

# İŞLETİM SİSTEMLERİ

# İçindekiler

[SORU 1: diff komutu](#)

[SORU 2: One Lane Bridge \(Tek Şeritli Köprü\) Problemi](#)

[SORU 3: Sleeping Barber \(Uyuyan Berber\) Problemi](#)

[SORU 4: POSIX Thread pthread kütüphanesi](#)

[SORU 5: exec fonksiyonları](#)

[SORU 6: fork fonksiyonu ve işlem çatallanması](#)

[SORU 7: Producer Consumer Problem \(Üretici Tüketici Problemi\)](#)

[SORU 8: Filozofların Akşam Yemeği \(Dining Philosophers\)](#)

[SORU 9: Banker Algoritması \(Banker's Algorithm\)](#)

[SORU 10: Peterson's Algorithm](#)

[SORU 11: Dekker's Algorithm](#)

[SORU 12: Birbirini Dışlama \(Mutually Exclusive\)](#)

[SORU 13: İşletim Sistemlerinde Hafıza Güvenliği](#)

[SORU 14: CFS \(Completely Fair Scheduling, Tam Adil Zamanlama\)](#)

[SORU 15: O\(1\) Zamanlaması \(O\(1\) Scheduling\)](#)

[SORU 16: Overhead \(Ek Yük\)](#)

[SORU 17: Fair Share Scheduling \(Adil Paylaşımlı Zamanlama\)](#)

[SORU 18: Translation Lookaside Buffer \(TLB, Dönüşüm Hafızası\)](#)

[SORU 19: Ön Hafıza \(Cache\)](#)

[SORU 20: Thread \(iplik, lif, iz\)](#)

[SORU 21: İşlemci Zamanları \(CPU Timing\)](#)

[SORU 22: Çok Seviyeli Sıralar \(Multi Level Queues\)](#)

[SORU 23: İçerik Değiştirme \(Context Switching\)](#)

[SORU 24: Sembolik Bağ \(Symbolic Link\)](#)

[SORU 25: Makine Dilleri \(Machine Language\)](#)

[SORU 26: Eşlemeli Metotlar \(Synchronized Methods\)](#)

[SORU 27: Priority Queue \(Öncelik Sırası, Rüçhan Sırası\)](#)

[SORU 28: Cluster Computing \(Bilgisayar Kümeleri\)](#)

[SORU 29: Turing Makinesi \(Turing Machine\)](#)

[SORU 30: Sanal Hafıza \(Virtual Memory\)](#)

[SORU 31: Sayfalama \(Paging\)](#)

[SORU 32: Sayfa Değiştirme Algoritması \(Page Replacement\)](#)

[SORU 33: CPU Utilization \(MİB Meşguliyeti\)](#)

[SORU 34: Meşguliyet \(Utilization, Kullanım\)](#)

[SORU 35: Semafor \(Semaphore, Flama, İşaret\)](#)

[SORU 36: Atomluluk \(Atomicity\)](#)

[SORU 37: Gizli Dosya \(Hidden File\)](#)

[SORU 38: C ile Zaman İşlemleri](#)

[SORU 39: İşlem Çatallanması \(Process Forking\)](#)

[SORU 40: Kabuk \(Shell\)](#)

[SORU 41: Çekirdek \(Kernel\)](#)

[SORU 42: Dahili Parçalar \(Internal Fragments\)](#)

[SORU 43: Kıtalamak \(Bölütlemek, Segmentation\)](#)

[SORU 44: Harici Parçalar \(External Fragments\)](#)

[SORU 45: Yükleyici \(Loader\)](#)

[SORU 46: Hafıza Yönetimi \(Memory Management\)](#)

[SORU 47: İşletim Sistemi \(Operating System\)](#)

[SORU 48: Yığın İş \( Batch Job, Batch Process \)](#)

[SORU 49: Kilitlenme \(Deadlock\)](#)

[SORU 50: Kıtlık \(Starvation\)](#)

[SORU 51: En Kısa İş İlk \(Shortest Job First\)](#)

[SORU 52: İlk Gelen Çalışır \(First Come First Serve, FCFS, FIFO\)](#)

[SORU 53: Round Robin](#)

[SORU 54: Kesmeyen Zamanlama \(non-preemptive Scheduling\)](#)

[SORU 55: Kesintili Zamanlama \(Preemptive Scheduling\)](#)

[SORU 56: İşlemci Zamanlama \(CPU Scheduling\)](#)

[SORU 57: Görevlendirici \(Dispatcher\)](#)

[SORU 58: Bekleme Sırası \(Ready Queue\)](#)

[SORU 59: Çevirici \(Assembler\)](#)

[SORU 60: Dinamik Bağlantı Kütüphaneleri \(Dynamic Link Library \(.dll\)\)](#)

[SORU 61: Yerleştirme Algoritmaları \(Fitting Algorithms\)](#)

[SORU 62: Rastgele Erişilebilir Bellek \(Random Access Memory , RAM\)](#)

[SORU 63: C ve Komut Satırı \(C Console Parameters\)](#)

[SORU 64: Sıralama Algoritmaları \(Sorting Algorithms\)](#)

[SORU 65: Bağlayıcı \(linker\)](#)

[SORU 66: sunucu \(server\)](#)

[SORU 67: istemci \(client, talebe\)](#)

[SORU 68: bit \(ikil\)](#)

[SORU 69: Çok işlemlik \(Multi processing\)](#)

[SORU 70: İşlem \(Process\)](#)

[SORU 71: İşlemler arası iletişim \(Inter process communication \(IPC\)\)](#)

**SORU 72: Disk Yönetimi (Disk Management)**

## SORU 1: diff komutu

Bu yazının amacı, bir UNIX komutu olan diff komutunu açıklamaktır. diff komutu, iki dosyanın arasındaki farklılıkları (ve dolayısıyla benzerlikleri) bularak ekranda göstermeye yarar.

Algoritma basitçe en uzun ortak kısmı (longest common subsequence) bulmaya dayanır.

Algoritmanın çalışmasını bir örnek üzerinden görebiliriz:

Örneğin, a ve b isminde iki dizgi (string) alalım:

a: a b c d k l p s

b: a b e f g o k l z

diff komutunun yukarıdaki girdiler için çıktısı aşağıdaki şekildedir:

3,4c3,6

< c

< d

—

> e

> f

> g

> o

7,8c9

< p

< s

—

> z

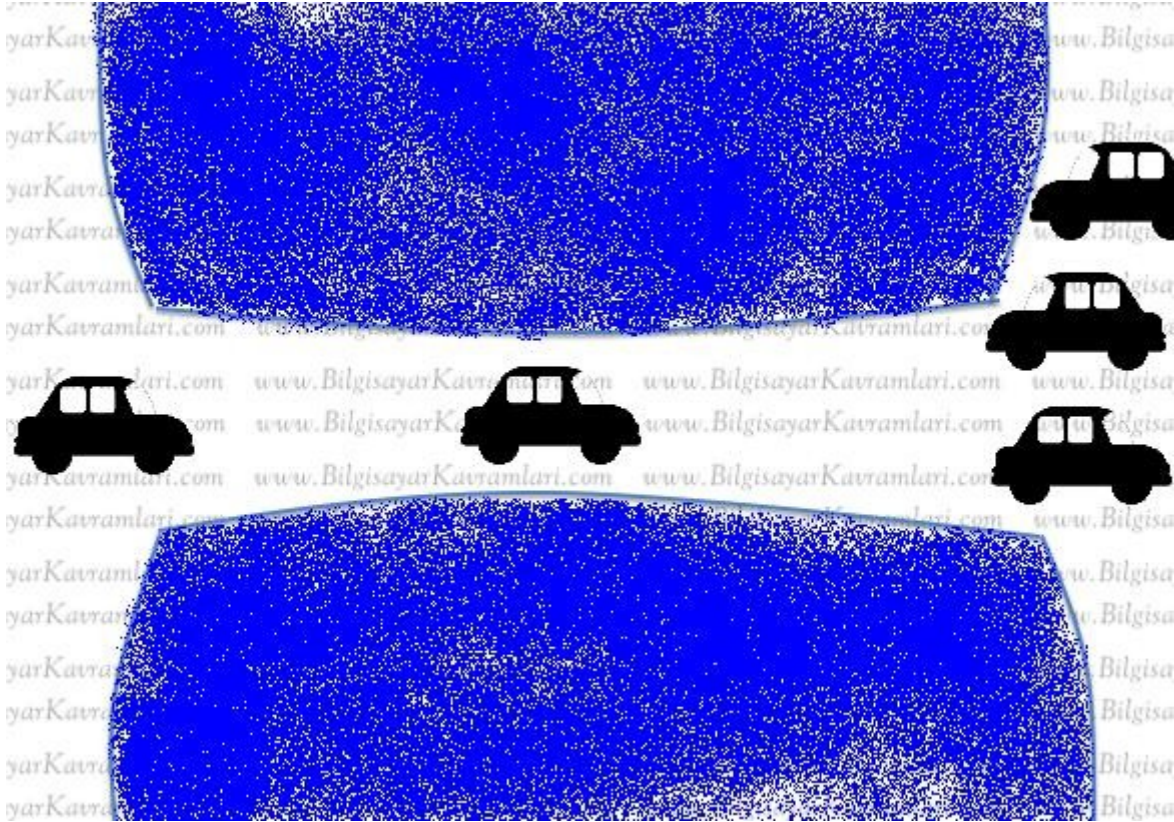
Diff komutunun kullanımı için linux (veya unix, macos) işletim sisteminde komut satırında aşağıdaki şekilde komutun yanına iki dosya ismi parametre olarak verilir.

diff a.txt b.txt

Bu işlem sonucunda iki dosyanın içeriğindeki farklılıklar ekrana basılır.

## SORU 2: One Lane Bridge (Tek Şeritli Köprü) Problemi

Bu problemde, tek şeritli bir köprünün iki ucundan gelen araçların karşıya geçmesini senkronize etmemiz isteniyor. Köprü tek şeritli olduğu için anlık olarak bir yönde araç geçişi mümkün olabiliyor. Bir aracın karşıya geçmesi, ancak karşı taraftan gelen araç olmadığı zaman mümkündür. Köprünün iki tarafında da duyargaların (sensor) yerleştirildiğini düşünelim ve aynı zamanda iki tarafta da trafik ışıkları bulunsun. Bu durumda her aracın bir işlem (process) olarak düşünüldüğü sistemde Varma () ve Ayrılma() fonksiyonlarını kodlamamız isteniyor. Ayrıca bu fonksiyonlara, işlemler tarafından yön bilgisinin geçirilebileceğini de kabul edebilirsiniz.



Şekilde görüldüğü üzere köprünün iki tarafında da bekleyen araçlar bulunmakta ve bu araçlardan anlık olarak bir araç karşı tarafa geçebilmektedir.

### Çözüm:

Problemin çözümü için öncelikle bir global değişken tanımlamamız gerekiyor. Bu değişken, o anda bulunan araç sayısını tutacak. İkinci bir değişken ise o andaki köprünün akış yönünü tutmaktan sorumlu olsun. Yani bütün işlemler (processes) bu değişkenlere erişebilecek ve dolayısıyla anlık olarak hangi yönde akışın olduğu ve kaç araç bulunduğu bilgisi bütün işlemler için erişilebilen tek bir yerde tutulacak. Elbette bu tanım anında bir senkronizasyon problemini beraberinde getiriyor.

Temel olarak eş programlama (concurrent programming) problemlerinde iki problem bulunur. Bunlar:

- [Eş farklılık \(Mutual Exclusion\)](#)
- İlerleme (Progress)

Olarak bilinir. Yani aynı anda bir işlemin çalışması ve birden fazla işlemin çalışmasının engellenmesi eş farklılık problemi olarak geçerken, herhangi bir işlemin [kilitlenmemesi \(deadlock\)](#) kısıtlığa girmemesi (starvation) veya problemin çözümüne ilerleme kat edilmesi ilerleme (progress) olarak geçer.

Bu soruları eğitim amacıyla çözdüğümüz için bu problemde iki farklı senkronizasyon (synchronisation) yöntemini beraber kullanacağız. Öncelikle [eş farklılık \(mutex\)](#) için kilit (lock) senkronizasyonu kullanacağız, ardından kritik alanımız olan (critical section) yön değişimi için koşullu değişken (conditional variable) kullanacağız.

Eş farklılık için (mutex) bütün işlemlerden önce kilit koyup bütün işlemlerden sonra kilidi açıyoruz.

İlerleme için (yani bir taraftan araç geçiyorken, diğer taraftan araç gelmesi halinde, diğer taraftaki araçta günün birinde geçmesini sağlamak için) koşullu değişken (conditional variable) kullanacağız. Bunu yön değiştirme işleminde yapıyoruz. Bir araç köprüyü terk edince (depart) yön değiştirmek mümkün hale gelir. Bu durum olunca köprünün iki yanında bekleyen araçlara duyurularak bir yarış koşulu (racing condition) oluşturulur, kim önce bu yarışı kazanırsa o taraftan araçlar geçmeye başlar.

```
int arac_sayisi = 0;
enum yon = {acik, sola, saga};
yon kopru_yonu = acik;
Lock *lock = new Lock();
Condition *cv = new Condition();
```

```
void Depart (yon aracin_yonu) {
    lock->Acquire();
    arac_sayisi--;
    if (arac_sayisi == 0) {
        kopru_yonu = acik;
        cv->Broadcast(lock);
    }
    lock->Release();
}
```

```
void Arrive (yon aracin_yonu)
    lock->Acquire();
    while (kopru_yonu != aracin_yonu || kopru_yonu != acik) {
        cv->Wait(lock);
    }
    arac_sayisi++;
    kopru_yonu = aracin_yonu;
    lock->Release();
}
```

Yukarıdaki çözümde, yön değiştirme durumu olan kopru\_yonu = aracin\_yonu şeklindeki satırdan önce aracin yönündeki hareketin mümkün olup olmadığı bir while döngüsü ile

kontrol edilmektedir. Bu kontrolden geçildikten sonra kritik alanımız olan (critical section) köprünün yönünü değiştirme işlemi çalıştırılabilir.

Yukarıdaki çözümde, [kilitlenme riski \(deadlock\)](#) bulunmaz çünkü anlık olarak tek bir işlem (process) çalışabilir, bu durum mutex uygulaması ile güvenceye alınmıştır.

Yukarıdaki çözümde tek sorun, [kıtlık olma ihtimalidir \(starvation\)](#) bunun sebebi bir yönden gelen araçların diğer yönden gelenlere göre öncelik kazanması ve yarış durumu (racing condition) düşünüldüğünde sıranın hiç karşı tarafa geçmemesi ihtimalidir. Bu durumu da çözmek için belirli bir sayı belirleyelim, örneğin n olsun. Bu n sayısından fazla bir yönden araç, diğer yönden araç geçmeden arka arkaya geçememeli kuralı koyabiliriz. Örneğin n = 5 olsun, bu durumda sağdan sola arka arkaya en fazla 5 araç geçebilir ve sonra durup soldan sağa geçmeyi bekleyecektir. Bu kuralımızı koda eklersek aşağıdaki gibi bir durum ortaya çıkar:

```
int arac_sayisi = 0;
int n = 5;
int solsayac = 0;
int sagsayac = 0;
enum yon = {acik, sola, saga};
yon kopru_yonu = acik;
Lock *lock = new Lock();
Condition *cv = new Condition();
```

```
void Depart (yon aracin_yonu) {
    lock->Acquire();
    arac_sayisi--;
    if(aracin_yonu == sola){
        sagsayac = 0;
        solsayac++;
    }
    else{
        sagsayac++;
        solsayac= 0;
    }
    if (arac_sayisi == 0) {
        kopru_yonu = acik;
        cv->Broadcast(lock);
    }
    lock->Release();
}
```

```
void Arrive (yon aracin_yonu)
    lock->Acquire();
    if(aracin_yonu == sola && solsayac >= n)
        cv->Wait(lock);
    if(aracin_yonu == saga && sagsayac >= n)
        cv->Wait(lock);
    while (kopru_yonu != aracin_yonu || kopru_yonu != acik) {
        cv->Wait(lock);
    }
    arac_sayisi++;
    kopru_yonu = aracin_yonu;
    lock->Release();
```



}

}

Yukarıdaki yeni çözümde, aracın geldiği yön ile bu yönün sayaçlarına bakılmakta, şayet aracın geldiği yönde, tanımlı olan n değeri kadar veya daha fazla araç geçiş yapmışsa bu araç bekletilmektedir.

Araçların ayrılması sırasında ise, ayrılan aracın tersi istakemtindeki sayaç sıfırlanmaktadır (çünkü bu yöndeki araçların arka arkaya geçmesi durumu bozulmuş, ters yönden bir araç geçmiştir).

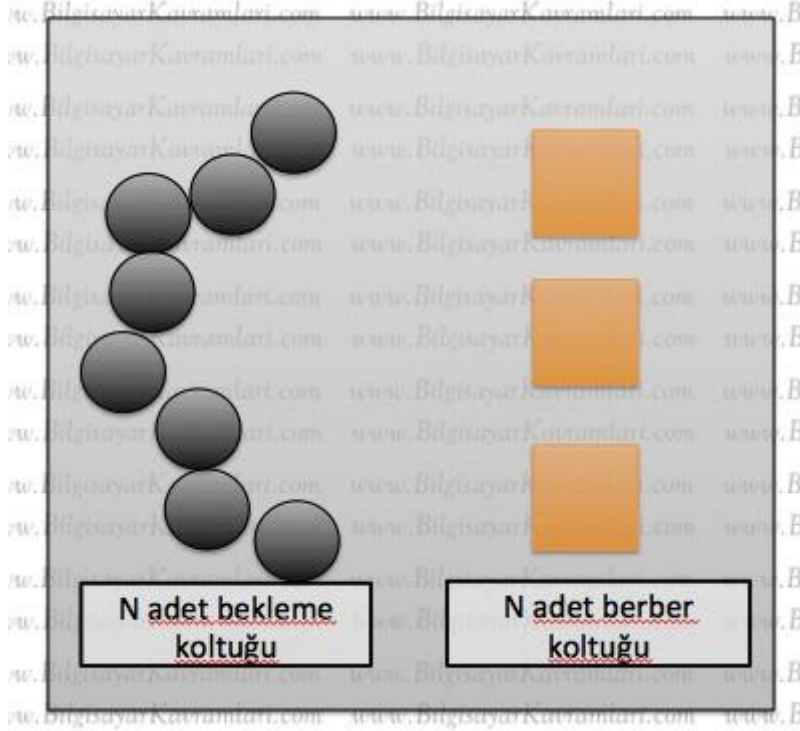
Ayrıca araçlar köprüyü terk ederken, artık geçme işlemi tamamlandığına göre aracın yönünden geçen araçların sayısı bir arttırılmaktadır.

Yukarıdaki yeni çözümde, kıtlık ve kilitlenme problemleri çözülmüş olur. Ancak bir tehlike kıtlık (starvation) çözümünü getirirken tanımladığımız n değerinin 1 olması durumudur. Bu durumda kilitlenme olma ihtimali bulunur ve okuyucu bunu kendisi deneyerek bulabilir.

Ayrıca ben işletim sistemleri dersini alırken (sene 1998 ) vizede aynı soru sorulmuş ve soruyu boruhattı (pipeline) yaklaşımı ile çözmeye çalışmıştım. Neden böyle bir zorluk çıkardığımı bilmiyorum ama o zamanki yaklaşımına göre bir yönde bir araç geçerken arkasından birden fazla aracın takip edebileceği ve köprü üzerinde anlık olarak i tane araç bulunabileceği kabulü ile soruyu çözmüştüm (sanırım gerçek hayatta bu şekilde olduğu için). Bu yaklaşımın genel olarak senkronizasyona bir etkisi olmamakla birlikte, yön değiştirme kararlarının köprüden geçmeyi bekleyen araçların bekleme sürelerine etkisi olmaktadır. Bu durum aslında senkronizasyon problemi ile birlikte işlemci zamanlamasında kullanılan yaklaşımların da tasarımı eklenmesini gerektirir. Okuyucu bu durumu çözerek deneyebilir.

### **SORU 3: Sleeping Barber (Uyuyan Berber) Problemi**

Problemin tanımı : Bir berber dükkanında, bir bekleme salonu, bu salonda n adet sandelye ve ayrıca m adet berber koltuğu bulunmaktadır. Sistemimizde m adet de berber olduğunu düşünelim. Şayet hiç bekleyen müşteri yoksa berber uyumaya gider (diğer berberlerin traş ettiği müşterileri beklemesine gerek yoktur). Şayet bekleme salonundaki bütün sandelyeler doluyken bir müşteri içeri girerse, salon dolu olduğu için müşteri salonu terk eder (müşteri kaybedilmiş olur). Şayet berber o sırada birisini traş ediyorsa ama boş sandelye varsa, içeri giren kişi, boş sandelyelerden birisine oturur. Şayet berber uyurken bir müşteri girerse, müşteri berberi uyandırır. Bizden istenen bu problemin çözümünü yapan kodu (berber ve müşteri tarafları için) kodlamamız.



Şekilde görüldüğü üzere bir berber salonunda, n adet bekleme koltuğu ve n adet berber koltuğu şeklinde problem düşünülebilir.

### Çözüm:

Problem tanımında bulunan ve senkronize edilmesi gereken durumları bulalım. Öncelikle problemimizde [birbirini dışlama \(mutually exclusion\)](#) problemi bulunmaktadır. Problem tanımındaki birden fazla işlemin (process) birbiri ile yarış durumuna (racing condition) girmesi halinde bazı kaynaklarda problem yaşanabilir. Örneğin aynı sandalyeye birden fazla müşterinin oturması veya berber koltuğuna birden fazla müşterinin oturması veya bir müşteri berber koltuğuna diğer müşteri kalkmadan oturmaya çalışırsa bu durumda yaşanacak problemler gibi çok sayıda problem bulunmaktadır.

Bu problemlerin en hızlı ve doğru çözümü [birbirini dışlar şekilde \(mutex\)](#) işlemlerin senkronize edilmesidir. Diğer bir deyişle aynı anda tek bir işlemin çalışmasına izin verilirse problem çözülür.

Öncelikle tasarımımda [semafor kullanacağız \(semaphore\)](#). Problemimizde iki yaklaşımı çözmemiz gerekiyor. Birisi daha önce bahsedilen [mutex](#) diğeri ise problemin ilerlemesi açısından berber ve müşteri işlemleri (process) arasındaki ilişki. İki problem için de [semafor](#) kullanalım. Bu durumda çözüm aşağıdaki şekilde olur:

```
sem_init(&mutex,0,1);
sem_init(&musteri,0,0);
sem_init(&berberler,0,1);
void berber(void *arg)/*Berber
Process*/
{
    sem_wait(&musteri);
    sem_wait(&mutex);
```

```

        sayac--;
        printf("Barber:saçlı kesilen:%d. kişinin",sayac);
        sem_post(&mutex);
        sem_post(&berberler);
        sleep(3); // burada müşteriyi traş ediyor
    }

void musteri(void *arg)/*musteri Process*/
{
    sem_wait(&mutex);
    if(sayac<N){
        sayac++;
        printf("Bekleyen musteri sayisi :%d
oldun",sayac);

        sem_post(&mutex);
        sem_post(&customer);
        sem_wait(&berberler);
        printf("Müşteri: traş oldum");
    }
    else{
        printf("Musteriyi kaybettik");
        sem_post(&mutex);
    }
    sleep(1);
}

```

Yukarıdaki çözümde, N sayısı, bekleme salonundaki sandeyle sayısını belirtmektedir. Sayac değişkeni anlık olarak kaç müşterinin salonda olduğunu belirtmektedir. Ayrıca aynı anda bir adet müşteri veya berber çalışabilmektedir. (mutex çözümünden dolayı)

Her iki işleminden bşalancısında sem\_wait(&mutex) satırı bulunmakta (dolayısıyla çalışan işlem her kim olursa olsun (berber ya da müşteri) o anda başka bir işlemin çalışmasına izin verilmemektedir. ) ve iki işlemin de çıkışında sem\_post(&mutex) bulunmaktadır. Bu durumda bir işlem bittikten sonra diğer işlemler çalışabilmektedir.

Şayet müşteri tarafındaki müşteri sayısı N'den küçükse, bekleme salonunda yer var demektir. Şayet daha büyükse yer yok demektir. Bu durumda müşteri kaybederiz. Dikkat edilirse müşteri fonksiyonunda iki durumda da çıkışta sem\_post(&mutex) yapılmaktadır.

Ancak şayet yer varsa (sayac < N durumu) ve müşteri salona alınırsa (sayac++ yapılmış demektir) ve boş berber varsa bu durumda müşteri traş olur.

Bu tasarım sorunun çözümü açısından doğrudur ve senkronizasyon başarılı bir şekilde yapılmıştır. Ancak bu çözümde problem olacak bir durum müşterilerin geliş sırasının dikkate alınmamasıdır. Örneğin salona ilk gelen ve bekleyen müşteriye sıra henüz gelmeden sonradan gelen müşteriler traş olup çıkabilir. Yani salonda bekleyen bütün müşteriler aynı derecede değerlendirilmektedir. Şayet sıra dikkate alınmak isteniyorsa, bu durumda sıra teorisi (queueuing theory) kullanılmalıdır ve bu sorunun kapsamı dışındadır. Ancak burada bir [kıtlık durumu \(starvation\)](#) söz konusu olduğu bilinmelidir.

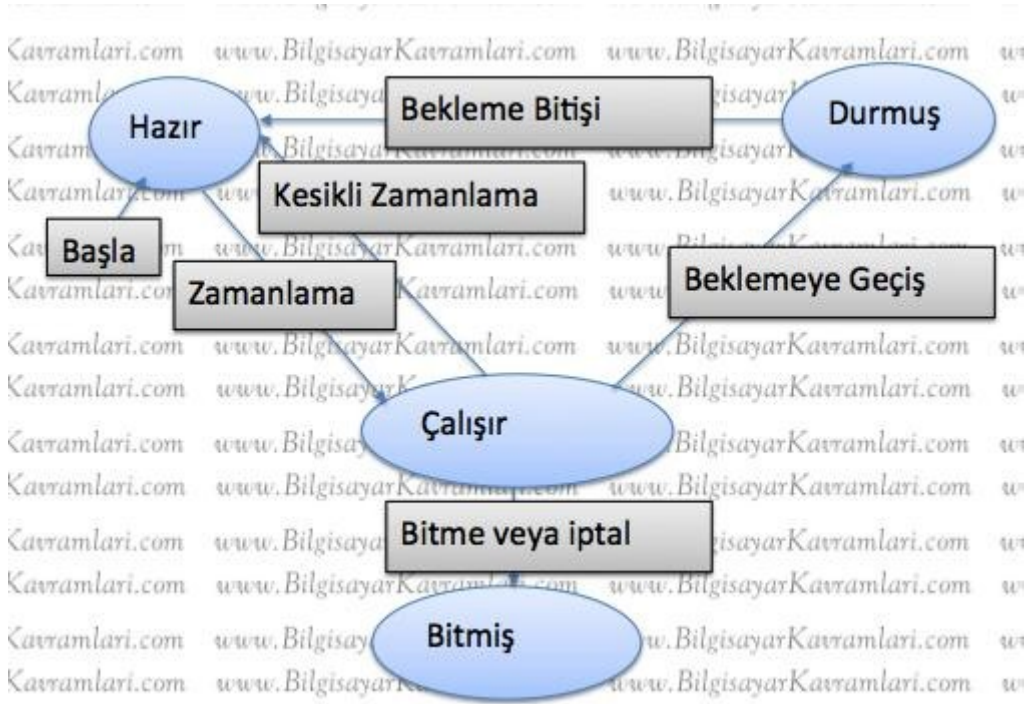
Yukarıda çözülmüş diğer bir problem ise, müşterilerin berber, berberlerin de müşteri beklemesi durumudur. Buna göre bir müşteri traş olmak için öncelikle boş bir berber bulmalıdır. Bu durum müşteri fonksiyonunda sem\_wait(&berber) olarak belirtilmiştir. Aynı şekilde bir berber de traş etmek için bir müşteri beklemektedir. Bu durumda berber fonksiyonunda sem\_wait(&musteri) olarak belirtilmiştir. Bu bekleme durumları da sem\_post

fonksiyonları ile çözülür. Şöyle ki bir müşteri geldiğinde sem\_post(&musteri) çağrılarak müşteriyi bekleyen berberler uyandırılır, bir müşteri traş olup berber koltuğundan ayrıldığında da sem\_post(&berber) denilerek o anda boşalan bir berber koltuğu (veya berber) olduğu haber verilir. Örneğin bir berber boşaldığı zaman, bekleme salonunda bulunan müşteriler yarış durumuna (racing condition) geçerler, aralarından ilk çalışan ve yarışı kazananı traş olmak için berber koltuğuna oturur. Şayet hiç müşteri yoksa yani salon boşsa, berber uyumaya gider.

#### SORU 4: POSIX Thread pthread kütüphanesi

Bilgisayar bilimlerinde geçen [lif \(thread, iplik, sicim\)](#) kavramının C dili ile kodlanabilmesi için genellikle UNIX türevi işletim sistemlerinde geliştirilen programlama kütüphanesidir. Kütüphane UNIX ortamında (ve dolayısıyla LINUX ortamında da ) POSIX kütüphanesi olarak geçmektedir ve bu kütüphanenin baş harfi olan P harfi ile thread kelimesinin birleşmesinden türemiştir (pthread).

Kütüphane kullanılarak temel lif işlemleri yapılabilir. Aşağıda, kütüphanede kullanılan bazı temel fonksiyonlar, tanımları ile birlikte verilecektir. Ardından meşhur, [üretici / tüketici \(producer consumer\)](#) örneği üzerinden nasıl uygulandığı gösterilecektir



Yukarıdaki şekilde 4 farklı durum arasındaki geçişler gösterilmiştir. Buna göre bir [lif \(thread\)](#) başlatıldıktan sonra, hazır durumuna geçer. [Zamanlama algoritmasına bağlı olarak \(scheduling algorithm\)](#), çalışır duruma geçebilir. Çalışma işlemi sırasında lif işini bitirirse veya iptal edilirse (kendi kendini iptal edebilir veya farklı bir lif tarafından iptal edilebilir) bitmiş duruma geçer. Çalışır durumdayken bir sebeple bekleme durumuna geçebilir. Örneğin farklı bir lifi bekleyebilir veya bir sistem kaynağına erişmek isteyebilir veya belirli bir süre için uyutulmuş olabilir. Durma sebebi ortadan kalktıktan sonra, hazır duruma geri geçer ve [hazır sırasında \(ready queue\)](#) beklemeye başlar.

Konuyu örnek bir kod ile anlamaya çalışalım:

### Örnek 1:

```
#include <pthread.h>
#include <stdio.h>
void *thread_routine(void* arg){
    printf("Yeni üretilen lif n");
}
void main(){
    pthread_t thread_id; void *thread_result;
    pthread_create( &thread_id, NULL, thread_routine, NULL );
    printf("Ana lifte n");
    pthread_join( thread_id, &thread_result);
}
```

Yukarıdaki örnek kodun çalışması aşağıda verilmiştir.

```
SADIs-MacBook-Air:yedekler sadievreenseker$ gcc p.c -lpthread
SADIs-MacBook-Air:yedekler sadievreenseker$ ./a.out
Ana lifte
Yeni üretilen lif
```

Yukarıdaki çalışma örneğinde görüldüğü üzere, derleme işlemi sırasında gcc derleyicisine -lpthread parametresi verilmiştir. Bunun sebebi derleme sırasında pthread kütüphanesinin bağlanması (link) gereğidir. Ardından derlenen kod çalıştırılmış ve ekrana sırasıyla “Ana lifte” ve “Yeni üretilen lif” mesajlarını basmıştır.

Kodun çalışması sırasında main fonksiyonu içerisinde sırasıyla, önce bir lif bilgisini tutabilecek ve pthread\_t [yapısından \(struct\)](#) üretilen bir değişken tanımlanmıştır. Bu değişkeni daha sonraki örneklerde life doğrudan erişmek için kullanacağız. Bu örneğimizin en önemli satırı olan pthread\_create fonksiyonunun ilk parametresidir. Bu parametre [atıf ile çağırma \(call by reference\)](#) işlemi olarak düşünülmelidir ve aslında yapılan lif üretimi sırasında (pthread\_create) parametre vererek, üretilen yeni lifin bilgisini almaktır.

Ardından ekrana “Ana lifte” mesajı basılmıştır. Burada dikkat edilecek husus, aslında yeni bir [lif \(thread\)](#) üretilirken en az iki lif bulunduğu durumdur. Birisi yeni üretilen [lif \(thread\)](#) diğeri ise bu lifi üreten lif olan ana liftir (main thread). Bu durum aşağıdaki şekilde ifade edilebilir:



Temsili resimcikte görüldüğü üzere, program akışı içerisinde çağrılan bir pthread\_create fonksiyonu ile hafızada iki farklı [lif \(thread\)](#) üretilmiş ve bunlar aynı anda çalışmaya devam etmiştir. Ana lifteki çalışma bittikten sonra, pthread\_join ile, yeni üretilen lifin (thread) bitmesi beklenmiştir.



Hafızada anlık olarak iki lifin bulunduğu durum, yukarıdaki şekilde gösterilmiştir. Bu gösterimden anlaşılacağı üzere ana lif ve yeni lif için iki ayrı yığın (stack) alanı bulunmaktadır. Bu alanlarda liflerin kendisine özel bilgileri tutulmaktadır. Örneğin anlık olarak hangi fonksiyonun hangi satırını çalıştırdıkları, bu fonksiyonu bitirdikten sonra hangi fonksiyona geri dönecekleri gibi bilgiler durur. Buna karşılık, lifin üretilmesi esnasında tanımlı olan bütün değişkenler, iki fonksiyon arasında paylaşılmıştır. Bu değerlere de paylaşılmış değişkenler (shared variable) ismi verilir.

## Örnek 2:

```
#include <pthread.h>

#include <stdio.h>

#include <string.h>

void *thread_routine(void* arg){

    printf("Inside newly created thread n");

    return (void*) strdup("Bir dizgi geliyor");

}

void main(){

    pthread_tthread_id;

    void *thread_result=0; pthread_create( & thread_id, NULL,
thread_routine, NULL );

    printf("Ana liftenn");

    pthread_join( thread_id,&thread_result);

    if ( thread_result!= 0 )

        printf("Ana life gelen %sn", thread_result);

}
```

Yukarıdaki yeni kodda, pthread\_join fonksiyonu, bir önceki örnekte olduğu gibi yeni üretilen lifi (thread) beklemek için kullanılmıştır. Bu kullanımda ikinci bir parametre de üretilen yeni liften gelen veriyi almak için [atıf ile çağırma \(call by reference\)](#) şeklinde verilmiştir.

Kodda yeni olan ve koyu renkler ile ifade edilen kısımlara bakılacak olursa, thread\_routine isimli fonksiyon, bir önceki koddan farklı olarak bir değeri döndürmektedir (return). Ayrıca bu fonksiyonun döndürdüğü değer, pthread\_join fonksiyonunun ikinci parametresinde yakalanmıştır. Bu yakalama işlemi sırasında kullanılan değişkenin tipine dikkat edilirse void \* olduğu görülür. Bu değişken tipi, C dilinde tipin belirsiz olması durumunda tercih edilir. Kısaca void \* tipi, tipin belirsizliği anlamındadır. Gelen değer, if kontrolünden geçirilerek bir değer döndürülmesi halinde ekrana ana liften basılmaktadır.

Yukarıda kullanımlarından bahsettiğimiz fonksiyonların detayları aşağıda verilmiştir:

```
int pthread_create( pthread_t *tid, cons pthread_attr_t *attr, void* funk,
void * arg);
```



Yeni bir [lif \(thread\)](#) oluşturmak için kullanılan fonksiyondur. 4 parametre alır. Bunlar sırasıyla, oluşturulacak olan yeni lifin (thread) bilgisini tutan tid, lif oluşumu sırasında verilecek olan özellikler (attr), lifin çalıştıracağı fonksiyon (buradaki parametre bir fonksiyon göstericisi (function pointer) olarak verilmektedir), fonksiyon göstericisi olarak alınan parametreye verilecek olan parametreler. Yani lif, bir fonksiyonu çalıştırırken bu fonksiyona geçirilecek olan parametreler şeklindedir.

Fonksiyon değer olarak bir int döndürür ve şayet int değeri 0 ise başarılı, diğer durumlarda ise hata değerini taşımaktadır.

```
int pthread_exit(pthread_t *tid);
```

Parametre olarak aldığı lif bilgisini bitirir. Lifin çalışması sonlandırılarak sistemden kaldırılır.

```
int pthread_equal(pthread_t *tid1, pthread_t *tid2);
```

Parametre olarak aldığı iki lifi (thread) karşılaştırır ve sonuçta şayet eşitse 0 değilse eşitlik durumuna göre 0 dışında bir değer döndürür. Buradaki karşılaştırma iki lifin içeriğine bakılarak yapılmaktadır ve derin karşılaştırma (deep compare) olarak kabul edilebilir. [Sığ karşılaştırma \(shallow compare\)](#) için == işlemi (operator) kullanılabilir.

```
int pthread_join(pthread_t tid, void ** ptr);
```

Verilen parametrelili lif bitene kadar bekler (join) ve bu lif tarafından döndürülen değeri ikinci parametresi olan ptr ile alır. Buradaki değeri alma işlemi [atıf ile çağırma \(call by reference\)](#).

```
int pthread_detach (pthread_t tid);
```

parametre olarak aldığı lifi hafızadan kaldırır. Bu fonksiyon ile lif sonlandırılmaz sadece hafızadaki kopyası silinir. Bu şekildeki lifler iptal edilemez (cancel) veya beklenemez (join).

```
int pthread_cancel(pthread_t tid);
```

parametre olarak aldığı lifi iptal eder (cancel). Bu işlem sayesinde o anda çalışan life artık ihtiyaç kalmadığı anlatılır ve o anda çalışan liflerin paylaşılmış durumlarını ayarlamaları sağlanır.

```
pthread_t pthread_self();
```

Bu fonksiyon çağrıldığında, değer olarak o andaki çalışan lifin bilgisi döndürülür.

```
int sched_yield();
```

Bu fonksiyon, [zamanlayıcıya \(scheduler\)](#) içinden çağrıldığı lifin çalışma durumundan alınarak bekleme durumuna geçirilmesini söyler. Bu sayede o anda beklemekte olan başka bir lif çalışabilecektir.

### Örnek 3:

```
void *thread_routine(void* arg){  
  
    printf("Lif olustu n");  
  
    sleep( 30 );  
  
    printf("Uykudan sonra n");  
  
}
```



```

void main(){

    pthread_t thread_id; void *thread_result=0;

    pthread_create( & thread_id, NULL, thread_routine, NULL );

    sleep(3);

    printf("Ana lifn");

    pthread_cancel( thread_id);

    printf("Sonn");

}

```

Yukarıdaki yeni kodda, ana lif içerisinde bulunan 3 mili saniyelik uyuma fonksiyonuna karşılık, üretilen lifin çalıştırdığı fonksiyon içerisinde 30 mili saniyelik uyuma fonksiyonu bulunmaktadır. Ayrıca ana lifte bir pthread\_cancel fonksiyonu çağrılmıştır. Bunun anlamı, ana lifin, 3 saniyelik uyuma işleminden sonra ekrana “Ana lif” mesajını basması ve ardından da oluşturulan yeni lifi iptal etmesidir (cancel). Bu esnada üretilen lif, “lif olustu” mesajını basacak ve 30 mili saniye uyuyacaktır. Tahminen ana lif, oluşturulan liften önce uyanacak ve oluşturulan lif henüz ekrana “uykudan sonra” yazmadan araya girecek ve bu lifi sonlandıracaktır. Neticede ekrana bir son yazısı yazılacak ama “ uykudan sonra” yazısı yazılamayacaktır.

#### Örnek 4:

Bu örnekte klasik bir problem olan [üretici / tüketici \(producer /consumer\) probleminin eş farklılık \(mutual exclusion\)](#) kısmını çözeceğiz. Amacımız, birden fazla üretici veya tüketici olması durumunda bu üretici veya tüketicilerden sadece birisinin çalışmasını sağlamaktır. Üretici / tüketici modelinde iki temel problem olduğunu hatırlayınız ( mutex ve progress (ilerleme)), bu kodda, bu problemlerden sadece bir tanesi çözülecektir.

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

```
int shared_data=1;
```

```

void *consumer(void* arg) {

    for(int i =0; i < 30 ; i ++ ){

        pthread_mutex_lock( &mutex);

        shared_data--; /* Critical Section. */

        pthread_mutex_unlock( &mutex);

    }

```

```

    printf("Returning from Comsumer=%dn", shared_data);
}

void main() {

    pthread_t thread_id;

    pthread_create( & thread_id, NULL, consumer, NULL );

    for(int i =0; i < 30 ; i ++ ){

        pthread_mutex_lock( &mutex);

        shared_data++; /* Producer Critical Section. */

        pthread_mutex_unlock( &mutex);

    } /*pthread_exit(0); /* Return from main thread. */

    printf("End of main =%dn", shared_data);

}

```

Yukarıdaki kodda, pthread kütüphanesinde bulunan mutex fonksiyonları kullanılmıştır. Bu fonksiyonlar, kritik alana (critical section) erişimi kontrol altına almışlar ve shared\_data isimli değişkenin arttırım ve azaltım işlemlerini [bölünemez \(atomic\)](#) hale getirmişlerdir. Bu sayede 30 üretim ve 30 tüketim işleminin tamamı [bölünmeden \(atomic\)](#) gerçekleşmiş ve değişken değerleri hatasız olarak değiştirilmiştir.

Kodda görüldüğü üzere mutex kullanımı pthread kütüphanesi içerisinde 3 adımdan oluşur:

- başlangıç (init)
- kilit (lock)
- açma (unlock)

Öncelikle pthread\_mutex\_t [yapısından \(struct\)](#) bir değişken tanımlanır. Yukarıdaki örnekte bu değişkene mutex ismi verilmiştir. Ardından istenildiği kadar bu değişken lock veya unlock fonksiyonları içerisinde çağırılabilir. Kilit fonksiyonunun (lock) çağırılması durumunda artık açma (unlock) fonksiyonu çağırılana kadar başka bir lifin (thread) çalışması engellenir. Bu sayede anlık olarak ilgili satırlar, tek bir [lif \(thread\)](#) tarafından çalıştırılmış olur.

Mutex için pthread kütüphanesinde kullanılan fonksiyonlara bakacak olursak aşağıdaki gibi bir liste çıkarılabilir:

```
pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Bu fonksiyon, yeni bir mutex oluşturulurken kullanılır ve verilen değişken ile ifade edilen ve ikinci parametrede verilen özellikleri taşıyan yeni bir lif mutex oluşturur. Bu değişken daha sonraki kilitleme ve açma fonksiyonlarına da parametre olacaktır.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Verilen parametredeki mutex kilidini siler ve yok eder.

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Verilen parametredeki mutex değişkenine dayanarak bir kilitleme işlemi başlatır. Bu kilitleme işlemi sırasında, aynı değişkeni kullanan tek bir [lif \(thread\)](#) çalışabilir. Aynı değişkenin kilitli olması şartı bulunan diğer bütün lifler (thread) bekletilir. Ancak aynı anda birden fazla değişken ismi kullanılabilir.

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Bu fonksiyonda bulunan parametredeki mutex değişkeni, daha önceden pthread\_mutex\_lock ile kilitlenen kilidi açılır. Böylelikle o anda bu kilidi bekleyen liflerden sadece bir tanesi daha kilidi geçerek kritik alana (critical section) girebilir.

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Yukarıdaki fonksiyon, kilit koymadan kilit konulup konulamayacağını sorgular. Örneğin kilitlenme ihtimali olmayan bir mutex için kilit konulur ve kilit açılana kadar (unlock) lif işlemeye devam eder. Öte yandan zaten kilitli bir lif için kilitleme işlemi yapılmaksızın meşgul mesajı alınır (EBUSY).

Bu fonksiyon aslında oldukça kullanışlıdır. Şöyle bir durum düşünün ki kritik bir iş yapmamız gerekiyor ancak o anda kritik işlem için beklediğimiz mutex kilitlenmiş. Bu durumda kritik işlemi ertelleyerek lifimizde bulunan diğer işleri yapıp sonra geri dönerek kilidin açılıp açılmadığını kontrol edebiliriz. Şayet kilit açıksa kritik alana girer şayet açık değilse yine başka işleri yapmaya devam edebiliriz.

### Örnek 5:

```
pthread_mutex_t read_mutex=PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t write_mutex=PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t qempty_cond_mutex=PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t q_notempty_cond=PTHREAD_COND_INITIALIZER;

pthread_mutex_t qfull_cond_mutex= PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t q_notfull_cond= PTHREAD_COND_INITIALIZER;

void *consumer(void* arg) {

    int i;

    n_consumer++;

    for (i=0; i< ITERATIONS; i++) {

        pthread_mutex_lock( &read_mutex);
```

```

        while ( queue_is_empty() ){

                                pthread_cond_wait(&q_notempty_cond,
&qempty_cond_mutex);

        } /*read from queue[ in ] */

        in = (in +1) % QUEUE_SIZE;

        pthread_mutex_unlock( &read_mutex);

        pthread_cond_signal(&q_notfull_cond);

    }

    printf("Returning from Consumern"); n_consumer--;

}

void *producer(void* arg) {

    int i ;

    for (i=0; n_consumer; i++) {

        pthread_mutex_lock( &write_mutex);

        while ( queue_is_full() ){

            pthread_cond_wait(&q_notfull_cond, &qfull_cond_mutex);

        } /* write to queue[out] */

        out = (out +1) % QUEUE_SIZE;

        pthread_mutex_unlock( &write_mutex);

        pthread_cond_signal(&q_notempty_cond);

    }

    printf("Returning from Producern");

}

int queue_is_empty(){

    if ( in == out ) return 1;

    else return 0 ;

```

```

}

int queue_is_full(){

    if ( in == (out+1 %QUEUE_SIZE) ) return 1;

    else return 0 ;

}

void main() {

    pthread_tthread_id;

    pthread_create(&thread_id,NULL, consumer, NULL);

    pthread_create(&thread_id,NULL, consumer, NULL);

    sleep(5);

    pthread_create(&thread_id,NULL, producer, NULL);

    pthread_create(&thread_id,NULL, producer, NULL);

    pthread_exit(0);

}

```

Yukarıdaki yeni kodda, [üretici tüketici problemi \(producer consumer problem\)](#) iki açıdan da çözülmüştür. Yukarıdaki kodda bulunan mutex fonksiyonları ile aynı anda en fazla tek bir [lif \(thread\)](#) çalışması garantilenirken, koyu renkte gösterilen ve yeni gelen koşullu değişken (conditional variable) uygulaması ile ilerleme (progress) problemi de çözülmüştür. Buna göre problemde şayet üretim için ayrılan alan kalmadıysa üreticilerin durup tüketicilerden en az birisinin bir tüketim yapmasını ve yeni alan açılmasını beklemesi gerekir. Benzer şekilde şayet hiç ürün yoksa, bu durumda da tüketicilerin durup en az bir üreticinin bir ürün üreterek hatta koymasını beklemesi gerekir. İşte bu problemin çözümü için yukarıda görülen boşluk ve doluluk kontrollerini yapan queue\_is\_empty ve queue\_is\_full fonksiyonları çağırılmıştır. Ayrıca mutex problemi oluşturmak için birden fazla üretici ve tüketici lifi (thread) main fonksiyonunda üretilmiştir.

Son olarak mutex kontrollerinden bağımsız olarak üretim hattının dolu olması halinde üretici lifleri durduran bir koşullu değişken (conditional variable) ve hattın boş olması durumunda tüketici lifleri durduran bir koşullu değişken (conditional variable) kodlamaya eklenmiştir.

Kodda dikkat edileceği üzere, in ve out değişkenleri hem mutex hem de cond kormuması altındadır.

Bu kodda kullanılan cond fonksiyonları ise aşağıda açıklanmıştır:

```

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t
*attr);

```

Yukarıdaki fonksiyon sayesinde yeni bir koşullu değişken (conditional variable) tanımlı yapılar ve bundan sonraki fonksiyonlarda kullanılabilir.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Yukarıdaki fonksiyon sayesinde, daha önceden tanımlanmış olan koşullu değişken (conditional variable) sistemden kaldırılarak yok edilir.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Yukarıdaki fonksiyon, basitçe lifin beklemesini sağlayan ve o andaki çalışmayı durdurarak, signal fonksiyonu çağrılana kadar bekleten fonksiyondur.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Bu fonksiyonla, o anda beklemekte olan liflere sinyal yollanır ve uyanmaları sağlanır. Burada önemli bir not, aynı anda çeşitli zamanlama algoritmalarına göre (örneğin FIFO veya RR ) beklemekte olan lifler (thread) bulunuyorsa, bu liflerden en yüksek öneme sahip (priority) lif uyandırılırken, şayet bütün bekleyen lifler aynı öneme sahipse bu durumda rast gelen bir lif uyandırılır.

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
```

Klasik wait fonksiyonundan farklı olarak bir zaman bilgisini parametre alan fonksiyondur. Bu fonksiyonun uyanması için bir sinyal gelmesi veya belirtilen sürenin dolması yeterlidir. Genelde gerçek zamanlı sistemlerde (real time systems) vaz geçilmez fonksiyonlardan birisidir çünkü işlerin ilerlemesi için zaman bilgisi kritik önem taşır.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Bu fonksiyonla o anda parametre olarak alınan bütün beklemeler uyandırılır.

