BURSA TEKNİK ÜNİVERSİTESİ

İŞLETİM SİSTEMLERİ DÖNEM ÖDEVİ RAPORU

Çok Katlı Apartman İnşaatı Üzerinden Process, Thread ve Senkronizasyon Kavramlarının Modellenmesi

Ders: İşletim Sistemleri
Dönem: 2024-2025 Bahar

Tarih: Mayıs 2025

Nazmi CİRİM – 21360859069

LATIF LATIF – 22360859306

1. GİRİŞ VE PROJE AMACI

Modern işletim sistemleri, bilgisayar kaynaklarını etkin bir şekilde yönetmek için çoklu işlem (multi-processing) ve çoklu iş parçacığı (multi-threading) kavramlarını kullanır. Bu kavramlar, aynı anda birden fazla işlemin veya iş parçacığının çalışmasına olanak tanıyarak sistem performansını artırır. Ancak bu paralel çalışma ortamında, kaynakların paylaşımı ve işlemler arası koordinasyon kritik önem taşır.

Bu proje kapsamında, 10 katlı ve her katta 4 daire bulunan bir apartmanın inşaat süreci üzerinden işletim sistemi kavramları modellenmiştir. Bu modelleme ile şu temel konular ele alınmıştır:

- **Process (Süreç) Kavramı:** Her kat bağımsız bir süreç olarak temsil edilmiştir
- **Thread (İş Parçacığı) Kavramı:** Her daire bir iş parçacığı olarak modellenmiştir
- **Senkronizasyon Mekanizmaları:** Süreçler ve iş parçacıkları arası koordinasyon • **Kaynak Paylaşımı:** Ortak kaynakların güvenli kullanımı
- **Yarış Koşulları (Race Conditions):** Eşzamanlı erişim problemleri ve çözümleri

Proje, C programlama dili kullanılarak gerçekleştirilmiş olup, POSIX thread kütüphanesi (pthread), semaphore yapıları ve paylaşılan bellek (shared memory) gibi sistem programlama araçları kullanılmıştır.

2. PROJE TASARIMI VE MİMARİ

2.1 Genel Sistem Mimarisi

Apartman inşaatı simülasyonu, hiyerarşik bir yapı üzerine kurulmuştur:

Ana Süreç (Main Process):

- Tüm sistemin koordinasyonunu sağlar
- Paylaşılan bellek alanını oluşturur ve yönetir
- Her kat için çocuk süreçler oluşturur
- Tüm inşaat sürecinin tamamlanmasını bekler

- **Kat Süreçleri (Floor Processes):**

- Her kat için bir tane olmak üzere toplam 10 süreç

- Kendi katındaki dairelerin inşaatını koordine eder
- Daire iş parçacıklarını oluşturur ve yönetir
- Alt katın tamamlanmasını bekler (sıralı inşaat kuralı)

Daire İş Parçacıkları (Apartment Threads):

- Her daire için bir tane olmak üzere kat başına 7 iş parçacığı
- Kendi dairesinin 8 aşamalı inşaat sürecini yürütür
- Ortak kaynakları paylaşır ve senkronizasyon kurallarına uyar

2.2 İnşaat Aşamaları Modeli

Her dairenin inşaat süreci 8 temel aşamadan oluşur:

1. **Kaba İnşaat (Rough Construction):** Yapının temel iskelet işleri
2. **Su Tesisatı (Plumbing):** Su ve kanalizasyon sistemlerinin kurulumu
3. **Elektrik Tesisatı (Electrical):** Elektrik kablolarının döşenmesi
4. **Yalıtım:** Yalıtım işlemleri (ses ve ısı)
5. **HVAC:** Havalandırma ve İklimlendirme sistemleri
6. **Döşeme:** Zemin döşeme (parke, fayans vb.)
7. **Sıva İşlemi (Plastering):** Duvarların sıvanması ve düzenlenmesi
8. **İç Tasarım (Interior Design):** Son rötuşlar ve dekorasyon işleri

Her aşama farklı ortak kaynaklar gerektirir ve farklı senkronizasyon kurallarına tabidir.

