Çoklu Kullanıcı İletişimli Kabuk Uygulaması Proje Raporu

İçindekiler

Proje Tanımı

Bu projede, çoklu kabuk örneklerinin birbirleriyle iletişim kurabildiği terminal benzeri bir uygulama geliştirdik. Her kabuk örneği (GTK arayüzü ile uygulanmış) standart bir kabuk gibi işlev görürken (`ls`, `grep` vb. komutları ayrıştırma), aynı zamanda kullanıcıların paylaşılan bir tampon üzerinden mesaj gönderip almasına olanak tanır. Proje, sistem programlama kavramlarını (süreçler, IPC, senkronizasyon) GUI geliştirme (GTK) ve modüler tasarım (MVC mimarisi) ile birleştirmeyi amaçlamaktadır.

Mimari Yaklaşım

MVC Yapısı

Projeyi, kodun bakımını ve genişletilebilirliğini kolaylaştırmayı amaçlayan Model-View-Controller (MVC) mimarisine göre tasarladık:

1. **Model (model.c)**:
   - Veri ve arka uç mantığını yönetir
   - Kabuk süreç yönetimi (`fork/exec`)
   - Kullanıcı oturum verilerini izleme
   - Paylaşılan mesaj tamponu yönetimi
   - Komut yürütme ve yönlendirme
2. **View (view.c)**:
   - GUI ve kullanıcı etkileşimlerini yönetir
   - Terminal emülatör widget'ları
   - Paylaşılan mesaj paneli
   - Sekme tabanlı kabuk arayüzü
3. **Controller (controller.c)**:
   - Kullanıcı girişini işler ve komutları model katmanına yönlendirir
   - Standart kabuk komutları ve mesaj komutları arasında ayrım yapar
   - Çıktıyı görünüm katmanına iletir
   - Mesaj güncellemelerini periyodik olarak kontrol eder

Sistem Tasarımı

Sistem, aşağıdaki ana bileşenlerden oluşur:

- **Çoklu Kabuk Ortamı**: GTK tabanlı bir arayüz üzerinden `n` adet etkileşimli kabuk örneği başlatılır
- **Komut Yürütme**: Her kabuk, `fork()`, `execvp()` ve yönlendirme kullanarak standart kabuk komutlarını yürütür
- **İletişim Tamponu**: Kabuklar arası iletişim için paylaşılan bellek (`shm_open`) kullanılır
- **Senkronizasyon**: Paylaşılan belleğe erişimi yönetmek için semaforlar (`sem_wait`, `sem_post`) kullanılır

Tasarım Seçimleri ve Gerekçeleri

GTK Arayüz Tasarımı

GTK kütüphanesini arayüz geliştirme için seçtik çünkü:

1. **Platformlar Arası Uyumluluk**: GTK, Linux, macOS ve Windows gibi farklı platformlarda çalışabilir
2. **C ile Entegrasyon**: C dilinde yazıldığından, projemizin geri kalanı ile sorunsuz entegrasyon sağlar
3. **Zengin Widget Seti**: Terminal emülasyonu için gerekli olan `GtkTextView`, `GtkEntry` gibi widget'lar sunar
4. **Olay Güdümlü Programlama**: GTK'nın sinyal mekanizması, kullanıcı etkileşimlerini işlemek için uygun bir yapı sağlar

c

```c
// Ana pencere oluşturma örneği
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "GTK Shell");
gtk_window_set_default_size(GTK_WINDOW(window), 800, 600);
```

Sekme Yapısı

Birden fazla kabuk örneğini yönetmek için sekme tabanlı bir arayüz tasarladık:

1. **Ölçeklenebilirlik**: Kullanıcılar ihtiyaç duydukları kadar kabuk örneği açabilir
2. **Kaynak Verimliliği**: Her kabuk için ayrı pencere açmak yerine, tek pencere içinde sekmeleri kullanarak ekran alanından tasarruf sağlar
3. **Kolay Erişim**: Sekmeleri kullanarak kabuklar arasında hızlı geçiş yapılabilir
4. **Dinamik Oluşturma**: "+" düğmesi ile istenen sayıda kabuk oluşturulabilir

c

```c
// Sekme ekleme fonksiyonundan bir kesit
notebook = gtk_notebook_new();
gtk_notebook_set_scrollable(GTK_NOTEBOOK(notebook), TRUE);
GtkWidget *add_button = gtk_button_new_with_label("+");
gtk_notebook_set_action_widget(GTK_NOTEBOOK(notebook), add_button, GTK_PACK_END);
```