## SORU 5: exec fonksiyonları

Bu yazının amacı, bilgisayar bilimlerinde, özellikle de işletim sistemlerinde kullanılan [exec\(\)](#) [fonksiyon](#) ailesini açıklamaktır. Bu fonksiyon grubu, kabaca bilgisayarımızın dosya sisteminde bulunan farklı bir programı, yazmış olduğumuz C programı içerisinden çağırmaya yarar.

Çalışma durumunu örnek bir kod üzerinden gösterelim:

```
1 #include <unistd.h>
2 main()
3 {
4     execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", (char *) 0);
5 }
```

Yukarıdaki kodda, görüldüğü üzere basit bir main fonksiyonu olan, oldukça küçük bir C kodunda, tek bir satır ile, bir unix komutu olan “ls” komutu çalıştırılmıştır. Bu işlem sırasında exec fonksiyon gruplarından execl fonksiyonu çağrılmıştır.

Programın çıktısı aşağıda verilmiştir:

```
new-host-2:deneme sadievrenseker$ gcc exec.c
new-host-2:deneme sadievrenseker$ ./a.out
total 32
-rw-r--r--  1 sadievrenseker  staff    98 12 Mar 17:22 exec.c
-rwxr-xr-x  1 sadievrenseker  staff  8696 12 Mar 17:23 a.out
```

```
new-host-2:deneme sadievrenseker$
```

Yukarıdaki çıktıda da görüldüğü üzere kod çalıştırıldıktan sonra, “ls” komudu çalıştırılmış ve ekrana bu kodun çıktısı basılmıştır.

Exec grubu fonksiyonlardan bir diğeri de `execv` fonksiyonudur. Bu fonksiyonu kullanan bir örnek kod aşağıda verilmiştir:

```
1 #include <unistd.h>
2 main()
3 {
4     char *args[] = {"/bin/ls", "-r", "-t", "-l", (char *) 0 };
5
6     execv("/bin/ls", args);
7 }
```

Yukarıdaki bu kodun çıktısı da aşağıdaki şekildedir:

```
new-host-2:deneme sadievrenseker$ gcc execv.c
new-host-2:deneme sadievrenseker$ ./a.out
total 40
-rw-r--r--  1 sadievrenseker  staff    98 12 Mar 17:22 exec.c
-rw-r--r--  1 sadievrenseker  staff   117 13 Mar 11:17 execv.c
-rwxr-xr-x  1 sadievrenseker  staff  8720 13 Mar 11:17 a.out
new-host-2:deneme sadievrenseker$
```

Görüldüğü üzere aynı işlem yeni fonksiyonumuz olan `execv` ile de çalıştırılmıştır. Exec grubu fonksiyonlar arasında ufak farklılıklar bulunur ve bu farklılıklar aldıkları parametrelerdir. Parametreler arasındaki farklar ise aşağıda listelenmiştir.

```
int execl(const char * path , const char * arg0 , ..., const char * argn ,
(char *)0);
```

```
int execlp(const char * path , const char * arg0 , ..., const char * argn ,
(char *)0, char *const envp []);
```

```
int execlpe(const char * file , const char * arg0 , ..., const char * argn ,
(char *)0);
```

```
int execlpe(const char * file , const char * arg0 , ..., const char * argn ,
(char *)0, char *const envp []);
```

```
int execv(const char * path , char *const argv []);
```

```
int execve(const char * path , char *const argv [], char *const envp []);
```

```
int execvp(const char * file , char *const argv []);
```

```
int execvpe(const char * file , char *const argv [], char *const envp []);
```

Yukarıdaki fonksiyonlarını parametrelerinde verilen kelimelerin anlamları kısaca şu şekilde:

**path:** programın çalıştırılacağı yol. Bu yolun sonunda programın da ismi yazılmalı. Örneğin `/bin/ls` şeklinde, bin dizini altındaki ls programı.

arg0... argn: bunun anlamı bir liste şeklinde virgül ile ayrılarak programa verilecek olan parametreler yazılabilir. Örneğin execl kodumuzda "-r", "-t", "-l" şeklinde parametreleri vermiştik.

argv[] : programa verilecek olan parametrelerin içinde bulunduğu bir dizi. Örneğin execv kodumuzda bir dizi tanımlayıp içerisine bu parametreleri eklemiştik.

envp[]: ortama özgü değişken göstericisi (environment pointer). Programın çalışacağı ortamı gösteren değişkendir ve bir [bağlı liste](#) mantığı ile çalışarak birbirlerini işaret ederler. Son elemanın gösterdiği değer null olmalıdır.

## SORU 6: fork fonksiyonu ve işlem çatallanması

Bu yazının amacı, bilgisayar bilimlerinin bir çalışma alanı olan işletim sistemlerinde sıklıkla kullanılan ve yeni bir [işlem \(process\)](#) oluşturmaya yarayan fork() ve exec() fonksiyonlarını açıklamaktır. Bu fonksiyonlar C programlama dilleri tarafından desteklenmekte ve unistd.h dosyasının içinde bulunmaktadır.

Örnek bir kod verip çalışmasını açıklayarak konuyu anlatmaya başlayalım:

```
#include <unistd.h> /* fork fonksiyonu için */
#include <sys/types.h> /* pid yapısı için */
#include <stdio.h> /* klasik girdi çıktı */
#include <stdlib.h> /* standart lib fonksiyonları */

int main()
{
    pid_t cocukpid; /* çocuğun pid değeri için */
    int cocukokunan; /* çocuğun döndürdüğü değer */
    int durum; /* çocuğun çıkış durumu */

    /* yeni bir işlem oluşturuyoruz */
    cocukpid = fork();

    if (cocukpid >= 0) /* şayet if sağlanırsa çocuk doğdu */
    {
        if (cocukpid == 0) /* çocuk için pid 0 olur*/
        {
            printf("COCUK: Ben çocuk islemim!\n");
            printf("COCUK: PID: %dn", getpid());
            printf("COCUK: Ata PID: %dn", getppid());
            printf("COCUK: cocukpid: %dn", cocukpid);
            printf("COCUK: Bir deger girin (0 ile 255 arasi): ");
            scanf(" %d", &cocukokunan);
            printf("COCUK: Cocuk oluyor!\n");
            exit(cocukokunan); /* cocuk degeri dondurup öldü */
        }
        else /* fork() ata için pid döndürdü demektir */
        {
            printf("ATA: Ata işlem!\n");
            printf("ATA: Ata PID: %dn", getpid());
            printf("ATA: cocugun pid kopyasi %dn", childpid);
            printf("ATA: cocugun bitmesi icin bekleniyor.n");
            wait(&durum); /*cocugun durumunu bekle */
            printf("ATA: Cocugun cikis kodu: %dn", WEXITSTATUS(durum));
            exit(0); /* parent exits */
        }
    }
}
```



```

}
}
else /* Fork hatasi olduysa -1 döner */
{
perror("fork"); /* hata durumu mesajı */
exit(0);
}
}

```

Yukarıdaki kodda, fork() fonksiyonu çağırılmış ve döndürdüğü değer, cocukpid isimli bir [yapıya \(struct\)](#) döndürülmüştür. Bu [yapı \(struct\)](#) daha önceden tanımlı olan ve sys/types.h dosyasından alınan pid\_t tipindedir. Aslında yapı incelendiğinde basit bir int yapısı olduğu görülür. Ancak işletim sistemi ile programlama sırasında bu şekilde yapılar kullanılması gerekmektedir. Bunun sebebi hem standartlaşma hem de güvenlik olarak düşünülebilir.

Sonuçta fork() fonksiyonu çağırıldıktan sonra dönen değer için iki ihtimal bulunmaktadır. Ya 0'dan büyük ve eşit ya da eksi bir değer olacaktır. Eksi değer dönmesi sadece fonksiyonun hatalı olması anlamındadır. Yani -1 hata kodudur ve bu durumda en altta bulunan else bloğu çalışacak ve hata mesajı ekrana basılacaktır.

Diğer taraftan 0'dan büyük eşit olması durumunda da iki ihtimal bulunmaktadır, ancak burada bir fark vardır. Fonksiyon çağırıldıktan sonra hem 0 hem de 0'dan büyük bir değer döner. Bunu aynı anda iki [işlemin \(process\)](#) çalışması ile açıklamak gerekir:



Yukarıdaki şekilde görüldüğü üzere fork() fonksiyonu çağırıldıktan sonra, iki ayrı işlem olduğu söylenebilir. Bunlardan ilki pid==0, if bloğuna girerken ikincisi pid>0 if bloğuna girer ve çalıştırır.

Yukarıdaki kodda, ayrıca iki if bloğu içerisinde de temel bazı bilgiler ekrana basılmıştır.

Bu basılan değerleri anlamak için örnek bir çalışma aşağıda verilmiştir:

```
SADIs-MacBook-Air:Downloads sadievrenseker$ ./a.out
ATA: Ata işlem!
ATA: Ata PID: 23256
ATA: cocugun pid kopyasi 23257
ATA: cocugun bitmesi için bekleniyor.
COCUK: Ben çocuk islemim!
COCUK: PID: 23257
COCUK: Ata PID: 23256
COCUK: cocukpid: 0
COCUK: Bir deger girin (0 ile 255 arasi): 44
COCUK: Cocuk oluyor!
ATA: Cocugun cikis kodu: 44
SADIs-MacBook-Air:Downloads sadievrenseker$
```

Yukarıda görüldüğü üzere, öncelikle ATA işlem çalışır. Bu işlem o anda işletim sistemi tarafından kendisine atanan işlem numarasını (process id , PID) ekrana basmıştır. Ardından çocuk işlemin PID değerine ulaşarak ekrana basar. Bu basma işlemleri için önce getpid() fonksiyonu çağırılarak mevcut işlemin pid değeri alınır ardından childpid değişkeninin değeri alınarak ekrana basılır. Bu iki değeri bastıktan sonra çocuktan gelecek değer beklenmeye başlanır ( wait fonksiyonu ile).

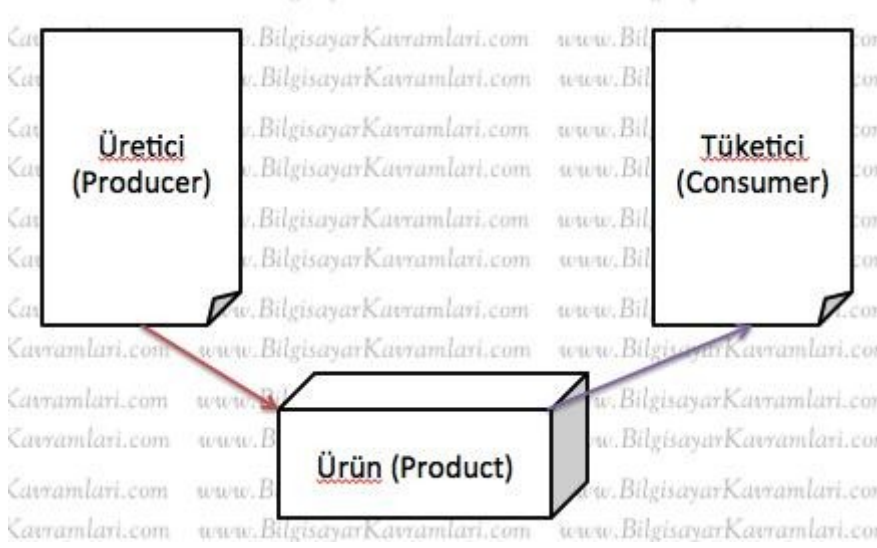
Bu esnada çocuk tarafında, sırasıyla getpid ve getppid fonksiyonları ile, önce çocuğun PID değeri ardından da çocuğun atasının PID değeri (getppid = get parent pid) alınarak ekrana basılır. Çocuk tarafından cocukpid değişken değeri ekrana basıldıktan sonra kullanıcıdan scanf fonksiyonu ile bir girdi okunur. Okunan değer, exit fonksiyonuna parametre verilerek ata işleme geri döndürülür. Bu sırada bekleyen ata işlem gelen kodu alarak ekrana basar. Yukarıdaki örnek çalışmada bu girdi kullanıcı tarafından 44 olarak verilmiştir.

## **SORU 7: Producer Consumer Problem (Üretici Tüketici Problemi)**

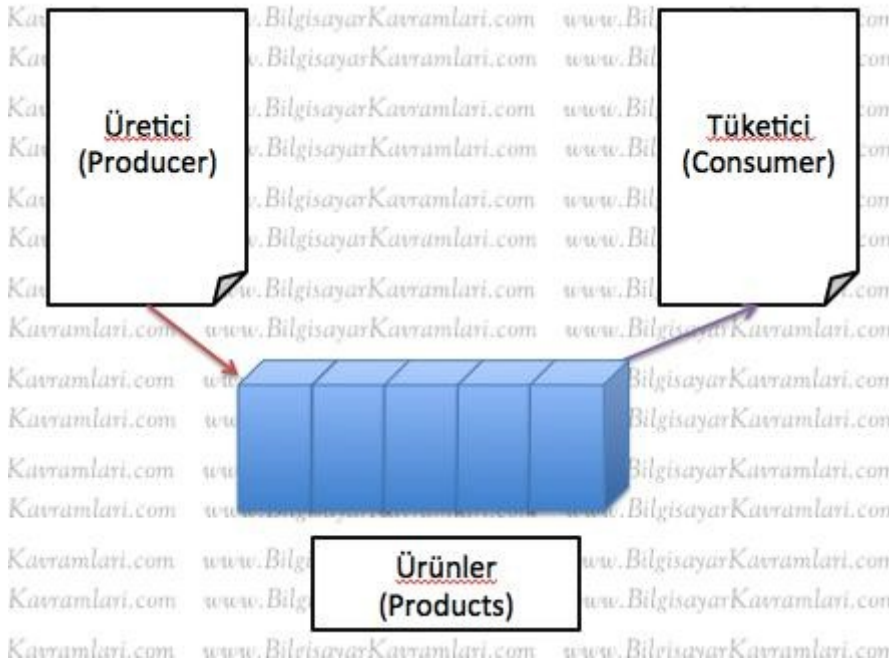
Bu yazının amacı, başta işletim sistemleri olmak üzere, bilgisayar bilimlerinin pek çok alanında geçen ve klasik bir koşut zamanlı (concurrent) problem örneği olan üretici / tüketici (producer / consumer ) örneğini açıklamaktır.

Problemin çeşitli değiştirilmiş tanımları olmasına karşılık, tanımı gayet basittir. Aynı anda çalışan iki [iş \(process\)](#) bulunmaktadır ve bunlardan birisi üretmekte diğeri de bu üretilen ürünü tüketmektedir.

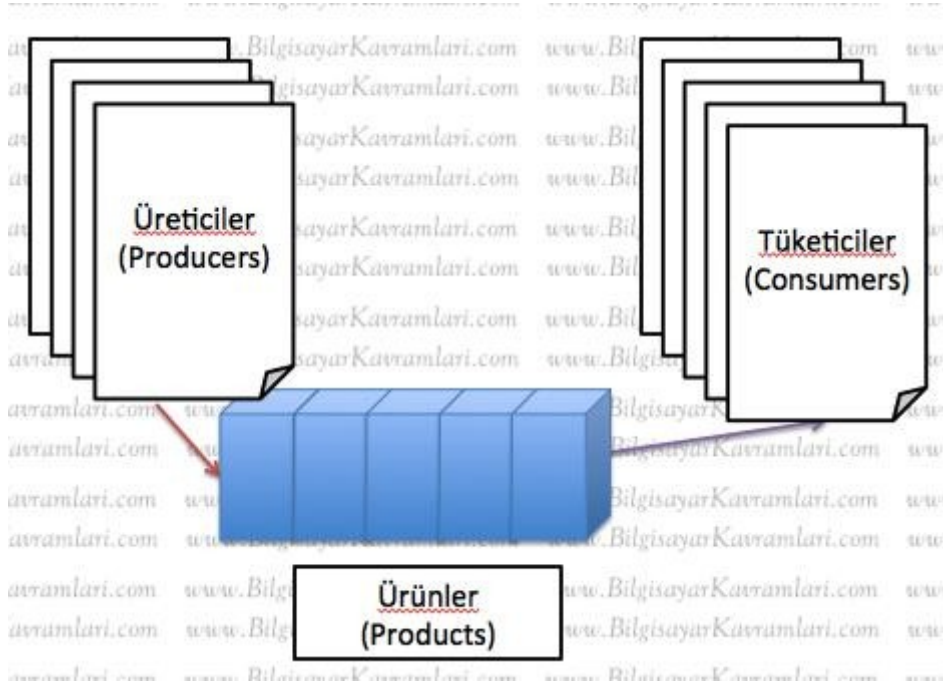
Buradaki sorun, üretilen ürün henüz tüketilmeden ikincisinin üretilmemesi veya henüz ürün üretilmeden tüketilmemesidir.



Problemin biraz değiştirilmiş halinde sınırlı bir bant üzerinde üretim yapılmakta, dolayısıyla üretim işi (producer process) henüz tüketim yapılmadan birden fazla ürün üretebilmektedir. Bu probleme sınırlı bant (bounded buffer) ismi verilmektedir ancak burada da bir sınır bulunmaktadır. Örneğin bant sınırı 10 ise, 10 ürün üretildikten sonra bir tane ürün tüketilmeden yeni ürün üretilmemektedir.



Problemin bir diğer hali de birden fazla üretici ve tüketici olması durumudur. Örneğin aynı bant üzerine aynı anda 3 ayrı üretici ürettikleri ürünleri koyabilmekteler veya aynı anda 5 ayrı tüketici üretilen bu ürünleri tüketebilmektedirler. Elbette bu yeni halinde, aynı ürünü iki tüketicinin tüketMEMesi veya aynı bant aralığına aynı anda iki üreticinin üretim yapmaması gerekir.



Problem tanımı itibariyle, işlemler arası iletişimi gerektirmektedir (interprocess communication, IPC) ve iki işlem birbiri ile iletişime geçerek senkronize olmalıdır. Yani tüketici, ne zaman tüketeyeceğine, üreticinin hareketlerine bakarak karar vermekte, benzer şekilde üretici de ne zaman üreteceğine, tüketicinin hareketlerine bakarak karar vermektedir.

### Yarış Durumu (Race Condition)

Problemin çözümü için bilinmesi gereken en önemli durum yarış durumudur. Buna göre kodlama sırasında iki [işlem \(process\)](#) aynı anda çalışacağına göre, hangisinin yazılan hangi satırı önce yapacağı bilinemez.

### Paylaşılmış Değişken (Shared Variable)

İki işlemin aynı anda erişebildiği, okuyabildiği veya içeriğini değiştirebildiği değişkenlere verilen isimdir. Herhangi bir cinsten olabilir (örneğin int) ve anlık olarak bir işlem erişebilir. Ancak, değişkene erişim, işlemler arası senkronizasyonla kontrol edilmezse anlık olarak kimin eriştiğinin bilinmesi imkansızdır. Kısaca, kim önce gelirse erişir.

### Örnek Çözüm

Aşağıdaki şekilde bir kodlama yapmaya çalışalım (bu kod tam çözüm değildir):

Kodlamak istediğimiz durum, yukarıdaki anlatılan problem çeşitlerinden ilkidir. Yani üretici tek bir ürün üretir ve tüketici de bu ürünü tüketir.

[View Code C](#)

```
1 int sayac = 0;
2 procedure producer() {
3     while (true) {
4         urun = uret();
5
6         if (sayac == LIMIT) {
7             sleep();
```

```

8
9     }
10
11     urunuSun(urun);
12     sayac = sayac + 1;
13
14     if (sayac == 1) {
15         wakeup(consumer);
16     }
17 }
18 }
19
20 procedure consumer() {
21     while (true) {
22
23         if (sayac == 0) {
24             sleep();
25         }
26
27         urun = urunuAl();
28         sayac = sayac - 1;
29
30         if (sayac == LIMIT - 1) {
31             wakeup(producer);
32         }
33
34         tuket(urun);
35     }
36 }

```

Yukarıdaki kodda, yapılmak istenen iki ayrı işlemin de çalışmaları sırasında kullanacakları iki fonksiyonu kodlamaktır. Buna göre üretici işlemi producer() fonksiyonunu çalıştırırken, tüketicisi işlemi consumer() fonksiyonunu çalıştıracaktır.

Dikkat edileceği üzere iki işlem de sonsuz döngüdedir, yani while(true) döngüsü altında sonsuza kadar dönmektedir, amacımız sonsuza kadar üreticinin ürünü üretip sunması, tüketicinin de tüketmesidir.

Basitçe bir üretici, sayaç isimli paylaşılmış değişkeni (shared variable) kullanarak tüketicisi ile iletişime geçer. Şayet sayaç değeri limite ulaşırsa, daha fazla üretim yapılması mümkün değil demektir ve bu durumda üretici beklemelidir. Şayet sayaç değeri 0'a ulaşırsa, bu durumda da daha fazla tüketim yapılması mümkün değildir ve tüketicisi beklemelidir.

Bu çözümde ölümcül kilitlenme olma ihtimali bulunduğu için tam çözüm olarak kabul edilmesi mümkün değildir.

Örneğin aşağıdaki senaryoyu ele alalım:

1. Tüketicisi tarafından sayaç değişkeni okundu ve değeri 0 olduğu için bekleme yapılan if bloğuna girildi.
2. Tüketicisi if bloğuna girdiği anda (ancak henüz uyumadan yani sleep() fonksiyonu çağırılmadan) üretici tarafı da çalıştı ve bir ürün üretip banta koydu.
3. Tüketicinin ürün üretmesi sırasında sayaç değeri de arttırıldı
4. Ayrıca tüketicisi, kendi sırası bitip sıranın üreticiye geçmesinden dolayı, üreticiyi de uyandırdı (üretici henüz uyumamıştı dolayısıyla boşuna yapılan bir uyandırma)

5. Ardından tüketici beklemekte olduğu sleep() fonksiyonuna girdi ve üretici de sayaç değeri 1 olduğu için uyuma durumuna geçti

Sonuç olarak iki işlem de uyumaktadır. Bu durumda bir [ebedi kilitleme \(deadlock\)](#) oluşacak ve işlemler asla çalışmayacaktır.

### Semaphore Kullanarak Çözüm:

Yukarıdaki çözümde kullanılan paylaşılmış değişken yaklaşımını bu sefer, işletim sistemlerindeki bir senkronizasyon yöntemi olan [semaforları](#) kullanarak çözmeye çalışalım.

Temel olarak [bir semafor \(ilgili yazıyı da okuyabilirsiniz\)](#) iki işlem arasında bir trafik ışığı benzeri şekilde davranarak anlık olarak bir tanesinin çalışmasını sağlar.

Semaforların üzerinde çalıştığı değişkenler bulunur ve bu değişkenlere erişim anlık olarak semafor yapısı tarafından kontrol altına alınmıştır. Semaforların iki temel fonksiyonu wait (bekle) ve signal (geç) şeklindedir. Trafik ışıklarının kırmızı (wait, bekle) ve yeşil (geç, signal) olması şeklinde düşünülebilir.

Yazının başında anlatılan 3 tip problemden, üçüncüsünün, yani birden fazla üretici ve tüketici olması durumunun, semafor olarak çözümü aşağıda verilmiştir (tam çözüm değildir):

[?View Code C](#)

```
1
2 semaphore doluluk = 0;
3 semaphore bosluk = LIMIT;
4
5 procedure producer() {
6     while (true) {
7         item = produceItem();
8         bekle(bosluk);
9         urunuKoy(item);
10        gec(doluluk);
11    }
12 }
13
14 procedure consumer() {
15     while (true) {
16         bekle(doluluk);
17         urun = urunuAl();
18         gec(bosluk);
19         tuket(urun);
20    }
21 }
```

Yukarıdaki çözümde, birden fazla işlem bir öncekine benzer şekilde aynı anda çalışmakta ve bu sefer doluluk ve boşluk isimleri verilen iki farklı semafor kullanılmaktadır. Basitçe, doluluk semaforu, üreticiyi bekletmekte, boşluk semaforu da tüketiciyi bekletmektedir (hat boşken tüketilemediğini ve hat doluyken üretilmediğini hatırlayınız).

Yukarıdaki kodda, daha sonra anlatılacak olan [eşbağımsızlık \(mutually exclusive\)](#) problemi bulunmaktadır. Bu problemi aşağıdaki adımların gerçekleştiği senaryoda görebiliriz:

1. İki üretici aynı anda boşluk değişkenini azaltır (bekle fonksiyonunu çağırır)
2. Üreticilerden sadece birisi üretim hattındaki boşluğu bulup ürünü koyar.

3. İkinci üretici, ürettiği ürünü, üretim hattına yerleştirmek istediğinde, birinci üretici tarafından zaten yerleştirilmiş olduğu için problem oluşur. Bu aşamada kodlanan dile göre, mevcut ürünün üzerine yazılır şeklinde düşünebiliriz.

Netice olarak iki üretici de aynı alana ürün koymu ve neticede iki üretici çalışmış ve sanki iki ürün üretilmiş gibi semafor değeri değiştirilmiş buna karşılık sadece bir ürün üretilmiştir. Bu durum tüketim sırasında sorunlara yol açacaktır.

Bu problemin çözümü için aynı cinsten, birden fazla [işlem \(process\)](#) olması durumlarının kontrol edilmesi gerekir. Bu kontrole [eşbağımsızlık \(mutual exclusion\)](#) ismi verilir ve aşağıdaki şekilde çözülebilir:

[View Code C](#)

```
1 semaphore mutex = 1;
2 semaphore doluluk = 0; // items produced
3 semaphore bosluk = LIMIT; // remaining space
4
5 procedure producer() {
6     while (true) {
7         item = produceItem();
8         bekle(bosluk);
9         bekle(mutex);
10        urunuKoy(item);
11        gec(mutex);
12        gec(doluluk);
13    }
14 }
15
16 procedure consumer() {
17     while (true) {
18         bekle(doluluk);
19         bekle(mutex);
20         urun = urunuAl();
21         gec(mutex);
22         gec(bosluk);
23         tuket(urun);
24     }
25 }
```

Yukarıdaki yeni çözümde, işlemler arasında (üretici <-> tüketici) eşitleme olduğu kadar, aynı cinsten işlemler arasında da eşitleme yapılmıştır.

mutex: Bu semafor, anlık olarak tek işlemin çalışmasını sağlar. Peki zaten tek işlem çalışıyorsa ilave semafora ne gerek var? Diyebilirsiniz ancak tek işlem çalışması, ürün yokken tüketilmesini veya hat doluyken üretilmesini engellemez.

Kısacası yukarıdaki yeni çözüm, bir öncekindeki üretici / tüketici eş zamanlamasına ilave olarak aynı anda birden fazla üretici ve tüketicinin de çalışmasını engellemektedir.

## **SORU 8: Filozofların Akşam Yemeği (Dining Philosophers)**

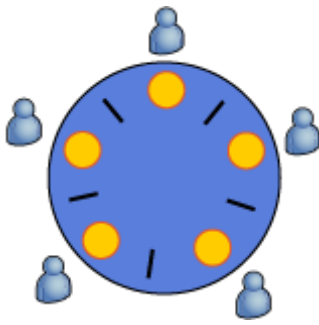
Bu yazının amacı, eş zamanlı işlemlerin (concurrent processes) yönetimini anlatmak için geliştirilmiş bir örnek olan yemek yiyen filozoflar konusunu açıklamaktır. Eş zamanlı



işlemler, işletim sistemleri (operating systems), ağ programlama (network programming) gibi pek çok bilgisayar bilimi konusunda geçmektedir.

Yemek yiyen filozoflar örneği, literatüre Dijkstra tarafından kazandırılmıştır ve eş zamanlı işlem yönetimini (concurrent process management) sembolize eder.

Öncelikle örneği anlatarak konuya başlayalım. Örneğe göre, ikiden fazla ( örnek için n kadar kabul edilebilir) filozof, bir yuvarlak masanın etrafına dizilerek yemek yerler. Literatürde örnek iki şekilde anlatılmaktadır ve orijinalinde filozoflar pirinç pilavı yemektedir. Buna göre pirinç yemek için iki adet yemek sopası (chopstick) gerekmektedir. Çinlilerin yemek yerken kullandıkları sopaları düşünebilirsiniz. Tek sopa ile yemek yenmesi imkansızdır ve her filozofun en az iki sopaya ihtiyacı vardır. Olayın daha iyi anlaşılması için aşağıdaki şekilde bu durum tasvir edilmiştir:



Şekilde de görüldüğü üzere, başlangıç durumunda, her filozofun iki yanında birer sopa durmaktadır.

Örneğin biraz daha batı dünyasına uyarlanmış halinde ( buna göre batıda çatal kullanılmaktadır), her filozof makarna yemek ister ancak makarna yemek için en az iki çatala ihtiyaç vardır. Filozofların iki yanında birer çatal olduğuna göre problem bir önceki pirinç ve Çinlilerin yemek sopası probleminden bir farkı kalmaz.

Problemde, yukarıda anlatılanlara ilave olarak, filozofların belirli bir süre düşünme süreci bulunur.

Problemde buna göre filozofların hepsi örneğin sağındaki sopayı alırsa, hepsinde birer sopa olacak ve yemek yiyemeyeceklerdir. Şayet hepsi iki yanındakini birden almaya kalkarsa, bu durumda, eş zamanlı işlemlerde karşılaşılan yarış durumu (racing condition) ortaya çıkacaktır ve hangisi önce davranırsa (ki bu konuda bir garantimiz bulunmamaktadır) o yemeğini yiyebilecektir. Ve belki de hepsi birer sopa alacağı için yine hiçbirisi yemek yiyemeyecektir. Şayet hepsi birden sopalarını bir diğeri yesin diye bırakırsa, bu durumda yine hiçbirisi yiyemeyecektir. Bu tip problemler, genelde [kıtık problemi \(starvation\)](#) olarak düşünülebilir. Buna göre her filozofun yemek yeme ihtimali bulunmaktadır ancak hiçbir şekilde yiyeceği garanti edilmemektedir. Örneğin filozoflardan birisi her durumda aç kalabilir ve asla sıra kendisine gelmeyebilir.

Problemde karşılaşılan diğer bir sorun ise ölümcül kilitlenmedir (deadlock). Yanlış bir tasarım sonucunda, tek çatal alan ve çatalı bırakmak için diğer filozofun bırakmasını bekleyen bir filozof sistemi kilitleyebilir. Bu da problemde bulunan ikinci risktir.



Son olarak problemin tanımında, filozoflar birbiri ile konuşamaz kuralı getirilmiştir. Bu kural önemli bir kural olmakla birlikte, aşağıdaki çözümlerin çoğunda bu kuralın ihlal edildiği görülebilir. Aslında filozoflar birbiriyle çatalar üzerinden iletişim kurmaktadır. Örneğin sağındaki veya solundaki filozofun o anda çatalı alıp almaması, yanındaki filozoflar hakkında bilgi vermekte ve bu da üstü kapalı bir iletişim olarak kabul edilmektedir. Aşağıdaki çözümlerin tamamında iletişim sadece çatalların durumuna göre sağlanmaktayken, sadece son çözüm olan chandy misra çözümünde, filozoflar doğrudan birbiri ile iletişime geçebilmektedir.

Problemin çözümü için farklı algoritmalar geliştirilmiştir. Bu algoritmalar aşağıda başlıklar altında anlatılacaktır.

## 1. Rastgele süre çözümü (Random Solution)

Bu çözümde, filozofların problemi çözmek için tamamen rastgele davranması öngörülür. Filozoflar, bir çatal aldıktan sonra ikincisini alabilirse yemeğini yer. Şayet ikinci çatalı alamazsa rastgele bir süre bekler ve bu süre içinde ikinci çatalın boşalmasını bekler. Şayet bu süre içerisinde diğer çatal, yanındaki filozof tarafından bırakılırsa, çatalı alır ve yemeğini yer. Şayet beklediği bu rastgele süre boyunca ikinci çatal bırakılmazsa bu durumda yemeğini yiyemeden elinde tuttuğu çatalı diğer filozofun yemesi için masaya geri bırakır.

Bu çözümde dikkat edileceği üzere, işlem tamamen rastgelelik üzerine kuruludur. Buna göre sistem tam başarı ile çalışmayabilir. Hatta sistemin çalışacağının hiçbir şekilde garantisi yoktur.

Çözüm yönteminin en önemli özelliği, çözümün gerçekleşmesinin (kodlamasının) diğer bütün sistemlere göre çok daha kolay olmasıdır.

## 2. Garson çözümü (Conductor Solution)

Problemin bir seviye daha karmaşık ancak yine de basit bir çözümü, masanın etrafında bir garsonun (literatürde conductor ismi verilmiştir ve bu yüzden conductor solution olarak geçer) dolaşmasıdır.

Garson, sürekli olarak masada boş duran ve filozoflar tarafından yemek için kullanılan çatalların sayılarını takip etmektedir. Bir şekilde her filozof, masadan çatal alabilmek için garsonun iznini istemek zorundadır. Şayet garson izin vermezse filozof masadan çatal alamaz. Bu çözümde filozofların [kıtlık problemi \(starvation\)](#) ile karşılaşmaları engellenir çünkü mantıklı bir garson tasarımı, bütün filozoflara yemek imkanı tanır. Aynı zamanda [ölümcül kilitleme \(deadlock\)](#) ihtimali de çözülmüştür çünkü garson hiçbir filozofu sonsuza kadar bekletmez. Yani filozofların birbirini bekleyerek sonsuza kadar yaşlanması sorunu çözülmüştür.

Çözümün daha iyi anlaşılabilmesi için, garsonun, saat yönünde masada döndüğünü, düşünelim. O anda işaretlediği filozof yemek yiyor, sonraki yemiyor sonraki yiyor ve böylece kaç filozof farsa, sırayla bir yiyor bir yemiyor şeklinde düşünülebilir. Bu durumda her filozofun yemek yemek için yeterli çatalı (veya sopası) bulunuyor demektir. Sonra garson, sırasıyla bir yönde (örneğin saat yönünde) dönerek masayı dolaşmakta ve sıradaki filozofa yemek yedirmekte (ve dolayısıyla sıradaki filozoftan sonraki yememekte ve sonraki yemekte ve böylece bütün masadakiler bir yer bir yemez şeklinde işaretlenmektedir).

### 3. Monitör Çözümü (Monitor Solution)

Bu çözüm, bir önceki garson çözümüne çok benzemektedir. Amaç sırasıyla her filozofun bir yiyen bir de yemeyen şeklinde sıralanmasıdır. Burada her filozof belirli bir sırayla sıralanmaktadır (örneğin saat yönünde veya saat yönünün tersi istikamette) ardından kendinden önceki filozofun durumunu kontrol ederek yemek yiyorsa yemez, kendinden önceki filozof yemek yemiyorsa bu durumda kendisi yemek yer.

Tek sayıda filozof olması durumunda yemek yeme eyleminin masada bir dalga şeklinde bir noktadan başlayarak sürekli döndüğü görülebilir. Şayet filozof sayısı çift ise bu durumda sürekli aynı filozoflar yemek yerken diğer filozoflar ölecektir. Bir çözüm olarak, şayet toplam filozof sayısı çift ise, sırasıyla tek ve çift filozoflara yemek yedirmek bir çözüm olabilir. Örneğin önce 1,3,5 numaralı filozoflar yemek yerken, sonra 2,4,6 numaralı filozoflar yemek yiyebilir.

### 4. Chandy Misra Çözümü (Chandy Misra Solution)

Bu çözüm, geliştiren iki kişinin ismi ile anılmaktadır (K. Mani Chandy ve J. Misra). Çözümün en önemli özelliği, merkezi bir karar mekanizmasını ortadan kaldırması ancak buna karşılık, filozoflar birbiri ile konuşamaz kuralını çiğnemesidir.

Çözüm aşağıdaki 4 adımdan oluşmaktadır denilebilir:

1. Her filozof ikilisi için bir çatal üretilir ve bu çatal en düşük sayı sahibi olan filozofa verilir. Her çatal kirli veya temiz olarak işaretlenebilir ve başlangıç durumunda bütün çatalar kirlidir.
2. Bir filozof, bir kaynak kümesini kullanmak istediğinde (yani yemek yemek istediğinde), komşusu olan çatalı kullanmak zorundadır. Elinde olmayan (ihtiyacı olan) bütün çatalara bir talep yollar.
3. Bir filozof, elindeki bir çatal için talep aldığında, şayet elindeki çatal temizse kullanmaya devam eder, şayet çatal kirli ise, çatalı masaya koyar. Ayrıca masaya konan çatal temiz olarak işaretlenerek konulur.
4. Bir filozof yemek yedikten sonra çatalı kirli olarak işaretler. Şayet bir filozof, daha önce bir çatalı talep ettiyse, çatalı temizleyerek masaya koyar.

Yukarıdaki algoritma adımlarını şu şekilde anlamak mümkündür. Öncelikle bütün çatalara bir işaret getiriliyor. Buna göre bir çatal, kirli veya temiz olabiliyor. Çatalın talep edilmesi ve edilmemesi arasındaki fark, bu işaret ile belirleniyor. Üzerinde talep olmayan çatalar temiz olarak tutulurken, üzerinde bir filozofun talebi bulunması halinde kirli oluyor. Bu durumda ikinci bir filozofun talepte bulunması engelleniyor. Her çatal sadece talep eden filozof tarafından kullanılacağı için de kullanma işlemi öncesinde tek bir filozofa atama yapılmış oluyor. Ayrıca filozofların talep işleminin gerçekleşebilmesi için çatal kümesinin (iki çatalın birden) atamasının yapılması gerekmektedir. Şayet tek bir çatal ataması yapılırsa bu durumda çatal filozofa ayrılmadan diğer filozofun kullanımı için serbest olmuş oluyor.

Burada akla, acaba bu çatal ayırımı sırasında, problemin orijinalinde bulunan kilitlenme ihtimali bulunmaz mı? Şeklinde bir soru gelebilir. Bu soru aslında mantıklı bir soru olmakla birlikte, kilitlenme probleminin önüne geçmek için, algoritma tasarımında bulunan en küçük numaraya sahip filozof koşulu getirilmiştir. Yani filozofların hepsi aynı önceliğe sahip olduğu durumlarda bir kilitlenme ihtimali bulunmakla birlikte, bu ihtimali bertaraf etmek için her

filozofa bir numara verilmiş ve bu numaraya göre en düşük değere sahip filozof öncelikli olmuştur. Peki sürekli bu filozof yemek yiyerek diğerlerini bir kıtlığa sokabilir mi? Bu durumda kirli ve temiz çatal ataması ile engellenmiştir.

### **SORU 9: Banker Algoritması (Banker's Algorithm)**

Bilgisayar bilimlerinde işletim sistemi tasarımı konusunda geçen ve kaynaklar üzerindeki kilitlenmeyi (deadlock)engelleme amaçlı algoritmadır. Algoritma Dijkstra tarafından geliştirilmiştir.

Algoritmanın temel 3 durumu ve 2 şartı bulunur:

Bilmesi gerekenler:

1. Her işlem (process) ne kadar kaynağa ihtiyaç duyar?
2. Her işlem (process) şu anda ne kadar kaynağı elinde tutmaktadır?
3. Şu anda ne kadar kaynak ulaşılabilir durumdadır?

Yukarıdaki bu bilgileri bildikten sonra kaynak ayrılması sırasında aşağıdaki şartları uygular:

1. Şayet talep edilen kaynak, azami kaynaktan (maximum) fazla ise izin verme
2. Şayet talep edilen kaynak eldeki kaynaktan fazla ise, kaynak boşalana kadar işlemi (process) beklet.

Yukarıdaki algoritma detayında, kaynak olarak geçen değer, işletim sistemi için herhangi bir şey olabilir (hafıza (RAM), giriş çıkış işlemleri (I/O), gerçek sistemler için çalışma zamanı gibi)

Banker algoritması ayrıca çalışması sırasında yukarıdaki takipleri yapabilmek için bazı veri yapılarına ihtiyaç duyar. Aşağıdaki veri yapıları için  $n$ , sistemdeki işlem (process) sayısı ve  $m$  birbirinden farklı kaynak sayısı (resource) olmak üzere:

Müsait :  $m$  adet elemanı olan bir dizidir. Dizinin her elemanı o kaynak tipinden ne kadar müsait olduğunu tutar. Örneğin  $M[i] = 5$  değerinin anlamı,  $i$  kaynak tipinden beşinin müsait olduğudur.

Azami : iki boyutlu dizi ile tutulur ve  $n \times m$  boyutlarındadır. Her işlem için ilgili kaynaktan ne kadar ayırım yapıldığı dizide işaretlenir. Örneğin  $Az[2][3]=5$  gösteriminin anlamı, 2. işlemin 3. kaynak üzerinde 5 birimlik ayırım hakkı olduğudur. Diğer bir deyişle 2. işlem, 3. kaynaktan 5 birimden fazla kullanmaz, kullanmak istemez, kullanamaz.

Ayırım : yine iki boyutlu bir dizidir ve yine boyutu  $n \times m$  olarak tutulur. Her işlemin her kaynağın ne kadarını kullandığını gösterir. Örneğin  $Ay[2][3] = 4$  gösteriminin anlamı, o anda, 2. işlemin 3. kaynaktan 4 birim kullanıyor olduğudur (ayırılmış olduğudur).

İhtiyaç: Benzer şekilde  $n \times m$  boyutlarında bir dizi olarak tutulur ve her işlemin her kaynaktan ne kadar ihtiyaç duyduğunu tutar. Örneğin  $I[2][3] = 1$  gösteriminin anlamı, 2. işlemin, 3. kaynaktan 1 birim ihtiyacı olduğudur.

Algoritmanın çıktısı güvenli veya değildir (safe state, unsafe state). Buna göre algoritma, işlemlerin çalışıp bitme ihtimali varsa güvenli sonucunu döndürürken, işlemler, birbirini kilitliyorsa bu durumda güvensiz sonucunu döndürür.

Hesaplama:

Algoritma, güveni veya güvensiz sonucuna ulaşmak için adıma bağlı olarak, aşağıdaki hesaplamaları yapar:

$$\text{Müsait} = \text{Müsait} - \text{İhtiyaç}$$

$$\text{Ayrım} = \text{Ayrım} + \text{İhtiyaç}$$

$$\text{İhtiyaç} = \text{Azami} - \text{Ayrım}$$

Bu durumu bir örnek üzerinden açıklayalım. Örneğin A,B,C isimli üç kaynağımız olsun ve bu kaynakların müsaitlik durumları aşağıdaki şekilde olsun:

Kaynaklar

	A	B	C
10	5	7	

Ayrıca 5 adet işlemimiz olsun ve bu işlemlerin anlık olarak azami ihtiyaçları, ayrılmış olan kaynaklar ve ihtiyaçları aşağıda verildiği gibi olsun:

	Azami			Ayrım		
	A	B	C	A	B	C
P0	7	5	3	0	1	0
P1	3	2	2	2	0	0
P2	9	0	2	3	0	2
P3	2	2	2	2	1	1
P4	4	3	3	0	0	2

Soru bu durumun güvenli bir durum olup olmadığıdır. Diğer bir deyişle acaba bir kilitlenme riski bulunur mu? Öncelikle yukarıdaki ayrım tablosunda bulunan kaynakları toplayalım:

$$\text{A kaynağı için : } 0 + 2 + 3 + 2 + 0 = 7$$

$$\text{B kaynağı için : } 1 + 0 + 0 + 1 + 0 = 2$$

$$\text{C kaynağı için : } 0 + 0 + 2 + 1 + 2 = 5$$

yukarıdaki toplama işlemleri, ilgili kaynaktan, başlangıçta ayrılmış miktarların toplamıdır ve kaynağın bulunduğu kolonun toplanması ile bulunabilir.

Şimdi başlangıçtaki kaynak miktarından bu toplamı çıkararak başlangıçta bir işlemin çalışması için müsait kaynakları bulalım:

$$\text{A : } 10 - 7 = 3$$

$$B : 5 - 2 = 3$$

$$C: 7 - 5 = 2$$

Bu kaynaklarla başlayarak acaba sistem kilitlenme olmadan çalışabilir mi?

Yukarıda verilen değerlerin üzerinden ihtiyaç tablomuzu hesaplayalım:

	Azami	-	Ayrım	=	İhtiyaç
	A B C		A B C		A B C
P0	7 5 3		0 1 0		7 4 3
P1	3 2 2		2 0 0		1 2 2
P2	9 0 2		3 0 2		6 0 0
P3	2 2 2		2 1 1		0 1 1
P4	4 3 3		0 0 2		4 3 1

Yukarıdaki işlemden anlaşılacağı üzere Azami tablosundan, Ayrım tablosu çıkarılmış ve ihtiyaçlar bulunmuştur.

Bu durumda, sistem güvenli denilebilir çünkü ihtiyaç tablosundaki hiçbir değer, müsait dizimizdeki değeri geçmemektedir. Demek ki sistem bütün ihtiyaçlara cevap verebilecek kapasitededir.

Bu durumun daha iyi anlaşılması için P1, P2, P3, P4, P0 sırası ile çalışmayı görelim:

	Azami	-	Ayrım	=	İhtiyaç	Müsait	
	A B C		A B C		A B C	A B C	
P1	3 2 2		2 0 0		1 2 2	3 3 2	3 3 2
P3	2 2 2		2 1 1		0 1 1	5 3 2	
P4	4 3 3		0 0 2		4 3 1	7 4 3	
P2	9 0 2		3 0 2		6 0 0	7 4 5	
P0	7 5 3		0 1 0		7 4 3	10 4 7	
						10 5 7	<<< Kaynaklar

Görüldüğü üzere işlemlerin sonucunda ilk kaynak değerlerine geri dönmüştür (aslında bütün işlemler bitince müsait olan kaynak değeri, başlangıçtaki değerdir). Yukarıdaki çalışma sırası (yani P1, P2, P3, P4, P0 sırası) güvenli çalışma olarak kabul edilir. Örneğimizi biraz daha ilerletelim : Acaba P1 işlemi (1,0,2) ihtiyacında olsaydı yine de güvenli durumda olur muyduk? İhtiyaç kontrolümüz (1,0,2)<= (1,2,2) olacaktı ve müsait durumumuz : (1,0,2)<= (3 3 2) olacaktı Bu durumda tablomuz aşağıdaki şekilde olabilirdi:

	Azami	Ayrım	İhtiyaç	Müsait
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	2 3 0<<<
P1	3 2 2	3 0 2<<<	0 2 0<<<	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

Bu durum güvenli midir? Evet !