2.3 Ortak Kaynaklar

Sistemde yedi temel ortak kaynak tanımlanmıştır:

- **Vinç (Crane):** Ağır malzemelerin taşınması için kullanılır
- **Asansör (Elevator):** İnşaat malzemelerinin katlara taşınması
- **Su Kaynağı (Water Supply):** Su tesisatı işlemleri için merkezi kaynak
- **Yalıtım Ekipmanları:** Su, ses emici malzemeler ve sıcaklık izolasyonu için kullanılır.
- **HVAC Uzman Ekibi:** Havalandırma ve iklimlendirme işini yapan personeller.
- **Döşeme Ekipmanları:** Epoksi Zemin Kaplama, PVC Zemin Kaplama gibi malzemeler
- **Elektrik Kaynağı (Power Supply):** Elektrik tesisatı için ana güç kaynağı

Bu kaynaklar aynı anda sadece bir daire tarafından kullanılabilir ve mutex mekanizmaları ile korunur.

3. KULLANILAN İŞLETİM SİSTEMİ KAVRAMLARI

3.1 Process (Süreç) Kavramı ve Uygulaması

Teorik Temel: İşletim sistemlerinde process, çalışmakta olan bir programın örneğidir. Her process kendi bellek alanına, sistem kaynaklarına ve çalışma durumuna sahiptir. Process'ler birbirinden bağımsız çalışır ve kendi adres alanlarında işlem yaparlar.

Projede Uygulama:

```
// Her kat için process oluştur
pid_t child_pids[TOPLAM_KAT];

for (int floor = 1; floor <= TOPLAM_KAT; floor++) {
    child_pids[floor-1] = fork();

    if (child_pids[floor-1] < 0) {
        perror("fork failed");
        exit(1);
    } else if (child_pids[floor-1] == 0) {
        // Çocuk process - bu katın inşaatını yap
        construct_floor(floor, shared_mem);
        exit(0);
    }
}
```

Bu yaklaşımla:

- Her kat bağımsız bir süreç olarak çalışır
- Süreçler kendi yerel değişkenleri ve durumları ile çalışır
- Paylaşılan bellek ile gerekli koordinasyon sağlanır
- Bir sürecin çökmesi diğerlerini etkilemez

3.2 Thread (İş Parçacığı) Kavramı ve Uygulaması

Teorik Temel: Thread'ler, bir process içinde çalışan hafif yürütme birimleridir. Aynı process'in thread'leri bellek alanını paylaşır ve birbirleriyle daha hızlı iletişim kurabilir.

Projede Uygulama:

```
// Her daire için thread oluştur ve parametreleri hazırla
for (int apt = 1; apt <= DAIRE; apt++) {
    params[apt-1].kat_numara = kat_numara;
    params[apt-1].daire_numara = apt;
    params[apt-1].resource_mutexes = resource_mutexes;
    params[apt-1].floor_mutex = &floor_mutex;
    params[apt-1].floor_condition = &floor_condition;
    params[apt-1].adim_completion = adim_completion;

    // Thread'i başlat
    pthread_create(&apartment_threads[apt-1], NULL, insa_et, &params[apt-1]);
}
```

Bu model ile:

- Aynı kattaki daireler paralel olarak inşa edilebilir
- Bellek paylaşımı ile hızlı iletişim sağlanır
- Thread'ler arası senkronizasyon daha esnek olur

3.3 Senkronizasyon Mekanizmaları

3.3.1 Semaphore Kullanımı

Teorik Temel: Semaphore'lar, eşzamanlı sistemlerde kaynak erişimini kontrol etmek için kullanılan senkronizasyon primitifleridir. Belirli sayıda kaynağa erişimi düzenler.

