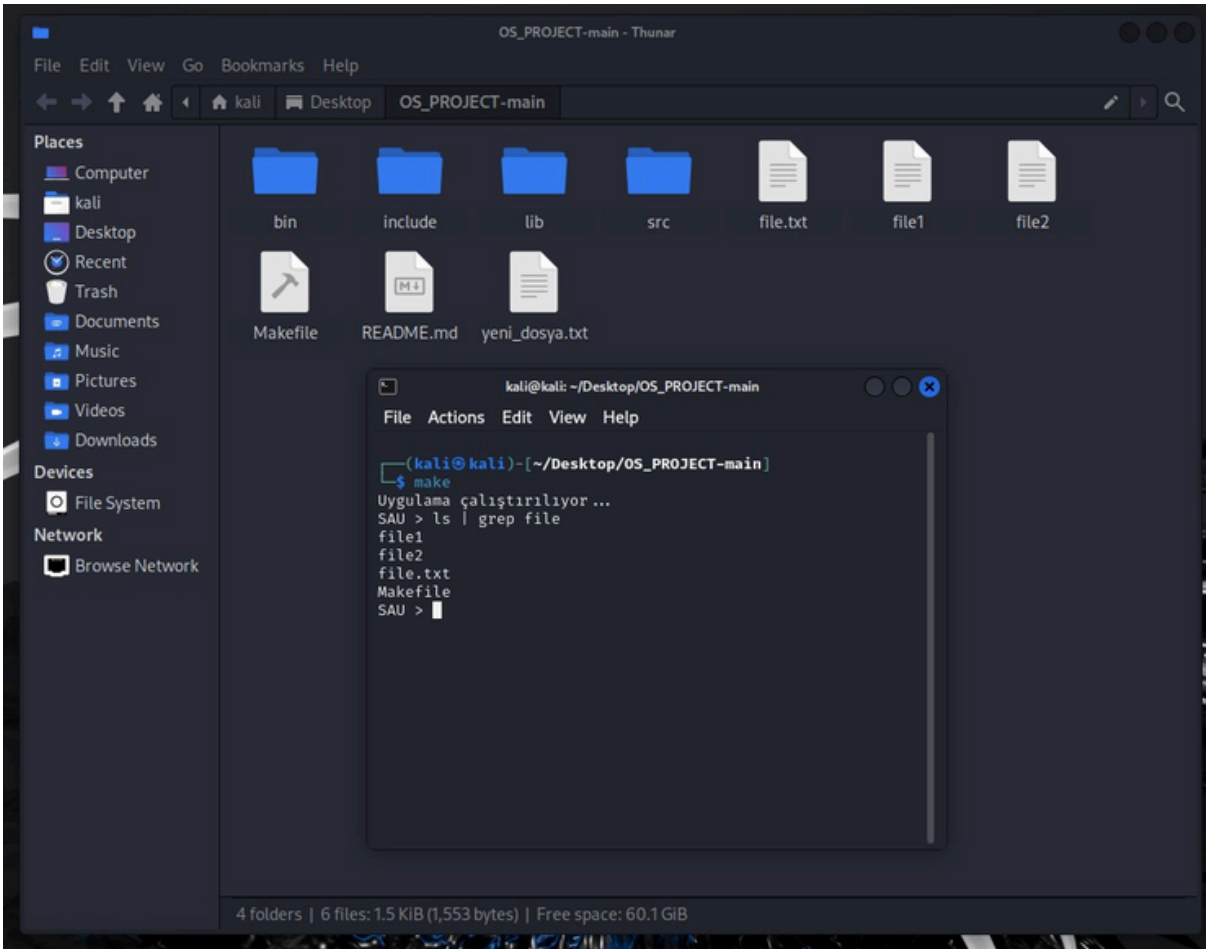
### 6. Hata Durumlarının Ele Alınması

Pipe oluşturulamadığında veya fork başarısız olduğunda uygun hata mesajları yazdırılır ve program sonlandırılır.

# Test Komutları ve Beklenen Sonuçlar

## Basit Boru:

SAU > ls | grep file



## İki Boru Kullanımı:

SAU > ls | grep file | wc -l