Çalışma sırası olarak P1, P3, P4, P0 ve P2 sırasında olması durumunda işlemler kilitlenme olmaksızın başarı ile çalışacaktır. Örneğin P4 için ihtiyacı (3,3,0) olarak yeniden ayarlasaydık

güvenli bir durum elde edemeyecektik çünkü bu adımdaki (3,3,0) değeri o anda müsait olan (2,3,0) değerinden büyük olacaktı ve işlem (process) çalışamayacaktı.

### SORU 10: Peterson's Algorithm

Bilgisayar mühendisliğinde, birbirinde ayrı işlerin kontrolü için kullanılır. Aynı anda çalışan [işlerin birbirinden tamamen ayrı olması için \(mutually exclusive\)](#), bazı kontrollerin yapılması gerekmektedir. Algoritma bu problemi aşağıdaki şekilde çözer.

```
1 // P0 için
2 flag[0] = 1;
3 turn = 1;
4 // bariyer kodlaması
5 while (flag[1] == 1 && turn == 1);
6 // kritik alan
7 // www.bilgisayarkavramlari.com
8 // critical section
9 ...
10 flag[0] = 0;
```

Yukarıdaki kodda, iki adet paylaşılmış değişken (shared variable) kullanılmış ve bu değişkenlere hem P0 hem de P1'in erişebileceği kabul edilmiştir. İki [işlem \(process\)](#) arasında senkronizasyonu sağlamak için kullanılan bu flag ve turn değişkenleri, P1 için aşağıdaki şekilde kontrol edilir.

```
1 // P1 için
2 flag[1] = 1;
3 turn = 0;
4 // bariyer kodlaması
5 while (flag[0] == 1 && turn == 0);
6 // kritik alan
7 // www.bilgisayarkavramlari.com
8 // critical section
9 ...
10 flag[1] = 0;
```

Dikkat edilirse, P0 için yazılan kod, turn değişkeninin değeri 1 iken beklemekte ve P1 için yazılan kod ise turn değişkeni 0 iken beklemektedir. Bekleme işlemleri kodların 5. Satırlarında yapılmaktadır. Bekleme işlemi için kontrol edilen ikinci bir değer ise flag değişkeninin değeridir. Flag değişkeni, çalışmakta olan işlemlerin kritik alana girmek istememeleri durumunda boşuna beklemelerini engeller. Örneğin P0, kritik alana girmek istemiyorsa bu durumda P1'in kritik alana girmek için bir kontrol yapmasına gerek yoktur çünkü zaten kritik alana erişecek başka bir işlem de yoktur.

Sistemde [kilitlenme \(deadlock\)](#) ihtimali bulunmaz. Bunun sebebi turn isimli değişkenin anlık olarak 1 veya 0 değerlerinden birisine sahip olacağıdır. Bu değer, hangi işlemin çalışacağını belirler dolayısıyla aynı anda iki işlemin bekleme veya çalışma ihtimali yoktur.

Buradaki bekleme işlemi meşgul bekleme (busy wait) olarak kodlanmıştır. Yani değişkenlerin değerleri kontrol edilerek döngü sürekli olarak dönmektedir. Değişken değerleri, diğer işlem (process) tarafından değiştirildiğinde döngü kırılır ve kritik alan başlar. Bir [işlem \(process\)](#) değişken değerini değiştirip, beklemede olan diğer işlemin çalışabileceği sinyali verirken

aslında kendisinin çalışmayacağını da kodlamış olur. Dikkat edilirse, koddaki `turn==0` kontrolü yapılmadan önce `turn = 0` veya benzer şekilde `turn==1` kontrolü yapılmadan önce `turn=1` ataması yapılmaktadır.

### SORU 11: Dekker's Algorithm

Bilgisayar mühendisliğinde, birbirinde ayrı işlerin kontrolü için kullanılır. Aynı anda çalışan işlerin birbirinden tamamen ayrı olması için (mutually exclusive), bazı kontrollerin yapılması gerekmektedir. Algoritma bu problemi aşağıdaki şekilde çözer.

```
2 //Bariyer kodlaması
3 flag[0] = true
4 while (flag[1] == true) {
5     if (turn != 0) {
6         flag[0] = false
7         while (turn != 0) {
8             }
9         flag[0] = true
10    }
11 }
12 // kritik alan kodlaması
13 // www.bilgisayarkavramlari.com
14 // critical section
15 ...
16 turn = 1
17 flag[0] = false
18 // kritik alan dışındaki kodlama
```

Yukarıdaki kodda görüldüğü üzere, kod üç kısımdan oluşmaktadır:

1. Bariyer
2. Kritik alan
3. Bitiş

Bariyer kodunda, işlem (process), diğer çalışmakta olan ve o anda kritik bir koda erişen işlemleri bekler. Kendi önündeki bariyerin kalkmasıyla kritik alana girer. Bu alana o anda giren tek işlemdir çünkü kendi önündeki bariyerin kalkması ancak diğer işlemin bitiş kodunu çalıştırması ile mümkündür. Kritik alandaki görevini yerine getirdikten sonra bitiş kodunu çalıştırır ve beklemeden olan diğer işlemin bariyerlerini kaldırır.

Yukarıdaki kodda kullanılan flag dizisi (array), boolean tipinde olup o anda hangi işlemin çalıştığını tutmaktadır. Örneğin yukarıdaki kodun simetriği olarak diğer işlemin kodu aşağıdaki şekilde verilebilir:

```

2 //Bariyer kodlaması
3 flag[1] = true
4 while (flag[0] == true) {
5     if (turn != 1) {
6         flag[1] = false
7         while (turn != 1) {
8             }
9         flag[1] = true
10    }
11 }
12
13 // kritik alan kodlaması
14 // www.bilgisayarkavramlari.com
15 // critical section
16 ...
17 turn = 0
18 flag[1] = false
19 // kritik alan dışındaki kodlama

```

İkinci kod ile ilk kod karşılaştırılırsa, flag[] ve turn değişkenlerine göre kimin çalışacağı belirlenmektedir.

Kodlarda bulunan 7. Satırdaki while döngüsü, aslında bekleme işlemini sağlar. Bu döngü turn değişkenine göre sonsuza kadar beklemektedir. Ancak diğer işlem tarafından değeri değiştirilince işlem çalışmasına devam edebilir.

Burada kullanılan turn değişkeni, iki işlem de aynı anda kritik alana girmek istiyorsa, anlık olarak bir tanesinin girmesini sağlar. Flag değişkeni ise o anda kritik alan girmek isteyen diğer işlem olup olmadığını kontrol eder.

Örneğin P1 işlemi, kritik alana girmek istemiyorsa, P2 işleminin kritik alana girerken turn değişkenine bakmasının bir anlamı yoktur. Benzer şekilde P2 işleminin kritik alan ihtiyacı yoksa, P1 de turn değişkenine bakmaz. Ancak iki işlem (yani P1 ve P2) aynı anda kritik alana girmek istiyorlarsa turn değişkeni anlamlı olur.

Bu tip beklemelere meşgul bekleme (busy wait) ismi verilir. Bu ismin verilme sebebi, işlemin sürekli olarak değişkenin değerine bakarak değeri değişene kadar beklemesidir. Bu bekleme işleminin sürekli bir kontrol yapıyor olması, işlemci üzerinde bir yük getirmektedir.

Algoritmanın diğer bir özelliği ise [kilitlenme \(deadlock\)](#) ve [kıtlık \(starvation\)](#) konularına karşı dayanıklı olmasıdır. Algoritmamızda [kilitlenme \(deadlock\)](#) olmayacağı, turn değişkeninin kullanılmasından anlaşılabilir. Bu değişkenin sistemde anlık olarak tek değeri bulunacak (1 veya 0) ve bu değere göre sistem iki işlemten bir tanesini çalıştıracaktır.

[Kıtlık \(starvation\)](#) problemini görmek için ise bir ihtimalin daha hesaplanması gerekir. Buna göre örneğin P1 çalışıyor ve P0 bekliyorken, P1 işini bitirip P0'ın kilidini kaldırdığında P0 henüz çalışmaya başlamadan P1 yeniden çalışıyor mu diye algoritmaya bakmalıyız. Diğer bir deyişle, sürekli olarak kritik alana erişmeye çalışan P1 ve P0 işlemlerinden örneğin P1 işini bitirip kritik alandan çıktından sonra henüz P0 kritik alana girmeden yeniden P1 kritik alana giriyorsa kıtlık (starvation) oluşuyor demektir.



Algoritmada böyle bir durum yaşanmaz. Bunun sebebi algoritmanın kullandığı flag değişkenidir. P1 işlemi, kritik alandan çıkarken, beklemede olan P0 işleminin bariyerini kaldırır ve bu kaldırma sonucunda P0 işlemi çalışırken P1'in yeniden çalışmasını engelleyici biçimde, diğer tarafın flag değişkenini atar.

Kısaca algoritma [kilitlenme \(deadlock\)](#) ve [kıtık \(starvation\)](#) ihtimallerine karşı başarılı bir şekilde çalışır.

## **SORU 12: Birbirini Dışlama (Mutually Exclusive)**

Birbirini dışlama özelliği, birden fazla işin birbiri ile ilişkisizliğini belirtmek için kullanılan bir terimdir. Örneğin iki [işlem \(process\)](#) veya iki [lifin \(thread\)](#) birbirinden bağımsız çalışmasını, aynı anda bir işlemi yapmamasını istediğimiz zaman birbirini dışlama özelliğini kullanabiliriz. Bazı kaynaklarda, kısaca mutex (mutually exclusive kısaltması) olarak da geçer.

İki adet [birbirine paralel ilerleyen iş için \(concurrent\)](#) aynı anda bir kaynağa erişme veya birbirleri için kritik olan işlemler yapma ihtimali her zaman bulunur. Örneğin bilgisayarda çalışan iki ayrı [lifin \(thread\)](#) JAVA dilinde ekrana aynı anda bir şeyler basmaya çalışması veya paylaşılan bir dosyaya aynı anda yazmaya çalışması, eş zamanlı programlamalarda, sıkça karşılaşılan problemlerdendir. Bu problemin çözümü için, [işlemlerin senkronize edilmesi](#) gerekir ve temel işletim sistemleri teorisinde 4 yöntem önerilir:

- Koşullu Değişkenler (Conditional Variable)
- [Semaforlar \(Semaphores\)](#)
- Kilitler (Locks)
- Monitörler (Monitors)

Yukarıda sayılan bu 4 yöntem, basitçe iki farklı işi senkronize hale getirmeye yarar. Şayet iki ayrı işin birbirine hiçbir şekilde karışmaması isteniyorsa (mutex) bu durumda çeşitli algoritmaların kullanılmasıyla sistemdeki işlerin ayrılması sağlanabilir.

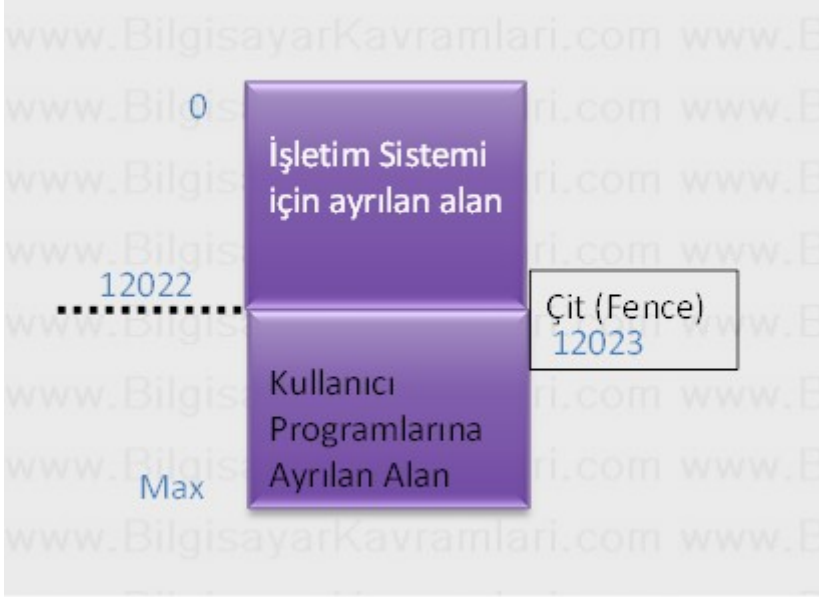
## **SORU 13: İşletim Sistemlerinde Hafıza Güvenliği**

Bu yazının amacı, [işletim sistemlerinde](#), özellikle de birden fazla işlemin çalıştığı ve aynı hafızayı paylaştığı [çok işlemlili \(multiprocessed\)](#) sistemlerde, hafızadaki güvenlik çözümlerini açıklamaktır.

Birden fazla işlemcinin çalıştığı ortamlarda karşılaşılan en kritik hafıza problemi, bir işlemin diğer işlemlerin [hafızada \(RAM\)](#) tutulan bilgilerini okuması veya değiştirmesidir. Bir [işlem \(process\)](#) tanım itibarıyla kendi hafıza alanında çalışır. Dolayısıyla kendisine ayrılan yer dışında bulunan, işletim sisteminin temel bilgilerine (ki işletim sisteminin kendisi de bir [işlemdir \(process\)](#)) veya diğer programlara eriştiği anda bir güvenlik ihlali olur. Bu ihlal veri çalınması, veri değiştirilmesi veya sistemin bozulması olarak karşımıza çıkabilir. (İngilizcedeki security ve reliability kelimelerinin ikisi için de Türkçede kullanılan güvenlik, buradaki iki durumu da karşılıyor, yani hem verinin korunması anlamında security, hem de sistemin sağlıklı çalışması anlamında reliability)

## **Çit Koruması (Fence)**

Hafızayı korumak için kullanılan en basit yöntemdir. İşletim sisteminin kendisini koruması için kullanılır. Bu yöntemde, işletim sistemi ile kullanıcı programları arasında bir çit varmış gibi kabul edilir ve çitin bir tarafındaki kullanıcı programlarının, diğer tarafa geçmesi engellenir.

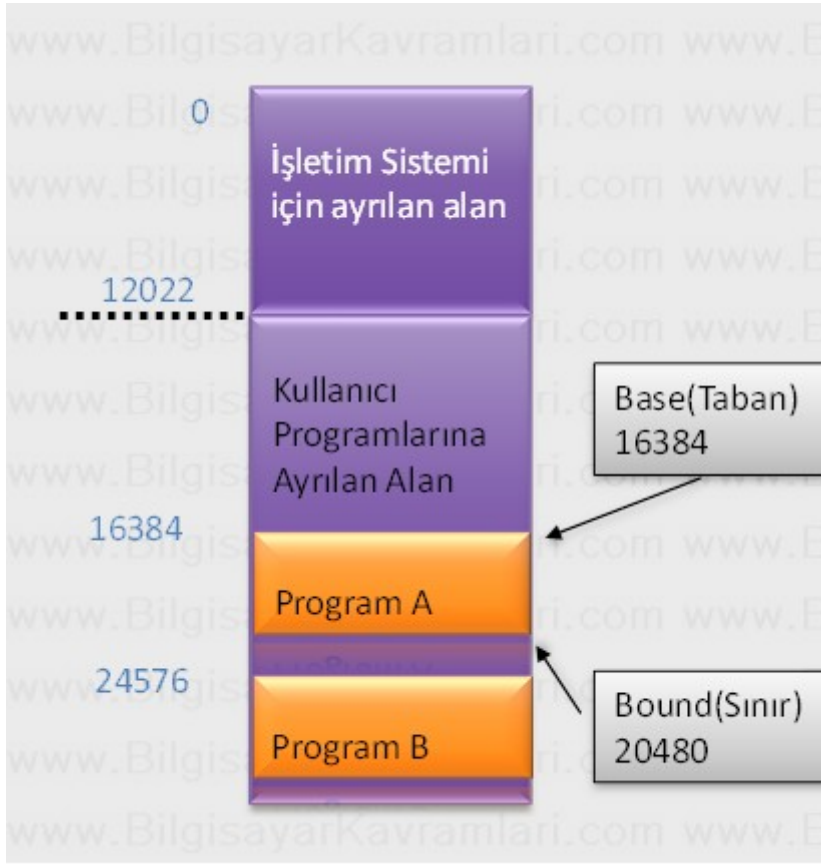


Yukarıdaki şekilde örneğin 12022 adresinde bir çit bulunmaktadır. Bu durumda kullanıcı programları 12023 numaralı adresten başlamakta ve daha büyük değerlere sahip olmaktadır. İşletim sistemi, herhangi bir işlemin (process) daha düşük değerdeki bir adrese erişmesine izin vermemektedir.

Bu yaklaşımda tek amaç, işletim sisteminin kendisini, kullanıcı programlarından korumasıdır. Kullanıcı programları arasında bir güvenlikten bahsedilemez.

### **Taban Sınır Yazmaçları (Base Bound Registers)**

Bu yaklaşımda, sistemde çalışan işlemler arasında güvenlik sağlanması amaçlanır. Buna göre sistemde bulunan bütün işlemlerin bir başlangıç (base) bir de sınır (bounds) değeri bulunur. Bu değerlerin dışına erişimleri yasaktır ve erişim talepleri, işletim sistemi tarafından engellenir.



Örneğin, yukarıdaki şekilde iki programın sistemde çalışıyor olduğunu kabul edelim. Program A'nın başlangıç değeri 16384 ve Program B'nin başlangıç değeri 24576 olarak verilmiş.

Programların her birisi için ayrı ayrı olmak üzere taban ve sınır değerleri belirlenmiştir. Örneğin yukarıdaki şekilde, Program A'nın taban ve sınır değerleri görülmektedir. Bu durumda Program A'nın belirtilen taban değerinden düşük ve belirtilen sınır değerinden yüksek adreslere erişimi engellenmiştir.

Ayrıca işletim sisteminde, [kıtalama \(segmentation\)](#) veya [sayfalama \(paging\)](#) kullanılması durumlarında, kullanılan yöntem, hafızadaki programın başlangıç ve bitiş değerlerini de tutmaktadır. Bu yöntemle göre sistemde hafıza erişim politikası kullanılabilir.

Ayrıca hafıza üzerinde yapılan bazı işlemler için de güvenlikten bahsedilebilir.

### İşlem Taşıma (Relocation)

Bir işlemin, hafızadaki bir konumdan farklı bir konuma taşınması işlemidir.



Yukarıdaki şekilde görüldüğü üzere, Program A, eski konumundan yeni konumuna taşınır. Taşıma işleminden sonra tasarıma bağlı olarak, programın adres erişimleri güncellenebilir veya yeni adrese göre kaydırma değeri eklenebilir.

### Etiketleme (Tagging)

Hafızaya yüklü bir programın, kendisine tanınan alanların dışında bir yere erişip erişmediğini kontrol etmeye yarayan bir yöntemdir. Öncelikle programda bulunan komutlar aşağıdaki üç değerden birisine göre etiketlenir.

- Veri (Data)
- Gösterici (Pointer)
- Kontrol (Control)

Bu değerlere göre işaretlemenin ardından, ilgili adres değerleri kontrol edilir ve erişim haklarının olup olmadığı sınanır.

D	0001
D	0002
D	0003
P	8192
C	Load A
C	Add B
C	Store C
P	16384
D	0004
D	0005
D	0006
P	10572

Yukarıdaki şekilde, programın her satırında ilgili işaretleme işlemleri yapılmıştır. Bu işaretleme işlemlerine göre erişilmek istenen hafıza alanı kontrol edilebilir.

#### **SORU 14: CFS (Completely Fair Scheduling, Tam Adil Zamanlama)**

Linux 2.6.23 sürümünden sonra [çekirdekte \(kenel\)](#) kullanılan [zamanlama algoritmasıdır \(CPU Scheduling\)](#). Algoritmanın özelliği, [CPU meşguliyetini \(CPU Utilisation\)](#) azami seviyeye getirmek ve işlemciden azami derecede istifade etmektir.

2.6.23 sürümünden önce Linux çekirdeğinde kullanılan [O\(1\) zamanlama \(O\(1\) scheduling\) algoritması](#), performans kriterini sistemdeki [bekleme sırası \(ready queue\)](#) üzerine kurmaktaydı. CFS algoritması ise işlemlerin bekletildiği veri yapısını, [kırmızı-siyah ağacına \(red black tree\)](#) çevirmiştir.

Ayrıca CFS algoritmasında, zaman aralıkları nano saniyeler cinsinden hesaplanmakta ve dolayısıyla çok küçük zaman aralıklarında yapılan işlemler, [sezgisel algoritmaların \(heuristic algorithm\)](#) kullanılması gereğini ortadan kaldırmaktadır.

Zamanlama yaklaşımında, ayrıca işlem grupları oluşturulmakta ve gruplar arasında adalet sağlanması hedeflenmektedir. Bu sayede kullanıcı ile iletişimde olan masa üstü uygulamaları gibi uygulamalara daha fazla öncelik verilerek, arka planda çalışan daha düşük öncelikli sunucu görevlerinin, işlemci üzerindeki yükü azaltılmaktadır.

### **SORU 15: O(1) Zamanlaması (O(1) Scheduling)**

İşlemci zamanlama (CPU Scheduling) yaklaşımlarından birisidir. Bu yaklaşımda, işlem (process) sayısına bakılmaksızın bütün işlemlere eşit miktarda zaman ayrılır. Genel olarak bir işletim sistemi zamanlama algoritmasının öncelikli amacı, sistemin verimli olarak kullanılması ve sistemdeki ek yükleri (overhead) azaltarak sistemde kıtlık (starvation) ve kilitlenmeleri (deadlock) önlemektir.

Ayrıca özel bir işletim sistemi grubu olan gerçek zamanlı işletim sistemleri (RTOS, real time operating systems) için zamanlama işlemlerinin belirgin olması gerekir. Yani bir işlemin çalışacağı ve biteceği zamanlar tam olarak bilinmelidir. O(1) zamanlama algoritmasında, bir işlemin sistemde çalışacağı süre kesin olarak belirlenmiştir. Linux 2.6.23 sürümü öncesinde kullanılan bu O(1) zamanlama sisteminde, işlemler kesin bir süre çalışıp ardından bitmemeleri halinde bekleme sırasına (ready queue) taşınmakta ve tekrar çalıştırılmak için beklemektedir. Linux zamanlaması 2.6.23 sonrasında CFS (completely fair scheduler, Tam Adil Zamanlama) ile değiştirilmiştir.

O(1) ismi, karmaşıklık teorisindeki big-oh gösteriminden gelmektedir ve sistemin en kötü durum analizinin 1 olduğunu belirtir. Bu değer karmaşıklık analizi (complexity class) için elde edilebilcek en iyi değerdir.

### **SORU 16: Overhead (Ek Yük)**

Genel olarak bir işin yapılması için, gereken ek maliyetlere verilen isimdir. Örneğin bir kamyonun, bir yükü taşıması için, kendisini de taşıması gerekir. Kendisini taşımasının maliyeti, bu işlemdeki ek yüküdür (overhead).

Bilgisayar bilimlerinde, çeşitli alanlarda farklı anlamlarla kullanılmaktadır.

Örneğin veri iletişimi (network) konusunda ek yük (overhead) denildiğinde genelde bir veriyi iletmek için kullanılan tesrifatın (protokol) kendi içindeki ilave haberleşmeleri kast edilir. Örneğin TCP/IP protokolünü ele alalım. Veriyi doğrudan iletmek yerine öncelikle 3 yönlü el sıkışma (three way hand shaking) işlemi ile taraflar arasında iletişim kurulur. Ardından her taşınan veri için TCP/IP paketinin başlık ve sonluk bilgileri de taşınır (ki bu bilgilerin içerisinde örneğin paketin nereden gelip nereye gittiği bilgisi bulunur). İşte veri iletmek için ayrılan kanal veya kaynakların bir kısmının, veriyi taşımak yerine protokole özgü ilave bilgileri taşıması genelde veri iletişimi (network) açısından ek yük (overhead) olarak isimlendirilir.

Programlama dilleri açısından çağırma ek yükü (call overhead) ismi verilen kavram, bir fonksiyonun çağırılması sırasında yaşanan ek yüküdür. Bir programlama dilinde, herhangi bir fonksiyon çağrıldığında, bu fonksiyonun çalışması sonucunda döneceği yer bir yığında (stack) tutulur. Fonksiyon çalışıp işi bittikten sonra program akışına bu noktadan devam edilir. Bu şekilde fonksiyonlar birbirini çağırarak devam edebilir. Örneğin A fonksiyonu B'yi, B fonksiyonu C'yi ... şeklinde birbirilerini çağırdıklarında fonksiyonun çalışması için gereken

yere ilave olarak fonksiyon bilgisi için ilave bir yer tutulması gerekir. Bu yere ve bu yer üzerine yapılan işlemlere çağırma ek yükü (call overhead) ismi verilir.

İşletim sistemleri açısından zamanlama ek yükü (scheduling overhead) ismi verilen kavram, bir zamanlama algoritmasının, işlemler arasında geçiş yaparken kaybettiği kaynaklardır. Örneğin [kesintili zamanlama \(preemptive scheduling\)](#) kullanılan algoritmalarda bu ek yük miktarı (overhead) artacaktır.

### **SORU 17: Fair Share Scheduling (Adil Paylaşımlı Zamanlama)**

[İşlemci zamanlama \(CPU Scheduling\)](#) konusunda kullanılan kavramsal bir yaklaşımdır. Bu zamanlama algoritmasının amacı, en kıymetli sistem kaynaklarından birisi olan işlemcinin (CPU), adil bir şekilde dağıtılmasıdır. Örneğin 10 kullanıcı bir sistemde, kaynaklar bütün kullanıcılara %10 oranında dağıtılacak ve sonuçta adalet sağlanacaktır.

Şayet kullanıcılardan birinin iki farklı işlemi bulunursa, bu durumda kaynakları %5 oranında paylaşılacaktır çünkü kullanıcının toplam CPU hakkı %10'dur.

FSS yaklaşımında ayrıca grup kullanımı da mümkündür. Kullanıcı gruplarına ve bu gruptaki kullanıcılar arasında işlemcinin paylaşılması mümkündür. Örneğin 4 gruplu bir sistemde, her gruba işlemci zamanının %25'i ayrılacaktır. Örneğin 5 kullanıcı bir grupta, kullanıcı başına %5 kaynak ayrılırken, 10 kullanıcı ikinci bir grupta, kullanıcı başına %2,5 kaynak ayrılacaktır. Bu yaklaşımda grup kullanıcıları arasında da adalet sağlandığı var sayılmıştır. Gruplar arası adalet sağlayıp (fairness) grup içerisinde farklı algoritmalar da kullanılabilir.

Benzer bir adalet sağlama yaklaşımı da [round robin algoritmasının](#) kullanılmasıdır. Adalet sağlanmak istenen varlıklar arasında (örneğin gruplar, kullanıcılar, [işlemler \(process\)](#)) belirli bir zaman aralığında dönme sağlanarak adalet hedeflenebilir.

Burada anlatılan adaletli dağıtım yaklaşımı sadece kavramsal bir yaklaşımdır, bu yaklaşımın gerçek uygulamaları da bulunmaktadır. (örneğin [CFS, completely fair scheduling, tam adil zamanlama](#) gibi)

### **SORU 18: Translation Lookaside Buffer (TLB, Dönüşüm Hafızası)**

TLB, [sayfalama işleminin \(paging\)](#) hızını arttırmaya yarayan bir hafıza bölümüdür. Kısaca TLB olması için sayfalama olmalıdır. Günümüzdeki çoğu bilgisayar mimarisi tarafından desteklenmektedir. TLB kullanılabilmesi için sayfalama (paging) sistemde yapılıyor olmalıdır.

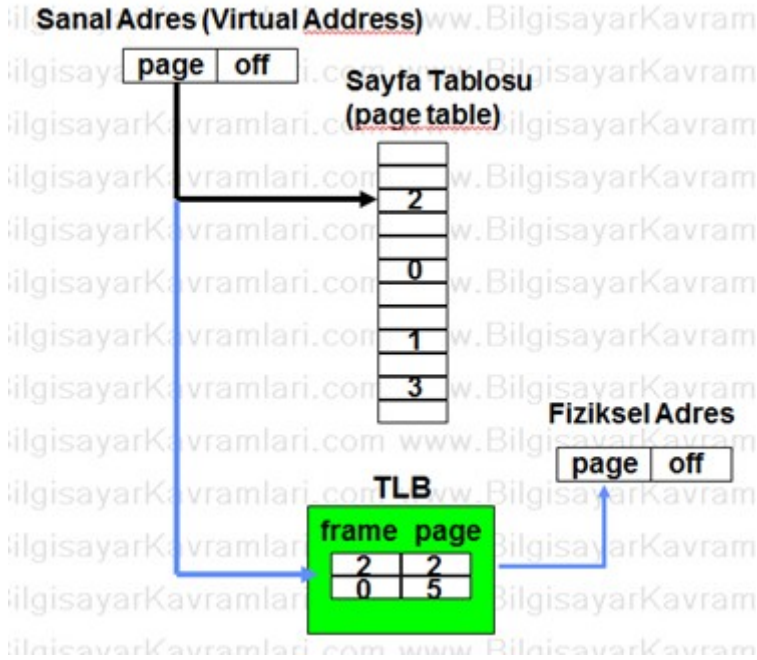
Basitçe, RAM'in yetersiz olduğu durumlarda hafıza ihtiyacının diskten karşılanmasını sağlayan sanal hafızanın verimli kullanılması için, diskte tutulan veri ve RAM'de duran veri arasında bir dönüşüm işlemi gerekmektedir. Bu işin hızlanması için dönüşümü TLB üzerinde tutar ve hız artışı sağlarız.

Bu yapıp, disk ile RAM arasında bir [önbellek \(cache\)](#) olarak düşünmek mümkündür. Basitçe diskte olan bir veriye erişilmek istendiğinde TLB üzerinde bu veri aranır. Şayet veriye ulaşırsa buna TLB bulma (TLB hit) ismi verilir ve dönüşüm sağlanır. Şayet veri bulunamazsa buna TLB kayıp (TLB miss) ismi verilir ve sanki TLB hiç yokmuş gibi klasik [sayfalama tablosunda \(page table\)](#) arama işlemi yapılarak bu hafıza sayfasına (page)



erişilmeye çalışılır. Elbette bu işlem çok daha maliyetlidir. Dolayısıyla TLB üzerindeki bulma işlemini (hit) arttırmak isteriz.

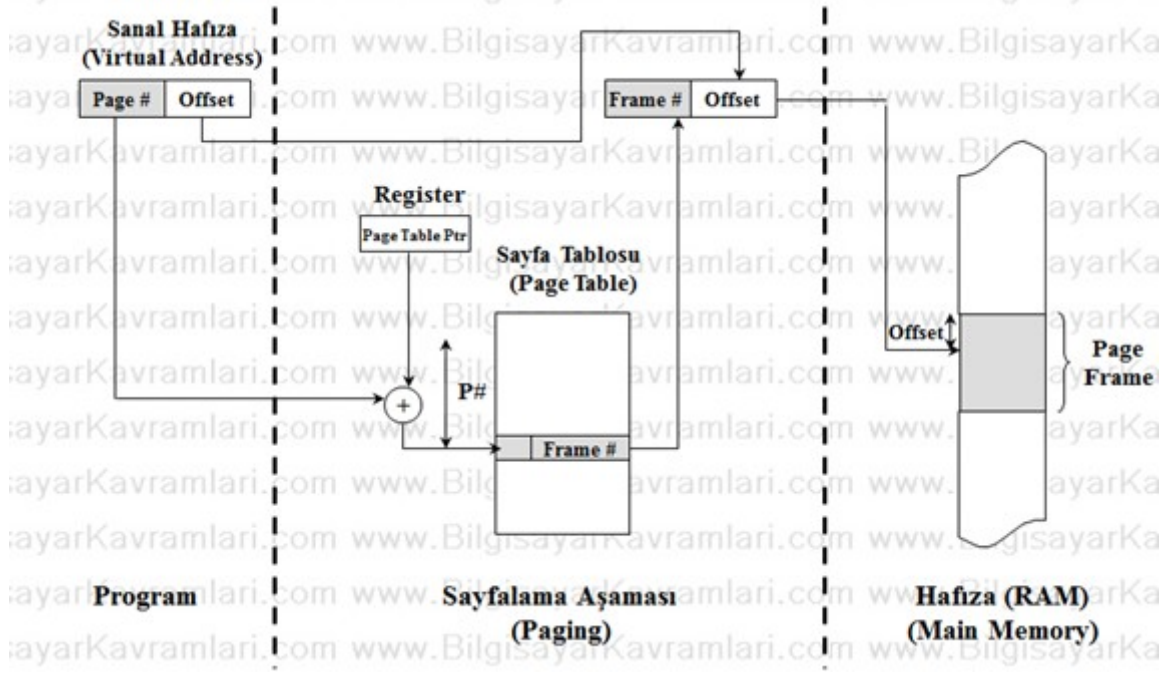
TLB yapısını, aşağıdaki şekil üzerinden anlamaya çalışalım:



Yukarıdaki örnekte, bir sanal adresin, fiziksel adrese dönüşmesi sırasında geçtiği yol gösterilmiştir. Örneğin gelen sanal adres talebimiz, 2 veya 0 ise, bu değerler TLB üzerinden karşılanır. Şayet talep 0 veya 1 ise bu değerler sayfa tablosu üzerinde aranarak karşılanır. Sayfa tablosu üzerindeki arama işlemi, TLB üzerindeki arama işlemine göre çok daha uzun sürmektedir.

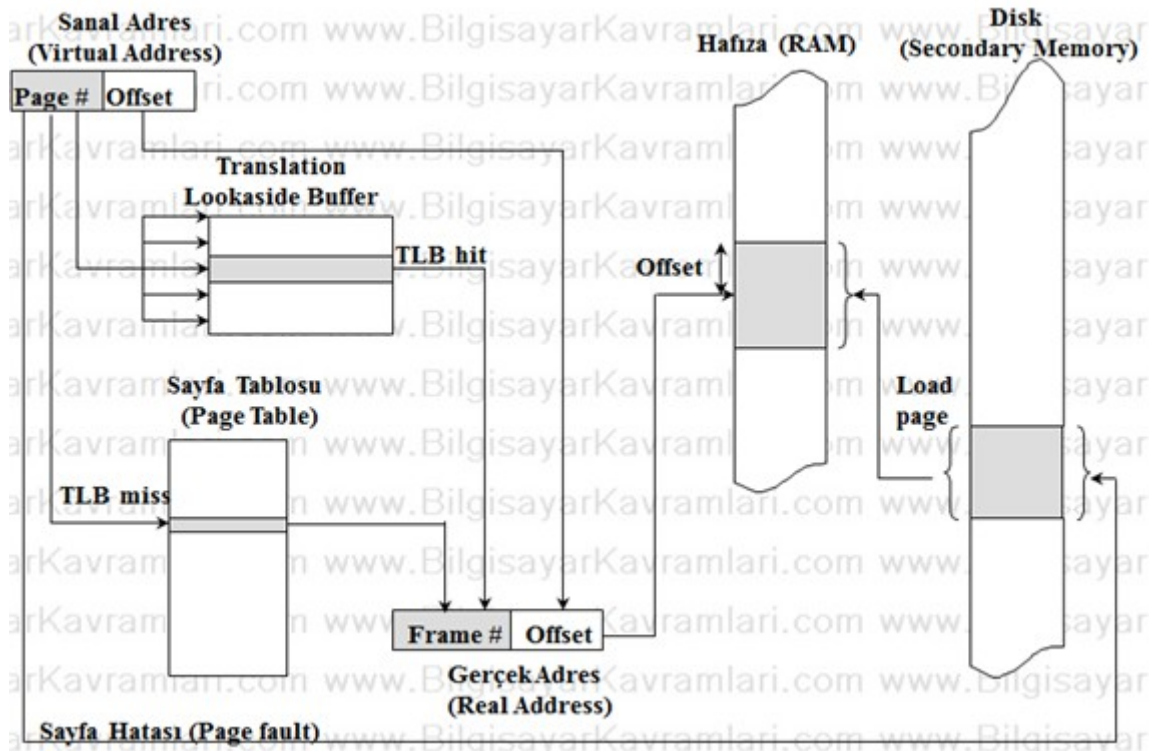
Konuyu, Sayfalama (paging) ile birlikte ele almak için, öncelikle sayfalama işlemini hatırlayalım.





Yukarıda görüldüğü üzere, sayfalama işlemi, programın kendi adresinden, gerçek adrese (fiziksel adres) dönüşümü sırasında kullanılır. Bu sırada, sayfalama yaklaşımı hafızadaki bilgilerin dağılımını tutan bir sayfalama tablosu bulundurur.

Şayet yukarıdaki şekle, TLB ve sanal hafıza (virtual memory) konularını da eklersek, aşağıdaki şekli elde ederiz:



Yukarıdaki yeni şekilde, bir sanal adresin, talep edilmesi durumunda yaşananları anlatmaya çalışalım.

Öncelikle bu adresin sayfa numarası, (ki [sayfalama konusundan hatırlanacağı](#) üzere sayfa numarası, adresin sayfa boyuna (page size) bölünmesi ile elde edilir).

Bu sayfa numarası, önce TLB üzerinde aranır. Bulunursa çerçeve numarasına ofset eklenerek hafızadan alınır.

Şayet TLB üzerinde bulunamazsa, TLB kayı (TLB miss) olur ve bu durumda Sayfa Tablosu üzerinde aranır. Şayet sayfa tablosunda bulunursa, yine ofset eklenir ve hafızadaki yerine ulaşılmış olunur.

Şayet Sayfa Tablosu üzerinde de bulunamazsa bu durumda sayfa hatası (Page fault) olur ve bunun anlamı, bu bilginin hafızada olmadığıdır. Artık bilgi diskten aranır ve bulunur. Ardından hafızaya (RAM) yüklenir ve sayfa tablosunda ilgili güncellemeler yapılır.

Yukarıdaki yazıda bazı terminolojik problemleri çözmek için bu notu eklemeyi önemli buluyorum. Bazı kaynaklarda, yukarıda sanal adres (virtual address) olarak geçen terime mantıksal adres (logical address), gerçek adres (real address) olarak geçen terime ise fiziksel adres (physical address) isimleri verilmektedir.

### **TLB tablosunun güncellenmesi**

Klasik bir TLB tablosunda, bir verinin TLB üzerinde bulunamaması halinde bu veri güncellenir. Yani şayet veri TLB üzerinde bulunuyorsa, bu durumda bir güncellemeye ihtiyaç duyulmaz. Ama bir şekilde TLB kayıp oluşursa (TLB miss), bu durumda talebin Sayfa Tablosundan (Page Table) veya diskten karşılanması halinde TLB tablosu güncellenir.

Ancak daha önce de bahsedildiği üzere TLB aslında bir önbellek olarak düşünülebilir ve klasik önbellek güncelleme algoritmaları veya erişimli önbellek algoritmaları (associative cache) TLB için de kullanılabilir.

### **SORU 19: Ön Hafıza (Cache)**

Bilgisayar bilimlerinde sıklıkla kullanılan ve herhangi bir hafıza işleminin nispeten daha küçük ve daha hızlı dolayısıyla da daha pahalı ilave bir hafızada yapılmasını ifade eden terimdir.

Aslında kelime anlamı olarak Türkçedeki zula kelimesi ile de karşılanabilen cache kelimesi, değerli şeylerin saklandığı yer anlamına gelmektedir. Bilgisayar bilimlerinde ise sık erişilen ve dolayısıyla bizim için daha değerli olan bilgilerin saklandığı yere verilen isimdir.

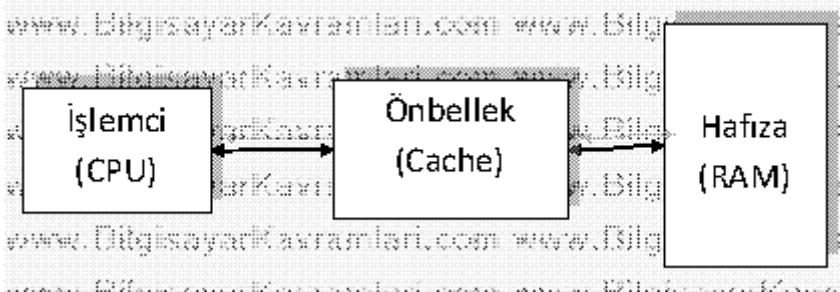
Kısacası bir ön hafıza (cache) gerçek hafızaya erişmeyi azaltmak ve daha hızlı bir şekilde çözmek için tasarlanır. Gerçek hafızadan daha hızlı ve daha pahalıdır. Gerçek hafızadaki her şeyi içeremez dolayısıyla küçük bir kısmını içerir.

En çok kullanıldığı yerler aşağıdaki şekilde sıralanabilir:

- İşlemci önbelleği
- Disk önbelleği
- Web önbelleği

Elbette yukarıdakiler dışında farklı alanlarda da veriye erişim hızını arttırmak için önbellekler kullanılabilir.

Örneğin aşağıda, [RAM \(hafıza\)](#) ile işlemci (CPU) arasına yerleştirilmiş ve işlemci ön belleği (CPU Cache) ismi verilen erişim şekli gösterilmiştir:



Buradaki amaç, işlemcinin hafızada ihtiyaç duyduğu verilerin bir kısmını önbellekten karşılamasıdır. Benzer şekilde hafızadaki veriler ile disk arasında da önbellek olabilir. Bu durumda bir verinin diskten ihtiyaç duyulması halinde diskten daha hızlı olan ön bellek devreye girerek hafızaya yükleme işlemi gerçekleştirebilir. Elbette burada iki ihtimal vardır:

Hit: isabet, işlemcinin isteğinin önbellekten karşılanması

Miss: kayıp, işlemcinin isteğinin önbellekten karşılanamayarak hafızaya erişim yapılması

Bu sayıların oranlanması ile de aşağıdaki değerler bulunur:

İsabet oranı (hit ratio) = isabet (hit) / Toplam talep (veya isabet + kayıp)

Kayıp oranı (miss ratio) = kayıp (miss) / Toplam talep (veya isabet + kayıp)

Örneğin 45 hafıza erişim talebinin olduğunu (request) ve bunların 21'inin önbellekten karşılandığını düşünelim. Bu durumda

$$\text{Kayıp (miss)} = 45 - 21 = 24$$

$$\text{İsabet oranı (hit ratio)} = 21 / 45 = \%47$$

$$\text{Kayıp oranı (miss ratio)} = 24 / 45 = \%53$$

Olarak hesaplanabilir.

### **Önbellek yazma politikası (write policy)**

Önbelleklerin birinci görevi, ihtiyaç duyulan verinin hızlı karşılanmasıdır. Ancak bu durum veri okunurken avantaj sağlar. Verinin değiştirilmesi veya yazılması gibi durumlarda önbellekte nasıl bir politika izleneceğine de yazma politikası (write policy) ismi verilir. Buna göre 3 farklı politika izlenebilir:

Üzerine yazma (write through) : her yazma işlemi, önbellekte bir isabet olsa bile hafızada güncelleme gerektirir. Buna göre önbellekteki veri ile hafızadaki veri birebir aynı olur.

Birisindeki deęiřiklik dięerini etkiler ve verinin iki ayrı kopyası arasında bir eřleřme gerektirmez.

Geri yazma (write back): Bu yazma politikasında, bir bilginin deęiřtirilmesi durumunda, bilgi ön bellekte bulunuyorsa, önbellek üzerinde deęiřiklik yapılır ve hafızadaki kopya hemen deęiřtirilmez. Dolayısıyla anlık olarak bilginin iki farklı kopyası bulunur. Bilginin iki kopyasının birbirinden farklı olduęunu göstermek için de kirli (dirty) bit kullanılır. Dolayısıyla bu politikada, ön bellek üzerinde ilave bir bite ihtiyaç vardır.

Bu yazma politikasında veriler ön bellekten kaldırılırken hafızadaki veri ile güncellenir. Yani ön bellek sürekli olarak deęiřtirilmiř veriyi tutar, veri ön bellekten kaldırılıp yerine yeni veri geleceęi zaman, bu veri hafızadaki verinin üzerine yazılarak iki kopya aynı hale getirilir.

Bu yazma politikasının hafızayı deęiřtirmesindeki tutumundan dolayı tembel yazma (lazy write) ismi de verilir.

Bu yazma politikasında ayrıca herhangi bir verinin önbellekten kaldırılıp yerine yeni veri gelmesi durumunda iki kere hafıza eriřimi gerekir. Birincisi eski verinin hafıza ile güncellenmesi ikincisi ise yeni verinin hafızaya yüklenmesi için. Üzerine yazma politikasında ise sadece tek eriřim yeterlidir.

Yazmama politikası (no-write allocation): bu politikada, önbellek üzerinde bir yazma iřlemi yapılmaz. Veri önbellekte sadece okunmak için tutulur ve bir yazma iřlemi gerekleřtięinde bu iřlem doęrudan hafızaya yazılır.

Bu durumda önbellekteki verinin doęruluęunda problem olacaktır. Yani hafızadaki veri daha güncel ve önbellekteki veri eski kalmıř olacaktır. Bunu belirtmek için önbellek üzerinde bir doęruluk biti (valid bit) kullanılır. Bu bit, verinin deęiřtięini ve önbellekteki verinin eski kaldıęını belirtir. Herhangi bir řekilde bu deęiřen veriye okumak için eriřim olmazsa önbellekte güncellemeye gerek de kalmaz (örneęin bu veri bir daha hi eriřilmeden önbellekten kaldırılabilir) ancak bir řekilde bu veriye tekrar okumak için eriřim olursa bu durumda hafızadaki veri ile güncellenmesi için hafızadan verinin okunup önbelleęe yeni halinin yazılması gerekir.

Bu politikada bir veri deęiřtirildięinde ardından gelen yazma iřlemlerinin önbellek tarafından karřılanması gerekmez. Ancak bir yazma iřleminin sonraki okuma iřlemlerinin hafızadan yeni veriyi önbelleęe yüklemesi gerekir.

### **Önbellek iliřkilendirmesi (Cache Associativity)**

Ön bellek kullanılırken bir verinin önbellek üzerinde nereye yazılacaęını belirleyen yöntemlere verilen isimdir. Bu yöntemler temel olarak üç grupta toplanabilir.

- Doęrudan haritalama (direct mapped cache) yönteminde veri önbellek üzerinde tek bir yere yazılabilir.
- Küme iliřkilendirme (set associativity) yönteminde veri önbellek üzerinde birden fazla yere yazılabilir.
- Tam iliřkilendirme (full associativity) yönteminde ise veri önbellekteki her yere yazılabilir.