Projede Uygulama:

```
// Katlar arası sıralı ilerleme için
```

```
    // Alt katın tamamlanmasını bekle (Kat 1 dışındaki tüm katlar için)
    if (kat_numara > 1) {
        printf("Kat %d, Kat %d'in tamamlanmasını bekliyor...\n", kat_numara, kat_numara - 1);
        sem_wait(&shared_mem->floor_semaphores[kat_numara - 1]);
        printf("Kat %d tamamlandı, Kat %d inşaatına geçiliyor.\n", kat_numara - 1, kat_numara);
    }
```

```
// İnşaat tamamlandıktan sonra
```

```
    // Bir sonraki katın başlayabileceğini bildir
    printf("Kat %d inşaatı tamamlandı. Kat %d inşaatına izin veriliyor.\n", kat_numara, kat_numara + 1);
    sem_post(&shared_mem->floor_semaphores[kat_numara]);
}
```

Bu mekanizma ile:

- Alt kat tamamlanmadan üst kat başlayamaz
- Yapısal istikrar kuralı korunur
- Process'ler arası güvenli sıralama sağlanır

3.3.2 Mutex (Mutual Exclusion) Kullanımı

Teorik Temel: Mutex'ler, kritik bölgelere sadece bir thread'in aynı anda erişebilmesini sağlar. Yarış koşullarını önlemek için kullanılır.

Projede Uygulama:

```
// Ortak kaynak için mutex kilidini al
```

```
// Ortak kaynak için mutex kilidini al
printf("Kat %d, Daire %d, %s için %s kaynağını talep ediyor...\n",
       floor, apt, adim_names[adim], resource_names[required_resource]);
pthread_mutex_lock(&params->resource_mutexes[required_resource]);

printf("Kat %d, Daire %d, %s için %s kaynağını aldı.\n",
       floor, apt, adim_names[adim], resource_names[required_resource]);
```

```
//     Kritik     bölge     -     kaynak     kullanımı
```

```
// İnşaat aşamasını simüle et
printf("Kat %d, Daire %d, %s aşaması başladı (%d saniye).\n",
       floor, apt, adim_names[adim], construction_time);
sleep(construction_time);
printf("Kat %d, Daire %d, %s aşaması tamamlandı.\n",
       floor, apt, adim_names[adim]);
```

```
// Mutex kilidini serbest bırak
```

```
// Ortak kaynağı serbest bırak
pthread_mutex_unlock(&params->resource_mutexes[required_resource]);
printf("Kat %d, Daire %d, %s kaynağını serbest bıraktı.\n",
       floor, apt, resource_names[required_resource]);
```

Bu yaklaşım ile:

- Ortak kaynaklar güvenli şekilde paylaşılır
- Aynı anda sadece bir daire belirli bir kaynağı kullanabilir
- Yarış koşulları önlenir

3.3.3 Condition Variable Kullanımı

Teorik Temel: Condition variable'lar, thread'lerin belirli koşulların gerçekleşmesini beklemelerini sağlar. Mutex ile birlikte kullanılarak güçlü senkronizasyon mekanizmaları oluşturur.

Projede Uygulama:

```
// Diğer dairelerin aynı aşamayı tamamlamasını bekle
```

```
printf("Kat %d, Daire %d, diğer dairelerin %s aşamasını tamamlamasını bekliyor.\n",
       floor, apt, adim_names[adim]);
pthread_cond_wait(params->floor_condition, params->floor_mutex);
```

```
// Tüm daireler aşamayı tamamladığında sinyal gönder
```

```
if (all_complete) {
    printf("Kat %d'deki tüm dairelerde %s aşaması tamamlandı.\n",
           floor, adim_names[adim]);
    pthread_cond_broadcast(params->floor_condition);
}
pthread_mutex_unlock(params->floor_mutex);
```

Bu mekanizma ile:

- Su ve elektrik tesisatı gibi kritik aşamalarda eşzamanlı tamamlama sağlanır
- Thread'ler gereksiz CPU döngüsü yapmadan bekler
- Verimli senkronizasyon gerçekleştirilir

3.4 Paylaşılan Bellek (Shared Memory)

Teorik Temel: Paylaşılan bellek, farklı process'lerin aynı bellek alanına erişebilmelerini sağlar. Process'ler arası iletişim (IPC) için etkili bir yöntemdir.