Paylaşılan Mesaj Tamponu

Kabuklar arası iletişim için POSIX paylaşılan bellek mekanizmasını (`shm_open`) seçtik:

1. **Performans**: Paylaşılan bellek, süreçler arası iletişim için en hızlı mekanizmalardan biridir
2. **Kullanım Kolaylığı**: Belleği normal bir dizi gibi kullanabilmemiz, mesaj işlemeyi basitleştirir
3. **Doğrudan Erişim**: Her süreç, aynı bellek bölgesine doğrudan erişebilir
4. **Senkronizasyon Desteği**: Semaforlarla kombinasyon halinde güvenli erişim sağlar

c

```c
// Paylaşılan bellek başlatma
int fd = shm_open(SHARED_FILE_NAME, O_CREAT | O_RDWR, 0600);
shmp = (ShmBuf *)mmap(NULL, BUF_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Kabuk İşlevselliği

Kabuk işlevselliğini gerçekleştirmek için `fork()` ve `execvp()` tabanlı bir yaklaşım seçtik:

1. **UNIX Uyumluluğu**: Standart UNIX araçlarıyla tam uyumluluk sağlar
2. **Esneklik**: Yönlendirme (`<`, `>`, `|`) ve arkaplan çalıştırma (`&`) gibi özellikler eklenebilir
3. **İzolasyon**: Her komut kendi sürecinde çalışır, bu da güvenlik ve kararlılık sağlar
4. **Denetim**: Ana süreç, çocuk süreçlerin durumunu izleyebilir ve yönetebilir

c

```c
// Komut yürütme yapısı
pid_t pid = fork();
if (pid == 0) {
    // Çocuk süreç
    execvp(args[0], args);
    exit(1);
} else if (pid > 0) {
    // Ana süreç
    if (background) {
        add_job(pid, command);
    } else {
        waitpid(pid, NULL, 0);
    }
}
```

Karşılaşılan Zorluklar ve Çözümleri

Süreç Yönetimi Zorlukları

Sorun: Çoklu süreçlerin oluşturulması, izlenmesi ve sonlandırılması karmaşık bir görevdi.

Çözüm:

1. Süreç yönetimi için yapılandırılmış bir yaklaşım geliştirdik:

```c
typedef struct {
    pid_t pid;
    char command[256];
    int is_running;
} struct job;
```

2. İş yönetimi fonksiyonlarını uyguladık:

```c
void add_job(pid_t pid, const char *command);
void remove_job(int index);
void list_jobs();
```

3. Zombi süreçleri önlemek için `waitpid()` kullanarak süreç durumlarını düzenli olarak kontrol ettik.

Paylaşılan Bellek Senkronizasyonu

Sorun: Birden fazla süreç aynı anda paylaşılan belleğe eriştiğinde veri bozulması riski oluşuyordu.

Çözüm:

1. POSIX semaforlarını kullanarak paylaşılan belleğe erişimi senkronize ettik:

```c
sem_wait(&shmp->sem);
// Paylaşılan bellek üzerinde işlemler
sem_post(&shmp->sem);
```

2. Semaforları paylaşılan bellek yapısının bir parçası olarak tanımladık, böylece tüm süreçler aynı senkronizasyon mekanizmasını kullanabildi.
3. Kritik bölümleri minimize ederek performans kaybını önledik.

GTK Arayüz Entegrasyonu

Sorun: Çoklu süreçler ve GTK'nın olay döngüsü arasındaki entegrasyon karmaşıktı.

Çözüm:

1. Çocuk süreçlerin çıktısını izlemek için GIOChannel kullandık:

```c
tab->stdout_channel = g_io_channel_unix_new(tab->stdout_pipe[0]);
g_io_add_watch(tab->stdout_channel, G_IO_IN | G_IO_HUP, read_terminal_output, tab);
```

2. Mesaj güncellemelerini periyodik olarak kontrol etmek için zamanlayıcı kullandık:

```c
g_timeout_add(500, update_shared_messages, NULL);
```

3. Veri yapılarını GTK widget'larına bağlamak için `g_object_set_data()` kullandık, böylece sekme değişikliklerinde doğru verilere erişebildik.

Komut Yürütme ve Yönlendirme

Sorun: Komut yönlendirme (`>`, `<`, `|`) ve arkaplan çalıştırma (`&`) gibi özelliklerin uygulanması karmaşıktı.

Çözüm:

1. Komutları ayrıştırmak için özel bir fonksiyon geliştirdik:

```c
void parse_command(char *command, char *args[]);
```

2. Yönlendirmeleri işlemek için dosya tanımlayıcıları yönetimi:

```c
int handle_redirections(char *args[], int *input_fd, int *output_fd);
```

3. Pipe'lar için özel işleme:

```c
int pipefd[2];
pipe(pipefd);
// İlk komut için çıktıyı pipe'a yönlendir
// İkinci komut için girdiyi pipe'dan al
```

Ek Özellikler

Çoklu Sekme Desteği

Kullanıcıların birden fazla kabuk örneğini yönetebilmesi için dinamik sekme oluşturma ve yönetim sistemi ekledik:

- Her sekme bağımsız bir kabuk sürecine sahiptir
- "+" düğmesi ile istenen sayıda yeni sekme oluşturulabilir
- Her sekme kendi komut geçmişini ve mesaj tamponunu tutar
- Sekmeler arasında geçiş yaparken bağlam otomatik olarak güncellenir

Zaman Damgalı Mesajlar

Mesajların gönderilme zamanını takip etmek için her mesaja zaman damgası ekledik:

c

```c
time_t now = time(NULL);
char timestamp[20];
strftime(timestamp, sizeof(timestamp), "%H:%M:%S", localtime(&now));
snprintf(formatted_msg, BUF_SIZE, "[%s] %d.sekme: %s", timestamp, current_tab, msg);
```

Bu özellik, birden fazla kabuk arasındaki iletişimi takip etmeyi kolaylaştırır ve mesajların gönderilme sırasını net bir şekilde gösterir.

İş Yönetimi

UNIX benzeri kabukların temel özelliklerinden biri olan iş yönetimini ekledik:

- `jobs` komutu ile çalışan işlerin listesini görüntüleme
- `bg <job_id>` komutu ile durdurulmuş işleri arka planda çalıştırma
- `fg <job_id>` komutu ile arka plandaki işleri ön plana getirme
- `&` işareti ile komutları arka planda çalıştırma

Bu özellik, profesyonel bir kabuk deneyimi sunmamızı sağlar ve kullanıcı verimliliğini artırır.

Sonuç ve Değerlendirme

Bu projede, sistem programlama, süreç yönetimi, paylaşılan bellek kullanımı ve GUI geliştirme gibi çeşitli alanları birleştiren kapsamlı bir uygulama geliştirdik. MVC mimarisini başarıyla uygulayarak, kodun bakımını ve genişletilebilirliğini kolaylaştırdık.

En önemli kazanımlarımız:

1. **Süreç Yönetimi**: Çoklu süreçlerin oluşturulması, izlenmesi ve iletişimi konusunda derin bilgi edindik
2. **Paylaşılan Bellek**: IPC mekanizmalarını ve senkronizasyon tekniklerini etkili bir şekilde kullanmayı öğrendik
3. **GUI Programlama**: GTK kullanarak kullanıcı dostu bir arayüz geliştirdik
4. **MVC Mimarisi**: Büyük bir projeyi modüler ve sürdürülebilir bileşenlere ayırmayı başardık

Projenin daha da geliştirilmesi için potansiyel alanlar:

- Kullanıcı kimliklerini ekleyerek kişiselleştirilmiş mesajlaşma
- Özel mesajlaşma özelliği
- Dosya paylaşım desteği
- Komut geçmişi ve arama özelliği

Bu proje, sistem programlama kavramlarını pratik bir uygulama ile pekiştirmemize olanak sağladı ve ekip üyelerimizin becerilerini geliştirme fırsatı sundu.