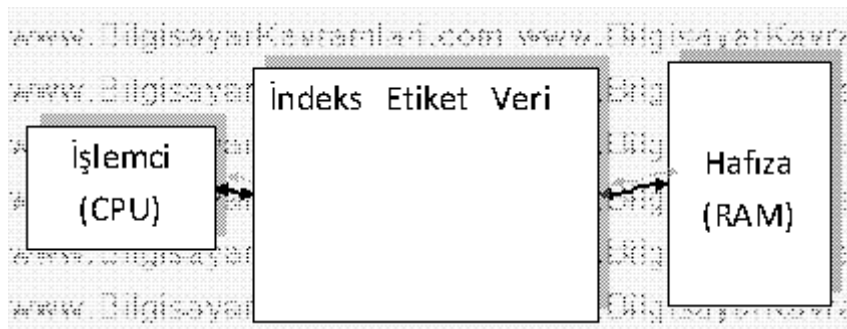
Yukarıdaki bu ilişkilendirme yöntemlerini aşağıda daha detaylı açıklamaya çalışalım.

#### Doğrudan haritalama (direct mapped cache)

Bu yöntemde, hafızada tutulan veriler ile önbellek adresleri arasında tam bir bağlantı vardır. Örneğin 4 satırdan oluşan bir önbelleğimiz olsun (yani önbelleğimizin kapasitesi 4 veri tutmaya yetiyor). Bu durumda önbelleğin adres için ayrılan bit sayısı 2 olur ( $2^2 = 4$ )

Yine örnek olarak hafızadaki verilerin uzunluğu da 8 bit olsun.

En basit doğrudan haritalama için mod alma işlemi kullanılabilir. Yani örneğin adresin son iki bitine göre (yada ilk iki bitine göre) hafızada erişilmek istenen adresi önbellekte adresleyeceğiz.



Örneğin yukarıdaki şekilde görüldüğü üzere ön bellek 3 sütundan oluşmuştur. Bu sütunların ilki önbellek satır numarasını belirtmeye yarayan ve bir bilginin önbellekte olması durumunda hangi satırda tutulduğunu gösteren sütundur.

Etiket kısmı, önbellekteki verinin aslında hafızadaki gerçek adresini belirten sütundur.

Veri sütunu ise önbellekte duran veridir. Bilginin önbellek tarafından karşılanması durumunda buradaki veri sütununda bulunan veri okunur.

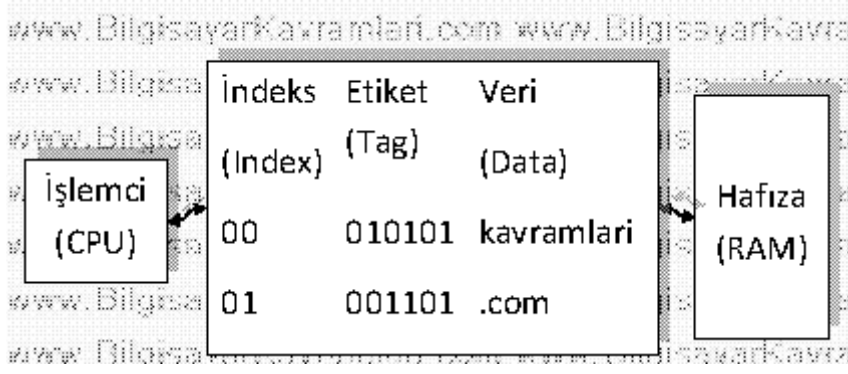
Yukarıda anlatıldığı üzere kullanılan yazma politikasına göre önbellekte ilave sütunlar bulunabilir (kirli (dirty) veya doğru (valid) bitleri gibi).

Örneğin adres olarak tutulan veri 8 bit uzunluğunda olsun ve önbellekte bu verinin ilk iki bitine göre indeksleme yapılıyor olsun.

Örnek hafıza erişimlerini aşağıdaki şekilde ele alalım:

Hafıza adresi	Veri
11010100	www.
10101001	bilgisayar
00010101	kavramlari
01001101	.com

Bu verilerin ilk iki bitleri birbirinden farklıdır ve hepsi önbellekte farklı sıralara yerleştirilecektir. Bu verilerin yerleştirilmiş hali aşağıdadır:



Yukarıda görüldüğü üzere, veriler önbellek üzerinde doğru yerlere yerleştirilmiş ve etiket kısmında adresi tamamlayıcı bilgiler bulunmaktadır. Bu yerleştirme işleminden sonra herhangi bir veriye erişilmek istendiğinde önbellek şu şekilde çalışır:

Örneğin erişilecek hafıza adresi 01001101 olsun.

Bu durumda ilk iki bitine bakılacak ve 01 indekisinde bulunan etiket kontrol edilecektir.

Buradaki etiket değeri ulaşmak istenen adresin son 6 biti ile aynı olduğu için ( 001101 = 001101 olduğu için) verinin önbellekte bulunduğuna karar verilip hafıza erişimi önbellekten karşılanacaktır ve veri olarak “.com” işlemciye iletilecektir.

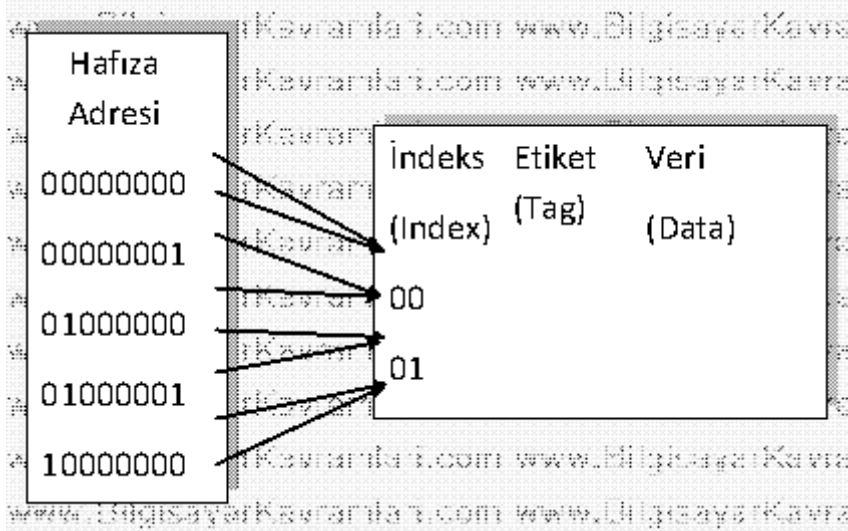
Örneğin erişilecek hafıza adresi 01001111 olsun.

Bu durumda ilk iki bitine bakılacak ve 01 indekisinde bulunan etiket kontrol edilecektir. Bu etiket ulaşmak istenen hafıza adresi ile uyuşmamaktadır ( 001101 != 001111 olduğu için).

Dolayısıyla veri önbellekte olmadığı için verinin hafızadan karşılanması gerekir. Hafızadan karşılanan bu veri, önbellek değiştirme algoritmasına göre (aşağıda anlatılacaktır) önbellekte bir değiştirmeye sebep olur veya olmaz ancak her halükarda bir hafıza erişimi gerekecektir.

Örneğin hafıza erişiminin ardından verinin değiştirileceğini düşünelim. Bu durumda önbellekteki 01 indekisinde olan veri değiştirilecek ve hafızadan yüklenen veri bu indekse yazılacaktır. Doğrudan erişim algoritmasında verinin yazılabileceği tek adres bulunur.

Doğrudan haritalama yöntemini örnek bir şekil üzerinden gösterecek olursak:



Yukarıda görülen şekilde hafızada, örnek adresler verilmiştir. Elbette 8 bitlik hafızada daha fazla adres vardır ancak yukarıdaki temsili şeklin amacı durumu anlatmak olduğu için ilk iki biti aynı olan ikişer adres üzerinden örnek gösterilmiştir.

Yukarıda görüldüğü üzere hafızadaki adresler doğrudan önbellek üzerinde tek bir indekse haritalanmış ve birden fazla hafıza adresi aynı indekse haritalanmış durumdadır.

#### Küme ilişkilendirme (Set Associativity)

Bu erişim yönteminde bir önceki doğrudan erişimden farklı olarak veri kesin ve net bir şekilde bir yere eklenmez. Bunun yerine bir bilginin önbellek üzerinde gidebileceği birden fazla yer bulunur.

Bu küme ilişkilendirme yönteminde bir bilginin önbellek üzerinde gidebileceği alana göre sayılar belirtilir. Örneğin ön bellek üzerinde 2 farklı yere eklenebiliyorsa 2 yönlü küme ilişkilendirme (2-way set associativity) veya 4 farklı yere erişilebilirse dört yönlü küme ilişkilendirme (4-way set associativity) ismi verilir. Bu durumu aşağıdaki örnek üzerinden anlamaya çalışalım.

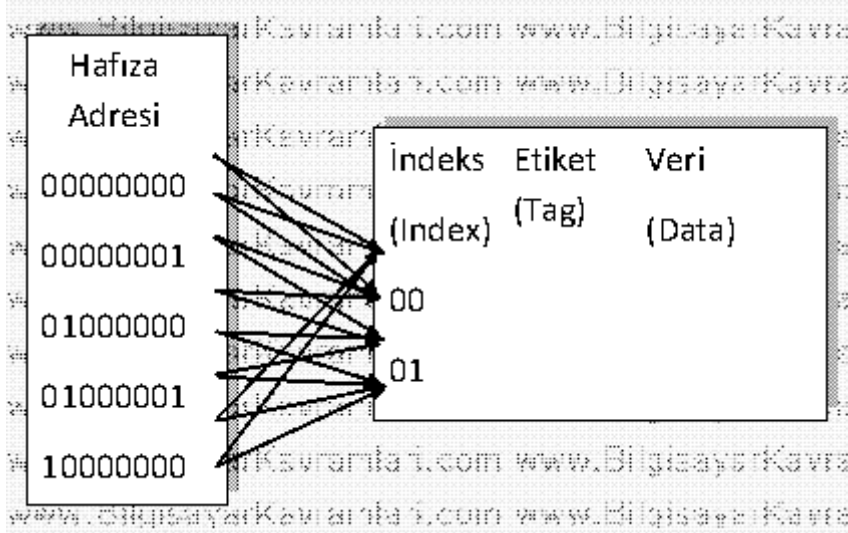
Örneğimizde bir önceki örnekle aynı yapıyı kullanalım ancak farklı olarak veri, hafıza adresinin ilk iki biti ve ilk iki biti +1 indekslerine yazılabilir.

Yani bir bilgi aşağıdaki iki adresten birisinde indekslenecektir:

A : ilk iki bit değerindeki indeks

B: (ilk iki bit + 1) mod önbellek boyutu değerindeki indeks.

Bu durumda erişim aşağıdaki şekilde haritalanmış olacaktır:



Yukarıda görüldüğü üzere ilk iki bitine göre veri hem bu adrese hem de bu adresin bir sonraki adresine haritalanmıştır. Bu durumu aşağıdaki tablo ile de gösterebiliriz:

Hafıza adresi	Önbellek 1	Önbellek 2
00	00	01
01	01	10
10	10	11
11	11	00

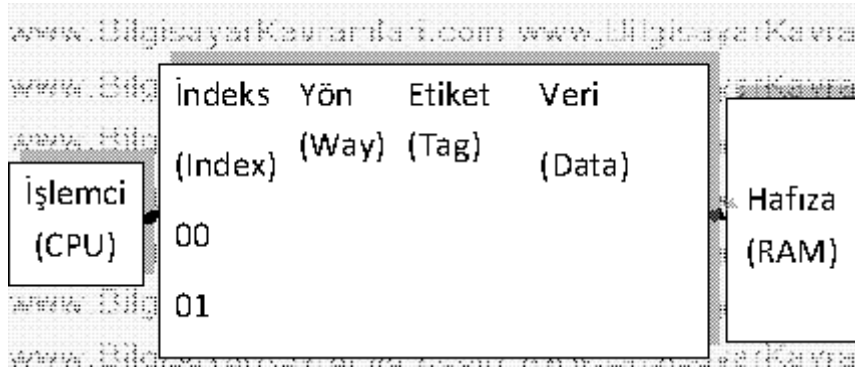
Yukarıdaki tabloda da gösterildiği üzere bir hafıza adresi iki farklı önbellek adresi ile haritalanmıştır.

Bu adresleme için örnek veri erişim tablomuz aşağıdaki şekilde olsun:

Hafıza adresi	Veri
11010100	www.
10101001	bilgisayar
10010101	kavramlari
01001101	.com
11010101	Sadi
00101111	Evren

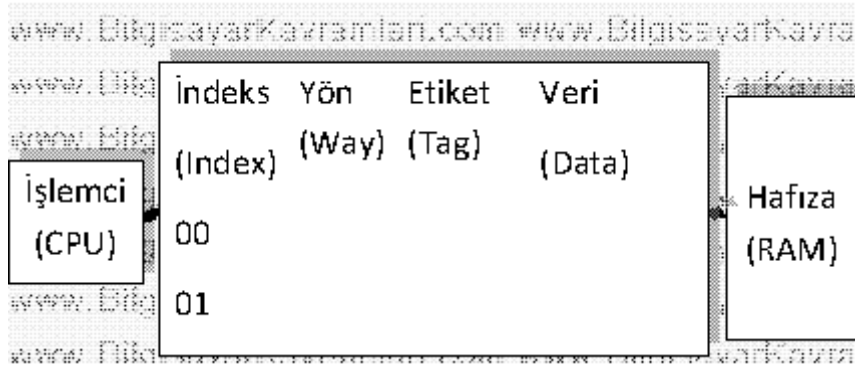
Yukarıda verilen sırayla erişimi aşağıdaki şekiller üzerinden açıklamaya çalışalım:



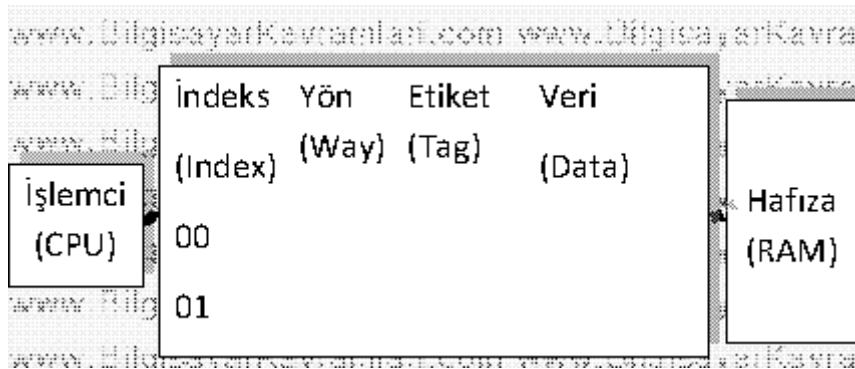


Yukarıdaki şekilde, daha önceki doğrudan haritalamadan farklı olarak yön (way) bitleri tutulmuştur. Bu bitlerin amacı bu indekste bulunan verinin aslında hangi yönden geldiğini tutmaktır. Örneğin 11 indeksine, 11 veya 10 yönünden veri yazılabilir bu durumda verinin hafıza adresini tam olarak bulabilmek için hangi yönden verinin yazıldığı tutulmalıdır.

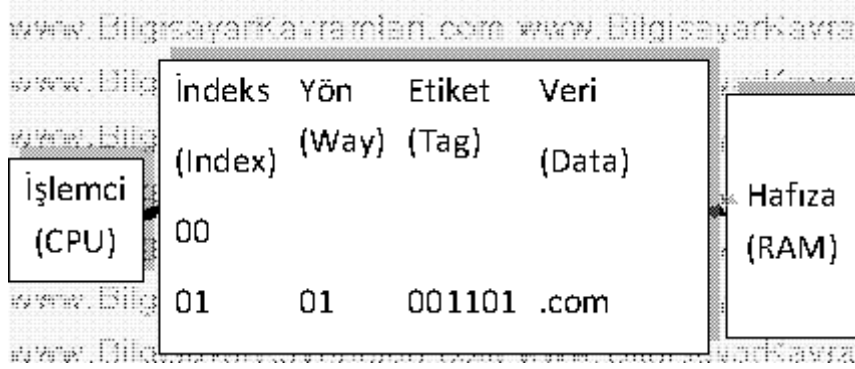
İlk olarak 11 indeksine veriyi yerleştiriyoruz. Ardından gelen 10101001 adresindeki veri ilk iki biti itibariyle 10 adresine yerleştiriliyor.



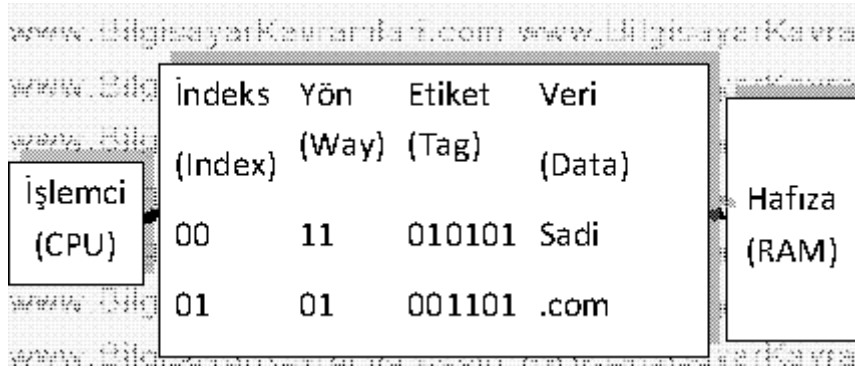
Sonraki verimiz 10010101 adresinde. Bu verinin yerleştirileceği ilk adres, 10 indeksi dolu, 2 yönlü haritalama kullandığımız için bir sonraki alternatif adrese bakıyoruz ve 11 adresi kontrol ediliyor. Bu adres de dolu olarak görülüyor. Sonuçta önbellek üzerinde yerleştirilebilir bütün alanlar dolu olduğu için önbellek değiştirme algoritması devreye giriyor ve önbellekteki bilgilerden birisinin değiştirilmesi gerekiyor. Örneğin kullanacağımız değiştirme algoritması FIFO (ilk giren ilk çıkar (first in first out)) olsun. Bu durumda 10 ve 11 adreslerinden eski olanı ile yeni gelen veri değiştirilecektir. Şu anda eski olan bilgi 11 adresindeki bilgidir dolayısıyla önbelleğin yeni hali aşağıdaki şekilde olacaktır:



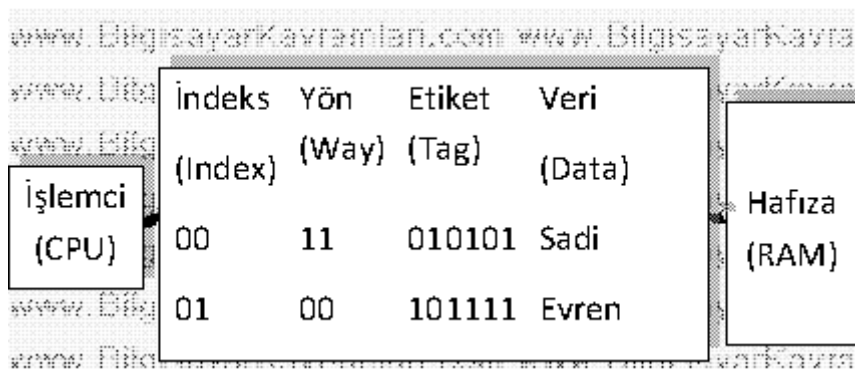
Yukarıdaki önbellek değiştirme (cache replacement) işleminin ardından 01001101 adresine erişim yapılıyor. Bu adres boş olduğu için ön bellek burada bilgiyi tutabilir:



Ardından gelen 11010101 erişimi ise yine önbellekte dolu olan indekse yapılmaktadır. Ancak 2 yönlü kümeleme kullanıldığı için veriyi diğer alternatifi olan ve şu anda boş olan 00 indeksine koyabiliriz:



Sonraki erişim 00101111 erişimidir. Burada yine FIFO önbellek değiştirme algoritması (cache replacement algorithm) gereği 00 ve 01 indekslerinden önce girenin değiştirilmesi gerekir. Bu bilgilerden 01 indeksteki veri daha önce önbelleğe alındığı için veri buraya yerleştirilir:



Yukarıdaki önbellek yerleştirilmesinden sonra herhangi bir bilgiye erişilmek istendiğinde ilk iki bitine bakılır, ardından bu iki bitin girebileceği indeksler taranır.

Örneğin erişilmek istenen veri 11010101 adresinde olsun. Bu durumda ilk iki bite bakılıp 11 değeri ile arama yapılacaktır. 11 değeri ise 2 yönlü küme ilişkilendirmesi gereği 11 ve 00 adreslerinde tutulabilecek bilgidir. Bu durumda önbellekte iki yere de bakılacak 00

indeksindeki yön + etiket bilgisi aranan adresin bu indekste olduğunu gösterecek ve veri buradan karşılanacaktır.

Küme ilişkilendirme yöntemlerinde, verinin birden fazla yere yazılabilmesi, veriye erişim sırasında bütün bu alanların aranmasını gerektirir. Dolayısıyla önbellek üzerinde birden fazla erişime sebep olarak belki de önbellekte hiç tutulmayan bir veri için uzun süreli bir arama ile sonuçlanabilir.

Küme ilişkilendirme yöntemlerinin avantajı ise hiç erişilmeyen hafıza alanlarının önbellekte durmasını engellemesidir. Örneğin yukarıdaki şekil için, 00 ile başlayan hafıza alanına uzun süre hiç erişim olmayacağını düşünelim. Bu durumda doğrudan haritalama (Direct mapping) yöntemi bu önbellek alanını hiçbir zaman kullanmaz. Oysaki küme ilişkilendirme yönteminde alternatif bir önbellek indeksi bu alanı kullanabilir. Böylelikle önbelleğin daha verimli kullanılması sağlanır.

#### Bütün küme ilişkilendirme (full set associativity)

Bu yaklaşımda önbellekteki her indekste her adres durabilir. Yani bir önceki örnekte iki yönlü küme ilişkilendirme sırasında bir hafıza adresi, sadece iki indeksten birisine gidebilirken şimdi bütün önbelleğe yerleştirilebilir. Bu durumda önbellekteki arama süresi uzarken, önbellekte atıl kalan indeks miktarı azaltılmış olunur.

#### **Önbellek değiştirme algoritmaları (Cache replacement algorithms)**

Bu algoritmaların amacı, önbellek üzerinde küme ilişkilendirilmesi kullanıldığında ve verinin önbellekte birden fazla alana yazılabileceği durumlarda nereye yazılacağını belirlemektir.

Temel olarak aşağıdaki algoritmalar en bilinenleri olarak sayılabilir:

FIFO , first in first out , ilk giren ilk çıkar

LRU, least recently used, en eski erişilen

LFU, least frequently used, en seyrek erişilen

MRU, most recently used, en son erişilen

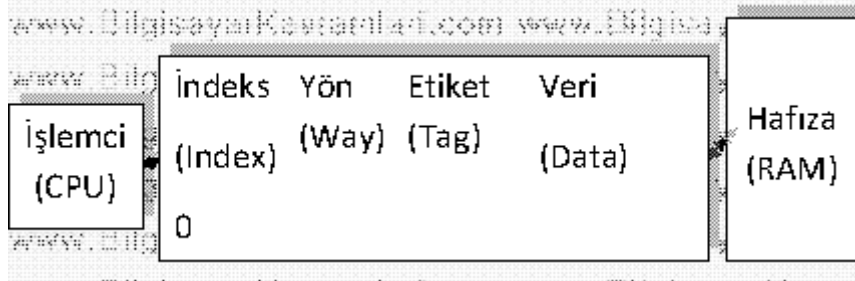
Belady's, Belady's algorithm, Belady algoritması

Yukarıdaki bu algoritmaları sırasıyla örnek üzerinden anlatmaya çalışalım. Örneğimizdeki önbelleğin iki indeksi bulunsun ve bütün küme ilişkilendirmesi (fully set associative) yapısında olsun. Buna göre aşağıdaki veriler önbelleğe geldiğinde sırasıyla yukarıdaki değiştirme algoritmalarının nasıl çalıştığını görelim:

Hafıza adresi	Veri
11010100	www.
10101001	bilgisayar
00010101	kavramlari

01001101	.com
----------	------

Yukarıdaki verileri aşağıdaki önbellek yapısına sırasıyla yerleştireceğiz. 2 satırdan oluşan önbelleği indekslemek için tek bit yeterlidir. Bu durumda hafıza adresinin ilk bitini önbellek indekslemesi için kullanacağız.

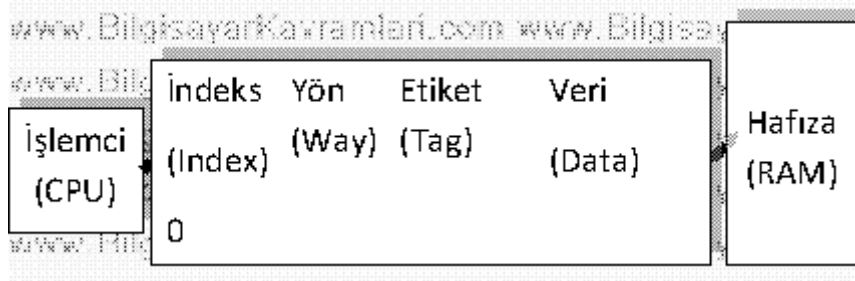


Bu durumda yukarıdaki ön bellek üzerinde bütün küme ilişkilendirme kullanılacaktır. Aslında önbellek iki satırdan oluştuğu için 2 yönlü küme ilişkilendirme de denilebilir.

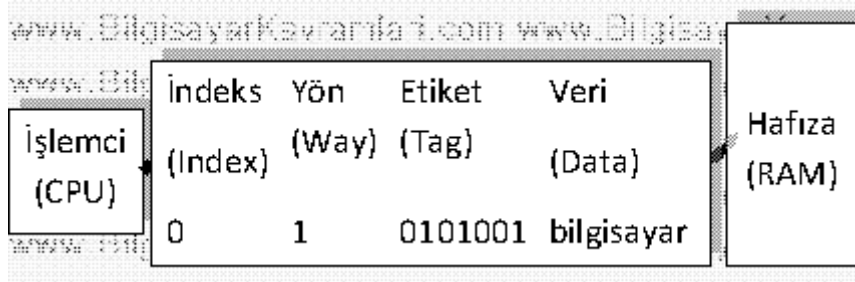
FIFO (First in first out, ilk giren ilk çıkar)

Bu algorithmada, önbellekte bir bilgi yazılacağı sırada, önbelleğe ilk girmiş, en eski verinin üzerine yazılması tercih edilir.

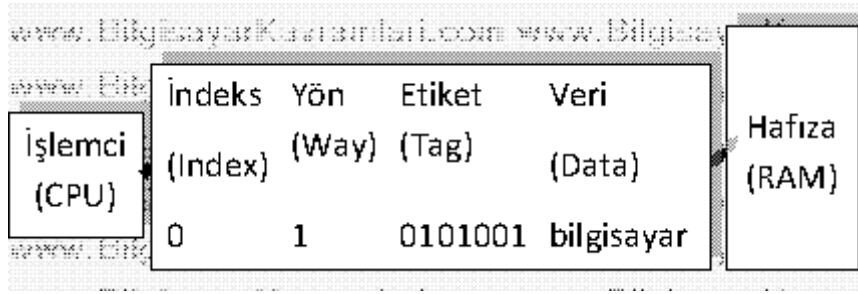
Yukarıdaki verilere erişimi sıra ile aşağıdaki şekiller üzerinden anlamaya çalışalım. İlk erişim 11010100 adresine yapılıyor.



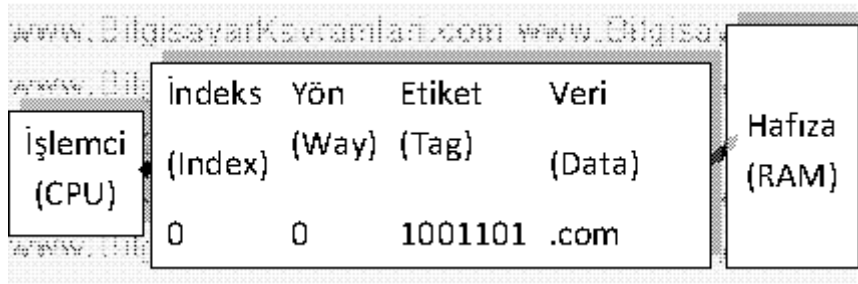
Bu adres 1 ile başladığı için 1. İndekse yerleştiriyoruz. Ardından gelen veri de 1 ile başladığı için 1. Adrese yerleşmesi gerekiyor ancak bu adres dolu ve 0. Adres boş, o halde boş olan yere yerleştiriliyor:



Şimdi gelen veri ise hem 0 hem de 1 e yerleşebilir (bütün küme ilişkilendirmesi olduğu için) dolayısıyla FIFO devreye giriyor ve en eski olan “www.” Bilgisi kaldırılıp yerine yazılıyor.



Sonraki veri yine iki indekse de yazılabilecek durumda ve veri en eski olan “bilgisayar”ın üzerine yazılıyor.



#### LRU (Least Recently Used, En Eski Erişilen)

Bu önbellek değiştirme algoritmasında amaç, önbellekte yapılan erişimleri takip etmek ve en geç erişilmiş olan veriyi değiştirmektir. Örneğimizi yine yukarıdaki gibi 2 satırlı bir önbellek üzerinden ve aşağıdaki erişim sırasıyla takip etmeye çalışalım:

1010 1010

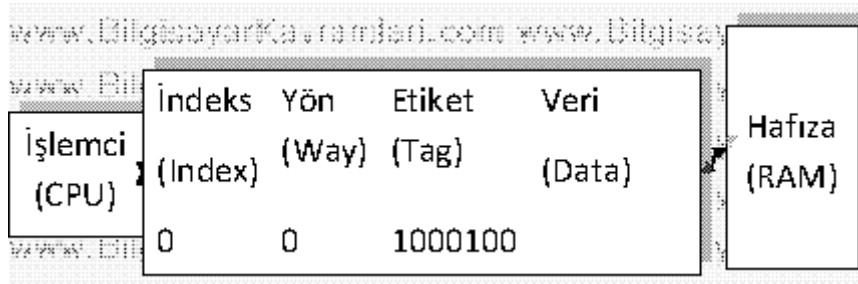
0100 0100

1010 1010

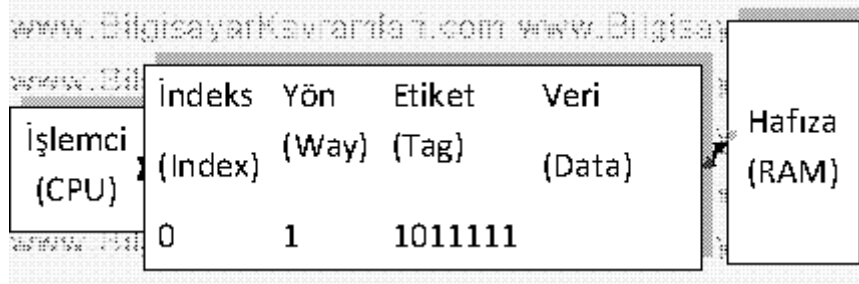
1101 1111

0100 0100

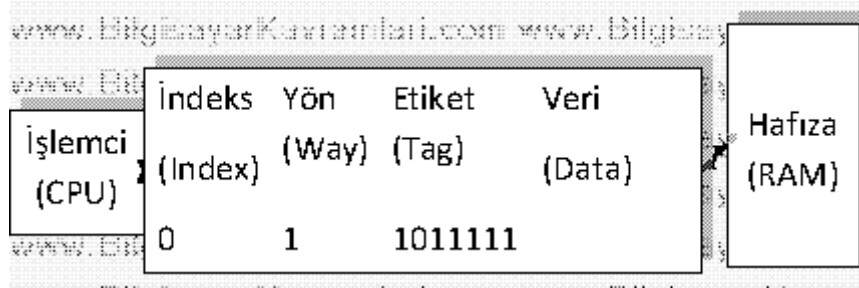
Yukarıdaki bu erişim sıralarına göre önbellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla önbelleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Buradan anlaşılacağı üzere en son erişilen veri ikinci satırdaki veridir. Yeni gelen veri en eski erişilmiş olan verinin üzerine yazılır:



Şu anda en son erişilen veri yeni yazdığımız veridir ve yeni gelen veri diğer verinin üzerine yazılır:



Görüldüğü üzere bu algorithmada, her zaman için en eski erişilmiş olan verinin üzerine yeni gelen veri yazılmaktadır.

#### LFU (Least frequently used, en nadir kullanılan veya en seyrek kullanılan)

Bu algorithmada amaç ön bellekte bulunan verilere yapılan erişim miktarlarını saymak ve bu erişim miktarlarından en az olanını değiştirmektir.

Bu durumu yine bir önceki alt başlıkta incelediğimiz örnek üzerinden anlamaya çalışalım.

1010 1010

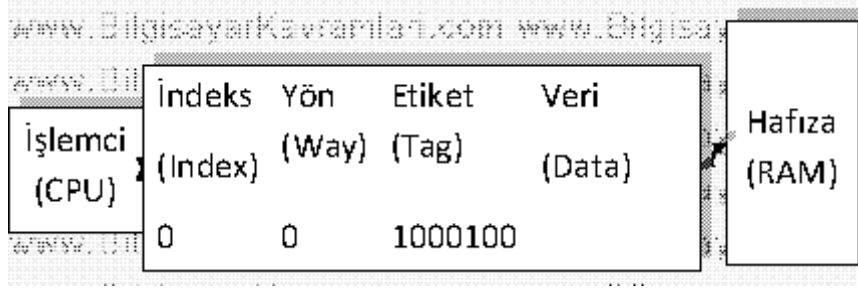
0100 0100

1010 1010

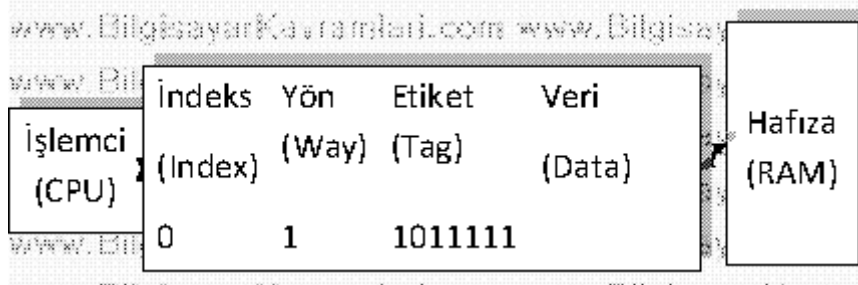
1101 1111

0100 0100

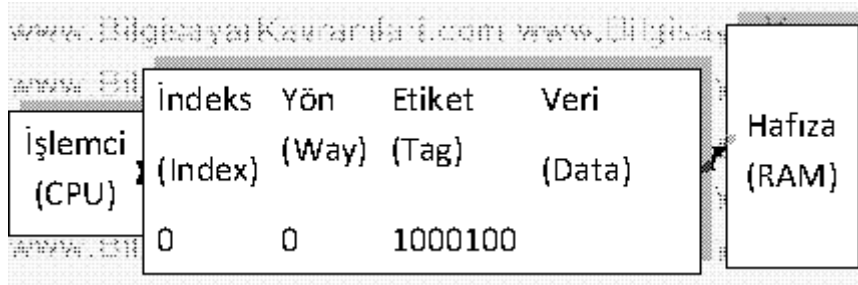
Yukarıdaki bu erişim sıralarına göre ön bellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla ön belleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Buradan anlaşılacağı üzere en son erişilen veri ikinci satırdaki veridir. Yeni gelen veri en az erişilmiş olan verinin üzerine yazılır. Burada en az erişilen veri 0. İndekste olan veridir.



Yeni yüklenen veri, şu andaki en az erişilmiş olan veridir. Dolayısıyla değiştirme işlemi yine 0. Satırdaki veri üzerine yapılır.



Görüldüğü üzere değiştirme işlemi sürekli olarak, o ana kadar en az erişilen önbellek alanında olmaktadır.

#### MRU (Most Recently Used, En sık kullanılan)

Bu değiştirme algoritmasında amaç en son erişilen veriyi değiştirmektir. Bu algoritma ilk başta çok anlamlı gelmeyebilir, sonuçta önbellekteki en taze bilgi en son erişilen bilgidir ve dolayısıyla işlemcinin bir sonraki adımda erişme ihtimali yüksek olan veridir.

Ancak bazı durumlarda anlamlı olabilir. Örneğin bir dosyadan sürekli olarak okuma yapıldığını veya bir sıkıştırma algoritmasının büyük bir dosyayı açmak için uğraştığını dolayısıyla hafızada yüklü bu dosyanın sürekli işlemci üzerinde işlendiğini ve işlenen bir veriye bir daha geri dönülmeyeceğini düşünün.

Örneğin 100MB boyutundaki sıkıştırılmış bir dosyayı açmak istiyoruz. Bu durumda dosyadan okunan ve işlenen bir bilgi kısmı açılacak ve işlemcinin dosyanın bu kısmı ile olan işi bitecektir.

Bundan sonraki adımlarda önbelleğe yüklenen verilerin tamamı taze olacak ve daha önceden yüklenen bir veriye ihtiyaç duyulmayacaktır.

Bu durumda önbelleğin bir daha kullanılmayacak veriler ile işgal edilmesi yerine önbellek üzerinde en son erişilen verinin değiştirilmesi ve diğer işlemlerin önbellek üzerinde kullandıkları verilerin daha sonraki erişimler için saklanması mantıklı olur.

Bu algoritmanın çalışmasını yine bir örnek üzerinden inceleyelim.

1010 1010

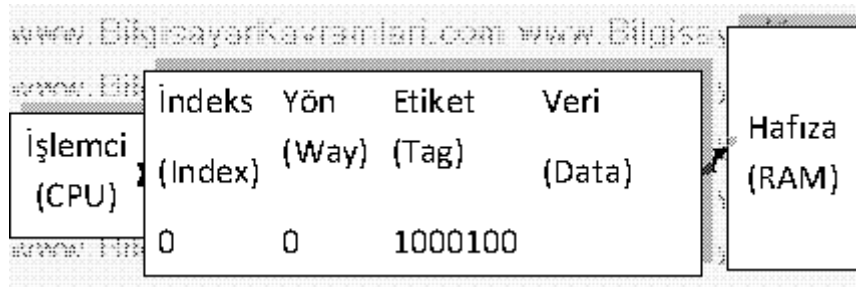
0100 0100

1010 1010

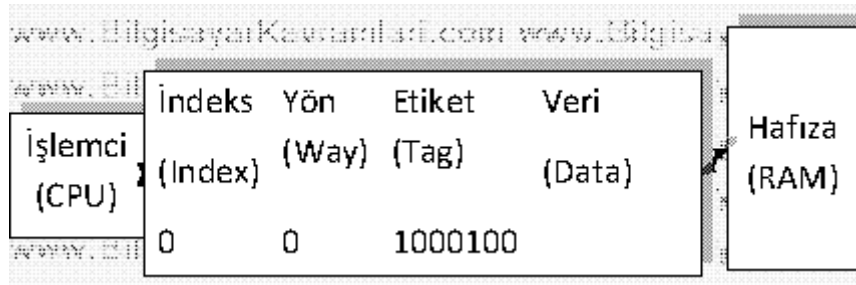
1101 1111

0100 0100

Yukarıdaki bu erişim sıralarına göre önbellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla önbelleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Buradan anlaşılacağı üzere en son erişilen veri ikinci satırdaki veridir. Yeni gelen veri en son erişilmiş olan verinin üzerine yazılır. Burada en az erişilen veri 1. İndekste olan veridir.



Yeni yüklenen veri, şu andaki en son erişilmiş olan veridir. Gelen erişim talebi ise zaten önbellekte bulunan veridir ve herhangi bir değiştirme işlemi gerekmeden çalışma sonuçlanır.

Belady's Algorithm (Belady algoritması)



Teorik olarak tanımlı olan bu algoritmanın gerçekte uygulanması mümkün değildir. Bu algoritma temel olarak ilerde en çok erişilecek hafıza adreslerini önbellekte tutmayı hedefler. Elbette gerçekte bir çalışma olmadan ilerde neyin çalışacağı bilinemeyeceği için de bu algoritmanın kullanılması imkansızdır.

Yine bir örnek üzerinden algoritmanın çalışmasını inceleyelim:

1010 1010

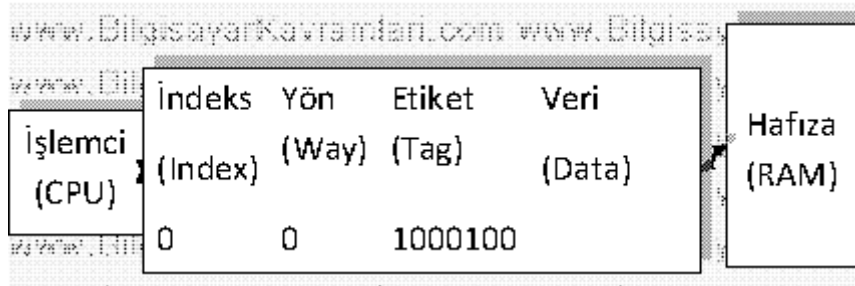
0100 0100

1010 1010

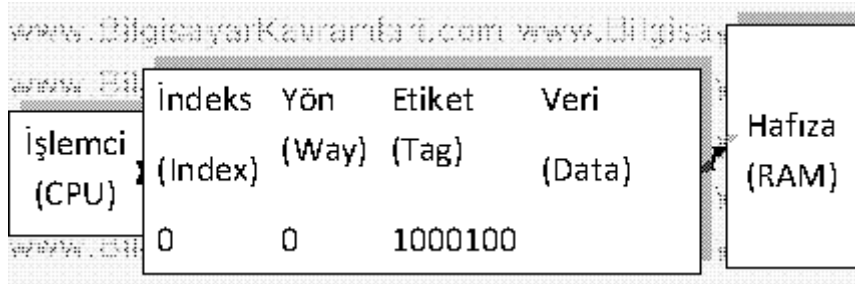
1101 1111

0100 0100

Yukarıdaki bu erişim sıralarına göre önbellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla önbelleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Daha sonra gelen erişimler incelendiğinde sıradaki erişim olan 11011111, şu anda 1. İndekste olan değer ile değiştirilmelidir çünkü 0100 0100 adresine ilerde erişim olacaktır ve bu bilgi ile değiştirilmesi durumunda ilerde yine bir önbellek değiştirme işlemi gerekecektir.



Görüldüğü üzere ilerde yapılacak değişiklik önceden hesaplanmış ve bunu engellemek için en makul değişiklik yapılmıştır.

Sonuçta bu algoritma ile her zaman en az önbellek kayıp oranı (cache miss) yakalanır.

**Kaynakça**

Aşağıdaki kitaplar kaynak olarak kullanılmıştır.

- Morris Mano, Computer System Architecture
- Tanenbaum, Structured Computer Organization
- J. Hayes, Computer Architecture and Organization
- R. Williams, Computer System Architecture
- D.A. Patterson, J.L. Hennessy, Computer Organization and Design
- V.C. Hamacher, Z. Vranesic, S. G. Zaky, Computer Organization

## **SORU 20: Thread (iplik, lif, iz)**

Bilgisayar bilimlerinde özellikle işletim sistemi (operating systems) konusunda kullanılan önemli terimlerden birisidir. Bir işletim sisteminde ya da yazılan bir programda birden fazla işin aynı anda yapılması için kullanılırlar.

Bilindiği üzere işlemci (CPU) anlık olarak tek iş çalıştırabilir, ancak işletim sistemi tasarımında kullanılan bazı yöntemlerle birden fazla iş aynı anda yapılıyor gibi işlemcinin kullanılması mümkündür. Bu noktada birden fazla işi yapmak için iki ayrı teknolojiye bahsedilebilir. Bunlardan en sık kullanılanı [işlemdir. \(process\)](#) . Bir işlem, işletim sisteminin kendisinden [çatallanan \(Fork\)](#) ve kendi başına hafızada yer kaplayan, işletim sisteminin [işlemci zamanlama algoritmasına \(cpu scheduling algorithm\)](#) bağlı olarak sırada bekleyen veya çalışan yapıdır.

Bu yazının konusu olan iplikler (threads) ise bazı kaynaklarda hafif işlem (lightweighted process) olarak geçmekle birlikte varlıkları bir işleme bağlı olan ve yine aynı anda birden fazla işi yapmaya yarayan yapılardır. Kısacası bir ipliğin (thread) vâd olması için önce bir işlemin var olması gerekir. Tersisi ise gerekmez yani her işlem çatallanması sonucunda bir iplik oluşturulmaz.

İpliklerin diğer önemli bir özelliği ise bir işlemin altında çalışıyor olmalarından dolayı bu işlemin içerisindeki bütün haklara sahip olmalarıdır. Bu haklardan önemli bir tanesi de hafıza erişim hakkıdır. Dolayısıyla bir işlem (process) tarafından üretilen bütün işlemler, bu işlemin hafızada kapladığı yere erişebilirler. Bu ise bize paylaşılmış hafıza (shared memory) olarak bilinen ve ipliklerin kodlanmasında önemli bir rol oynayan özelliği kazandırır.

Kabaca bir işlemdeki bir değişkeni sadece bu işlem değiştirebilir. Bir iplikteki değişkeni ise aynı işlem tarafından üretilmiş bütün iplikler değiştirebilir veya okuyabilir. Bu şekilde iplikler arasında paylaşılan değişkenlere de paylaşılan değişken anlamında shared variable ismi verilir.

İplikler ayrıca teknolojik olarak işlem üretme sorunu olan ortamlarda önem arz eder. Örneğin java teknolojisinde işlem üretme fırsatı yoktur. Bunun sebebi java kodlarının çalıştığı ve JVM (Java Virtual Machine) olarak bilinen java sanal makinesinin zaten bir işlem olarak çalışıyor olması ve bu işlemin kendisi dışında işlemler üretmediği durumlarda da başarılı bir şekilde çalışması isteniyor olmasıdır. Örneğin java kodları çok işlemli olmayan ve birden fazla işlemin üretilmeyeceği ortamlarda da çalışabilmelidir. (Burada java'nın WORA , write once run anywhere (bir kere yaz, her yerde çalıştır) sloganını hatırlamak gerekiyor). Bu yüzden bir java programı çalışacağı ortamda işlem üretebilecek olsa bile JAVA dilinde buna izin verilmemiştir ve bütün aynı anda yapılması istenen işlemler çoklu iplik (multi threaded) yapıya yüklenmiştir.

## SORU 21: İşlemci Zamanları (CPU Timing)

Bu yazının amacı, işletim sistemleri teorisinde sık kullanılan zamanlama kavramlarını açıklamaktır. Bu yazıda anlatılması hedeflenen kavramlar:

- Bekleme zamanı (waiting time)
- Dönüş süresi (turnaround time)
- İş üretimi (throughput)
- Cevap süresi (response time)

Yukarıdaki bu kavramların kısa tanımlarını yaptıktan sonra bir örnek üzerinden açıklamaya çalışalım.

CPU zamanlama algoritmalarında (CPU scheduling algorithms) kullanılan bekleme zamanı, bir işlemin (process) [bekleme sırasında \(ready queue\)](#) geçirdiği toplam süredir. Benzer şekilde dönüş süresi, [işlemin \(process\)](#) çalıştırılması ile bitmesi arasındaki toplam zamandır. Throughput (iş üretimi) ise bir işlemcinin birim zamanda tamamladığı [işlem \(process\)](#) sayısıdır. Bir algoritmadaki cevap süresi (response time) ise işlemin algoritma tarafından, ne kadar zaman içerisinde ilk kez cevap verildiğidir.

Bu durumların hepsini örnek bir çalışma üzerinden görmeye çalışalım.

Örneğin işlemcilerin sisteme varış zamanları (arrival time) ve çalışma süreleri (CPU burst time) aşağıdaki şekilde olsun.