Projede Uygulama:

```
// Paylaşılan bellek oluşturma
```

```
// Paylaşılan bellek oluştur
key_t key = ftok("apartment_simulation", 'R');
int shmid = shmget(key, sizeof(SharedMemory), IPC_CREAT | 0666);
if (shmid < 0) {
    perror("shmget failed");
    exit(1);
}

SharedMemory* shared_mem = (SharedMemory*)shmat(shmid, NULL, 0);
if (shared_mem == (void*)-1) {
    perror("shmat failed");
    exit(1);
}
```

Bu yapı ile:

- Process'ler arası veri paylaşımı sağlanır
- Katların tamamlanma durumu global olarak izlenir
- Semaphore'lar process'ler arası paylaşılır

4. DETAYLI KOD ANALİZİ

4.1 Ana Fonksiyon Analizi

Ana fonksiyon, tüm sistemin orchestrasyonunu gerçekleştirir:

```
int main()
{     //Sistem başlatma
    printf("Apartman İnşaatı Simülasyonu Başlıyor\n");
srand(time(NULL));


    // Paylaşılan bellek kurulumu
    SharedMemory* shared_mem = setup_shared_memory();


    //      Process'leri      oluştur     ve    çalıştır
create_floor_processes(shared_mem);


    // Temizlik işlemleri     cleanup_resources(shared_mem);
}
```

4.2 Kat İnşaat Fonksiyonu Analizi

Her kat için çalışan süreç fonksiyonu:

```
// Kat inşaat süreci (her process bu fonksiyonu çalıştırır)
void construct_floor(int kat_numara, SharedMemory* shared_mem) {
    printf("Kat %d inşaatı başlıyor.\n", kat_numara);

    // Alt katın tamamlanmasını bekle (Kat 1 dışındaki tüm katlar için)
    if (kat_numara > 1) {
        printf("Kat %d, Kat %d'in tamamlanmasını bekliyor...\n", kat_numara, kat_numara - 1);
        sem_wait(&shared_mem->floor_semaphores[kat_numara - 1]);
        printf("Kat %d tamamlandı, Kat %d inşaatına geçiliyor.\n", kat_numara - 1, kat_numara);
    }
```

// Kaynakları hazırla

```
// Kaynak ihtiyaçlarını belirleme
Resource required_resource;
switch (adim) {
    case KABA_INSAAT:
        required_resource = VINC;
        break;
    case SU_TESISATI:
        required_resource = SU_KAYNAGI;
        break;
    case ELEKTRIK:
        required_resource = ELEKTRIK_KAYNAGI;
        break;
    case YALITIM:
        required_resource = YALITIM_EQUIPMENT;
        break;
    case HVAC:
        required_resource = HVAC_TEAM;
        break;
    case SIVA:
        required_resource = ASANSOR;  // Sıva malzemeleri için asansör kullanılır
        break;
    case DOSEME:
        required_resource = DOSEME_EKIPMANLARI;
        break;
    case DESIGN:
        required_resource = ASANSOR;  // İç tasarım malzemeleri için asansör kullanılır
        break;
}
```

//Daire thread'lerini oluştur

```
// Her daire için thread oluştur
pthread_t apartment_threads[DAIRE];
ApartmentParams params[DAIRE];
```

// Thread'lerin tamamlanmasını bekle

```
// Tüm dairelerin tamamlanmasını bekle
for (int apt = 0; apt < DAIRE; apt++) {
    pthread_join(apartment_threads[apt], NULL);
}
```

// Bir sonraki kata izin ver

```
// Bu katın tamamlandığını işaretle
shared_mem->floor_complete[kat_numara - 1] = 1;

// Bir sonraki katın başlayabileceğini bildir
printf("Kat %d inşaatı tamamlandı. Kat %d inşaatına izin veriliyor.\n", kat_numara, kat_numara + 1);
sem_post(&shared_mem->floor_semaphores[kat_numara]);
```