İşlem	Ulaşma Zaman (Arrival Time)	Çalışma Süresi (Burst Time)
P1	0	5
P2	0	3
P3	7	5
P4	8	9

Yukarıdaki örneği [round robin algoritmasına](#) göre çalıştıralım ve zaman birimi (time quadrant) olarak 2 birim kabul edelim. Bu durumda her 2 birimlik zamanda bir işlem değişimi (context switch ) olacak ve [görevlendirici \(dispatcher\)](#) bekleme sırasından (ready queue) yeni bir işlemi alarak işlemciye gönderecektir.

Süre	0-2	2-4	4-6	6-7	7-8	8-10	10-12	12-14	14-16	16-17	17-22
İşlem	P1	P2	P1	P2	P1	P3	P4	P3	P4	P3	P4

Yukarıdaki gösterilen zaman çizgisinde önce geliş sırasına göre P1 işleminden başlanıyor ve zaman birimi olan 2 birim sonra bekleme sırasından bir sonraki işlem olan P2 çalıştırılıyor. Örneğin 7. Zaman biriminde P3 ulaşıyor ancak o sırada bekleme sırasında önde bulunan P1 işlemi bitirilerek 8. Zamanda P3 işlemi çalıştırılıyor.

Yukarıdaki bu örnekte, açıklamaya çalıştığımız zamanları hesaplayalım:

Sistemdeki işlemlerin bekleme zamanları aşağıdaki şekildedir:

P1 işlemi, 2-4 ve 6-7 zaman aralıklarında toplam 3 birim için bekleme sırasında (ready queue) beklemiştir.

P2 işlemi, 0-2 ve 4-6 zaman aralıklarında toplam 4 birim için bekleme sırasında (ready queue) beklemiştir.

P3 işlemi 7-8 , 10-12 ve 14-16 zaman aralıklarında toplam 5 birim için bekleme sırasında (ready queue) beklemiştir.

P4 işlemi 8-10, 12-14 ve 16-17 zaman aralıklarında toplam 5 birim için bekleme sırasında (ready queue) beklemiştir.

Ortalama bekleme zamanı için bütün işlemlerin bekleme zamanlarını toplayarak işlem sayısına bölüyoruz. Bu durumda  $3 + 4 + 5 + 5 = 17 / 4 = 4.25$  birimlik ortalama bekleme süresi vardır denilebilir.

Dönüş süresi hesaplanırken bir işlemin sisteme ulaşması ile bitmesi arasındaki zaman hesaplanmalıdır. Bu hesap her işlem için aşağıda verilmiştir.

P1 işlemi sisteme 0. Zamanda ulaşmış ve 8. Zamanda bitmiştir. Bu durumda dönüş süresi 8. Birimdir.

P2 işlemi sisteme 0. Zamanda ulaşmış ve 7. Zamanda bitmiştir. Bu durumda dönüş süresi 7. Birimdir.

P3 işlemi sisteme 7. Zamanda ulaşmış ve 17. Zamanda bitmiştir. Bu durumda dönüş süresi 10. Birimdir.

P4 işlemi sisteme 8. Zamana ulaşmış ve 22. Zamanda bitmiştir. Bu durumda dönüş süresi 14. Birimdir.

Ortalama dönüş süresi (average turn around time) ise bu değerlerin ortalamasıdır :  $8 + 7 + 10 + 14 = 39 / 4 = 9.75$  birim zaman olarak hesaplanabilir.

İş üretimi (throughput) hesabı için, sistemde toplam 22 birim zamanlık iş yapılmış ve bu süre içerisinde 4 işlem çalıştırılmıştır. Bu durumda  $22 / 4 = 5.5$  olarak hesaplanabilir.

Cevap sürelerini yine her işlem için ayrı ayrı hesaplayacak olursak:

P1 işlemi sisteme 0. Zamanda ulaşmış ve 0. Zamanda işlemcide ilk kez çalıştırılmaya başlamıştır. Bu durumda cevap süresi (response time ) 0. Birimdir.

P2 işlemi sisteme 0. Zamanda ulaşmış ve 2. Zamanda işlemcide ilk kez çalıştırılmaya başlamıştır. Bu durumda cevap süresi (response time ) 2. Birimdir.

P3 işlemi sisteme 7. Zamanda ulaşmış ve 8. Zamanda işlemcide ilk kez çalıştırılmaya başlamıştır. Bu durumda cevap süresi (response time ) 1. Birimdir.

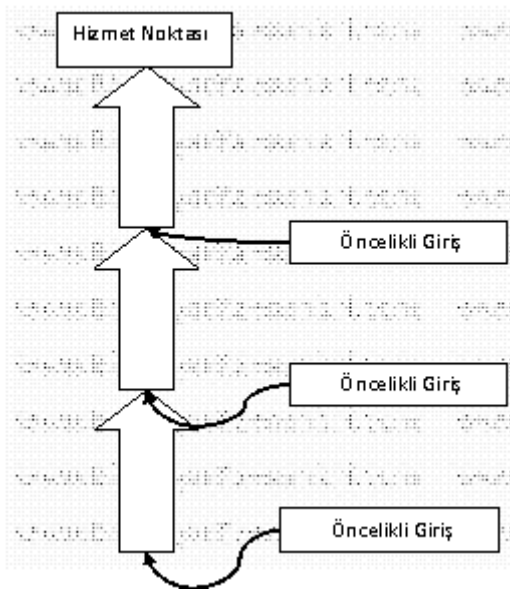
P4 işlemi sisteme 8. Zamanda ulaşmış ve 10. Zamanda işlemcide ilk kez çalıştırılmaya başlamıştır. Bu durumda cevap süresi (response time ) 2. Birimdir.

Ortalama cevap süresi (average response time) hesaplanırken yine bu sayıların ortalaması bulunur:  $0 + 2 + 1 + 2 = 5/4 = 1.25$  birim zamandır.

## SORU 22: Çok Seviyeli Sıralar (Multi Level Queues)

Bilgisayar bilimlerinde kullanılan bir veri yapısı (data structure) çeşididir. Çalışma yapısı olarak sıraya (queue) benzetilebilir.

Çok seviyeli sıralarda, [sıralara \(queue\)](#) benzer şekilde [ilk giren ilk çıkar \(first in first out, FIFO\)](#) mantığı geçerlidir. Ancak sıranın birden çok girişi vardır ve bu anlamda, değişik hızlarda ilerleyen birden çok sıranın birleşimi gibi düşünülebilir.



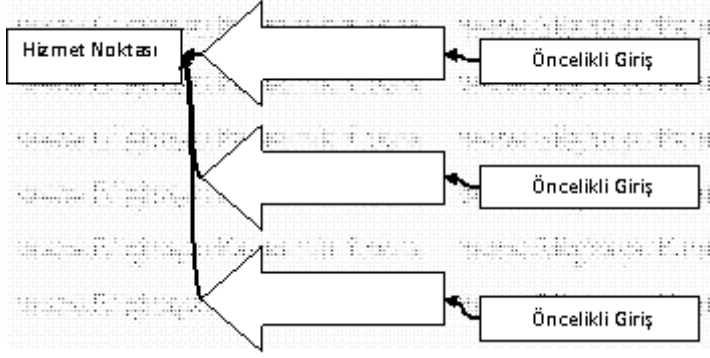
Yukarıdaki şekilde, 3 seviyeli bir sıra gösterilmiştir. Sonuçta bütün sıralar, aynı hizmet noktasına doğru hareket etmektedir. Ancak bazı sıra elemanları, diğerlerine göre daha öncelikli noktalardan başlayabilmektedir. Bu anlamda bazı işlemler daha öncelikli olmaktadır. Bu açıdan çok seviyeli sıralar (multi level queues) birer [rüşhan sırası \(priority queue, öncelik sırası\)](#) olarak düşünülebilir.

Yukarıdaki benzetmeyi gerçek hayatta bir bankada sıra bekleyen kişilere benzetmek mümkündür. Örneğin banka kendi müşterilerine 1. Önceliği verirken, kredi işlemi yapan müşterilere ikinci ve diğer müşterilere 3. Önceliği verebilir.

Elbette yukarıdaki çizimde ve anlatımda problem olabilecek nokta, [kıtlıktır \(starvation\)](#). Yani sürekli olarak bankaya kendi müşterisi gelmesi durumunda, diğer müşterilere sıra asla gelmeyebilir.

Bunun çözümü olarak, düşük seviyelerden, daha yüksek seviyelere belirli aralıklarla bazı kişilerin geçirilmesi gerekir.

Yukarıdaki anlatımı aşağıdaki şekilde düşünmek de mümkündür:



Yukarıdaki örnekte 3 ayrı sıra (3 seviye) bulunmakta ve bu sıralar birbirine bağlanmak yerine doğrudan hizmet noktasına bağlanmaktadır. Bu yeni çizimimiz ile gösterilen veri yapısı ise çok seviyeli geri besleme sırası (multilevel feedback queue) olarak isimlendirilmektedir. İki yapı arasındaki en önemli fark, ilkinde işlemlerin sıralar arasında geçiş yapabilmesi ancak bu yeni yapıda işlemlerin girdikleri sırada kalmasıdır.

Bu gösterimde her sıra, farklı hızda ilerlemekte ve hizmet noktası, her sıradan farklı hızda işlem kabul etmektedir. Örneğin her turda, ilk sıradan 6, ikinci sıradan 3 ve son sıradan tek kişi alındığını kabul edelim. Bu durumda ilk sıradaki işlemler daha yüksek öncelikte olacak ve diğer sıralarda kendi hızlarına göre birer öncelik belirlemiş olacaklardır.

İşletim sistemleri (operating systems) açısından da oldukça kullanışlı olan çok seviyeli sıralar [işlemci zamanlamasında \(CPU Scheduling\)](#) kullanılmaktadırlar. Örneğin CPU ile daha yoğun çalışan ve nispeten kısa olan işler, en hızlı sıraya konulmakta, Giriş çıkış ( I / O ) işlemleri nispeten yavaş olduğu için daha düşük seviyeye ve nihayet en son seviyeye kullanıcının özel olarak zamanını verdiği zamanlanmış işlemler yerleştirilmektedir. Bu sayede uzun süren I/O işlemlerini daha nadir yapmak ve daha önemli ve kısa olan CPU işlemlerini daha çabuk yapmak mümkün olmaktadır.

### **SORU 23: İçerik Değiştirme (Context Switching)**

Bu yazıda anlatılan içerik değiştirme (context switching) konusunu anlamadan önce bilgisayarlarda bulunan işlemcinin (CPU) anlık olarak tek bir iş ile uğraşabileceğini söylememiz gerekiyor. İşletim sistemi tasarımında (operating system design) bulunan bir özellik sayesinde, anlık olarak işlemcide tek iş çalıştırılması ve yinede birden fazla işin bilgisayarda aynı anda çalışıyormuş gibi hissettirilmesi mümkündür.

Aslında tek işlem çalıştırıp birden fazla iş yapıyormuş gibi gösteren işletim sistemi özelliği kısaca [çok işlemlilik \(multiprocess\)](#) olarak geçer ve bu özellik işlemcinin (CPU), [işlemleri \(process\)](#) sırayla çalıştırması sayesinde elde edilir. Yani bilgisayarda iki program çalışıyorsa (birinci programa A ikincisine B diyelim). Bilgisayar sırayla bir A programını bir de B programını çalıştırır. Bu sayede programlardan hiçbirisi aslında bekletilmeden işlemciye belirli aralıklarla erişme imkanı bulur ve biraz A programı biraz B programı çalıştırılarak aynı anda çalışıyor izlenim olur.

Bu programlar arasında işlemcinin geçiş yapmasına da içerik değiştirme (context switching) ismi verilir. Aslında her içerik değiştirmenin bilgisayar açısından bir maliyeti vardır. Dolayısıyla içerik değiştirmek aslında iyi bir özellik olmasına karşılık bir de maliyeti vardır ve oranı yükseldikçe dezavantaj haline gelir.

## SORU 24: Sembolik Bağ (Symbolic Link)

Bilgisayar bilimlerinde, işletim sistemi konusunda kullanılan bir terimdir. İşletim sisteminin temel fonksiyonlarından birisi de bilgisayarın [sabit diskini \(hard disk\)](#) ve bu disk üzerindeki dosyalama yapısını kontrol etmektir. Bu anlamda çeşitli işletim sistemi üreticileri çeşitli tasarımlar yapmış ve farklı dosyalama sistemleri geliştirilmiştir. Bunlardan en çok bilinenleri FAT, NTFS ve ext şeklinde sayılabilir.

Bu dosya sistemlerinin hepsinde bilgiler dosyalar içerisinde tutulur. Yani bir dosyanın ismi, tarihi ve sahibi bulunmakta yarıca dosyanın boyutunu da belirleyen bir içeriği bulunmaktadır. Örneğin bir kelime işlem programı ile (örn. Word) yazılan yazıların saklandığı tek bir dosya diskte saklanabilir ve istenildiğinde bu dosyaya dönülerek saklanan bilgilere ulaşılabilir.

Dosyalama sistemlerinin (file systems) ortak özelliklerinden birisi de dosyaların klasörler içerisinde saklanma imkanındır. Dizin (directory) veya klasör (folder) ismi verilen bu yapılar birden fazla dosyayı veya yine kendi cinslerinden dizin veya klasörü barındırabilir.

Dolayısıyla her dosya aslında bir dizinin (veya klasörün) içerisinde bulunur. İşte tam bu noktada sembolik bağlar anlam kazanır. Şayet dosyamızı bir klasörün içinde bulunuyor ve aynı zamanda farklı bir klasöründe içinde olmasını istiyorsak, yani dosyanın birden fazla kopyasının olmasını değil ama birden fazla yerden erişilmesini istiyorsak, dosyayı bu farklı konumlara kopyalamak yerine birer bağ oluşturabiliriz. Bu sayede dosya tek bir yerde saklanacak ve diskte bir kere yer kaplayacak ancak bağ oluşturulan her yerden erişilebilecektir.

Linux üzerinde sembolik bağ oluşturmak için “ln” komutu kullanılır (link kelimesinin kısaltması) ve sembolik bağ için -s parametresi kullanılır (Sembolik kelimesinin baş harfi)

Şayet -s parametresi kullanılmazsa bu tip kurulan bağlara sıkı bağ (hard link) ismi verilir.

Bağlantı kurmak için aşağıdaki şekilde komut verilebilir:

ln -s hedef kaynak

Örneğin /home/sadievrenseker dizini altında yeni bir link oluşturulmak istensin. Bağlantının hedefi de /usr/bin/xterm olsun. Bu durumda komut aşağıdaki şekilde verilecektir:

ln -s /home/sadievrenseeker/xterm /usr/bin/xterm

Yukarıdaki komut verildikten sonra /home/sadievrenseker altında “xterm” isimli bir bağlantı oluşacaktır ve ls -l komutu ile bu dizinin içeriği gösterildiğinde aşağıdaki şekilde bir görüntü oluşur:

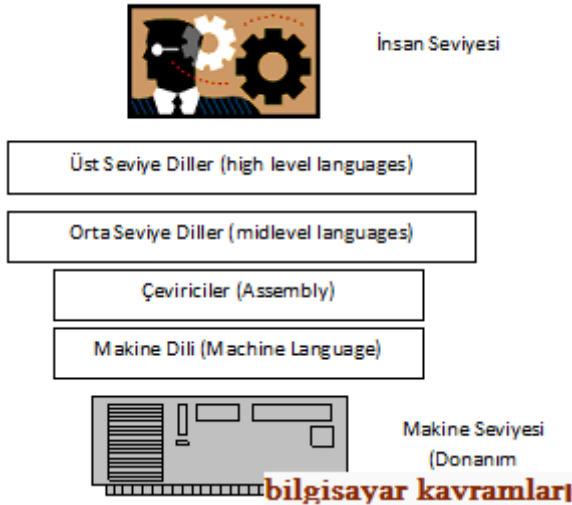
```
-rwx-----+ 1 shedai None 13520 Sep 29 13:10 shedai.rar
-rwx-----+ 1 shedai None 438 Sep 29 13:05 tokens.h
drwx-----+ 4 shedai None 0 Oct 17 00:22 ttk-1.0
lrwxrwxrwx 1 shedai None 14 Nov 8 22:10 xterm -> /usr/bin/xterm
shedai@shedai-PC ~
$
```

www.BilgisayarKavamlari.com

Yukarıdaki şekilde görüldüğü üzere, ls komutu ile dizinin içeriği görüntülendiğinde, en baştaki sembol “l” harfi olarak belirtilmiştir. Bunun anlamı bu dizin elemanının bir bağlantı (link) olduğudur ve ayrıca detay kısmında xterm elemanının /usr/bin/xterm’e bir bağlantı olduğu da görüntülenmiştir.

### SORU 25: Makine Dilleri (Machine Language)

Bilgisayar bilimlerinde programlama dillerinin en alt seviyesini oluşturan ve insan tarafından neredeyse tamamen anlaşılmaz buna karşılık makine için en anlaşılır dildir. Aslında makine dilini anlamak için dillerin seviyelerini anlamakta yarar vardır.



Yukarıdaki şekilde de tasvir edildiği üzere dilleri seviyelere bölecek olursak en altta donanıma en yakın ve donanım tarafından en anlaşılır olan makine dilinden (machine language) bahsedebiliriz. Esas itibarıyla makine dili [ikilik tabanda \(binary\)](#) sayılardan ibarettir ve bu sayılar makinede işlenen dijital sinyallerin birer göstereimidir. Yani diğer bir dille makinede kullanılan ve her birisi farklı anlamlara gelen sinyallere makine dili ismi verilebilir.

Makine dilinin hemen üzerinde makine diline çok yakın olan ve makinedeki her şeyin kodcu tarafından bilinmesi ve takip edilmesi gereken [çeviriciler \(assembly\)](#) gelir. Bunun üzerindeki dil sınıfları literatürde farklılık gösterir. Bazı kaynaklarda doğrudan üst seviye diller olarak nitelendirilen bir katman bulunurken bazı kaynaklarda ara bir seviye olan orta seviye (midlevel) diller bulunur. Burada kast edilen hala donanıma [ve çevirici \(assembly\)](#) seviyesinin mümkün olduğu ancak istenirse bu tip kodlamalarla hiç muhatap olunmayan dillerdir.



Örneğin C bu seviyedeki dillere bir örnek olabilir. Dilenirse RAM (hafıza) ve işlemci (CPU) üzerinde doğrudan düşük seviye işlemler yapılabilir veya bu işlemlere hiç girilmeden C kodları da yazılabilir. Bu açıdan C dili hem düşük hem de yüksek seviye programlama dili olarak düşünülebilir kullanılabilen orta seviye bir dildir.

Yüksek seviye dillerle kast edilen ise genelde bilgisayarın donanımından uzak, nispeten insanlar tarafından anlaşılması daha kolay dillerdir. Örneğin pascal, ada, cobol, fortran gibi diller bu seviyeden kabul edilebilir.

Ayrıca literatürde yüksek seviye dillerin üzerinde de seviyeler belirlenmektedir. Bazı kaynaklarda görsel diller (visual languages) 5. Seviye olarak kabul edilir. Örneğin visual basic, oracle forms and reports, Microsoft Access veya SAP menu painter gibi görsel yazılım ortamları ve bu ortamları destekleyen diller (ki dil oldukları bazı kaynaklarda tartışmalıdır) 5. Seviye dil olarak kabul edilebilirler.

Benzer şekilde [nesne yönelimli dilleri \(object oriented languages\)](#) 5. Seviye kabul eden kaynaklar olduğu gibi insanın algılaması ve anlaması nispeten daha kolay olan modelleme dillerini 5. Seviye dil olarak kabul eden kaynaklarda bulunur. Örneğin UML ve benzeri çizimleri bu grupta kabul edebiliriz.

Dillerin sınıflandırılmasına yukarıdaki şekilde değindikten sonra konumuz olan makine dillerine dönelim.

Makine dillerini tanımlarken her zaman için mikrokodları (microcode) ayrı tutmak gerekir. Mikrokodlar veya bazı kaynaklarda firmware (fabrika yazılımları) olarak da geçen yazılımlar basitçe bir cihaz üzerine fabrikadaki üretimi sırasında yazılmış olan ve değiştirilemeyen yazılımlardır. Örneğin bir kol saatinin veya hesap makinesinin veya ADSL modem üzerinde basılı olarak gelen yazılımlardır. Bu yazılımların makine dillerinden ayrı tutulmasının sebebi değiştirilemez olmasıdır. Makine dilinin en temel özelliği bir dil olması ve makinenin anlayabileceği seviyeye yakın olmasıdır. Dolayısıyla bu tanımda makineyle iletişimin var olduğu kabul edilmiştir (dil zaten iletişimi gerektirir). İşte bu sebeple mikro kod (veya fabrika kodu) bir makine dili olarak sınıflandırılmamıştır (istisnalar sayılmazsa).

Makine dilini daha iyi anlayabilmek için C dilinde yazılmış bir kodun çevrimini incelemeye çalışalım:

```
#include <stdio.h>
int main() {
    printf("sadi evren seker");
    return 0;
}
```

**bilgisayar kavramları**

Yukarıdaki basit C kodunu [derlediğimiz \(compile\)](#) zaman makine kodu çıkıyor. Bu kodu basit bir editorle (bu örnekte notepad++ kullanılmıştır) açar ve incelersek aşağıdakine benzer bir dosya görürüz:

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayar kavramları.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

```
00000f90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000fa0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000fb0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000fc0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000fd0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000fe0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000ff0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00001000 73 61 64 69 20 65 76 72 65 6e 20 73 65 6b 65 72 sadi evren seker
00001010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00001020 2d 4c 49 42 47 43 43 57 33 32 2d 45 48 2d 32 2d -LIBGCCW32-EH-2-
00001030 53 4a 4c 4a 2d 47 54 48 52 2d 4d 49 4e 47 57 33 SJLJ-GTHR-MINGW3
00001040 32 00 00 00 77 33 32 5f 73 68 61 72 65 64 70 74 2...w32_sharedpt
00001050 72 2d 3e 73 69 7a 65 20 3d 3d 20 73 69 7a 65 6f r->size == sizeo
00001060 66 28 57 33 32 5f 45 48 5f 53 48 41 52 45 44 29 f(W32_EH_SHARED)
00001070 00 25 73 3a 25 75 3a 20 66 61 69 6c 65 64 20 61 .%s:%u: failed a
00001080 73 73 65 72 74 69 6f 6e 20 60 25 73 27 0a 00 00 ssertion '%s'...
00001090 2e 2e 2f 2e 2e 2f 67 63 63 2f 67 63 63 2f 63 6f ../../gcc/gcc/co
000010a0 6e 66 69 67 2f 69 33 38 36 2f 77 33 32 2d 73 68 nfig/i386/w32-sh
000010b0 61 72 65 64 2d 70 74 72 2e 63 00 00 47 65 74 41 ared-ptr.c..GetA
000010c0 74 6f 6d 4e 61 6d 65 41 20 28 61 74 6f 6d 2c 20 tomNameA (atom
bilgisayar kavramları
```

Yukarıda bu makine koduna çevrilmiş halinin sadece bir kısmı alınmıştır. Burada anlamlı yazılar çıktığına dikkat edebilirsiniz. Bunun sebebi editörümüzün makine dilindeki ikilik tabandaki bilgileri okunabilir halde işlemeye çalışmasıdır (bu örnekte notepad++ ANSI olarak ayarlanmıştır).

Aslında makine dilini tam olarak görmek için [onaltılık tabanda \(hexadecimal\)](#) okumak gerekir.

Yukarıdaki metni onaltılık tabana çevirecek olursak (hex editor ile görüntüleyecek olursak, bu örnekte notepad++ için hex eklentisi kullanılmıştır):

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000f90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000fa0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000fb0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000fc0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000fd0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000fe0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000ff0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00001000	73	61	64	69	20	65	76	72	65	6e	20	73	65	6b	65	72	sadi evren seker
00001010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00001020	2d	4c	49	42	47	43	43	57	33	32	2d	45	48	2d	32	2d	-LIBGCCW32-EH-2-
00001030	53	4a	4c	4a	2d	47	54	48	52	2d	4d	49	4e	47	57	33	SJLJ-GTHR-MINGW3
00001040	32	00	00	00	77	33	32	5f	73	68	61	72	65	64	70	74	2...w32_sharedpt
00001050	72	2d	3e	73	69	7a	65	20	3d	3d	20	73	69	7a	65	6f	r->size == sizeo
00001060	66	28	57	33	32	5f	45	48	5f	53	48	41	52	45	44	29	f(W32_EH_SHARED)
00001070	00	25	73	3a	25	75	3a	20	66	61	69	6c	65	64	20	61	.%s:%u: failed a
00001080	73	73	65	72	74	69	6f	6e	20	60	25	73	27	0a	00	00	ssertion '%s'...
00001090	2e	2e	2f	2e	2e	2f	67	63	63	2f	67	63	63	2f	63	6f	../../gcc/gcc/co
000010a0	6e	66	69	67	2f	69	33	38	36	2f	77	33	32	2d	73	68	nfig/i386/w32-sh
000010b0	61	72	65	64	2d	70	74	72	2e	63	00	00	47	65	74	41	ared-ptr.c..GetA
000010c0	74	6f	6d	4e	61	6d	65	41	20	28	61	74	6f	6d	2c	20	tomNameA (atom

bilgisayar kavramları

Yukarıdaki resimde de görüldüğü üzere en sol kolonda adres, ortada onaltılık kodlama ve en sağda ise kodlamanın ansi karşılığı görüntülenmektedir.

Yukarıdaki örnek kod için 18.3kb uzunluğunda olan dosyanın kaynak kodu 79bayttır. Bu dosyanın bilgisayarın anlayacağı seviyeye çevrilmesi ve aslında onaltılık tabanda gösterilen sinyallerin işlenmesi sonucunda bilgisayarda çalıştığını söyleyebiliriz. Burada CPU, RAM gibi donanımın yanında işletim sisteminin de etkisi bulunmaktadır.

## **SORU 26: Eşlemeli Metotlar (Synchronized Methods)**

JAVA, C++ veya C# gibi nesne yönelimli programlama dillerinde kullanılan bir terimdir. Basitçe, aynı anda çalışan birden fazla [lifin \(thread\)](#) veya işlemin (process) sıralı olmasını ve birbiri ile iletişim halinde çalışmasını sağlar.

Nesne yönelimli programlama ortamında iki farklı kavram birbirine sıkça karışmaktadır. Aslında anlam olarak birbirine yakın olan synchronized methods (eşlemeli metotlar, synchronous method) ve synchronized statements (eşlemeli satırlar) kullanımda ufak farklılıklara sahiptir.

Bir metodun eşlemeli olması durumunda metottaki bütün işlemler, bu metodu çağıran [lifler \(threads\)](#) tarafından sırayla yapılır. Yani bir lif (thread) bu metodu çalıştırırken bir diğeri beklemek zorundadır.

Benzer şekilde eşlemeli satırlarda ise kritik alan (critical section) ismi verilen bir veya daha fazla satırdan oluşan bir blok olur. Bu blok eşlemeli metotlarda olduğu gibi liflerin (threads) sırayla buradaki komutları çalıştırmasını gerektirir.

Öncelikle JAVA dilinde metotların nasıl eşlemeli (synchronized) yapıldığına bakalım:

```
public class Sayac {
    private int c = 0;
    public void arttir() {
        c++;
    }
    public void azalt() {
        c--;
    }
    public int deger() {
        return c;
    }
}
```

Yukarıdaki kodda bir sınıf tanımlanmış ve c isminde bir int değişken bu [sınıf \(class\)](#) içerisinde bir sınıf değişkeni (class variable) olarak tanımlanmıştır. Yukarıdaki bu sınıfı kullanan bütün lifler (threads) bu değişkene erişme ve değiştirme hakkına sahiptir. Örneğin iki farklı lif (thread) aynı anda arttir() metodunu çağırırsa ve c değişkeninin ilk değeri 0 ise, değişken iki kere arttırılarak 2 olabilir veya iki lif (thread) tarafından da birer kere arttırılarak 1 olabilir.

Yani iki lif (thread) de aynı anda değişkenin değerini aldılar (0 olarak) arttırdılar (1 oldu) ve aynı anda değişkenin içine yazdılar. Dolayısıyla değişken iki farklı lif (thread) tarafından arttırılmasına rağmen 1 değerine sahip olmuş olabilir. Bu problem bilgisayar bilimlerinde sıkça karşılaşılan get and set (alma ve koyma) problemidir ve bu iki işlemin [bölünemez \(atomic\)](#) olmamasından kaynaklanır.

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayar kavramları.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

Yukarıdaki bu klasik problemin çözümü fonksiyona bir [lif \(thread\)](#) erişirken başka birisinin erişmemesidir. Böylelikle fonksiyona erişen her lif (thread) mutlaka değeri bir arttıracak ve sonuç olarak c değişkeninin değeri tam olarak bilinebilecektir.

Yukarıdaki bu eşlemeli erişimi sağlamak için kodun aşağıdaki şekilde değiştirilmesi gerekir:

```
public class synchronizedSayac {  
    private int c = 0;  
    public void synchronized arttir() {  
        c++;  
    }  
    public void synchronized azalt() {  
        c--;  
    }  
    public int deger() {  
        return c;  
    }  
}
```

Yukarıdaki yeni kodda dikkat edileceği üzere metodlar synchronized kelimesi ile başlamakta ve dolayısıyla erişim anlık olarak tek life (thread) izin vermektedir.

Yukarıdaki kodda deger() fonksiyonu synchronized yapılmamıştır. İlk bakışta bu durum doğru gibi gelir. Genelde şu hatalı kanı yaygındır: Bunun sebebi bu fonksiyonun içeriğinde karışıklık sebebi olacak bir iş yapılmamasıdır. Yani anlık olarak değişkenin değerinin okunması mümkündür ve bu okuma işlemi diğer liflerde (threads) bir probleme yol açmaz.

Bu kanı doğrudur ve gerçektende diğer liflerde (threads) bir soruna yol açmaz ancak deger() metodunu çağıran lif(thread) için sorun vardır. Yani o anda bir lifin (thread) arttir() fonksiyonuna eriştiğini ve bizim de deger() fonksiyonuna eriştiğimizi düşünelim. Acaba deger() fonksiyonu arttırılmadan önceki değerimi arttırıldıktan sonraki değerimi döndürecek?

İşte bu yüzden bu fonksiyonu da eşlemeli yapmak gerekir ve kodun doğru hali aşağıdaki şekildedir:

```
public class synchronizedSayac {  
    private int c = 0;  
    public void synchronized arttir() {  
        c++;  
    }  
    public void synchronized azalt() {  
        c--;  
    }  
    public int synchronized deger() {  
        return c;  
    }  
}
```

[Yapıcılar \(constructors\)](#) eşlemeli olamaz. Yani bir [yapıcı fonksiyonunun \(constructor\)](#) başına `synchronized` kelimesi yazılması hatadır. Bu gayet açıktır çünkü zaten bir yapıcıya (constructor) anlık olarak tek lif (thread) erişebilmektedir.

C# dilinde senkron metotlar için sadece yazılış farklılığı vardır. Örneğin yukarıdaki JAVA koduyla aynı işi yapan csharp kodu aşağıdaki şekildedir:

```
using System;
using System.Runtime.CompilerServices;
public class synchronizedSayac {
    private int c = 0;
    [MethodImpl(MethodImplOptions.Synchronized)]
    public void arttir() {
        c++;
    }
    [MethodImpl(MethodImplOptions.Synchronized)]
    public void azalt() {
        c--;
    }
    [MethodImpl(MethodImplOptions.Synchronized)]
    public int deger() {
        return c;
    }
}
```

Görüldüğü üzere metodun başındaki `synchronized` terimi yerine `[MethodImpl(MethodImplOptions.Synchronized)]` gelmekte ve `System.Runtime.CompilerServices` paketi programa dahil edilmelidir.

## Eşlemeli satırlar

Programlama dillerinde bazen bir metodun tamamına değil de sadece bir veya birkaç satırı içeren bir bloğa anlık olarak tek bir lifin (thread) girmesi istenebilir. Bu durumda ilgili bu satırların eşlemeli (synchronized) yapılması yeterlidir. Bunun için JAVA dilinde, aşağıdakine benzer bir kodlama gerekir:

```
public void isimEkle(String isim) {
    synchronized(this) {
        sayac++;
    }
    isimListesi.add(isim);
}
```

Yukarıdaki kodda görüldüğü üzere `isimListesi` isimli bir listeye yada vektöre (vector) isim eklenmektedir. Bu işlemin sıralı olması gerekmez. Yani listeye her lif(thread) değişik zamanlarda ekleme yapabilir. Ancak eklenen isimlerin sayısının doğru tutulması açısından sayacın artırılma işlemine erişen liflerin (thread) aynı anda değiştirme yapmaması gerekir. Bunun için sadece bu satırı eşlemeli (synchronized) yapan kod eklenmiştir.

Aynı kod C# için aşağıdaki şekilde yazılabilir:

```
public void isimEkle(String isim) {
    lock(this) {
        sayac++;
    }
}
```

```
    isimListesi.add(isim);  
}
```

Görüldüğü üzere iki dil arasındaki tek fark, synchronized terimi yerine lock teriminin kullanılmasıdır.

### **SORU 27: Priority Queue (Öncelik Sırası, Rüçhan Sırası)**

Bilgisayar bilimlerinin özellikle veri yapıları (data structures) konusunda sıkça kullanılan bir veri yapısının ismidir. Basitçe klasik bir [sıranın \(queue\)](#) üzerine öncelik değerinin eklenmesi ile elde edilir.

Bilindiği üzere normalde [sıralar \(queue\)](#) ilk giren ilk çıkar (FIFO , first in first out) mantığı ile çalışırlar. Yani bir bilet sırasında olduğu gibi sıraya ilk giren kişi hizmete ilk ulaşan, son giren kişi ise son ulaşan kişidir.

Öncelik sırasında (rüşhan sırası, priority queue) ise bu sırada bekleyenlerden en öncelikli olan kişinin ilk erişmesi beklenir. Bu durumu bir örnek üzerinden anlatmak gerekirse sırada bekleyen kişiler ve öncelikleri aşağıdaki şekilde verilmiş olsun:

Kişi No	Öncelik
1	5
2	3
3	1
4	2
5	4

Yukarıdaki tabloda kişi numaraları ve öncelikleri sıralanmıştır. Bu listeye göre sırada bekleyenlerden ilk çıkması gereken kişi en yüksek önceliğe sahip olan 1 numaralı kişidir. Ardından 4 önceliğine sahip 5 numaralı kişi sıradan çıkar. Sıradan önceliklerine göre çıkanları sıralayacak olursak : 1,5,2,4,3 numaralı kişilerdir.

Öncelik sırasının işlenmesini aslında sırada bekleyen kişilerin önceliklerine göre [sıralanması \(Sort\)](#) ve ardından normal bir [sıra \(queue\)](#) gibi veri yapısının işlenmesi olarak düşünebiliriz. Örneğin yukarıdaki durumda öncelik sırasına göre sırayı yeniden düzenleyecek olursak :

Kişi No	Öncelik
1	1
5	2
2	3
4	4
3	5

Sıralamasını elde etmiş oluruz ki bu da zaten beklemekte olan kişilerin çıkış sırasıdır.

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayar kavramları.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

Öncelik sıralarında aynı önceliğe sahip kişilerin nasıl sıradan çıkacakları da ayrıca bir problem olarak görülebilir. Bu noktadaki en klasik çözüm geliş sırasında göre öncelik vermektir. Örneğin yukarıdaki durumda 3 önceliğine sahip bir kişi daha olsaydı:

Kişi No	Öncelik
1	5
2	3
3	1
4	2
5	4
6	3

Bu durumda öncelik sıralamasına göre hangi 3 önceliğine sahip olan kişinin (2. Kişi veya 6. Kişi) belirlerken önce gelenin önce çıkması ilkesi korunabilir. Bu durumda beklemekte olan kişiler aşağıdaki sırada sıralanabilir:

Kişi No	Öncelik
1	1
5	2
2	3
6	3
4	4
3	5

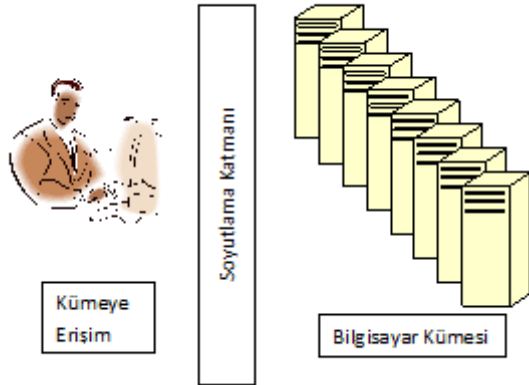
Öncelik sıraları bilgisayar bilimlerinin veri yapısı konusunda kullanılan bir kavram olmasına karşılık uygulama alanları oldukça geniştir. Örneğin işletim sistemlerinde kullanılan [en kısa iş ilk \(shortest job first\) işlemci zamanlama \(CPU scheduling\)](#) algoritması tam bir öncelik sırası örneğidir. [Bekleme sırasındaki \(Ready queue\) işlemlerin \(process\)](#) çalıştırılma sırası uzunluklarına göre önceliğe sahiptir.

## **SORU 28: Cluster Computing (Bilgisayar Kümeleri)**

Bilgisayar bilimlerinde, daha fazla işlem gücü elde etmek amacıyla birden fazla bilgisayarın tek bir bilgisayar gibi çalışmasına verilen isimdir. Genelde birden fazla bilgisayar birbirine oldukça hızlı bir ağ bağlantısı ile bağlanır ve bilgisayarların üzerinde çalıştırılan özel yazılımlar ile istenen işin paylaştırılması hedeflenir. Literatürde kümeleme veya İngilizce olarak clustering terimleri de kullanılmaktadır.

Bilgisayarların üzerinde çalışan işletim sisteminin tek bir işletim sistemi gibi davranması durumuna dağıtık işletim sistemi (distributed operating system) ismi verilir. Burada kullanıcı

veya çalıştırılması istenen işler kümeye (cluster) tek bir noktadan erişir ve işin arka planda nasıl yapıldığı ile arasında bir soyutlama katmanı (abstraction layer) bulunur.



Diğer bir deyişle bilgisayar kümelerinde hedeflenen amaç, kullanıcıların yada [çalışan işlemlerin \(process\)](#) kümenin iç yapısından bağımsız olması ve kümenin her durumda istenen işi en verimli şekilde çalıştırmasıdır.

Ancak bu durum ne yazık ki günümüzde tam olarak gerçekleştirilememiş bir hayaldir. Bunun en önemli sebebi yük dağılımının (load balancing) her durum için en verimli şekilde yapılmasını sağlayacak bir otomat bulunamamasıdır.

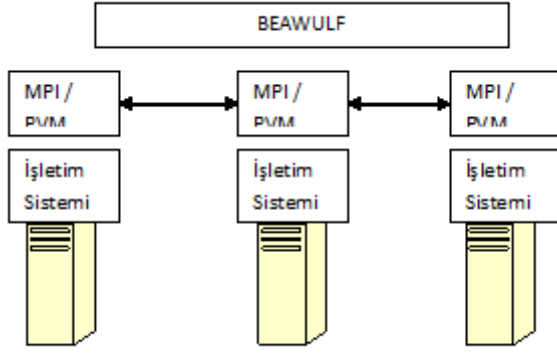
Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayararkavramlari.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

Örneğin bir [matris çarpımı](#) işlemini yaparken kullanılacak dağıtım ile bir fraktal üretimi yada resim işleme sırasında yapılacak dağıtım farklı olabilir.

Tam bu noktada işlem kümeleri (Computing clusters) terimi devreye girer. Bu tip özel kümelerde amaç sadece özel bir iş yada iş tipi için verimli hale getirilmiş kümeler ve yazılımlar üretmektir.

Örneğin sadece hava durumu tahmini veya sadece sonlu eleman analizi (finite element analysis) için geliştirilmiş bir kümede özel bir müdahale gerekmeden işlem yapılabilir. Genellikle PVM paralel virtual machine) veya [MPI \(message passing interface\)](#) benzeri alt yapılar kullanılarak sağlanan bu paralelleştirme işlemlerine işletim sistemlerinin üzerinde çalışan beawulf gibi katmanlar örnek gösterilebilir.





Yukarıdaki şekilde üç bilgisayar üzerinde çalışan bu yapı temsil edilmiştir. Bilgisayarların üzerinde çalışan işletim sistemleri. Bu işletim sistemleri üzerinde çalışan paralel işlem katmanı ve en üstte çalışan işlem kümesi yazılımı.

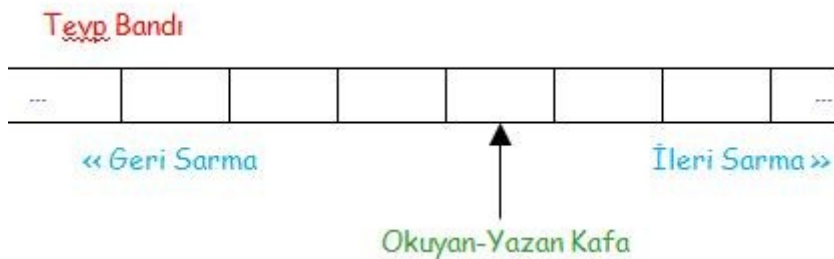
Küme bilgisayarların farklı bir uygulama şeklide ızgara hesaplamalarıdır (grid computing). Bu tip dağıtımli işlemelerde (distributed computing) amaç yukarıda anlatılan işlem kümelerinden ( computing clusters) farklı olarak daha genel amaçlara yönelik işlem gücü elde etmektir. Ayrıca burada tam bir bölünmüşlükten bahsedebiliriz. Bazı durumlarda ağ üzerinde iletişimin oldukça az olması ve işlem yapan bilgisayarların birbiri ile neredeyse hiç konuşmaması bile söz konusudur. Örneğin kullanan SETI@Home projesi buna örnek gösterilebilir. Bu projede insanlar bilgisayarlarını kullanmadıklarında devreye giren basit bir program bütün dünyadan insanların bilgisayarında işlem yapıp sonuçları merkezi sunucuya bildiriyor. Bu sayede uzun süre boş duran bilgisayarların işlem gücü zayi olmamış oluyor ve astronomi konusunda çeşitli hesaplamalarda kullanılıyor.

## SORU 29: Turing Makinesi (Turing Machine)

Bilgisayar bilimlerinin önemli bir kısmını oluşturan [otomatlar \(Automata\)](#) ve [Algoritma Analizi \(Algorithm analysis\)](#) çalıştırmalarının altındaki dil bilimin en temel taşlarından birisidir.1936 yılında Alan Turing tarafından ortaya atılan makine tasarımı günümüzde pekçok teori ve standardın belirlenmesinde önemli rol oynar.

### Turing Makinesinin Tanımı

Basitçe bir kafadan (head) ve bir de teyp bandından (tape) oluşan bir makinedir.



Makinede yapılabilecek işlemler

- Yazmak

- Okumak
- Bandı ileri sarmak
- Bandı geri sarmak

şeklinde sıralanabilir.

### Chomsky hiyerarşisi ve Turing Makinesi

Bütün teori bu basit dört işlem üzerine kurulmuştur ve sadece yukarıdaki bu işlemleri kullanarak bir işin yapılıp yapılamayacağı veya bir dilin bu basit 4 işleme indirgenip indirgenemeyeceğine göre diller ve işlemler tasnif edilmiştir.



Bu sınıflandırma yukarıdaki venn şeması ile gösterilmiştir. Aynı zamanda [chomsky hiyerarşisi \(chomsky hierarchy\)](#) için 1. seviye (type-1) olan ve Turing makinesi ile kabul edilebilen diller bütün tip-2 ve tip-3 dilleri yani içerik bağımsız dilleri ve düzenli dilleri kapsamaktadır. Ayrıca ilave olarak içerik bağımsız dillerin işleyemediği (üretmediği veya parçalayamadığı (parse) )  $a^n b^n c^n$  şeklindeki kelimeleri de işleyebilmektedir. Düzenli ifadelerin işleyememesi konusunda bilgi için [düzenli ifadelerde pompalama savı \(pumping lemma in regular expressions\)](#) ve [içerik bağımsız dillerin işlemeyemesi için de içerik bağımsız dillerde pompalama savı \(pumping lemma for CFG\)](#) başlıklı yazıları okuyabilirsiniz.

### Turing Makinesinin Akademik Tanımı

Turing makineleri literatürde akademik olarak aşağıdaki şekilde tanımlanır:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$$

Burada M ile gösterilen makinenin parçaları aşağıda listelenmiştir:

Q sembolü sonlu sayıdaki durumların kümesidir. Yani makinenin işleme sırasında aldığı durumardır.

$\Gamma$  sembolü dilde bulunan bütün harfleri içeren alfabeyi gösterir. Örneğin ikilik tabandaki sayılar ile işlem yapılyorsa  $\{0,1\}$  şeklinde kabul edilir.

$\Sigma$  sembolü ile makineye verilecek girdiler (input) kümesi gösterilir. Girdi kümesi dildeki harfler dışında bir sembol taşıyamayacağı için  $\Sigma \subseteq \Gamma$  demek doğru olur.

$\delta$  sembolü dilde bulunan ve makinenin çalışması sırasında kullanacağı geçişleri (transitions) tutmaktadır.

$\diamond$  sembolü teyp bandı üzerindeki boşlukları ifade etmektedir. Yani teyp üzerinde hiçbir bilgi yokken bu sembol okunur.

$q_0$  sembolü makinenin başlangıç durumunu (state) tutmaktadır ve dolayısıyla  $q_0 \subseteq Q$  olmak zorundadır.

F sembolü makinenin bitiş durumunu (state) tutmaktadır ve yine  $F \subseteq Q$  olmak zorundadır.

### Örnek Turing Makinesi

Yukarıdaki sembolleri kullanarak örnek bir Turing makinesini aşağıdaki şekilde inşa edebiliriz.

Örneğin basit bir kelime olan  $a^*$  düzenli ifadesini (regular expression) Turing makinesi ile gösterelim ve bize verilen  $aaa$  şeklindeki 3 a yı makinemizin kabul edip etmediğine bakalım.

Tanım itibariyle makinemizi aşağıdaki şekilde tanımlayalım:

$$M = \{ \{q_0, q_1\}, \{a\}, \{a, x\}, \{q_0 a \rightarrow a R q_0, q_0 x \rightarrow x L q_1\}, q_0, x, q_1 \}$$

Yukarıdaki bu makineyi yorumlayacak olursak:

Q değeri olarak  $\{q_0, q_1\}$  verilmiştir. Yani makinemizin ik idurumu olacaktır.

$\Gamma$  değeri olarak  $\{a, x\}$  verilmiştir. Yani makinemizdeki kullanılan semboller a ve x'ten ibarettir.

$\Sigma$  değeri olara  $\{a\}$  verilmiştir. Yani makinemize sadece a girdisi kabul edilmektedir.

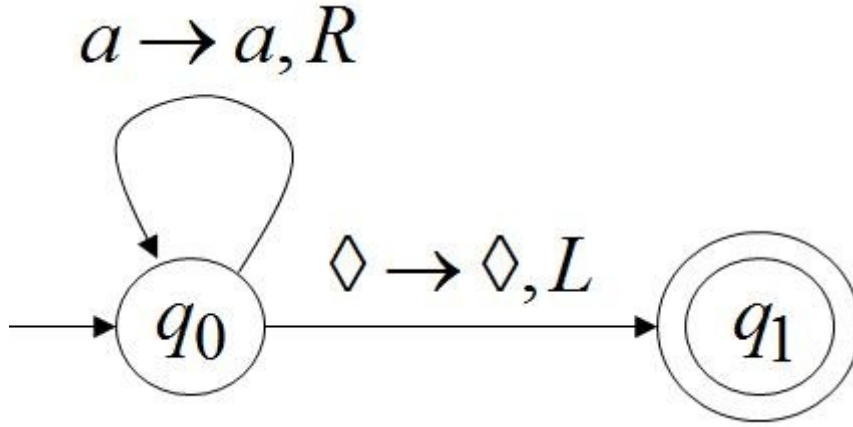
$\delta$  değeri olarak iki geçiş verilmiştir  $\{q_0 a \rightarrow a R q_0, q_0 x \rightarrow x L q_1\}$  buraadki R sağa sarma L ise sola sarmadır ve görüleceği üzere Q değerindeki durumlar arasındaki geçişleri tutmaktadır.

$\diamond$  değeri olarak x sembolü verilmiştir. Buradan x sembolünün aslında boş sembolü olduğu ve bantta hiçbir değer yokken okunan değer olduğu anlaşılmaktadır.

$q_0$  ile makinenin başlangıç durumundaki hali belirtilmiştir.

F değeri olarak  $q_1$  değeri verilmiştir. Demek ki makinemiz  $q_1$  durumuna geldiğinde bitmektedir (halt) ve bu duruma gelmesi halinde bu duruma kadar olan girdileri kabul etmiş olur.

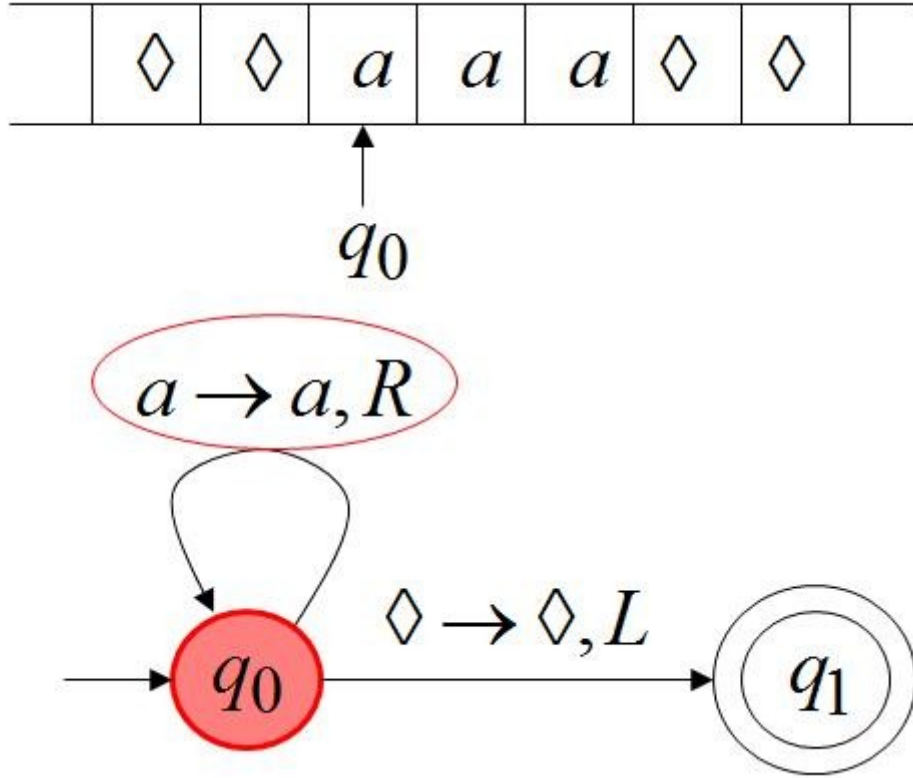
Yukarıdaki bu tanımlı görsel olarak göstermek de mümkündür:



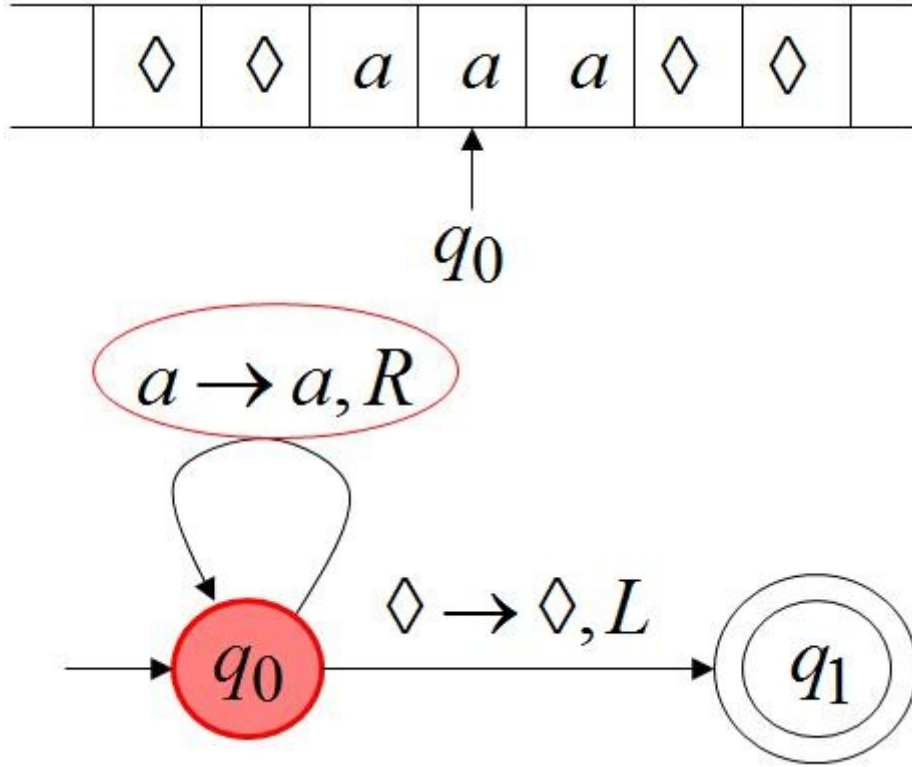
Yukarıdaki bu temsili resimde verilen turing makinesi çizilmiştir.

Makinemizin örnek çalışmasını ve bant durumunu adım adım inceleyelim.

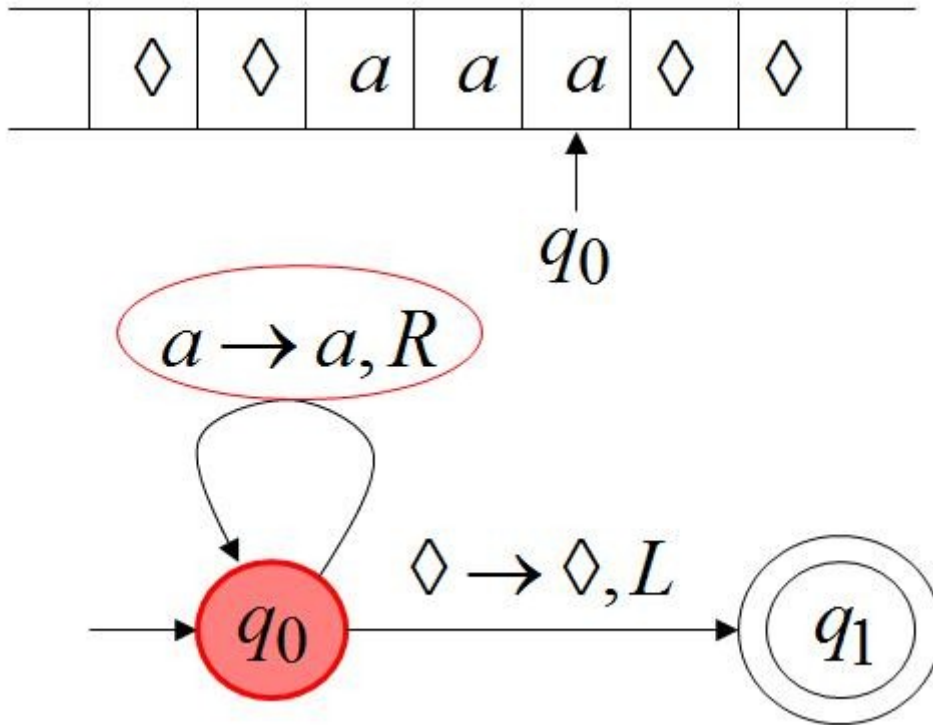
Birinci adımda bandımızda aaa (3 adet a) yazılı olduğunu kabul edelim ve makinemizin bu aaa değerini kabul edip etmeyeceğini adım adım görelim. Zaten istediğimiz de aaa değerini kabul eden bir makine yapabilmektir.



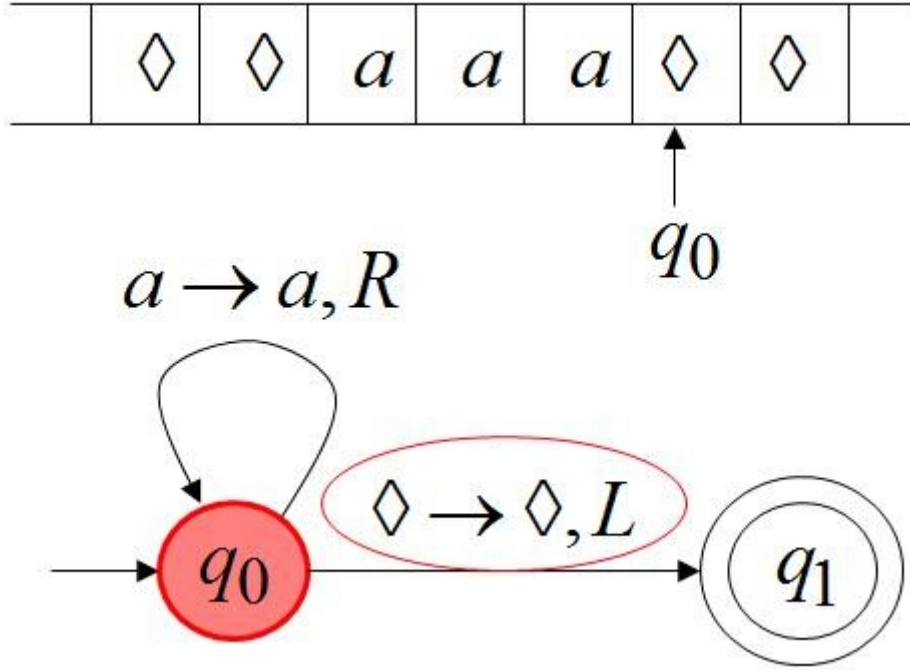
Yukarıdaki ilk durumda bant üzerinde beklenen ve kabul edilip edilmeyeceği merak edilen değerimiz bulunuyor. Makinemizin kafasının okuduğu değer  $a$  sembolü. Makinemizin geçiş tasarımına göre  $q_0$  halinde başlıyoruz ve  $a$  geldiğinde teybi sağa sarıp yine  $q_0$  durumunda kalmamız gerekiyor.



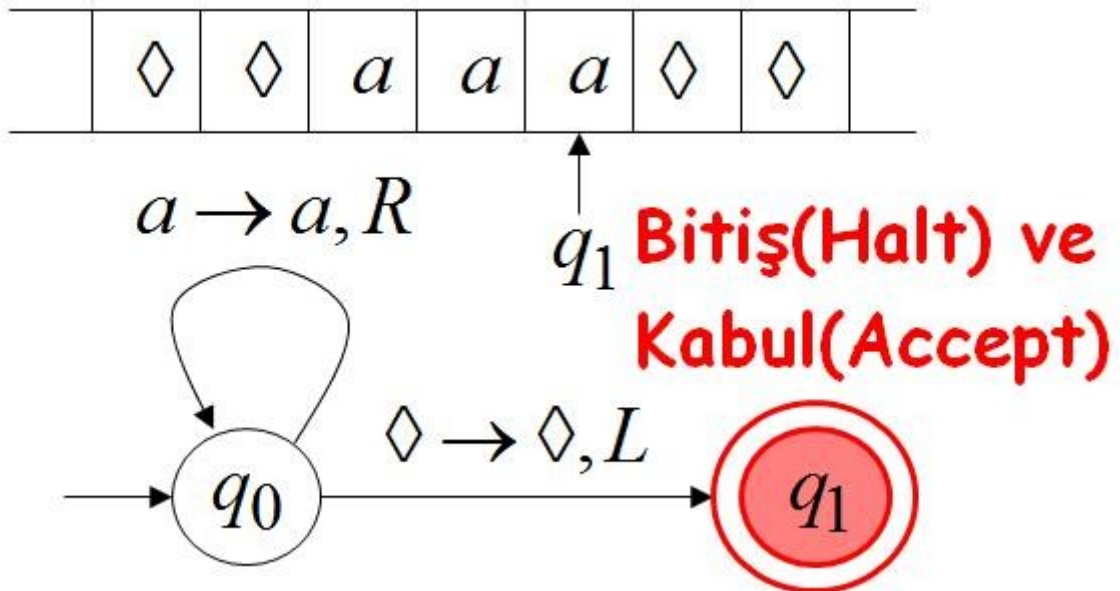
Yeni durumda kafamızın okuduğu değer banttaki 2. a harfi ve bu durumda yine  $q_0$  durumundayken teybi sağa sarıp yine  $q_0$  durumunda kalmamız tasarlanmıştır



3. durumda kafamızın okuduğu değer yine  $a$  sembolü olmakta ve daha önceki 2 duruma benzer şekilde  $q_0$  durumundayken  $a$  sembolü okumanın sonucu olarak teybi sağa sarıp  $q_0$  durumunda sabit kalıyoruz.



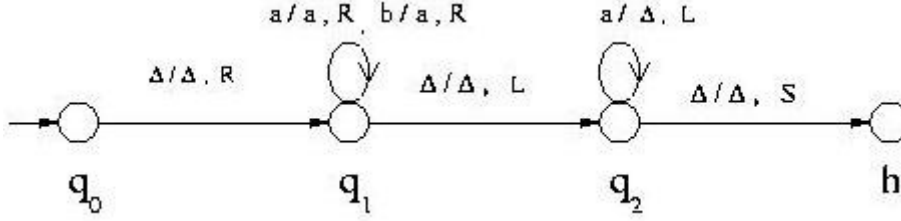
4. adımda teypten okuduğumuz değer boşluk sembolü  $x$  oluyor. Bu değer makinemizin tasarımında  $q_1$  durumuna gitmemiz olarak tasarlanmış ve teybe sola sarma emri veriyoruz.



Makinenin son durumunda  $q_1$  durumu makinenin kabul ve bitiş durumu olarak tasarlanmıştı ( makinenin tasarımındaki F kümesi) dolayısıyla çalışmamız burada sonlanmış ve giriş olarak  $aaa$  girdisini kabul etmiş oluyoruz.

## 2. Örnek

Hasan Bey'in sorusu üzerine bir örnek makine daha ekleme ihtiyacı zuhur etti. Makinemiz  $\{a,b\}$  sembolleri için çalışsın ve ilk durum olarak bandın en solunda başlayarak bandta bulunan sembolleri silmek için tasarlansın. Bu tasarımı aşağıdaki temsili resimde görülen otomat ile yapabiliriz:



Görüldüğü üzere makinemizde 4 durum bulunuyor, bunlardan en sağda olan h durumu bitişi (halt) temsil ediyor. Şimdi bu makinenin bir misal olarak “aabb” yazılı bir bandta silme işlemini nasıl yaptığını adım adım izah etmeye çalışalım.

Aşağıda, makinenin her adımda nasıl davranacağı bant üzerinde gösterilmiş ve altında açıklanmıştır. Sarı renge boyalı olan kutular, kafanın o anda üzerinde durduğu bant konumunu temsil etmektedir.



◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

q0 durumunda başlıyoruz. Ve boşluk ile bandı sağa sarıyoruz:

◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

a veya b değeri okundukça bant sağa sarılmaya devam ediyor ve q1 durumunda kalıyoruz.

◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

◇	◇	a	a	b	b	◇
---	---	---	---	---	---	---

Okunan değer b ise banda geri a değeri yazılıyor

◇	◇	a	a	a	b	◇
---	---	---	---	---	---	---

[www.bilgisayarkavramlari.com](http://www.bilgisayarkavramlari.com)

◇	◇	a	a	a	a	◇
---	---	---	---	---	---	---

En sağda boşluk değerini okuyunca (◇) Sağa sarma işlemi bitiyor ve geri dönüyoruz

◇	◇	a	a	a	a	◇
---	---	---	---	---	---	---

◇	◇	a	a	a	◇	◇
---	---	---	---	---	---	---

◇	◇	a	a	◇	◇	◇
---	---	---	---	---	---	---

◇	◇	a	◇	◇	◇	◇
---	---	---	---	---	---	---

◇	◇	◇	◇	◇	◇	◇
---	---	---	---	---	---	---

Tekrar boşluk (◇) görülünce makine bitiyor.

Geri sarma işlemi sırasında a değerleri silinmiş oluyor

Netice olarak Hasan Bey'in sorusuna temel teşkil eden ve örneğin q1 üzerindeki döngülerden birisi olan b/a,R geçişi, banttan b okunduğunda banta a değerini yaz manasındadır.

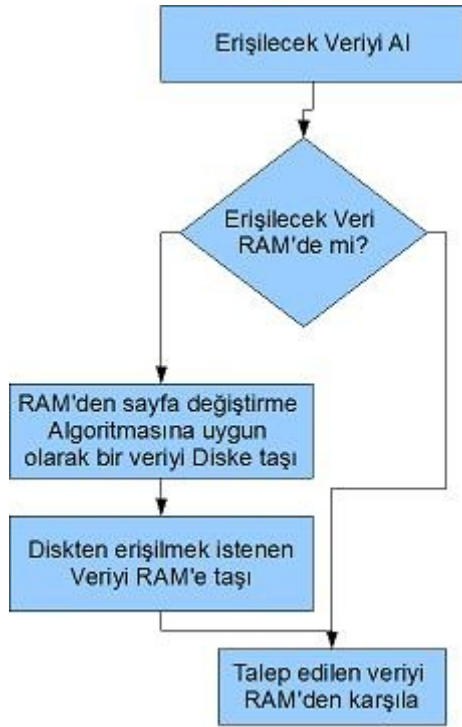
### SORU 30: Sanal Hafıza (Virtual Memory)

Sanal bellek olarak da isimlendirilen hafıza türü, bilgisayarın [birincil hafızası \(primary memory\) olarak bilinen RAM](#)'in yetersiz kaldığı durumlarda ikincil hafıza (secondary memory) olarak bilinen diskin bir kısmının kullanılmasıdır.

Unix/Linux terminolojisine göre takas alanı (Swap space) olarak isimlendirilen bu hafızada RAM ve disk arasında bulundurlan bilgiler sürekli olarak takaslanmaktadır (swapping).

Kısacası işlemlerin çalışmak için ihtiyaç duydukları Bellek kapasitesinin üzerindeki talepler için disk kullanılır ve çalışan programların eriştikleri veriler anlık olarak RAM'de durmalıdır. Yani şayet yer yetersizliğinden dolayı bir bilgi diske taşınmışsa ve bu bilgiye tekrar erişilmek istenirse, bu bilgi diskten RAM'e geri yüklenmelidir.

Bu yükleme işlemi tahmin edileceği üzere oldukça zaman almaktadır çünkü diske erişim, RAM'e erişime göre oldukça yavaştır. Ayrıca diske iki kere bilgi yazılmalıdır. Bu algoritma aşağıdaki şekilde özetlenebilir:



Yukarıdaki [akış diyagramında \(flow chart\)](#) gösterilen algoritmaya göre şayet veri RAM'de bulunmuyorsa hem yer açmak için RAM'den bir veri diske yüklenmeli hem de diskte RAM'e talep edilen veri yüklenmelidir.

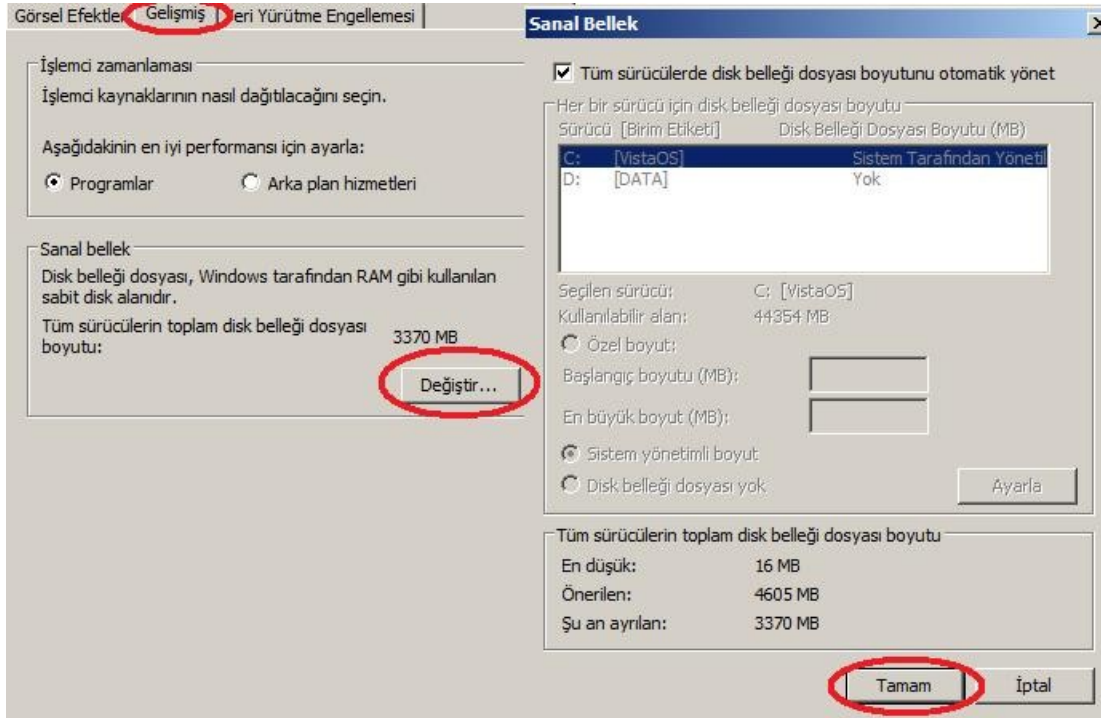
Sanal belleğin kullanılabilemesi için işlemlerin kitalama (Segmentation) veya Sayfalama (Paging) ile hafızada tutulmaları gerekir. Bunun sebebi mantıksal adreslerin fiziksel adreslere dönüşüm olanağıdır. Ayrıca sayfalama (paging) ile hafızada tutulan işlem verilerinin parçalı olarak da disk ve RAM arasında takaslanması mümkündür.

Örneğin Window işletim sistemi üzerinde paging (sayfalama) kullanılarak disk üzerinde bir dosya oluşturulur ve işlem verileri buraya atılır.

IO.SYS	0 KB	Sistem Dosyası
MSDOS.SYS	0 KB	Sistem Dosyası
NTDETECT.COM	47 KB	MS-DOS Uygulaması
ntldr	245 KB	Sistem Dosyası
pagefile.sys	2.095.104 KB	Sistem Dosyası

Yukarıda bilgisayarın diskinde bulunan dosyaların listesi verilmiştir. Bu listenin en sonunda pagefile.sys ismi verilen dosyanın amacı sayfalama verilerinin sonucunda işlemlerin bir kısmının diskte tutulmasıdır.

Diskte bu işlem için ayrılan miktar windows'un gelişmiş ayarlarından ayarlanabilir:



Görüldüğü üzere Windows random Access olan bir hafıza alanını ardışık (Sequential) erişimi olan disk üzerinde tek bir dosya olarak tutması çeşitli problemlere yol açabilmektedir. Örneğin bu dosya üzerinde fragmentation (parçalanma) olabilir. Yani çalışan ve ölen işlemler zaman içinde dosyada boşluklar oluşturmakta bu durumda performans kaybına sebep olmaktadır. Bunun için bu dosyanın defragment edilmesini de içeren bir takım yöntemler geliştirilmiştir. Ancak hafıza yodun işlemlerde performans artışı sağlayan en kesin yöntem fiziksel hafızanın artırılmasıdır.

Sanal belleğin kapatılması için yukarıdaki şekilde görülen ekran kullanılabilir veya Swappingin tamamen kapatılması için :

Linux sistemlerde `/proc/sys/vm/swappiness` parametresinin değiştirilmesi

Windowsta ise `DisablePagingExecutive` registry ayarının değiştirilmesi mümkündür.

### **SORU 31: Sayfalama (Paging)**

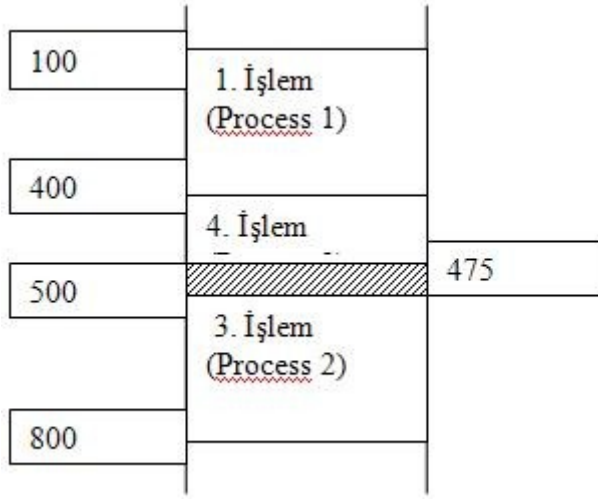
İçerik

Hafıza	Yönetim	Problemleri
<a href="#">Sayfalama</a>		<a href="#">(Paging)</a>
<a href="#">Sayfa Tablosu (Page Table)</a>		

Bilgisayar bilimlerinin önemli konularından birisi olan işletim sistemlerinin bir görevi de [hafızayı verimli yönetmektir \(memory management\)](#). Kısaca sınırlı miktarda [hafıza \(RAM, Bellek, Memory\)](#) bulunmakta ve çalışan her program bir miktar hafızaya ihtiyaç duymaktadır. İşletim sistemi (operating system) bu hafızayı ne kadar verimli kullanırsa ve [işlemleri \(process\)](#) ne kadar düzgün yerleştirirse hafızanın içerisine sığabilen program mikatrı o kadar fazla olur.

### **Hafıza problemleri**

Hafızanın yönetimi sırasında karşılaşılabilecek problemlerden birisi [harici hafıza kırıntılarıdır \(external fragments, harici parçalar\)](#). İlgili yazı okunursa görülür ki hafızaya yeni işlemlerin yüklenmesi ve biten işlemlerin kaldırılması sırasında, hafızada verimsiz boşluklar oluşur.



Örneğin yukarıdaki şekilde daha önceden 400-500 fiziksel adresleri (physical address) arasında yer alan 2. işlem (process 2) kaldırılmış ve oluşan boşluğa, 2. işlemten daha az yer kaplayan 4. işlem yerleştirilmiştir. 2. işlem 100, 4. işlem ise 75 boyutunda olduğu için 25 boyutunda bir boşluk oluşmuştur.

Yukarıdaki örnekteki 25 uzunluğundaki boşluk ancak 25 ve daha az hafıza ihtiyacı olan işlemler için kullanılabilir. Hafızanın çeşitli yerlerinde benzer şekillerde küçük parçaların kalması sonucunda hafızada toplamda yeterli yer olmasına karşılık yeni bir işlem bu parçalara bölünemeyeceği için yetersiz yer problemi ile karşılaşılacaktır. Örneğin yukarıdaki şekilde 5 farklı yerde 25 boyutunda hafıza boşluğu olsun. Bu durumda toplam 125 boyutunda yer olacak ancak 100 boyutunda yeni bir işlem bu parçalara bölünemediği için hafızadaki bu boş yer kullanışsız olacaktır.

### Çözüm olarak Sayfalama (Paging)

Çözüm olarak sayfalama (paging) kullanılabilir. Sayfalama çözümünde hafıza (RAM) basitçe sayfa boyutu kadar ufak parçalara bölünür. Aynı durum işlemler (process) için de geçerlidir. Yani hafıza ihtiyacı olan bütün işlemler verilen sayfa boyutu (page size) kadar parçaya bölünür. Ardından hafızada ilgili işlem sayfaları, sayfa sayfa yüklenir.

Örneğin sayfa boyutumuz (page size) 100 olsun.

275 boyutundaki bir işlem aşağıdaki şekilde 3 sayfaya bölünecektir.

İşlem 1 (Process 1)	
Adres	Sayfa
0-100	1
100-200	2
200-275	3

Benzer şekilde 325 uzunluğuna sahip başka bir işlem de 4 parçaya bölünür:

İşlem 2 (Process 2)	
Adres	Sayfa
0-100	1
100-200	2
200-300	3
300-325	4

Örneğin 1000 boyutunda bir hafızamız (ram) bulunması halinde ise bu hafıza aşağıdaki şekilde 10 parçaya bölünür.

Hafıza (RAM)	
Adres	Çerçeve
0-100	1
100-200	2
200-300	3
300-400	4
400-500	5
500-600	6
600-700	7
700-800	8
800-900	9
900-1000	10

Yukarıda adres sütununda bulunan değerler başlangıç ve bitiş değerleridir. Örneğin 500-600 aralığı, 500 adresinden 600 adresine kadar olan aralıktır.

Hafızadaki parçalara çerçeve (frame) ismi verilir ve yukarıda görüldüğü üzere 1000 boyutundaki bir hafızada çerçeve boyutu (frame size) 100 olması durumunda 10 parça bulunur. Genelde çerçeve boyutu ile sayfa boyutu eşit alınır ancak bu bir şart değildir.

Tanım olarak işlemlerin adreslerine mantıksal adres (logical address) ve hafızadaki adreslere fiziksel adres( physical address) ismi verilir. Bilindiği üzere işlemler gerçekte hangi adreste

olduklarını yani fiziksel adreslerini bilmezler. İşlemler kendilerini sanal bir dünyada kendi adres uzayında (address space) zannederler.

Yukarıdaki şekilde bölünen sayfaların hafızaya yerleştirilişi ise çalışma sırasına göre olur. Örneğin İşlem 1 önce, işlem 2 ise sonra çalışıyor olsun. Bu durumda işletim sistemi işlemleri aşağıdaki şekilde yerleştirebilir:

Hafıza (RAM)		
Adres	Çerçeve	Sayfa
0-100	1	İ1-1
100-200	2	İ1-2
200-300	3	İ1-3
300-400	4	İ2-1
400-500	5	İ2-2
500-600	6	İ2-3
600-700	7	İ2-4
700-800	8	
800-900	9	
900-1000	10	

Yukarıdaki şekilde işlem numarası ve sayfa numarası şeklinde kısaltma kullanılmıştır. Örneğin İ1-3, işlem 1 sayfa 3 anlamına gelmektedir.

Yukarıdaki tabloda görüldüğü üzere ilk 3 çerçeve işlem1 ve sonraki 4 çerçeve ise işlem 2 için ayrılmıştır.

Bu durumda yukarıdaki tabloda dikkat edilirse 3. çerçeve aslında 100 boyutunda olmasına karşılık 75 boyutundaki işlem 1'in 3. sayfasını ve 7 çerçeve de 100 boyutunda olmasına karşılık işlem2 'nin 25 boyutundaki 4. sayfasını tutmaktadır.

Bu durumda 3. çerçevede 25 ve 7. çerçevede 75 boyutlarında [iç hafıza kırıntısı \(internal fragments\)](#) oluşmuştur. Bu durum sayfalamanın (paging) dez avantajıdır. Yani aslında sayfalama (paging) [harici hafıza kırıntılarını \(external fragments\)](#) engellerken bu defa [dahili hafıza kırıntıları \(internal fragments\)](#) oluşturmaktadır. Ancak görüldüğü üzere dahili kırıntılar, harici kırıntılara nispetle kontrol edilebilir. Yani azami dahili hafıza kırıntısı yukarıdaki örnekte 99 boyutunda olabilir (çerçeve boyutunun 100 ve o çerçeveye gelen işin uzunluğunun 1 olduğu kabul edilirse). Dolayısıyla dahili kırıntılar, harici kırıntılara göre nispeten daha tercih edilebilir denilebilir.

### Sayfa tablolarının tutulması (Page Tables)

Yukarıdaki örnekte görüldüğü üzere iki ayrı [işlem \(process\)](#) hafızaya yerleştirilmiştir. [Hafıza yönetimi \(memory management\)](#) yapan işletim sistemi, çalışan herhangi bir programın mantıksal adresini (logical address) fiziksel adrese (physical address) çevirmek zorundadır. Bu işlem için sayfa tabloları (page tables) kullanılır.



Yukarıdaki şekilde hafızaya yerleşen iki işlem için sayfa tablosu (page table) aşağıdaki şekilde olur:

Sayfa Tablosu	
Sayfa	Çerçeve
İ1-1	1
İ1-2	2
İ1-3	3
İ2-1	4
İ2-2	5
İ2-3	6
İ2-4	7

Yukarıdaki tabloda hangi işlemin hangi sayfasının, RAM'deki hangi çerçeveye ait olduğu gösterilmiştir. Bu tablo ile mantıksal adresten fiziksel adrese dönüşüm mümkün olur.

Örneğin 2. işlem mantıksal adresi 327'ye erişmek istesin. bu durumda öncelikle 327 numaralı adresin hangi sayfada olduğu bulunmalıdır:

Mantıksal Adres / Sayfa Boyutu = Sayfa Numarası

$$327 / 100 = 3$$

Yukarıda görüldüğü üzere erişilmek istenen adres sayfa boyutuna bölünmüş (tam sayı bölmesi) ve sonuç olarak 3 çıkmıştır. Yukarıdaki sayfa ve çerçeve numaraları 1'den başlamıştır bu durumda bu sayıya 1 eklenecektir. Şayet sayfa numarası ve çerçeve numarası 0'dan başlıyorsa bu durumda ekleme işlemine gerek kalmaz:

$$3 + 1 = 4$$

Yukarıdaki hesaptan 4. sayfadaki bir adrese erişilmek istendiği anlaşılmıştır. Şimdi bu sayfanın fiziksel adres kaşılığını bulmak için hangi çerçeveye yüklendiğini bulmamız gerekir. Sayfa tablosu (page table) da tam burada devreye girer. 4. sayfa yukarıdaki tablodan görüleceği üzere 7. çerçevededir.

O halde önce fark miktarını (offset) hesaplayalım:

Mantıksal Adres % Sayfa Boyutu = fark miktarı (offset)

$$327 \% 100 = 27$$

Yukarıda erişilmek istenen mantıksal adresin sayfa boyutuna bölümünden kalan (remainder, modulo) değeri bulunmuştur. Bu durumda 7. çerçevenin başlangıç adresine 27 eklenmesi durumunda karşılık olan fiziksel adres bulunmuş olacaktır.

7. çerçeve 600. adresten başlamaktadır bu durumda fiziksel adres:

$$600 + 27 = 627$$

olarak bulunmuş olunur.

Yukarıdaki bu işlemler aşağıdaki şekilde de formülüze edilebilir:

$$FA = ST (MA / SB) + MA \% SB$$

Yukarıdaki formülde FA : Fiziksel Adres, ST : Sayfa Tablosundaki o işlemin karşılığı (örneğin ST (2) sayfa tablosundaki 2. sayfanın karşılığı olan çerçeve), MA : Mantıksal Adres, SB : Sayfa Boyutu olarak tanımlanmıştır.

### **SORU 32: Sayfa Değiştirme Algoritması (Page Replacement)**

Bilgisayar bilimlerinde özellikle işletim sistemi konusunda kullanılan ve hafızanın daha verimli çalışması için geliştirilmiş algoritmaların ismidir.

İçerik

[Arkaplan](#) [ve](#) [ön](#) [bilgiler](#)  
[FIFO](#)  
[LRU](#)  
[Optimal Replacement](#)

#### **Algoritmanın arka planı ve gerekli ön bilgiler**

Bilindiği üzere bilgisayarda hafızanın yönetimini (Memory management) işletim sistemi yapmaktadır. Dolayısıyla başarılı bir hafıza yönetiminde, hafızada (RAM) bulunan boşluk kırıntıları (Fragment) en az olmalıdır. Bu boşlukları en aza indirmek için sayfalama (Paging) yöntemi kullanılır. Sayfalama yönteminde harici boşluk kırıntısı (External Fragment) bulunmaz. Bunun yerine bir sayfaya sığmayan işlemler için iç boşluk kırıntıları (internal fragment) bulunabilir ve bu boşlukların azami değeri sayfa boyutunu aşamaz. Yani bir anlamda hafızanın verimsizleşmesine sebep olan boşluklar kontrol altına alınmış olunur.

Hafıza yönetiminin diğer bir problemi ise, bilgisayar hafızasının yetersiz olduğu durumlarda işlemleri çalıştıramamasıdır. Örneğin 1GB boş hafızası olan bir bilgisayarda 3GB boyutunda işlemlerin çalışması normalde imkansız gibidir. Ancak bu duruma karşı çözüm olarak sanal hafıza (Virtual memory veya Unix terminolojisinde takas alanı, swap) geliştirilmiştir. Basitçe RAM'e sığmayan bilgiler diskin bir kısmında saklanmakta ve gerekli oldukça diskten RAM'e yüklenmektedir. Tabi diskten RAM'e yükleme sırasında tam dolu bir RAM'de yer açmak için de bir kısım bilgi RAM'den diske geri yazılmaktadır.

İşte sayfa değiştirme algoritmaları tam bu noktada devreye girmektedir. Basitçe bilgisayarın hafızası sayfalara (pages) bölünmekte ve çalışan işlemler (processes) bu sayfalara yerleştirilmektedir. Sonuçta sınırlı hafıza ve sınırın üzerinde hafıza ihtiyacı olduğu durumlarda bazı sayfalar (page) diske yazılmaktadır. Hangi sayfanın hafızada (RAM) ve hangi sayfanın diskte bulunacağını ve ne zaman yer değiştireceklerini belirleyen algoritmalar sayfa değiştirme algoritmalarıdır.

#### **fifo Page Replacement Algorithm (İlk giren ilk çıkar sayfa değiştirme algoritması)**



Bu algoritmaya göre bir sayfa ihlali (page fault) olduğunda, yani hafızada (RAM) bulunmayan bir sayfaya erişilmek istendiğinde, yani diskteki bir sayfaya erişilmek istendiğinde, Diskten ilgili sayfa hafızaya (RAM) yüklenirken, hafızadaki en eski sayfa yerine yüklenir ve bu en eski sayfa da diske geri yazılır.

Bu algoritmayı bir örnek üzerinden inceleyelim.

Örneğin işletim sisteminden sırasıyla aşağıdaki çerçeveler (Frames) yani sayfalar talep ediliyor olsun:

1, 2, 3, 2, 3, 4, 5, 3, 1

ve ayrıca hafızamıza (RAM) sadece 3 çerçeve anlık olarak sığabiliyor olsun. Bu durumda her talepten sonra hafızadaki çerçeveler (frames, sayfalar, pages) aşağıdaki şekilde olacaktır.

1. sayfa talebinde sayfa ihlali (page fault) olur çünkü henüz hafızada olmayan bir sayfa talep edilmiştir.

| 1 | | |

2. sayfa talebinde sayfa ihlali yolur çünkü 2. sayfa da henüz hafızada bulunmamaktadır.

| 1 | 2 | |

3. sayfa talebinde yeniden bir sayfa ihlali bulunur sebebi 3. sayfanın da henüz hafızada olmamasıdır:

| 1 | 2 | 3 |

3. sayfadan sonra artık hafızamız tamamen dolmuştur çünkü hafıza kapasitesi 3 sayfa taşıyacak şekilde tanımlanmıştır.

4. sayfa talebinde zaten hafızada bulunan 2. sayfa istenir ve bir sayfa ihlali oluşmaz.

5. sayfa talebinde de benzer şekilde zaten hafızada bulunan 3 numaralı sayfa talep edilir ve bir sayfa ihlali oluşmaz.

6. sayfa talebinde hafızada bulunmayan 4 numaralı sayfa talep edilir ve bir sayfa ihlali oluşur. İşte tam bu noktada FIFO algoritması devreye girer. Çünkü hafıza tamamen doludur ve 4 numaralı sayfanın hafızaya yüklenmesi için hafızadaki sayfalardan birisinin kaldırılması gerekmektedir. Soru hangisinin kaldırılacağıdır.

FIFO algoritmasına göre en eskisi kaldırılır. Yani 1 numaralı sayfa ilk gelen olduğu için ilk kaldırılan olur ve 1 numaralı sayfa yerine 4 numaralı sayfa yerleştirilir:

| 4 | 2 | 3 |

7. sayfa talebinde yine hafızada o anda bulunmayan 5 numaralı sayfa talep edilmiştir. Bu durumda yeni bir sayfa ihlali oluşacak ve 5 numaralı sayfa o anda hafızada bulunan en eski sayfa yerine yerleştirilecektir. Bu en eski sayfa 2 numaralı sayfadır ve sayfa yerleştirilmesi (page replacement) işleminden sonra hafızadaki sayfalar aşağıdaki şekildedir:

| 4 | 5 | 3 |

8. sayfa talebinde 3 numaralı sayfaya erişilmek istenmiştir ve bu sayfa hal-i hazırda bellekte bulunmaktadır dolayısıyla bir sayfa ihlali oluşmaz ve doğrudan hafızadan ihtiyaç karşılanır.

9. sayfa talebinde ise daha önceden hafızada olan ancak şu anda hafızada bulunmayan 1 numaralı sayfa istenmiştir. Bu durumda bu sayfa hafızada olmadığı için sayfa ihlali olur ve 1 numaralı sayfa hafızadaki en eski sayfanın yerine yani 3. numaralı sayfanın yerine yerleştirilir ve son halinde:

| 4 | 5 | 1 |

şeklinde bir hafıza sayfa dizilimine erişilmiş olur.

Yukarıdaki bu örnekte toplam 9 sayfa için 6 sayfa ihlali olmuştur. Bu durumda sayfa ihlal oranı (Page fault rate) 0.66 olarak bulunmuş olunur. **Least Recently Used (LRU) Page replacement (En nadir kullanılan sayfa değiştirme algoritması)**

Bu algoritmaya göre bir sayfa ihlali (page fault) olduğunda, yani [hafızada \(RAM\)](#) bulunmayan bir sayfaya erişilmek istendiğinde, yani diskteki bir sayfaya erişilmek istendiğinde, Diskten ilgili sayfa [hafızaya \(RAM\)](#) yüklenirken, hafızadaki en az erişilen sayfa yerine yüklenir ve bu en az kullanılan sayfa da diske geri yazılır.

Bu algoritmayı bir örnek üzerinden inceleyelim.

Örneğin işletim sisteminden sırasıyla aşağıdaki çerçeveler (Frames) yani sayfalar talep ediliyor olsun:

1, 2, 3, 2, 3, 4, 5, 3, 1

ve ayrıca hafızamıza (RAM) sadece 3 çerçeve anlık olarak sığabiliyor olsun. Bu durumda her talepten sonra hafızadaki çerçeveler (frames, sayfalar, pages) aşağıdaki şekilde olacaktır.

1. sayfa talebinde sayfa ihlali (page fault) olur çünkü henüz hafızada olmayan bir sayfa talep edilmiştir.

| 1 | | |

2. sayfa talebinde sayfa ihlali yolur çünkü 2. sayfa da henüz hafızada bulunmamaktadır.

| 1 | 2 | |

3. sayfa talebinde yeniden bir sayfa ihlali bulunur sebebi 3. sayfanın da henüz hafızada olmamasıdır:

| 1 | 2 | 3 |

3. sayfadan sonra artık hafızamız tamamen dolmuştur çünkü hafıza kapasitesi 3 sayfa taşıyacak şekilde tanımlanmıştır.

4. sayfa talebinde zaten hafızada bulunan 2. sayfa istenir ve bir sayfa ihlali oluşmaz.

5. sayfa talebinde de benzer şekilde zaten hafızada bulunan 3 numaralı sayfa talep edilir ve bir sayfa ihlali oluşmaz.

6. sayfa talebinde hafızada bulunmayan 4 numaralı sayfa talep edilir ve bir sayfa ihlali oluşur. İşte tam bu noktada LRU algoritması devreye girer. Çünkü hafıza tamamen doludur ve 4 numaralı sayfanın hafızaya yüklenmesi için hafızadaki sayfalardan birisinin kaldırılması gerekmektedir. Soru hangisinin kaldırılacağıdır.

LRU algoritmasına göre en eski erişilen kaldırılır. Buna göre en son erişilen iki sayfa, 5. ve 6. adımlardaki 2. ve 3. sayfalardır. Dolayısıyla 1. sayfa, şu anda en eski erişilmiş olandır ve hafızadan kaldırılır.

| 4 | 2 | 3 |

7. sayfa talebinde yine hafızada o anda bulunmayan 5 numaralı sayfa talep edilmiştir. Bu durumda 2 numaralı ve en eski erişilmiş olan sayfa yerine 5 numaralı sayfa yerleştirilir.

| 4 | 5 | 3 |

8. sayfa talebinde 3 numaralı sayfaya erişilmek istenmiştir ve bu sayfa hal-i hazırda bellekte bulunmaktadır dolayısıyla bir sayfa ihlali oluşmaz ve doğrudan hafızadan ihtiyaç karşılanır.

9. sayfa talebinde ise daha önceden hafızada olan ancak şu anda hafızada bulunmayan 1 numaralı sayfa istenmiştir. Bu durumda bu sayfa hafızada olmadığı için sayfa ihlali olur 1 numaralı sayfa o anda hafızada bulunan en nadir erişilmiş sayfa yerine yerleştirilecektir. Bu en az erişilen sayfa 4 numaralı sayfadır ve sayfa yerleştirilmesi (page replacement) işleminden sonra hafızadaki sayfalar aşağıdaki şekildedir:

| 1 | 5 | 3 |

şeklinde bir hafıza sayfa dizilimine erişilmiş olur.

Yukarıdaki bu örnekte toplam 9 sayfa için 6 sayfa ihlali olmuştur. Bu durumda sayfa ihlal oranı (Page fault rate) 0.66 olarak bulunmuş olunur.

### **Optimal Replacement (Mükemmel Sayfa Değiştirme Algoritması)**

Bu algoritma, hiç bir zaman gerçekleştirilemeyecek hayali bir algoritmadır. Akademik olarak ortaya atılmıştır ve algoritmanın çalışması için daha sonra gelecek olan sayfa ihtiyaçlarının önceden bilinmesi gerekir. Bu sayfa değiştirme algoritmasına göre bir sayfa ihlali (page fault) olduğunda, yani [hafızada \(RAM\)](#) bulunmayan bir sayfaya erişilmek istendiğinde, yani diskteki bir sayfaya erişilmek istendiğinde, Diskten ilgili sayfa [hafızaya \(RAM\)](#) yüklenirken, hafızada bundan sonra en uzun süre erişilmeyecek olan yerine yüklenir ve bu en az kullanılan sayfa da diske geri yazılır.

Bu algoritmayı bir örnek üzerinden inceleyelim.

Örneğin işletim sisteminden sırasıyla aşağıdaki çerçeveler (Frames) yani sayfalar talep ediliyor olsun:

1, 2, 3, 2, 3, 4, 5, 3, 1

ve ayrıca hafızamıza (RAM) sadece 3 çerçeve anlık olarak sığabiliyor olsun. Bu durumda her talepten sonra hafızadaki çerçeveler (frames, sayfalar, pages) aşağıdaki şekilde olacaktır.