4.3 Daire İnşaat Fonksiyonu Analizi

Her daire için çalışan thread fonksiyonu:

```
// Daire inşaat süreci fonksiyonu (thread tarafından çalıştırılır)
void* insa_et(void* args) {
    ApartmentParams* params = (ApartmentParams*)args;
    int floor = params->kat_numara;
    int apt = params->daire_numara;
```

// Her inşaat aşaması için

```
// Her inşaat aşaması için
for (int adim = 0; adim < INSAAT_ADIMLARI; adim++) {
    // İnşaat aşaması süresini (saniye) rastgele belirleme (1-3 saniye arası)
    int construction_time = (rand() % 3) + 1;
```

{

// Gerekli kaynakları belirle

```
    // Kaynak ihtiyaçlarını belirleme
    Resource required_resource;
    switch (adim) {
        case KABA_INSAAT:
            required_resource = VINC;
            break;
        case SU_TESISATI:
            required_resource = SU_KAYNAGI;
            break;
        case ELEKTRIK:
            required_resource = ELEKTRIK_KAYNAGI;
            break;
        case YALITIM:
            required_resource = YALITIM_EQUIPMENT;
            break;
        case HVAC:
            required_resource = HVAC_TEAM;
            break;
        case SIVA:
            required_resource = ASANSOR;  // Sıva malzemeleri için asansör kullanılır
            break;
        case DOSEME:
            required_resource = DOSEME_EKIPMANLARI;
            break;
        case DESIGN:
            required_resource = ASANSOR;  // İç tasarım malzemeleri için asansör kullanılır
            break;
    }
```

// Kaynağı talep et ve kullan

```
    // Ortak kaynak için mutex kilidini al
    printf("Kat %d, Daire %d, %s için %s kaynağını talep ediyor...\n",
           floor, apt, adim_names[adim], resource_names[required_resource]);
    pthread_mutex_lock(&params->resource_mutexes[required_resource]);

    printf("Kat %d, Daire %d, %s için %s kaynağını aldı.\n",
           floor, apt, adim_names[adim], resource_names[required_resource]);
```

// Senkronizasyon kurallarını uygula

```
// Kat içi senkronizasyon için mutex ve condition variable
pthread_mutex_t floor_mutex;
pthread_cond_t floor_condition;
pthread_mutex_init(&floor_mutex, NULL);
pthread_cond_init(&floor_condition, NULL);
```

```
// Katlar arası senkronizasyon için semaphore'ları başlat
for (int i = 0; i <= TOPLAM_KAT; i++) {
    if (sem_init(&shared_mem->floor_semaphores[i], 1, (i == 0) ? 1 : 0) != 0) {
        perror("sem_init failed");
        exit(1);
    }
}
```

```
    }
return NULL; }
```

5. SENKRONİZASYON KURALLAري VE ÖZELLİKLER

5.1 Process Düzeyinde Senkronizasyon

Sıralı İnşaat Kuralı:

- Kat N+1'in inşaatı, Kat N tamamlanmadan başlayamaz
- Bu kural semaphore mekanizması ile uygulanır
- Yapısal istikrar ve güvenlik gereksinimlerini simüle eder

Uygulama Detayı:

```
// Her kat için bir semaphore (başlangıçta sadece Kat 0 için açık)
```

```
// Katlar arası senkronizasyon için semaphore'ları başlat
for (int i = 0; i <= TOPLAM_KAT; i++) {
    if (sem_init(&shared_mem->floor_semaphores[i], 1, (i == 0) ? 1 : 0) != 0) {
        perror("sem_init failed");
        exit(1);
    }
}
```

Thread Düzeyinde Senkronizasyon

Ortak Kaynak Erişimi:

- Aynı kaynağı kullanan daireler sıraya girer
- Her kaynak için ayrı mutex koruması
- Deadlock önleme stratejileri uygulanır