1. sayfa talebinde sayfa ihlali (page fault) olur çünkü henüz hafızada olmayan bir sayfa talep edilmiştir.

| 1 | | |

2. sayfa talebinde sayfa ihlali yolur çünkü 2. sayfa da henüz hafızada bulunmamaktadır.

| 1 | 2 | |

3. sayfa talebinde yeniden bir sayfa ihlali bulunur sebebi 3. sayfanın da henüz hafızada olmamasıdır:

| 1 | 2 | 3 |

3. sayfadan sonra artık hafızamız tamamen dolmuştur çünkü hafıza kapasitesi 3 sayfa taşıyacak şekilde tanımlanmıştır.

4. sayfa talebinde zaten hafızada bulunan 2. sayfa istenir ve bir sayfa ihlali oluşmaz.

5. sayfa talebinde de benzer şekilde zaten hafızada bulunan 3 numaralı sayfa talep edilir ve bir sayfa ihlali oluşmaz.

6. sayfa talebinde hafızada bulunmayan 4 numaralı sayfa talep edilir ve bir sayfa ihlali oluşur. İşte tam bu noktada Optimal Replacement algoritması devreye girer. Çünkü hafıza tamamen doludur ve 4 numaralı sayfanın hafızaya yüklenmesi için hafızadaki sayfalardan birisinin kaldırılması gerekmektedir. Soru hangisinin kaldırılacağıdır.

Optimal Replacement algoritmasına göre en uzun süre erişilmeyecek olanı kaldırılır. Yani 2 numaralı sayfaya çalışma sonuna kadar erişilmeyeceği için bu sayfa kaldırılacak ve yerine 4 numaralı sayfa konulacaktır. Diğer sayfalar (1 ve 3) 2'den daha önce erişilecektir.

| 1 | 4 | 3 |

7. sayfa talebinde yine hafızada o anda bulunmayan 5 numaralı sayfa talep edilmiştir. Bu durumda hafızada bulunan sayfalar tekrar incelenir ve 4 numaralı sayfanın çalışma sonuna kadar bir daha erişilmeyeceği görülerek bu sayfa yerine 5 konulur.

| 1 | 5 | 3 |

8. sayfa talebinde 3 numaralı sayfaya erişilmek istenmiştir ve bu sayfa hal-i hazırda bellekte bulunmaktadır dolayısıyla bir sayfa ihlali oluşmaz ve doğrudan hafızadan ihtiyaç karşılanır.

9. sayfa talebinde ise daha önceden hafızada olan, 1 numaralı sayfa istenmiştir. Bu durumda da bir sayfa ihlali oluşmaz

| 1 | 5 | 3 |

şeklinde bir hafıza sayfa dizilimine erişilmiş olur.

Yukarıdaki bu örnekte toplam 9 sayfa için 5 sayfa ihlali olmuştur. Bu durumda sayfa ihlal oranı (Page fault rate) 0.55 olarak bulunmuş olunur.

### **SORU 33: CPU Utilization (MİB Meşguliyeti)**

Bilgisayar bilimlerinde en önemli kaynaklardan birisi de merkezi işlem birimidir (MİB, central processing unit CPU). Özellikle işletim sistemi çalışmaları sırasında bir işletim sisteminin bu en kıymetli kaynağı daha verimli kullanması amaçlanır.

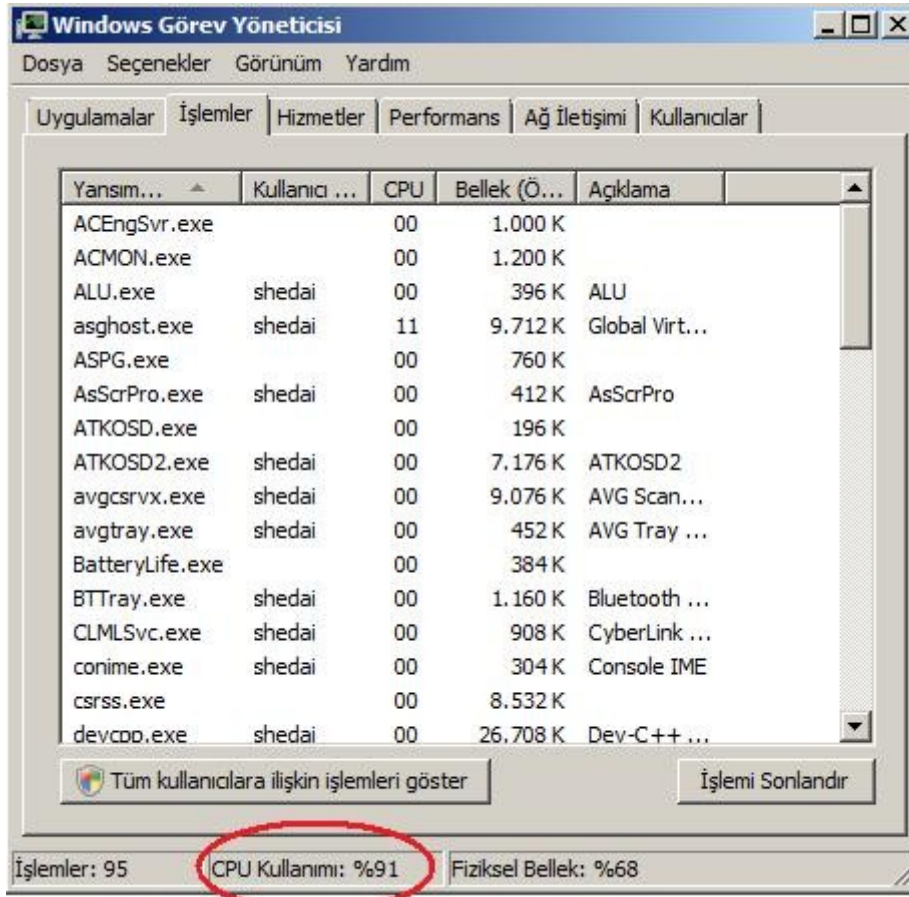
MİB Meşguliyeti (Utilization) ise işlemcide çalışmak için bekleyen [işlemler \(process\)](#) için sistemin meşguliyet oranını (utilization) bulmaya yarayan bir hesaplama.

Aslında çalışacak olan her işlem [bekleme sırası \(ready queue\)](#) ismi verilen bir sırada bekler. İşletim sisteminde [görevlendirici \(dispatcher\)](#) ismi verilen bir işlem ise bu sıradan [işlemci zamanlama algoritmasına \(cpu scheduling algorithm\)](#) uygun olan bir işlemi alarak işlemcide çalıştırır. Bu sırada sistemin ne kadar işleme cevap verebildiği (arz) ile sistemde üretilen ve işlemcide çalıştırılmak için bekleyen işlemlerin (talep) oranı işlemcinin meşguliyeti olarak hesaplanır.

Basitçe sıra teorisinde (queue theory) aşağıdaki şekilde gösterilen oranla bulunan meşguliyet değeridir:

$$\rho = \lambda / \mu$$

Çeşitli işletim sistemlerinde bu değeri okumak için yöntemler bulunur. Örneğin windows işletim sistemlerinde CPU Usage (CPU Kullanımı) olarak geçen bu terim, task manager (görev yöneticisinden) okunabilir:



Yukarıdaki resimde bu değer %91 olarak verilmiştir. Yani bir yoruma göre işlemcinin gelen taleplerin %91'ini karşıladığını söyleyebiliriz.

Örneğin Linux sistemlerde top komutu çalıştırıldığında aşağıdakine benzer bir ekran çıkar ve burada işlemci kullanımı görülebilir:

```

top - 17:07:00 up 2:38, 1 user, load average: 0.45, 0.70, 0.62
Tasks: 104 total, 1 running, 102 sleeping, 0 stopped, 1 zombie
Cpu(s): 23.8% us, 3.3% sy, 0.0% ni, 72.8% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 700000k total, 691768k used, 76832k free, 37960k buffers
Swap: 979956k total, 0k used, 979956k free, 368184k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4421	tv	15	0	148m	81m	18m	S	9.9	10.8	13:56.36	firefox-bin
4090	root	6	-10	173m	41m	3036	S	6.9	5.5	7:32.76	XFree86
6109	tv	15	0	29604	14m	12m	S	4.3	2.0	0:01.32	ksnapshot
4339	tv	15	0	31440	14m	11m	S	2.3	1.9	0:20.42	kicker
4335	tv	15	0	27736	12m	9.8m	S	2.0	1.6	0:13.47	kwin
4424	tv	15	0	31380	14m	11m	S	0.7	2.0	0:17.15	konsole
6090	root	16	0	2272	1152	872	R	0.3	0.1	0:00.25	top
1	root	16	0	1940	664	568	S	0.0	0.1	0:00.25	init
2	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
3	root	10	-5	0	0	0	S	0.0	0.0	0:00.35	events/0
4	root	11	-5	0	0	0	S	0.0	0.0	0:00.01	khelper
5	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	kthread
7	root	10	-5	0	0	0	S	0.0	0.0	0:00.08	kblockd/0
8	root	20	-5	0	0	0	S	0.0	0.0	0:00.00	kacpid
117	root	20	0	0	0	0	S	0.0	0.0	0:00.00	pdflush
118	root	15	0	0	0	0	S	0.0	0.0	0:00.03	pdflush
120	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	aio/0
119	root	25	0	0	0	0	S	0.0	0.0	0:00.00	kswapd0
709	root	10	-5	0	0	0	S	0.0	0.0	0:00.01	kseriod
811	root	15	0	0	0	0	S	0.0	0.0	0:00.39	kjournald
1404	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khudd
1906	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	scsi_eh_1
1909	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	usb-storage
3102	daemon	16	0	1712	376	280	S	0.0	0.0	0:00.00	portmap
3353	root	16	0	1660	624	512	S	0.0	0.1	0:00.26	syslogd
3359	root	16	0	2428	1304	380	S	0.0	0.2	0:00.08	klogd
3367	dnsmasq	16	0	1888	732	608	S	0.0	0.1	0:00.08	dnsmasq
3516	root	16	0	1764	696	572	S	0.0	0.1	0:00.02	automount
3523	root	16	0	1756	680	564	S	0.0	0.1	0:00.00	automount
3586	root	16	0	1764	696	572	S	0.0	0.1	0:00.00	automount

Yukarıda işletim sistemlerinin göstermiş olduğu işlemci meşguliyetinin nasıl hesaplandığını bir örnek üzerinden görmeye çalışalım.

Örnek olarak 3 işlemin çalıştığı sistemimizdeki her işlemin, işlemci zamanları (CPU time) ms cinsinden aşağıdaki şekilde verilmiş olsun:

İşlem	İşlemci Zamanı
A	11
B	21
C	31

Ayrıca sistemimizdeki işlemci zamanlama algoritmasının (CPU Scheduling Algorithm), [round robin](#) algoritması olduğunu ve zaman bölmesi (time quantum) olarak 20ms olduğunu kabul edelim. Sistemin ilk çalışması sırasında da sistemdeki bekleme sırasında (ready queue), işlemlerin A,B ve C sırasıyla bulunduğunu kabul edelim. Ayrıca soru kapsamında yer değiştirme zamanının (context switch) 1ms olduğunu kabul edelim. Bütün bu verilen şartlarda işlemci meşguliyetini (CPU Utilization) hesaplamaya çalışalım:

- İlk işlem 1. zaman bölmesinde çalışacak ve bitecektir. (zaman bölmesi 20ms ve işlem 11ms olduğu için)
- Ardından 1ms süresince yer değiştirme (context switch) olacaktır.
- İkinci işlem 12. ms'de çalışmaya başlayacak ve zaman bölmesi olan 20ms çalıştıktan sonra bekleme sırasına geri konulacaktır. (ikinci işlem olan B'nin bitmesi için 1ms zaman kalacaktır)
- Ardından 1ms süresince yer değiştirme (context switch) olacaktır.

- C işlemi 33. ms'de çalışmaya başlayacak ve zaman bölmesi olan 20ms çalıştıktan sonra bekleme sırasına geri konulacaktır. (ikinci işlem olan C'nin bitmesi için 11ms zaman kalacaktır)
- Ardından 1ms süresince yer değiştirme (context switch) olacaktır.
- Sıradaki işlem olan B işlemi 54. ms'de tekrar çalıştırılacak ve bitmesi için gereken 1ms süresince işlemciyi meşgul edecektir. Sonuçta 55. ms'de işlem bitecek ve yer değiştirme olacaktır.
- ms süresince yer değiştirme (context switch) olduktan sonra
- 56. ms'de C işleminin kalanı yapılmak üzere işlemciye yüklenecek ve kalan 11ms boyunca işlemciyi meşgul edecektir. Sonuçta 67. ms'de C işlemi de bitecektir.

İşlemci meşgulliyetini bulmak için çalışan işlemlerin toplam zamanına bakalım :

$$A + B + C = 11 + 21 + 31 = 63 \text{ ms'dir.}$$

Sistemin bu işlemleri ne kadar zamanda çalıştırdığına bakalım: 67 ms olarak yukarıda bulduk.

Dolayısıyla :

$$\rho = \lambda / \mu$$

oranında

$$\rho = 63 / 67$$

$$\rho = 0.94$$

olarak bulunur.

### **SORU 34: Meşgulliyet (Utilization, Kullanım)**

Bilgisayar bilimlerinde [sıra \(queue\) teorisinde](#), sıradaki bir varlığın ne kadar meşgul edildiğini ölçmeye ve bu ölçüme göre kararlar vermeye verilen isimdir.

İstatistiksel olarak  $\rho$  sembolü ile gösterilir ve şayet  $\rho$  değeri 1'den büyükse sıranın uzadığı, 1'e eşitse sıranın ne kısalıp ne de uzadığı ve şayet 1'den küçükse sıranın kısalacağı veya sırada kimse kalmadığı sonucu çıkarılır.

Burada  $\rho$  değerinin hesaplanması için aşağıdaki formül kullanılabilir:

$$\rho = \lambda / \mu$$

Yukarıdaki formülde  $\lambda$  değeri sıraya gelen varlıkların oranını,  $\mu$  değeri ise sistemin bu sıradaki varlıklara hizmet verme oranını göstermektedir. Yani basitçe sıraya girenler ve sıradan çıkanların oranı olarak düşünülebilir. Daha basit bir ifadeyle talep/arz oranı olarak düşünülebilir.

Basit bir örnek üzerinden anlatmak gerekirse örneğin bir bankada bulunan banka görevlileri ortalama olarak bir müşteriye 1 dakikada hizmet veriyor olsun. Saatte 50 müşterinin geldiği bir bankada meşgulliyet aşağıdaki şekilde hesaplanabilir:



Öncelikle arz ve talebi aynı birimlere getirmek gerekir. Yani bankanın arz ettiği hizmet değeri müşteri/saat şeklindeyken, bankanın talep değeri müşteri/saat cinsinden. Bu durumda bankanın arz değerini de müşteri/saat cinsine çevirelim.

Basit bir hesapla 1 dakikada bir müşteriye hizmet verilen bir bankada saatte 60 müşteriye hizmet verilir. Öyleyse bu bankanın meşguliyet oranı (utilization):

$$\rho = \lambda / \mu$$

$$\lambda = 50$$

$$\mu = 60$$

$$\text{için } \rho = 50 / 60$$

$$= 0.83$$

olarak bulunur. Bu durumda bankanın müşterilerine zamanında hizmet verebileceğini ve müşterilerin sıra beklemeyeceğini veya bir sıra varsa zamanla azalacağını yorumlayabiliriz.

### **SORU 35: Semafor (Semaphore, Flama, İşaret)**

Bilgisayar bilimlerinde özellikle de işletim sistemi ve müşterek programlamada (concurrent programming, eş zamanlı programlamada) sıkça kullanılan bir eşleme (synchronization) yöntemidir. Yani birden fazla işin (process) aynı anda çalışması halinde birbirleri için risk arzettikleri kritik zamanlarda (critical sections) birbirlerini beklemesini sağlayan bir mekanizmadır.

Basitçe bir değişken veya bir [mücerret veri yapısı \(abstract data type, soyut veri tipi, adt\)](#) üzerine kurulmuş olup, bu veri yapısı içerisindeki bilgiye ve fonksiyonlara göre çalışmaktadır.

Semaforların çalışması sırasında [bölünmezlik \(atomicity\)](#) ön plandadır. Yani bir semafor'un içerisinde yapılan birden fazla iş, program tarafından sanki tek bir iş gibi algılanmalı ve araya başka işin girmesine izin verilmemelidir.

Semaforlar kullanım alanları ve tasarımları itibariyle ikiye ayrılır:

- ikili semaforlar (binary semaphores)
- tam sayı semaforları (integer semaphores)

ilk semafor tipi olan ikili semaforlar sadece iki işlem (process) arasında eşleme (Synchronization) sağlar ve üçüncü bir iş için tasarlanmamıştır. Tam sayı semaforları ise istenilen miktarda işlemi kontrol edebilir.

### **SORU 36: Atomluluk (Atomicity)**

Latince bölünemez anlamına gelen atom kökünden üretilen bu kelime, bilgisayar bilimlerinde çeşitli alanlarda bir bilginin veya bir varlığın bölünemediğini ifade eder.

Örneğin programlama dillerinde bir dilin atomic (bölünemez) en küçük üyesi bu anlama gelmektedir. Mesela C dilinde her satır (statement) atomic (bölünemez) bir varlıktır.

Benzer şekilde bir verinin bölünemezliğini ifade etmek için de veri tabanı, veri güvenliği veya veri iletimi konularında kullanılabilir.

Örneğin veri tabanında bir işlemin (transaction) tamamlanmasının bölünemez olması gerekir. Yani basit bir örnekle bir para transferi bir hesabın değerinin artması ve diğer hesabın değerinin azalmasıdır (havale yapılan kaynak hesaptan havale yapılan hedef hesaba doğru paranın yer değiştirmesi) bu sıradaki işlemlerin bölünmeden tamamlanması (atomic olması) gerekir ve bir hesaptan para eksildikten sonra, diğer hesaba para eklenmeden araya başka işlem giremez.

Benzer şekilde işletim sistemi tasarımı, paralel programlama gibi konularda da bir işlemin atomic olması araya başka işlemlerin girmemesi anlamına gelir.

Örneğin sistem tasarımında kullanılan check and set fonksiyonu önce bir değişkeni kontrol edip sonra değerini değiştirmektedir. Bir değişkenin değeri kontrol edildikten sonra içerisine değer atanmadan farklı işlemler araya girerse bu sırada problem yaşanması mümkündür. Pekçok işlemci tasarımında buna benzer fonksiyonlar sunulmaktadır.

Genel olarak bölünemezlik (atomicity) geliştirilen ortamda daha düşük seviyeli kontroller ile sağlanır. Örneğin işletim sistemlerinde kullanılan [semafor'lar \(semaphores\)](#), kilitler (locks), koşullu değişkenler (conditional variables) ve monitörler (monitors) bunlar örnektir ve işletim sisteminde bir işlemin yapılması öncesinde bölünmezlik sağlayabilirler.

Kullanılan ortama göre farklı yöntemlerle benzer bölünmezlikler geliştirilebilir. Örneğin veritabanı programlama sırasında koşul (condition) veya kilit (lock) kullanımı bölünmezliği sağlayabilir.

### **SORU 37: Gizli Dosya (Hidden File)**

İşletim sistemlerinde kullanılan dosya tiplerinden birisidir. Basitçe sistemde kullanılan kritik dosyaların kullanıcı müdahalesinden korumak için geliştirilmiştir. Örneğin Windows™ işletim sisteminde kullanılan gizli dosyaların ağırlıklı amacı sistem dosyalarını ve önemli ayarlamaları içeren klasörleri korumaktır.

Linux / Unix gibi işletim sistemlerinde de gizli dosyaların isimleri “.” işareti ile başlamaktadır. İşletim sistemi otomatik olarak bu dosyaları gizli dosya adleder ve klasik dosya görüntüleme ve dosya sistemi dolaşma işlemlerinde bu dosyalar gizlenir.

Örneğin linux işletim sisteminde kullanılan ve dosya listelemeye yarayan “ls” komutunda gizli dosyaları göstermek için -h (hidden) parametresi verilmesi gerekir.

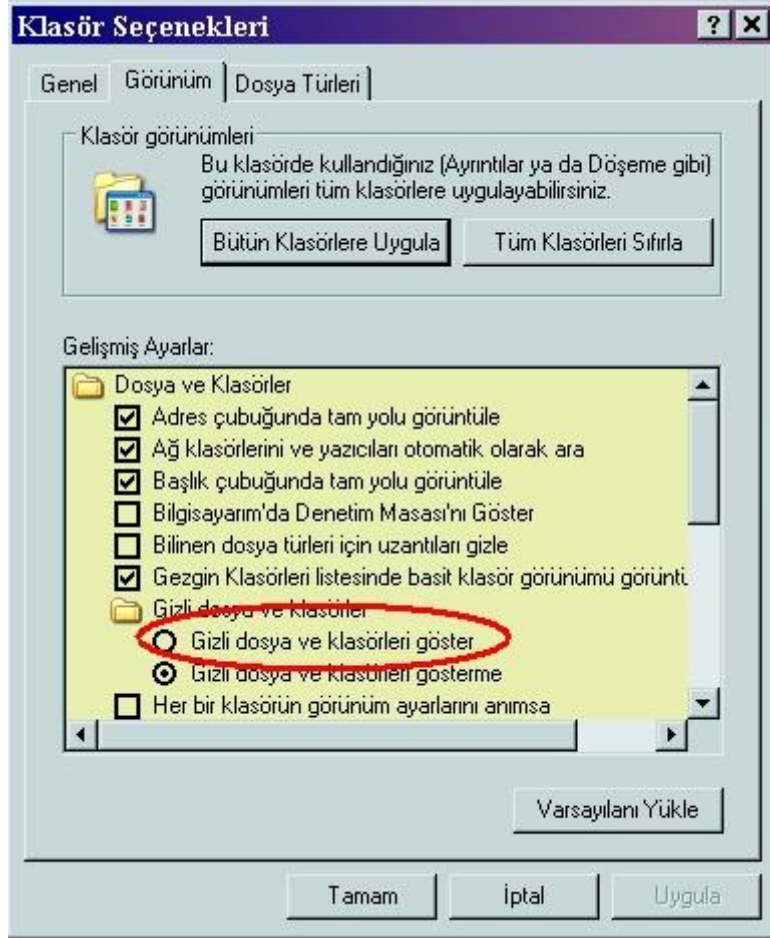
“ls -h” şeklinde. Arşiv dosyalarının da görülmesi için “ls -ah” şeklinde kullanılabilir.

DOS işletim sisteminde bu işlemi yapan “dir” komutunu da /h parametresi ile kullanabiliriz.

“dir /ah” şeklinde çağrılabilir ve görüntülenebilir.

Windows işletim sisteminde ise gizli dosyaların gösterilmesi için “klasör seçeneklerinden” gerekli ayarların yapılması gerekir.

Basitçe “bilgisayarım>Araçlar>Klasör Seçenekleri>Görünüm” ekranında aşağıdaki seçeneğin seçilmesi gizli dosyaların gösterilmesi için yeterlidir:



Ayrıca windows ve DOS işletim sistemlerinde dosyaların gizli veya arşiv dosyası olduğunu görmek veya tiplerini değiştirmek için “attrib” komutu kullanılabilir. Basitçe + ve – işaretleri ile dosyaya özellik eklenebilir veya çıkarılabilir. Örneğin bir dosyayı gizli dosya yapabilmek için

“attrib +h dosya” şeklinde komutu kullanmak gerekir.

Linux / Unix işletim sistemlerinde ise dosyanın isminin başına nokta “.” işaretinin eklenmesi yeterlidir.

### **SORU 38: C ile Zaman İşlemleri**

C dilinde mevcut zamanı almak ve işlemek mümkündür. Bunun için time.h dosyasının içerisinde bulunan fonksiyonlar kullanılabilir. Ayrıca time.h dosyasında bulunan [time\\_t oluşumu \(struct\)](#) zaman tutmak için geliştirilmiştir ve zamanı oluşturan alt unsurları da içerir.

Örneğin şu andaki zamanı ekrana basmak için aşağıdaki kod kullanılabilir:

```

struct timeval tv;
time_t curtime;
gettimeofday(&tv, NULL);
curtime=tv.tv_sec;
printf("%m-%d-%Y   %T.", localtime(&curtime));

```

Örneğin bir kullanıcıdan yazı okuyan ve ne kadar zamanda okuduğunu ekrana basan kod aşağıdaki şekilde yazılabilir:

```

#include <stdio.h>
#include <time.h>
#include <conio.h>
int main ()
{
    time_t start,end;
    time (&start);
    printf ("Bir tuşa basınız");
    getch();
    time (&end);
    printf ("Tuşa basmanız %.2lf saniye sürdü.n", difftime (end,start));
    return 0;
}

```

Yukarıdaki time\_t yapısı oldukça kullanışlı olmasına karşılık mikro saniye gibi düşük zamanları algılamakta yetersizdir. Bunun için biraz daha detaylı çalışan timeval yapısından ve gettimeofday fonksiyonundan faydalanılabilir:

```

struct timeval starttv, endtv;
struct timezone starttz, endtz; gettimeofday(&starttv, &starttz);
for (i=0;i<100;i++)
    for(j=0;j<256;j++)
        for(k=0;k<10;k++)
            printf("deneme") ;
gettimeofday(&endtv, &endtz);
float fark1=getdiff(endtv, starttv);

```

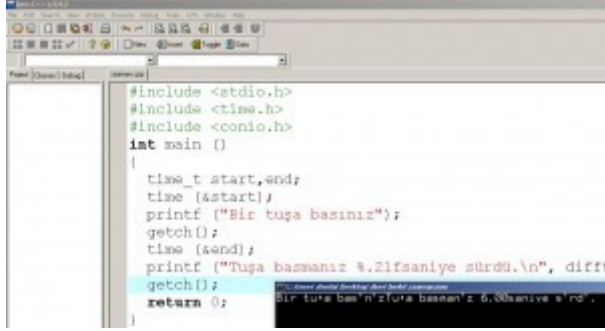
Örneğin yukarıdaki kodda iç içe 3 döngü içerisinde ekrana deneme yazdırılıyor (toplam 100x256x10 tane) bu işlemin aldığı vakti ölçmek için gettimeofday fonksiyonundan faydalanılmış ve en sonunda da fark hesaplanmış. Burada eksik olan getdiff fonksiyonu aşağıdaki şekilde yazılabilir:

```

float getdiff(struct timeval endtv, struct timeval starttv)
{
    float diff=0;
    diff=(endtv.tv_sec-starttv.tv_sec)*1000000+
        (endtv.tv_usec-starttv.tv_usec);
    return diff;
}

```

Serper Bey'in sorusu üzerine ekliyorum, büyütmek için resme tıklayabilirsiniz:



```
#include <stdio.h>
#include <time.h>
#include <conio.h>
int main ()
{
    time_t start,end;
    time (&start);
    printf ("Bir tuşa basınız");
    getch();
    time (&end);
    printf ("Tuşa basmanız %.2lf saniye sürdü.\n", difftime(end, start));
    getch();
    return 0;
}
```

## Dev-CPP üzerinde zaman işlemleri

Yukarıdaki, linux kodlarından farklı olarak Dev-CPP kullanmak isteyenler clock sınıfından yararlanabilir.

Kod basitçe zamanı clock() fonksiyonu ile okur ve dört işleme tabi tutar. Buradaki değişken tipi ise clock\_t cinsindendir.

```
clock_t start, end;
```

```
start=clock();
```

```
.... işlemler .....
```

```
end = clock();
```

```
printf("fark: %f",end-start);
```

şeklinde yazılan kod başarılı bir şekilde çalışma süresini ölçer. Elbette bu işlemler sırasında sys/time.h kütüphanesi include edilmelidir.

## SORU 39: İşlem Çatallanması (Process Forking)

Yazılan her program ilk başta tek bir işlem olarak çalışmaya başlar. Temel olarak [derlendikten\(compile\)](#) hemen sonra [bağlanarak \(link\)](#) hafızaya yüklenir (load).

Yüklenen programı, [işletim sistemi \(operating system\)](#) bir [işlem \(process\)](#) olarak çalıştırır. Ancak bazı işlemler yüklendikten ve çalışmaya başladıktan sonra yeni işlemler üretebilirler.

Aslında bu işlem üretme çalışan işlem üzerinde bir çatallanmaya sebep olmak demektir. Bu çatallanmayı destekleyen dillerde bu iş için özel fonksiyonlar bulunur. Örneğin C dilindeki fork() fonksiyonu bu amaçla geliştirilmiştir.

C dilinde yazdığımız bir programda daha önceden yazılmış ve işletim sisteminde bulunan (tercihen linux) "ls" komudunu çağıracağımızı düşünelim.

```
#include<stdio.h>
int main()
{
    int pid;
```

```

pid=fork();
if(pid==-1)
{
    printf("n Çatallamada hata oldu");
    exit(0);
}
if(pid==0)
{
    printf("n Çocuk işlem.....");
    execlp("/bin/ls", "ls", NULL);
}
else
{
    printf("n Ata işlem");
    wait(pid);
    printf("Tamamlandı");
    exit(0);
}
}

```

Yukarıdaki kodda önce bir fork fonksiyonu çağrılmış ardından if kontrolleri ile fork fonksiyonundan dönen integer değer kontrol edilmiştir. Bunun anlamı aslında çalışan işlemin (process) kopyalanmasıdır. Yani fork fonksiyonu çağrılınca aynı programdan iki kopya hafızada çalıştırılır. Arada tek fark fork fonksiyonundan dönen değerdir. Basitçe çocuk işlemde (child process) dönen değer 0 iken ata işlemde(parent process) dönen değer 1'dir.

Şayet fork işleminde bir hata olursa dönen değer -1 olur, bunu hata kontrolü için yukarıda da gösterildiği üzere kullanabiliriz.

## **SORU 40: Kabuk (Shell)**

Bilgisayar bilimlerinde kabuk kelimesi daha çok çevreleyici, kaplayıcı anlamında kullanılmaktadır. Örneğin işletim sistemlerinde [çekirdeğin \(kernel\)](#) dış dünya ve kullanıcılar ile iletişim kurmasını sağlayan işletim sisteminin parçasına kabuk (shell) adı verilmektedir.

### **İşletim Sistemlerinde Kabuk**

Kabuğun en temel görevini bir komut satırı (command line interface) olarak tanımlayabiliriz. Örneğin DOS, LINUX veya UNIX işletim sistemlerinde komutları alan ve bu komutlar dahilinde çekirdeğe işlemleri geçiren modül olarak düşünülebilir.

Örneğin basit bir taşıma işlemi:

```
mv a.txt b.txt
```

komuduyla yapılabilir. Bu dosya taşıma işlemi sırasında dosya sisteminde bazı bilgilerin değiştirilmesi söz konusudur (örneğin FAT Tablosundaki kayıt ya da inode değerleri gibi) bu değişiklikler çekirdek tarafından yapılır.

komut satırının daha gelişmiş olarak kabul edilebilen Grafik Arayüzü (Graphical User Interface, GUI) eklentileri ile kabuğun yaptığı işlerde pek farklılık olmasa da kullanıcıya sunulan işlemlerin şekli ve kullanıcının işlem yapabilme kabiliyeti arttırılmıştır. Ancak temelde bir işletim sisteminin çekirdeği ile kabuğun ilişkisi aynıdır.

Kabuk programlama (shell programming) ismi verilen bir programla tipi de kullanıcı işlemleri yapılan bu kabuktaki işlemleri bir program dahilinde kullanıcı iletişimi olmaksızın çalıştırmayı amaçlar.

DOS üzerindeki bat dosyaları ([Batch files](#)) ve linux ve unix üzerindeki shell programming ve windows üzerindeki vbscript ve javascript (csh ve jsh) bu tip kabuk programlamaya örneklerdir. Burada kabuğun yaptığı işlemler kullanılarak yine kabuğun yaptığı birleştirilmiş işlemler elde edilir. Örneğin sistemdeki her kullanıcının dizinlerinin içerisinde bir adet yardım dosyası koymak isteyelim. bunu yapan hazır bir komut yoktur. Ancak bir dosyayı kopyalayan komudumuz 'cp' her kullanıcı için tekrar tekrar çalıştırılarak bu işlem yapılabilir. İşte bir sistem yöneticisi örneğin 100 kullanıcı için teker teker bu kopyalama işlemini yapmak yerine bir kabuk programı yazarak bu işlemi yaptırmaktadır.

### **SORU 41: Çekirdek (Kernel)**

Bilgisayar bilimlerinin farklı alanlarında kullanılmasına karşılık, çekirdek kavramı genelde birşeyin merkezi veya kalbi şeklinde tabir edilebilecek anlamlara gelmektedir.

#### **İşletim sistemlerinde çekirdek:**

İşletim sisteminin temel fonksiyonlarının icra edildiği kısımdır. Kullanıcılar ile iletişim kuran kabuk (shell) sadece dış işleri yapmaktan sorumlu olup, işletim sisteminin bütün temel fonksiyonları çekirden üzerinde çözülür.

Bir çekirdeğin temel görevleri aşağıdaki şekilde sıralanabilir

- Giriş çıkış işlemlerinin yönetilmesi (I/O management): Örneğin klavye, fare veya ekran gibi dış donanımların yönetilmesi bu donanımların hafıza ve işlem ihtiyaçlarının sistem kaynakları içerisinde çözülerek tasarlanan zaman ve tasarlanan başarıyla çalışmalarını sağlamaktır.
- İşlem yönetimi (process management): Bir işletim sisteminde çalışan programların ve programların ürettiği işlemlerin (process) yönetilmesi işidir. Bilindiği üzere her işlemin sistemden sürekli olarak talepleri olmaktadır. Bu taleplerin karşılanması ve işlemlerin belirli bir ahenk ve adil bir sıra ile çalışmasını sağlamak gibi görevler işletim sisteminin çekirdeği tarafından yürütülür.
- Hafıza yönetimi (memory management): İşletim sisteminin çekirdeği, kendisi de dahil olmak üzere, o anda çalışan bütün işlemlerin hafıza gereksinimini, en verimli şekilde karşılamak zorundadır. Bunun için sayfalama (paging) ve kıtalama (Segmentation) işlemlerinin yapılması.
- Aygıt yönetimi: Sisteme bağlı çalışan aygıtların kontrolü, bu aygıtların işlemci ve hafıza ihtiyaçlarının karşılanması ve işletim sisteminin diğer parçalarının bu aygıtlara erişimi.
- Dosya yönetimi: Disk üzerinde tutulan dosyaların takibi, bu dosyaların disk üzerinde verimli bir şekilde tutulması, hızlı erişilmesi, güvenliğinin sağlanması ve dosyalama ile ilgili kopyalama, taşıma, okuma, yazma gibi işlemlerin icrası.

Yukarıdaki temel işlemler bir işletim sisteminin çekirdeğini oluştururken her zaman bulunması gereken durumlar değildir. Örneğin bazı işletim sistemleri disk bile olmayan ortamda çalışmaktadır (günümüzdeki cep telefonu ve kişisel asistanlar (PDA, personal digital

assistant) bunlara birer örnek olabilir) bu durumda doğal olarak bir disk kontrolünden bahsetmek mümkün değildir.

#### **SORU 42: Dahili Parçalar (Internal Fragments)**

Birden fazla işlemin bir işletim sistemi üzerinde çalıştırılması sırasında hafızadaki işlemlerini belirli bir düzene göre yerleştirilmesi gerekir.

Bu yerleştirme sırasında çıkan problemlerden birisi de parçalar (fragments) 'dir. Buna göre işletim sisteminin önünde iki ihtimal bulunmaktadır. Ya hafızayı sabit slotlara bölüp işlemlere bu slotlardan ihtiyaç duyduğu kadar verecektir (örneğin slotların boyu 200birimlik olsun, işlemlerden birisi 700 birim isterse 4 slot bu işleme verilecek, sonuç olarak  $4 \times 200 = 800$  birim ayrılmış olacak ve 100 birim hafıza israf olacaktır , bu yaklaşıma sayfalama (paging) ismi verilir ) ya da her işleme ihtiyaç duyduğu kadar hafıza alanı ayrılacaktır bu durumda da işlemler arasında boşluklar oluşacaktır. Bu boşluklara da harici parça (external fragment) ismi verilir.

Aşağıda sayfalama (paging) sırasında oluşan bir dahili boşluk (iç boşluk, internal fragment) tasvir edilmiştir:

Sayfa 1 ( Page 1)	İşlem 1 (Process 1)
Sayfa 2 ( Page 2)	İşlem 1 (Process 1)
Sayfa 3 ( Page 3)	İşlem 1 (Process 1)
Sayfa 4 ( Page 4)	İşlem 1 (Process 1)
Sayfa 5 ( Page 5)	İşlem 2 (Process 2)

Yukarıdaki örnekte de görüldüğü üzere 4. sayfada işlem 1'in tam kullanmaması dolayısıyla bir boşluk oluşmuştur. Bu boşluk başka işlemler tarafından asla kullanılamaz çünkü bu sayfa (page) artık işlem 1 için ayrılmıştır.

Örneğin 200 birimlik sayfa boyutuna sahip bir sayfalama da 700 birim kullanan bir işlem yukarıdaki şekilde görüldüğü gibi 3 tam sayfa bir de yarım sayfa kullanacaktır. İşte bu yarım sayfada oluşan boşluğa dahili boşluk (internal fragment) ismi verilir.

Bu sorunun çözümü için kıtalama (segmentation) kullanılır.

#### **SORU 43: Kıtalamak (Bölütlemek, Segmentation)**

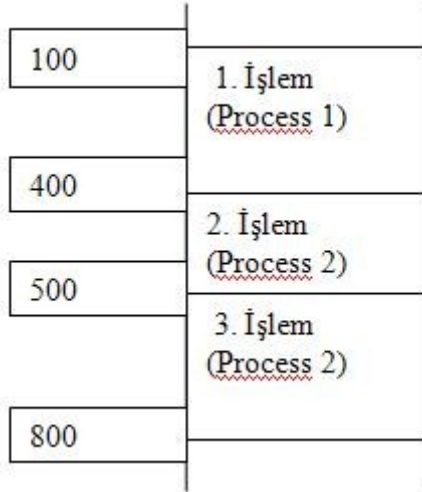
İşletim sistemlerinin temel görevlerinden birisi olan hafıza yönetimi (memory management) için kullandıkları çözüm yöntemlerinden birisidir. Bilindiği üzere bir işletim sistemi birden



fazla işlem (multi process) çalıştırıyorsa bu durumda işletim sisteminin hafızayı bu işlemler arasında ihtiyaçlarına uygun bir şekilde dağıtması gerekir.

Bu dağıtımda bir işleme tam olarak ihtiyacı olduğu kadar yer ayırma yaklaşımına kısıtlama (bölütlemek , segmantasyon , segmentation) ismi verilir.

yaklaşım basitçe aşağıdaki şekilde gösterildiği gibidir:



Yukarıda, hafızada (RAM) çalışmakta olan 3 ayrı işlem ve bu işlemlerin hafıza adresleri görülmektedir. Dolayısıyla bir işlemin fiziksel adresi kendi içerisindeki mantıksal adresten farklı olmaktadır.

Yani her işlem kendi adresleme sistemini kullanır ve hafızada nerede çalıştığından habersizdir. 1. İşlem için 50 numaralı adres, 2. işlem için 50 numaralı adres ve 3. işlem için 50 numaralı adres hep kendi 50 numaralı adresleridir. Basit bir hesaplama ile bu adresler 1. işlem için 150, 2. işlem için 450 ve 3. işlem için 550 numaralı gerçek adreslerken, işlemler bundan habersiz olarak kendi 50 numaralı adreslerine erişmeye çalışırlar.

İşlemlerin talep ettiği adresler (demanding addresses) kendi adresleridir ve bu adreslere mantıksal adres (logical addresses) ismi verilir. Gerçekte bilginin bulunduğu adresler ise fiziksel adreslerdir (physical addresses). Bu iki bilgi arasında dönüşüm aşağıdaki formülle yapılabilir:

fiziksel adres = kitanın başlangıcı + mantıksal adres

mantıksal adres = fiziksel adres – kitanın başlangıcı

Yukarıdaki bu formüllere göre iki adres arasındaki dönüşüm kitanın (segment) hafızadaki adresi kadardır. Bu adresler kita tablosu (segment table) ismi verilen bir tabloda aşağıdaki şekilde tutulur:

Limit	Başlangıç (Base)
300	100
100	400
300	500

Yukarıdaki sistemde 3 kıta (segment) olan bir sistem söz konusudur. Burada kıta numaraları verilmemiştir bunun sebebi yukarıdaki sıra ile birer kıta numarası olmasıdır. Örneğin 2 numaralı segment 300 limit ve 500 başlangıç değerine sahiptir.

Bu tabloda yer alan başlangıç değeri kıtamızın hafızaya yüklendiği adres olurken limit değeri de hafızada ne kadar yer kapladığını göstermektedir.

Limit değerinin tutulmasının sebebi bir işlemin başka işlemlerin adreslerine erişmesini engellemektir. Dolayısıyla her erişim aşağıdaki kontrolden geçer:

$\text{başlangıç} + \text{mantıksal adres} < \text{başlangıç} + \text{limit}$

şayet bu sorunun cevabı hayır ise bu durumda ilgili kıtanın (segment) dışında bir alana erişilmeye çalışılıyor demektir ve hata verilir.

kıtalama işlemi sırasında adresleme kıta numarası ve mantıksal adres ikilisinden oluşmaktadır.

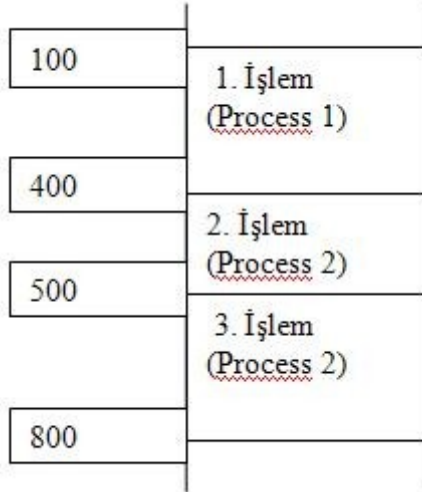
(s,d) ikilisi (segment, offset) (kıta, mantıksal adres) ikilisi anlamındadır.

Örneğin yukarıdaki kıta tablosunda (1,40) gerçek adres olarak (Fiziksel adres, physical address) 440 olmaktadır (1,120) bir hatadır.

#### **SORU 44: Harici Parçalar (External Fragments)**

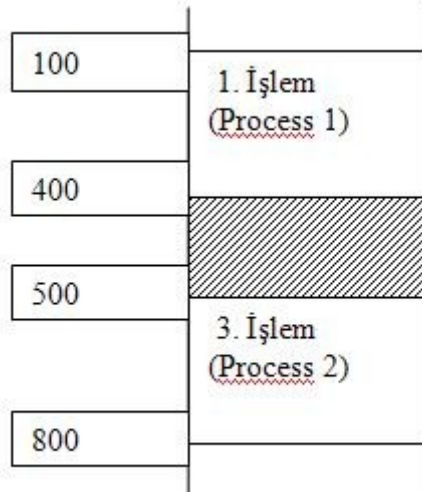
Hafıza yönetimi sırasında kullanılan kıtalama (bölütleme, segmentation) hafızadaki her işleme tam olarak istediği kadar yer ayırmaya çalışır. Bu yaklaşımda, işlemler arasında oluşabilecek boşluklara verilen isim harici parçalar (dış parçalar, external fragments)'dir.

Her işleme ihtiyaç duyduğu kadar yer ayırmak ilk başta daha verimli gibi görülse de bu çözümde de boşluklar ve hafıza israfı söz konusudur. Sayflama (Paging) yaklaşımında slotların içine olan boşluklara iç parça (internal fragment) ismi verilirken kıtalama (segmentation) sırasında oluşan boşluklara dış parça (external fragment) ismi verilir. Bu boşluklar aşağıdaki şekilde oluşur.

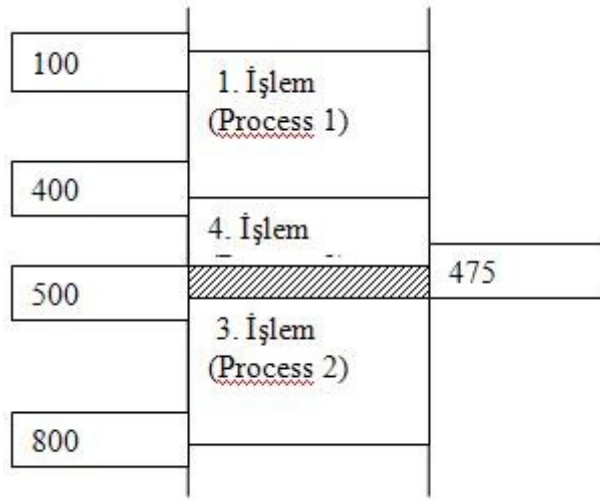


Yukarıdaki şekilde görüldüğü üzere sırasıyla 1, 2 ve 3. işlemler gelmiş hafızada gösterilen alanlar bu işlemlere ayrılmıştır. Şimdi bu işlemlerden örneğin 2. işlemin çalışıp bittiğini düşünelim. Bunun anlamı 2. işlem hafızadan kaldırılacak demektir.

2. işlem hafızadan kaldırıldıktan sonra oluşan tablo aşağıdaki şekildedir:



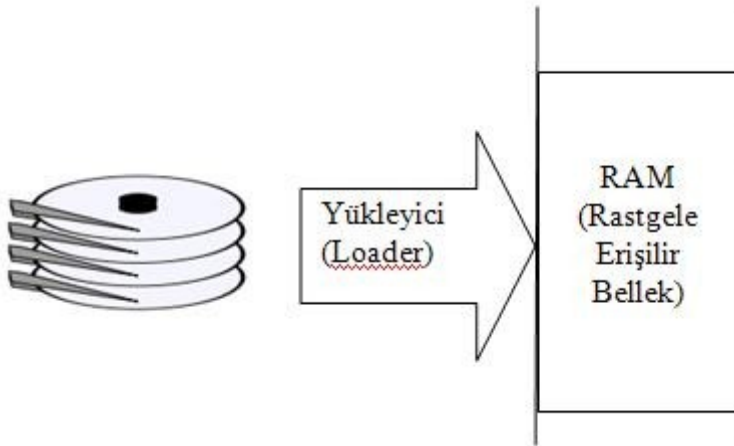
Yukarıda da görüldüğü üzere 400 ile 500 arasındaki hafıza alanı boşalmıştır. Buradaki boşluk verimsiz bir boşluktur bunun sebebi 100 boyutunda başka bir işlem gelmediği sürece bu alan kullanılamaz. Hele bir de aşağıdaki gibi örneğin 75 boyutunda bir işlem gelirse arada kalan 25 birimlik boşluğun kullanılabilirliği oldukça azalır.



işte bu alanlara bilgisayar bilimlerinde verilen isim harici parça (external fragment)'dir.

#### **SORU 45: Yükleyici (Loader)**

Yükleyiciler basitçe bir programı diskten alıp hafızaya yüklemekle sorumlu programlardır.



Bir program yazıldıktan ve [derlendikten \(compile\)](#) sonra programın makine dilindeki karşılığı elde edilir. Bu karşılık tam bir kod olmayıp harici kütüphanelerden faydalıyor olabilir. Bu kütüphaneler de programa dahil edilip tam bir program elde edildikten sonra (yani [bağlandıktan sonra \(linker\)](#) ) program artık çalıştırılmaya hazırdır.

Programın çalışması ise programın CPU üzerinde yürütülmesi ile olabilir ve bunun için programın öncelikle [hafızaya \(RAM\)](#) yüklenmesi gerekir. Burada yükleyici (loader) devreye girer. Yükleyici makine dilindeki bu kodu alarak işletim sisteminin işaret ettiği adrese programı yükler. Buradan sonrası işletim sistemi tarafından yürütülür.

#### **SORU 46: Hafıza Yönetimi (Memory Management)**

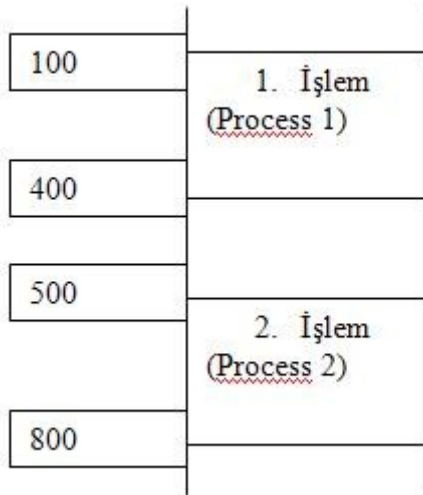
Bir [işletim sisteminin \(operating system\)](#) birden [fazla işlem çalıştırması durumunda \(multi process\)](#) bu işlemlerin [hafızayı](#) nasıl paylaşacakları ve hafızanın nasıl daha verimli kullanacağı hafıza yönetiminin konusudur.

Şayet işletim sisteminde tek işlem (process) çalışıyorsa bu çözülmesi çok daha kolay bir durumdur. İşlemler derlendikten (compile) sonra hafızaya yükleyici (loader) ismi verilen bir programla yüklenir ve işlemci (CPU) bu işlemi çalıştırmaya başlar. İşlem bitene kadar bilgisayar bu işlemin kontrolünde olur ve ancak işlem tamamen bittikten sonra hafızadan kaldırılarak yerine yenisi konulabilir ve farklı bir işlem çalıştırılabilir.

Ancak bir işletim sistemi birden fazla işlemi çalıştırmak gerekiyorsa burada bir iki problem ortaya çıkar.

İşlemlerin hafıza adresleri: Çalışan işlemler programlanır veya derlenirken hafızayla ilgili işlemleri gereği adres bilgileri içerirler. Örneğin bir program hafızadaki sabit bir adrese (sözgelimi 150 numaralı adres olsun) erişmek isteyebilir.

Bilindiği üzere hafızada aynı adrese sahip tek yer bulunmaktadır. Dolayısıyla örneğin 150 numaralı adrese erişme talebi tek bir yerdir oysaki programların adres erişme talepleri genelde yine programın farklı bir alanına erişmek içindir. Buradaki problem hafızada birden fazla işlemin yüklenmesi durumunda aynı adrese sadece bir işlemin yüklenecek olması ve her işlemin farklı adreslerde çalışacak olmasıdır. Dolayısıyla hafızadaki adresler hareketli olacak (dynamic, dynamic addressing) ve her işlem artık farklı bir adres aralığına yüklenecektir.



Örneğin yukarıdaki şeklin hafızayı (RAM) temsil ettiğini farz edelim. Bu resimde de görüldüğü üzere iki işlem aynı anda hafızaya yüklenmiş bu işlemlerden ilki 100-400 aralığında yüklü ikincisi ise 500-800 aralığında yüklüdür.

Örneğimize geri dönecek olursak 150 numaralı hafıza adresi 1. işlemin yüklü olduğu adrestir. 2. işlemin buraya erişmesi ise risklidir. Bunun çok basit iki temel sebebi vardır.

1. Güvenlik ihlali olabilir. Yani örneğin bu adreste bir kullanıcının şifresi yazılı olabilir ve başka bir işlem buraya erişirse (ki bu başka bir işlem başka birisinin işlemi de olabilir) bu şifreyi ele geçirebilir
2. çalışan işlemlerin birbirine müdahalesinin sonucu kestirilemez. Yani siz işletim sisteminde bir kelime işlem programı çalıştırdığınızı düşünün (örneğin word) herhangi birisi girip sizin wordde yazıklarınızı daha kaydetmeden ekranda yazılıyken değiştirebilir, sonuçlar her zaman bu örnek kadar masum da olmak zorunda değildir. Pek çok ihtimalde sistemin çökmesi veya kalıcı olarak zarar görmesi mümkündür.

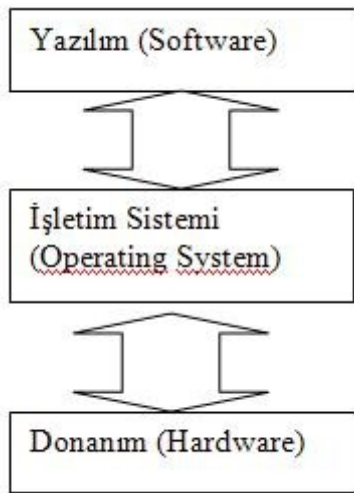
İşte hafıza yönetimi kapsamında

- bir işlemin talep ettiği (demanding address) ile RAM'in gerçek adresi (fiziksel adres, physical address) arasındaki bağı kurulması (ki bu işlem için mantıksal adres (Logical address) ismi verilen bir yöntem kullanılmaktadır. )
- Aynı anda çalışan birden fazla programın hafızayı daha verimli kullanması için hafızada oluşan boşlukların (fragmentation) azaltılması.
- [İşlemler arası haberleşme ve senkronizasyon](#) ihtiyaçları giderilerek her işlemin kendi ardes alanında (own address space) çalışmasının garanti edilmesi

gibi problemler çözülmektedir. Bu problemlerin çözülmesi için [sayfalama \(Paging\)](#) ve [kıtalama \(bölütleme, segmentation\)](#) ismi verilen yöntemler kullanılmaktadır.

### **SORU 47: İşletim Sistemi (Operating System)**

İşletim sisteminin görevi temel olarak donanım (ve diğer sistem kaynakları) ile bilgisayarda çalışan ve bu kaynakları talep eden program (veya processler) arasında ilişki kurmak ve kaynak yönetimini kontrol etmektir. Aşağıdakine benzer bir katmanlı yaklaşım bu anlamda doğru kabul edilebilir:



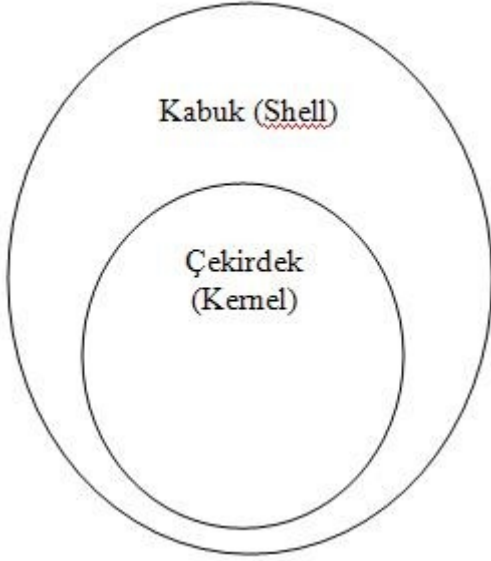
İşletim sisteminin günümüzdeki anlamını anlamak için belkide kelimenin gelişimini ve kısa bir tarihini bilmekte yarar olabilir. İşletim sistemleri olmadığı zamanlarda bilgisayarların yönetilmesinden sorumlu olan işletmenler (operators) bugün işletim sistemlerinin yaptığı işlemleri temel olarak elle yapıyorlardı.

Örneğin kullanıcıların çalıştırmak istedikleri programları varsa, bu işletmene gidip programlarını teslim ediyor sonra işletmenin verdiği tarih ve saatte tekrar giderek programlarının sonuçlarını alıyorlardır. Bu sırada işletmen bilgisayar üzerinde gerekli kaynak ayarlamaları ve sistem ayarlarını (configuration) yaparak programları belirli bir sırada çalıştırıyordu.

Gelişen yazılımlarla işletmenlerin bu yaptıklarını artık programlar yapabiliyor ve gelen program talebini yukarıdakine benzer şekilde çözüyorlar.

İşletim sistemlerinin bugün kü halini almasındaki en önemli tarihi kırılmalardan birisi de bir işletim sisteminde aynı anda birden fazla işlemin çalışabilmesidir (multi processing). Bu gelişme ile birlikte hafızanın ve işlemcinin paylaşılması problemi doğmuştur.

Bir işletim sistemini iki farklı parça olan çekirdek (kernel) ve kabuk (shell) 'den müteşekkil olarak görmek mümkündür. Buna göre çekirdek kısmında işletim sisteminin temel fonksiyonları icra edilirken, kabuk kısmı işletim sisteminin kullanıcı iletişiminden, kullanıcılardan gelen taleplerin çekirdeğe taşınması ve sonuçların kullanıcının talebi doğrultusunda işlenmesi gibi işlemlerden mesuldür.



İşletim sistemleri ile ilgili bilgisayar kavramları.com sitesinde yer alan pek çok yazıyı yine [aynı isimli kategorinin](#) altından okuyabilirsiniz.

#### **SORU 48: Yığın İş ( Batch Job, Batch Process )**

Bilgisayar bilimlerinde, belirli bir zamanda yapılması planlanan yoğunlukla kullanıcı etkileşimi gerektirmeyen işlerin biriktirilmesidir.

Örneğin sistemin yedeğinin alınması için 10 ayrı bilgisayara bağlanılarak her bilgisayardan dosyalar alınıp sunucuya kaydedilecek olsun. Bu işlemi şirketin kapalı olduğu gece 3.00'da yapmak istiyoruz. Bunun için bir yığın iş (batch process) hazırlayarak sistemde saklanır. Beklenen zaman geldiğinde bu iş otomatik olarak çalışır ve sistemin yedeğini alır.

Çeşitli işletim sistemlerinde farklı kullanımları vardır.

Örneğin windows işletim sisteminde bulunan zamanlanmış görevler (scheduled tasks ) vasıtasıyla istenilen zamanda bir işin çalışması mümkündür. Daha eskiden DOS işletim sisteminde .bat uzantılı dosyalar da batch file (yığın dosya) olarak literatürde geçerler. Bu dosyalarda çalışacak olan komutlar arka arkaya sıralanmakta ve sırayla çalıştırılmaktaydı.

Bu dosyalardan en meşhuru da Autoexec.bat dosyasıdır. İşletim sisteminin özel dosyalarından olan bu dosya ilk açılışta okunan ve sistem açılınca yapılması istenen özel işlerin sıralandığı

dosyaydı. Windows işletim sistemiyle birlikte başlangıç (startup) grubunda bulunan işler buna benzetilebilir.

Unix (veya linux) işletim sisteminde ise örneğin “at” komudu ile istenilen zamanda çalışmak üzere bir iş tanımlanabilir.

Örneğin “cp /etc/passwd /yedeck/passwd” komutu etc altındaki passwd dosyasını /yedeck dizinine kopyalar. Bu işlemi otomatik olarak saat 15’te çalıştırmak istersek