Aşamalı Senkronizasyon:

- Su ve elektrik tesisatı aşamalarında tüm dairelerin eşzamanlı tamamlaması
- Condition variable ile verimli bekleme
- Race condition'ların önlenmesi

5.3 Yarış Koşulu Önleme Stratejileri **Kaynak Koruma:**

- Her ortak kaynak için ayrı mutex
- Kısa süreli kilit tutma
- İç içe kilit alma yapmaktan kaçınma

Veri Tutarlılığı:

- Paylaşılan verilere erişimde mutex kullanımı
- Atomik operasyonların güvence altına alınması
- Memory barrier'ların doğru kullanımı

6. PERFORMANS VE ÖLÇEKLENEBILIRLIK ANALIZİ

6.1 Sistem Performansı

Paralellik Derecesi:

- Maksimum 40 thread (10 kat × 4 daire) eşzamanlı çalışabilir
- Ortak kaynak kısıtlamaları paralelliği sınırlar
- Her aşamada farklı paralellik seviyeleri gözlenir

Kaynak Kullanımı:

- Bellek kullanımı: O(katlar + daireler) complexity
- CPU kullanımı: Ortak kaynak erişimi ile sınırlı
- I/O işlemleri: Sadece konsol çıktısı

6.2 Ölçeklenebilirlik

Yatay Ölçeklendirme:

- Kat ve daire sayısı kolayca artırılabilir
- Sabit ortak kaynak sayısı performans darboğazı oluşturabilir
- Process ve thread sayısı sistem sınırları ile kısıtlı

Dikey Ölçeklendirme:

- Ortak kaynak sayısı artırılarak performans iyileştirilebilir
- Daha karmaşık senkronizasyon kuralları eklenebilir
- Gerçek zamanlı izleme ve raporlama eklenebilir

7. SONUÇ VE DEĞERLENDİRME

Bu proje kapsamında, apartman inşaatı simülasyonu üzerinden işletim sistemi kavramları başarıyla modellenmiştir. Elde edilen temel sonuçlar şunlardır:

Başarılan Hedefler:

- Process ve thread kavramlarının pratik uygulaması gerçekleştirildi
- Çoklu düzeyde senkronizasyon (process ve thread) başarıyla uygulandı
- Ortak kaynak paylaşımı ve yarış koşulu önleme mekanizmaları çalışır durumda
- Gerçek hayat senaryosu ile teorik kavramlar arasında köprü kuruldu

Öğrenilen Dersler:

- Senkronizasyon mekanizmalarının karmaşıklığı ve önemi
- Paralel programlamada tasarım kararlarının kritik rolü
- Deadlock ve race condition risklerinin minimize edilmesi
- Sistem kaynaklarının verimli kullanımının gerekliliği

Geliştirilmesi Gereken Alanlar:

- Daha sofistike deadlock detection ve prevention algoritmaları
- Dynamic load balancing mekanizmaları
- Fault tolerance ve error recovery yetenekleri
- Performance monitoring ve optimization araçları

Pratik Uygulama Değeri: Bu simülasyon, işletim sistemi kavramlarının gerçek dünya problemlerine nasıl uygulanabileceğini göstermektedir. Apartman inşaatı gibi karmaşık, çok aşamalı süreçlerin modellenmesi, sistem tasarımcılarına değerli perspektifler sunmaktadır.

Proje, teorik bilgilerin pratik uygulamaya dönüştürülmesi açısından başarılı olmuş ve işletim sistemi kavramlarının derinlemesine anlaşılmasına katkı sağlamıştır. Gelecekte bu tür simülasyonlar, daha karmaşık senaryolar ve gerçek sistem optimizasyonları için temel oluşturabilir.

Kaynaklar:

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts. John Wiley & Sons.
- Stevens, W. R., & Rago, S. A. (2013). Advanced Programming in the UNIX Environment. Addison-Wesley.
- POSIX Threads Programming Documentation
- Linux System Programming Manual Pages