```
cp /etc/passwd /yedeck/passwd | at 1500
```

veya benzer şekilde

```
at 1500
```

```
> cp /etc/passwd /yedeck/passwd
```

```
>^D (burada çıkmak için shift+D tuşlarına basıyoruz)
```

şeklinde sisteme 15.00’da çalışacak otomatik bir iş tanımlamış oluruz.

Bir işin belirli periyotlarla sürekli tekrarlanması isteniyorsa cron ismi verilen daemon (sistem arkaplan işi) üzerinde tanımlama yapılabilmektedir.

### **SORU 49: Kilitlenme (Deadlock)**

İşletim sistemlerinde çeşitli sebeplerle iki işlemin birbirini kitlemesi durumudur. Benzer bir örnek güncel hayatta da yaşanabilir.

Örneğin Ali kapıdan geçmek için Ahmet’in önce geçmesini şart koşuyor. Benzer şekilde Ahmet de Ali’nin önce geçmesini şart koşuyor. İki kişide karşısındaki önce geçmezse geçmiyor bu durumda iki si de kapıdan sonsuza kadar geçemez ve bu durum ismi kilitlenme (deadlocks)’tır.

Bu durum işletim sistemlerinde iki [işlem \(process\)](#) arasında sıkça yaşanır. Örneğin aşağıda [semafor \(semaphore\)](#) kullanılarak senkronize edilmiş iki işlemin kod örneğini inceleyelim:

```
Process A
{
    Wait(B);
    ....
    Signal(A);
}
Process B
{
    Wait(A);
    ....
    Signal(B);
}
```



Yukarıdaki örnekte Process A ve Process B isminde iki işlem bulunuyor. A işlemi B'yi, B işlemi ise A'yı bekliyor. Bu durumda ikisi de aynı anda başlayan bu işlemler kilitlenerek sonsuza kadar birbirini beklerler.

Yukarıdaki örnekte olduğu gibi işlemler arası senkronizasyon bir kilitlenme sebebidir. Ancak tek kilitlenme sebebi bu değildir. Diğer sık rastlanan bir kitleme sebebi de kaynaklara erişimde yaşanır.

Kaynak ayrımı (resource allocation) sırasında birden fazla işlem birden fazla kaynağa erişiyorsa şöyle bir durum olabilir:

A işlemi X kaynağını kendisine almış ve Y kaynağı için sıra bekliyor olsun

B işlemi ise Y kaynağını kendisine almış ve X kaynağı için sıra bekliyor olsun

Yukarıdaki bu durumda A işlemi, Y kaynağını almadan X kaynağını bırakmayacaktır, benzer şekilde B işlemi de X kaynağına erişmeden Y kaynağını bırakmayacaktır. Dolayısıyla bu iki işlem birbirini kitlemiş olurlar.

Kilitlenmenin (deadlock) çözülmesi için en kolay ve en etikli yol, programı yazarken kilitlenmeye engel olacak ve kilitlenme içermeyen kod yazmaktır.

### **SORU 50: Kıtlık (Starvation)**

Bir algoritmada sıra bekleyen işlere bir türlü sıra gelmemesi durumudur. Teorik olarak sıradaki her işe birgün sıra gelecektir ancak fiiliyatta bu bir türlü gerçekleşmeyebilir.

Bu tip problemler genelde öncelik tanımlanmış olan algoritmalarda çıkar.

Şöyle bir örnek düşünelim, elimizde uzunlukları 4,5,6 olan işler olsun ve en kısa işi tercih eden bir algoritmamız olsun ([Shortest Job First, SJF](#)). Algoritmamız en kısa olan 4 uzunluğundaki işten başlayacaktır. Ve diyelim ki 4 biter bitmez veya henüz bitmeden uzunluğu 3 olan başka bir iş gelsin. Algoritmamız 5 uzunluğundaki iş yerine daha kısa olan 3 uzunluğundaki işe öncelik verecektir.

Bu süreç böylece sonsuza kadar gidebilir. Yani henüz 5 ve 6 uzunluğundaki işlere sıra gelmeden hep daha kısa olan işlerin gelerek önceliği alması ve 5 ve 6 uzunluğundaki işlere hiçbir zaman sıra gelmemesi söz konusudur.

### **SORU 51: En Kısa İş İlk (Shortest Job First)**

Bir [zamanlama algoritması \(CPU Scheduling\)](#) şekli olan en kısa iş ilk (Shortest job first, SJF veya Shortest Job Next, SJN) algoritmasında o anda elde bulunan işler biritilmek için gereken süreye göre sıralanırlar. En kısa olan işe öncelik verilerek sırayla işler alınır.

Çalışma mantığı olarak [kesintisiz \(non-preemptive\)](#) bir algoritma olan bu yaklaşımda bir iş başladıktan sonra başka bir işin araya girme şansı yoktur.

Basitçe eldeki işlerin en kısasını yaparak performans artışı sağlamaya çalışır. Ancak algoritmanın bir dez avantajı [kıtlık \(starvation\)](#) doğurma riskidir.

Olayı şöyle düşünelim, elimizde uzunlukları 4,5,6 olan işler olsun. Algoritmamız en kısa olan 4 uzunluğundaki işten başlayacaktır. Ve diyelim ki 4 biter bitmez veya henüz bitmeden uzunluğu 3 olan başka bir iş gelsin. Algoritmamız 5 uzunluğundaki iş yerine daha kısa olan 3 uzunluğundaki işe öncelik verecektir.

Bu süreç böylece sonsuza kadar gidebilir. Yani henüz 5 ve 6 uzunluğundaki işlere sıra gelmeden hep daha kısa olan işlerin gelerek önceliği alması ve 5 ve 6 uzunluğundaki işlere hiçbir zaman sıra gelmemesi söz konusudur.

Bu algoritmanın çalışmasını aşağıdaki örnek için inceleyelim:

İşle m	CPU Zamanı
A	10
B	15
C	7

Yukarıda verilen işlerin aynı anda başladığını ve sıra beklediklerini düşünelim. Algoritmamız öncelikle bu işlemleri uzunluk sırasına koyacak ardından en kısa olan iş ile başlayacaktır. En kısa olan işlemimiz C işi ve uzunluğu 7'dir o halde çalışma sıramız aşağıdaki şekilde olur:

Zama n	İşle m	Çalışm a	Kala n
0	C	7	0
7	A	10	0
17	B	15	0

Yukarıda görüldüğü üzere bir işlem bir kere başladıktan sonra başka işlem araya girmemektedir. Ayrıca en kısa iş her zaman en önce çalışmaktadır.

Okuyucu bu tabloyu [round robin](#) ve [ilk gelen çalışır](#) algoritmalarındaki çalışma tabloları ile karşılaştırabilir.

Algoritma, bazı kaynaklarda, SPF (Shortest Process First , En kısa işlem ilk) olarak da geçmektedir. Ne yazık ki algoritma, bir [işlem \(process\)](#) çalıştırılmadan, tam olarak ne kadar zaman alacağını bilinmemesinden dolayı sadece teorik uygulamalara sahiptir. Ayrıca tam başarının mümkün olmadığı [sezgisel \(heuristic\) algoritmalarla](#) birlikte kullanılması da mümkündür.

## **SORU 52: İlk Gelen Çalışır (First Come First Serve, FCFS, FIFO)**

Bilgisayar bilimlerinin çeşitli alanlarında kullanılan bir yaklaşımdır. Bu yaklaşıma göre bir kaynak veya bir ısıraya ilk gelenin ilk önce işini bitirerek çıkması hedeflenir.

Örneğin CPU Scheduling (İşlemci zamanlama) problemi sırasında işlemciye gelen işlemlerin hangi sıra ile çalışacağı bu algoritmaya göre belirlenirse ilk gelen iş bitmeden ikinci iş başlayamaz.

Bir [sıra \(queue\)](#) için benzeri durum düşünülürse sıraya ilk giren ilk çıkar (first in first out, FIFO fifo)

[İşlemci zamanlama \(CPU Scheduling\)](#) algoritması olarak kullanılmasını aşağıdaki örnek üzerinden anlamaya çalışalım:

Örneğin işlemciye aşağıdaki işlemler verilen sıra ile gelmiş olsunlar ve yanlarında verilen zaman kadar işlemcide çalışarak bitecek olsunlar:

İşle m	CPU Zamanı
A	10
B	15
C	7

Yukarıdaki bu işlemlerin çalışma sırası ve zamanları aşağıda verildiği şekildedir:

Zaman	İşlem	Çalışma	Kalan
0	A	10	0
10	B	15	0
25	C	7	0

Görüldüğü üzere [kesintisiz \(nonpreemptive\)](#) bir zamanlama algoritması olan FSFC algoritmasında bir işlem, işlemci tarafından kabul edildikten sonra bitene kadar başka bir işlem araya giremez.

Okuyucu bu çalışma tablosunu round robin algoritmasındaki tablo ile karşılaştırarak farkı daha iyi anlayabilir.

### **SORU 53: Round Robin**

Bir zamanlama (scheduling) algoritmasıdır. Özellikle işletim sistemi tasarımında işlemcinin (CPU) zamanlamasında kullanılan meşhur algoritmalarından birisidir. Bu algoritmaya göre sırası gelen işlem, işlemcide işi bitmese bile belirli bir zaman biriminden sonra (time quadrant) işlemciyi terk etmek zorundadır.

Bu sayed işletim sisteminde kıtlık (Starvation) olma ihtimali kalmaz. Çünkü hiç bir zaman bir işlemin CPU'yu alıp diğer işlemlere sıra gelmesini engellemesi mümkün olmaz.

Örneğin aşağıda süreleri verilen işlemlerin aynı anda bekleme sırasına (Ready queue) geldiğini düşünelim ve round robin algoritmasına göre nasıl bir sıra ile çalıştırılacağına bakalım.

İşlem	CPU Zamanı
A	10
B	15
C	7

Yukarıda verilen bu işlemlerin Round Robin algoritmasına göre CPU’da çalışma süreleri ve sırasıyla CPU’da yer değiştirmeleri (context switch) aşağıda verilmiştir:

Zaman	İşlem	Çalışma	Kalan
0	A	3	7
3	B	3	12
6	C	3	4
9	A	3	4
12	B	3	9
15	C	3	1
18	A	3	1
21	B	3	6
24	C	1	0
25	A	1	0
26	B	3	3
29	B	3	0

#### **SORU 54: Kesmeyen Zamanlama (non-preemptive Scheduling)**

İşletim sistemi tasarımında önemli bir konu olan [işlemci zamanlama algoritmalarına \(CPU scheduling algorithms\)](#) göre de sırası gelen işlem bu bekleme sırasından alınarak [görevlendirici \(dispatcher\)](#) ismi verilen bir işlem tarafından CPU’ya gönderilir.

CPU’da yine işlemci zamanlama algoritmasının izin verdiği kadar (ya bitene ya da belirli bir zaman geçene kadar) çalışan program ya biter ve hafızadan kaldırılır ya da tekrar bekleme sırasına bir sonraki çalışma için yerleştirilir. Yukarıda basitçe birden çok işlemin tek işlemcide nasıl çalıştığı anlatıldı şimdi bu bekleme sırası ile işlemci arasında zamanlama ilişkisini kuran işlemci zamanlama algoritmalarını (cpu scheduling algorithms) tanımaya çalışalım.

Temel olarak 2 grupta incelenebilen bu algoritmalar:

- kesintili algoritmalar (preemptive algorithms)
- kesmeyen algoritmalar (nonpreemptive algorithms)

Bu algoritmalar arasındaki temel fark işlemcinin bir işleme başladıktan sonra o işlemi bitirmeden başka işleme başlayıp başlamamasıdır.

Örneğin işlemci sıradaki işlemi aldı ve bu işlem her ne olursa olsun 3ms sonra bekleme sırasına geri konulup yeni bir işlem alınacak dersek bu algoritmamız kesintili algoritma olmuş olur.

Tersine işlemci bir işi aldıktan sonra ne olursa olsun işi bitirip öyle bir sonraki işlemi alacak (işlem yarım kalmayacak, kesilmeyecek) dersek o zaman algoritmamız kesmeyen algoritmaya örnek olmuş olur.

Kesintili algoritmalara en meşhur örnek Round Robin algoritmasıdır.

İşlemci zamanlama konusunda çok meşhur En kısa iş ilk (shortest job first, sjf), ilk gelen ilk çıkar (first in first out), ilk gelen son çıkar (first in last out) gibi algoritmalar ise kesmeyen algoritmalara örnektir.

### **SORU 55: Kesintili Zamanlama (Preemptive Scheduling)**

İşletim sistemi tasarımında önemli bir konu olan [işlemci zamanlama algoritmalarına \(CPU scheduling algorithms\)](#) göre de sırası gelen işlem bu bekleme sırasından alınarak [görevlendirici \(dispatcher\)](#) ismi verilen bir işlem tarafından CPU'ya gönderilir. CPU'da yine işlemci zamanlama algoritmasının izin verdiği kadar (ya bitene ya da belirli bir zaman geçene kadar) çalışan program ya biter ve hafızadan kaldırılır ya da tekrar bekleme sırasına bir sonraki çalışma için yerleştirilir. Yukarıda basitçe birden çok işlemin tek işlemcide nasıl çalıştığı anlatıldı şimdi bu bekleme sırası ile işlemci arasında zamanlama ilişkisini kuran işlemci zamanlama algoritmalarını (cpu scheduling algorithms) tanımaya çalışalım. Temel olarak 2 grupta incelenebilen bu algoritmalar:

- kesintili algoritmalar (preemptive algorithms)
- kesmeyen algoritmalar (nonpreemptive algorithms)

Bu algoritmalar arasındaki temel fark işlemcinin bir işleme başladıktan sonra o işlemi bitirmeden başka işleme başlayıp başlamamasıdır. Örneğin işlemci sıradaki işlemi aldı ve bu işlem her ne olursa olsun 3ms sonra bekleme sırasına geri konulup yeni bir işlem alınacak dersek bu algoritmamız kesintili algoritma olmuş olur. Tersine işlemci bir işi aldıktan sonra ne olursa olsun işi bitirip öyle bir sonraki işlemi alacak (işlem yarım kalmayacak, kesilmeyecek) dersek o zaman algoritmamız kesmeyen algoritmaya örnek olmuş olur. Kesintili algoritmalara en meşhur örnek Round Robin algoritmasıdır. İşlemci zamanlama konusunda çok meşhur En kısa iş ilk (shortest job first, sjf), ilk gelen ilk çıkar (first in first out), ilk gelen son çıkar (first in last out) gibi algoritmalar ise kesmeyen algoritmalara örnektir.

### **SORU 56: İşlemci Zamanlama (CPU Scheduling)**

İşletim sistemi tasarımında en önemli hususlardan birisi de bir işlemcinin verimli kullanılmasıdır. Şayet işletim sistemi [çok işlemliliği \(multiprocessing\)](#) destekliorsa ve donanımsal olarak tek işlemci (CPU) bulunuyorsa yapılacak tek şey birden fazla işlemi hafızada bekletip sırayla çalıştırmaktır.

CPU çok hızlı olduğu için kullanıcı sanki bütün işlemler aynı anda çalışıyormuş gibi hissedebilir ancak için hakikati aynı anda bir işlemcide tek işlemin çalışıyor olduğudur.

Çalışan her program hafızaya yüklendikten sonra işlemcide çalıştırılmak için sıra beklemeye başlar, bu sıraya [bekleme sırası \(ready queue\)](#) adı verilir.

Bu yazının konusu olan işlemci zamanlama algoritmalarına göre de sırası gelen işlem bu bekleme sırasından alınarak [görevlendirici \(dispatcher\)](#) ismi verilen bir işlem tarafından CPU'ya gönderilir. CPU'da yine işlemci zamanlama algoritmasının izin verdiği kadar (ya bitene ya da belirli bir zaman geçene kadar) çalışan program ya biter ve hafızadan kaldırılır ya da tekrar bekleme sırasına bir sonraki çalışma için yerleştirilir.

Yukarıda basitçe birden çok işlemin tek işlemcide nasıl çalıştığı anlatıldı şimdi bu bekleme sırası ile işlemci arasında zamanlama ilişkisini kuran işlemci zamanlama algoritmalarını (cpu scheduling algorithms) tanımaya çalışalım.

Temel olarak 2 grupta incelenebilen bu algoritmalar:

- [kesintili algoritmalar \(preemptive\)](#)
- [kesmeyen algoritmalar \(nonpreemptive\)](#)

Bu algoritmalar arasındaki temel fark işlemcinin bir işleme başladıktan sonra o işlemi bitirmeden başka işleme başlayıp başlamamasıdır. Örneğin işlemci sıradaki işlemi aldı ve bu işlem her ne olursa olsun 3ms sonra bekleme sırasına geri konulup yeni bir işlem alınacak dersek bu algoritmamız kesintili algoritma olmuş olur.

Tersine işlemci bir işi aldıktan sonra ne olursa olsun işi bitirip öyle bir sonraki işlemi alacak (işlem yarım kalmayacak, kesilmeyecek) dersek o zaman algoritmamız kesmeyen algoritmaya örnek olmuş olur.

Kesintili algoritmalara en meşhur örnek [Round Robin algoritmasıdır](#).

İşlemci zamanlama konusunda çok meşhur [En kısa iş ilk \(shortest job first, sjf\)](#), [ilk gelen ilk çıkar \(first in first out\)](#), [ilk gelen son çıkar \(first in last out\)](#) gibi algoritmalar ise kesmeyen algoritmalara örnektir.

Ayrıca aşağıdaki zamanlama algoritmalarına bakabilirsiniz:

- [Fair Share Scheduling \(Adil Paylaşımlı Zamanlama\)](#)
- [CFS \(Completely Fair Scheduling, Tam Adil Zamanlama\)](#)
- [O\(1\) Zamanlaması \(O\(1\) Scheduling\)](#)

### **SORU 57: Görevlendirici (Dispatcher)**

İşletim sistemi tasarımında kullanılan görevlendirici, işlemci zamanlama algoritmasına (CPU scheduling algorithm) göre beklemekte olan işlemlerden sıradakini alıp işlemciye yollayan programın ismidir.

Buna göre bilgisayarda anlık olarak tek işlem çalışabilir ve bu işlem o anda çalışmakta olan diğer işlemler arasından seçilmiş bir işlemdir. Örneğin bilgisayarda 10 tane program açık olabilir ama CPU'da anlık olarak bir tanesi çalışır. Çalışan programların tamamı bekleme sırasında (ready queue) beklerler. Bir işlemci zamanlama algoritmasına göre bu işlemler sırayla CPU'ya gönderilerek çalıştırılırlar. İşte görevlendirici (dispatcher) bu işlemlerden sırası gelenin [bekleme sırasından \(ready queue\)](#) alınarak işlemciye gönderilmesi işlemini yerine getirir.

### **SORU 58: Bekleme Sırası (Ready Queue)**

İşletim sistemlerinde aynı anda birden fazla işin çalışıyormuş gibi yapılması için kullanılan bir sıradır. Buna göre bilgisayarın işlemcisinde anlık olarak sadece bir tane işlem çalışabilir.

Ama işletim sistemi sanki birden fazla işlem çalışıyormuş gibi gelen yeni çalıştırma taleplerine olumlu cevap verir.

Bunun sonucu olarak onlarca program açılabilir ama problem CPU'da tek işlemin çalışmasıdır. Bu problem de CPU'ya işlemlerin sırasıyla verilerek çalıştırılması şeklinde çözülür. Sonuçta işlemci oldukça hızlıdır ve yeterince hızlı çalışırsa bütün işlemleri sırayla çözerek kullanıcının birden fazla program talebini karşılayabilir.

Bu sırada çalışan programların işlemciye yollanmalarının bir sıraya sokulması gerekir. Örneğin 10 programdan bir tanesi işlemciyi meşgul ederken diğerleri çalışamaz ve şayet bu işgal çok uzun sürerse diğer 9 program çalışmadan bekler. dolayısıyla bu 10 programın hangisinin işlemcide çalışacağını belirleyen işlemci zamanlama algoritmalarına ihtiyaç duyulur. Bu algoritmalar da anlık olarak işlemcide çalışmayan ama [hafızaya](#) yüklenmiş ve sanki çalışıyormuş gibi olan programları bir veri yapısında tutmaya ihtiyaç duyarlar.

İşte CPU'da çalışmayan ama hafızaya yüklenmiş ve sanki çalışıyormuş gibi kullanıcıya gösterilen ve işletim sistemi açısından çalışan bu programların CPU için sıra bekledikleri yerin adı bekleme sırası (ready queue)'dur.

### **SORU 59: Çevirici (Assembler)**

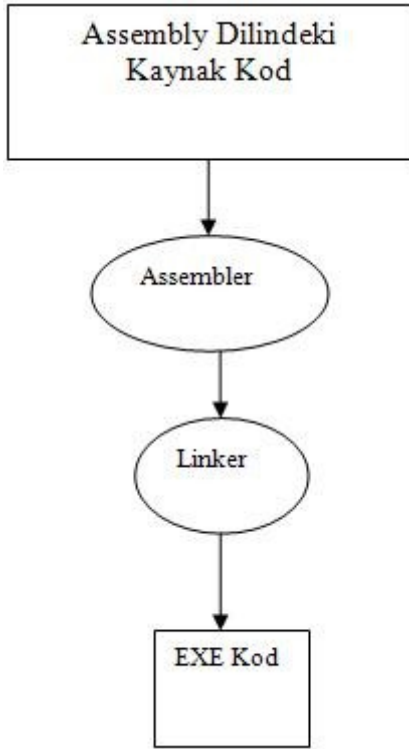
Bilgisayar bilimlerinde iki farklı kavram için assembler kelimesi kullanılmaktadır. Birincisi Assembly dili adı verilen ve makine diline (machine language) çok yakın düşük seviyeli (low level language) için kullanılan ve nesne kodunu (object code) makine koduna (machine code) çeviren dildir. İkincisi ise birleştirmek, monte etmek anlamında örneğin nesne yönelimli dillerde nesnelerin birleştirilmesi monte edilip büyük parçaların çıkarılması anlamında kullanılmaktadır.

Assembly dili, CPU üzerinde programcıya tanınmış olan yönergeleri (instructions) alarak bunları işlemcinin doğrudan çalıştırdığı makine kodları haline getirir. CPU üzerindeki bu yönergeler genelde üretici tarafından belirlenir ve tamamen teknoloji bağımlıdır. Diğer bir deyişle aynı kod farklı yapıdaki işlemciler üzerinde çalışmaz. Hatta çoğu zaman aynı mimarinin farklı nesilleri arasında bile sorun yaşanır.

Assembly dili macro destekleyen ve desteklemeyen olarak ikiye ayrılır. Tarihi gelişim sürecinde ihtiyaç üzerine dilde macro yazmak ve yazılan bu macroları tekrar tekrar kullanmak imkanı doğmuştur. Bu tip assembly dilini makine diline çeviren çeviricilere de macroassembler adı verilir.

Assembly dilleri temel olarak fonksiyon(function) veya prosedür (procedure) desteği barındırmazlar. Hatta döngü ve koşul operatörleri de yok denilebilir. Bunun yerine programın içerisinde istenen bir satıra (adrese) gidilmesini sağlayan GOTO komutu kullanılır.

[Yapısal programlama \(structured programming\)](#) dokusuna ters olan bu yaklaşım dilin performans kaybı ve düşük seviye olmasından kaynaklanmaktadır.



Yukarıdaki şekilde assembly kaynak kodunda verilen bir girişin assembler'dan geçerek [bağlayıcı \(linker\)](#) marifetiyle çalışabilir (Exe code) haline gelişi tasvir edilmiştir.

Ayrıca çeviriciler (assemblers) geçiş (pass) sayısına göre gruplanabilir:

- [Tek geçişli çeviriciler](#)
- [iki geçişli çeviriciler](#)
- [çok geçişli çeviriciler](#)

### **SORU 60: Dinamik Bağlantı Kütüphaneleri (Dynamic Link Library (.dll))**

Microsoft tarafından windows işletim sistemi üzerinde kullanıma açılan ve çalışma sırasında bağlanmaya izin verilen kütüphane yaklaşımıdır. Basitçe Linux ortamlarındaki .o (object file (nesne dosyası)) benzetilebilir. Bu dosyaların amacı birden fazla program tarafından kullanılan kütüphaneleri içermeleri ve her programın gerekli oldukça ilgili kütüphaneden dosyayı okumasıdır.

Windows öncesi microsoft işletim sisteminde (windows 3.x ve DOS gibi) kullanılan yaklaşımda program tek bir dosyadan oluşmaktaydı. Dolayısıyla programın kullandığı her kütüphane o programa özgü olarak bulunduruluyordu. İşletim sisteminin programın çalıştırılması sırasında herhangi bir bağlama yapması söz konusu değildi.

Windows 95 ile birlikte 32bit sisteme geçen windows, Unix ve dolayısıyla Linux dünyasında da rahatlıkla kullanılabilen object code kavramı windows dünyasına kazandırılmış oldu. Buradaki amaç ortak kullanılan fonksiyonları bir dosyada bulundurmak ve her programın çalıştırılması sırasında önce [bağlayıcı \(linker\)](#) tarafından bu dosyaların bağlanması ve ardından programın [yükleyici \(loader\)](#) tarafından hafızaya yüklenerek çalıştırılmasıydı.



Bu sayede aynı ortak fonksiyonu kullanan programların herbirisi için kod tekrarı olmayacağı gibi güncellemeler de tek elden takip edilebilecekti. Programcılığın modüler yaklaşımının bir ürünü olan .dll dosyaları günümüzde windows işletim sisteminde yaygın olarak kullanılmaktadır. Bu dosyalar derlenmiş (compiled) kod olduğu için gerekli görülmesi durumunda şifrelenebilmekte ve orjinal kodu koruma altında tutabilmektedir.

Gelişen web teknolojileri ile birlikte .dll dosyalarını windowsun web sunucusu (web server) olan IIS (internet information server) ile de uyumlu hale getiren microsoft, şu anda geliştirilen web projelerini .dll olarak sunucularda barındırıp internet kullanıcılarının erişimine açmaktadır.

.dll dosyalarının bir diğer avantajı da [hafızaya](#) bir kere yüklenen dosyaların birden fazla program tarafından paylaşılması bu sayede hafızanın verimli kullanılmasıdır.

### **SORU 61: Yerleştirme Algoritmaları (Fitting Algorithms)**

Bilgisayar bilimlerinde kısıtlı bir alanın verimli kullanılması için geliştirilmiş algoritmalar vardır. Örneğin sınırlı bir [hafıza \( RAM \)](#) içerisine en verimli şekilde programları yerleştirmek, işletim sistemleri için bir problemdir. Benzer problemlerle gerçek hayatta da sıkça karşılaşılmaktadır. Örneğin bir deponun verimli kullanılması veya bir kamyonun verimli yüklenmesi veya haftalık bir ders programına derslerin verimli yerleştirilmesi gibi.

Bu problemlerin çözümü için genelde bin packing ismi verilen bir algoritma kullanılır. Bu yazının amacı bin packing algoritması da dahil olmak üzere genel olarak izlenen yerleştirme stratejilerini açıklamaktır.

Temel olarak bir yerleştirme işleminde şu 3 yöntemden birisi izlenebilir

- İlk bulunan yere yerleştirme ( First fitting )
- En iyi yerleştirme ( best fitting )
- En kötü yerleştirme ( worst fitting )

Bu yöntemler isimlerinden de anlaşılacağı üzere kısaca:

Yerleştirmek istediğimiz şeyi ( hafızaya yerleştirilecek bir program, depodaki bir kutu gibi ) ilk bulduğumuz boş yere yerleştiriyorsak buna first fitting, en az boş yer kalacak şekilde bir yer arıyor ve yerleştiriyorsak buna best fitting şayet yerleştirdikten sonra en fazla boş yer kalacak yere yerleştiriyorsak buna da worst fitting ismi verilir.

### **SORU 62: Rastgele Erişilebilir Bellek (Random Access Memory , RAM)**

Bilgisayarların en önemli parçalarından birisidir. Özel bazı bilgisayarları dikkate alamazsak gündelik hayatta karşılaşılabilecek hemen her bilgisayarda bulunması gereken bir donanım parçasıdır. Birincil bellek (primary memory) ismi de verilen rasgele erişilebilir belleğin temel fonksiyonu işlemcinin (Merkezi işlem birimi ( Central processing unit (CPU)) program çalıştırırken geçici olarak verileri sakladığı ve sırası geldikçe bu verileri kullandığı alan olmasıdır.

Buna göre bir işlemci (CPU) çok basit toplama çıkarma seviyesinde işlemler yaparak programları çalıştırır. Komplike bir programın ise bu nevi basit işlemlere indirgenmesi

mümkün olsa da bu indirgenme sonucunda çok sayıda işlemin yapılması gerekir. İşte CPU anlık olarak bu işlemlerden sadece birisini yapabilir. Geri kalan işlemler ise sırasını beklemek ve bu sırada bir yerde saklanmak zorundadır.

Kısaca bilgisayarda çalışan her programın CPU'da çalışmak için bekletildiği ve o ana kadar çalışması sonucunda biriken verilerinin saklandığı hafıza ünitesidir.

Disklere göre daha hızlı (ve dolayısıyla daha pahalı) olan veri ünitelerinin içinde bulunan bilgiler elektrik kesilmesi sonucu kaybolur. Bu yüzden [ikincil hafıza \(secondary memory\)](#) ismi verilen [sabit disklerle \(hard discs\)](#) her zaman ihtiyaç vardır ve bilgisayarın kapanıp açılması sonucu verilerin korunması için ramdeki bilgiler diske yazılabilir.

Çok uzun süredir RAM'de bulunan verilerin elektrik kesintisinden sonra da hafızada kalması için üzerinde çalışılan ve çok uzun süredir bir ürün olarak alınabilen non-volatile (geçici olmayan) hafızalar ne yazık ki fiyatları yüzünden son kullanıcının alabileceği seviyelerde henüz değildir.

### **SORU 63: C ve Komut Satırı (C Console Parameters)**

C dilinde öncelikli olarak çalıştırılacak olan fonksiyon main fonksiyonudur. Main fonksiyonunun prototipi aşağıdaki şekildedir:

```
int main(int argc, char *argv[])
```

yukarıda görüldüğü üzere main fonksiyonu bir integer (tam sayı) döndürmektedir. Bu C90 standardına göre belirlenmiştir ancak çoğu [derleyici \(compiler\)](#) main fonksiyonunun void döndürmesine de izin verir.

Bu dönen değer C programımızı çalıştıran işletim sistemine dönmektedir. Örneğin başka bir program altından exec fonksiyonu ile çalıştırılırsa bu programa parametre olarak döner.

main fonksiyonunun parametresi olan argc ise fonksiyona konsoldan kaç tane parametre verildiğini sayar.

Örneğin copy komudunu yapan bir program yazdığımızı kabul edelim ve a.txt dosyasını b.txt dosyası olarak kopyalamak isteyelim. Bu durumda aşağıdakine benzer bir komut yazacağız:

```
copy a.txt b.txt
```

bu kullanımda C için toplam 3 argüman (parametre) vardır. bunlar copy, a.txt ve b.txt parametreleridir. İşte main fonksiyonunun parametresi olan argc bu sayıyı tutmaktadır. Hemen ardından gelen argv (array of char pointers) tipinden de anlaşılacağı üzere bu parametrelerin her birini tutar.

### **SORU 64: Sıralama Algoritmaları (Sorting Algorithms)**

Bilgisayar bilimlerinde verilmiş olan bir grup sayının küçükten büyüğe (veya tersi) sıralanması işlemini yapan algoritmalara verilen isimdir. Örneğin aşağıdaki düzensiz sayıları ele alalım:

5 9 2 3 7 11 -4 6

Bu sayıların sıralanmış hali

-4 2 3 5 6 7 11

olacaktır. Bu sıralama işlemini yapmanın çok farklı yolları vardır ancak bilgisayar mühendisliğinin temel olarak üzerine oturduğu iki performans kriteri buradaki sonuçları değerlendirmede önemli rol oynar.

- Hafıza verimliliği (memory efficiency)
- Zaman verimliliği (Time efficiency)

Temel olarak algoritma analizindeki iki önemli kriter bunlardır. Bir algoritmanın hızlı çalışması demek daha çok hafızaya ihtiyaç duyması demektir. Ters durumda da bir algoritmanın daha az yere ihtiyaç duyması daha yavaş çalışması demektir. Ancak bir algoritma hem zaman hem de hafıza olarak verimliyse bu durumda diğer algoritmalarından başarılı sayılabilir.

Genellikle verinin hafızada saklanması sırasında veriyi tutan bir bellekleyici özelliğinin olması istenir. Veritabanı teorisinde birincil anahtar (primary key) ismi de verilen bu özellik kullanılarak hafızada bulunan veriye erişilebilir. Bu erişim sırasında şayet bellekleyici alan sıralı ise erişimin logaritmik zamanda olması mümkündür. Şayet veri sıralı değilse erişim süresi doğrusal (linear) zamanda olmaktadır.

Aşağıda bazı sıralama algoritmaları verilmiştir:

- [Seçerek Sıralama \(Selection Sort\)](#)
- [Hızlı Sıralama Algoritması \(Quick Sort Algorithm\)](#)
- [Birleştirme Sıralaması \(Merge Sort\)](#)
- [Yığınlama Sıralaması \(Heap Sort\)](#)
- [Sayarak Sıralama \(Counting Sort\)](#)
- [Kabarcık Sıralaması \(Baloncuk sıralaması, Bubble Sort\)](#)
- [Taban Sıralaması \(Radix Sort\)](#)
- [Sokma Sıralaması \( Insertion Sort\)](#)
- [Sallama Sıralaması \(Shaker Sort\)](#)
- [Kabuk Sıralaması \(Shell Sort\)](#)
- [Rastgele Sıralama \(Bogo Sort\)](#)
- [Şanslı Sıralama \(Lucky Sort\)](#)
- [Serseri Sıralaması \(Stooge Sort\)](#)
- [Şimşek Sıralaması \(Flahs Sort, Bora Sıralaması\)](#)
- [Tarak Sıralaması \(Comb Sort\)](#)
- [Gnome Sıralaması \(Gnome Sort\)](#)
- [Permütasyon Sıralaması \(Permutation Sort\)](#)
- [Strand Sort \(İplik Sıralaması\)](#)

Yukarıda verilen veya herhangi başka bir sıralama algoritması genelde küçükten büyüğe doğru (ascending) sıralama yapar. Ancak bunun tam tersine çevirmek (descending) genelde algoritma için oldukça basittir. Yapılması gereken çoğu zaman sadece kontrol işleminin yönünü değiştirmektir.

Ayrıca sıralama işleminin yapılması sırasında hafızanın kullanımına göre de sıralama algoritmaları :

- [Harici Sıralama \(External Sort\)](#)
- Dahili Sıralama (Internal Sort)

şeklinde iki grupta incelenebilir.

Algoritmaların karşılaştırılması için aşağıdaki tablo hazırlanmıştır:

Algoritma	İngilizces i	Algoritma Analizi			Kararlılı	Yöntem	Açıklama
		En İyi	Ortalama	En Kötü			
Seçerek Sıralama	Selection Sort	$n^2$	$n^2$	$n^2$	Kararsız	Seçerek	
Hızlı Sıralama	Quick Sort	$n \log(n)$	$n \log(n)$	$n^2$	Kararsız	Parçala Fethet	
Birleştirme Sıralaması	Merge Srot	$n \log(n)$	$n \log(n)$	$n \log(n)$	Kararlı	Parçala Fethet	
Yığınlama Sıralaması	Heap Sort	$n \log(n)$	$n \log(n)$	$n \log(n)$	Kararsız	Seçerek	
Sayarak Sıralama	Counting Sort	$n + 2^k$	$n + 2^k$	$n + 2^k$	Kararsız	Sayarak	k ikinci dizinin boyutu.
Kabarcık Sıralaması	Bubble Sort	$n$	$n^2$	$n^2$	Kararlı	Yer Değiştirme	
Kokteyl Sıralması	Coctail Sort	$n$	$n^2$	$n^2$	Kararlı	Yer Değiştirme	Çift Yönlü kabarcık sıralaması (bidirectional bubble sort) olarak da bilinir ve dizinin iki ucundan işleyen kabarcık sıralamasıdır.
Taban Sıralaması	Radix Sort	$n(k/t)$	$n(k/t)$	$n(k/t)$	Kararlı	Gruplama / Sayma	k, en büyük eleman değeri, t ise tabandır
Sokma Sıralaması	Insertion Sort	$n$	$d+n$	$n^2$	Kararlı	Sokma	d yer değiştirme sayısıdır ve $n^2$

							cinsindendir
Sallama Sıralaması	Shaker Sort	$n^2$	$n^2$	$n^2$	Kararsız	Seçme	Çift yönlü seçme sıralaması (bidirectional selection sort) olarak da bilinir.
Kabuk Sıralaması	Shell Sort	$n^{3/2}$	$n^{3/2}$	$n^{3/2}$	Kararsız	Sokma	
Rastgele Sıralama	Bogo Sort	1	$n \cdot n!$	sonsuz	Kararsız	Rastgele	Algoritma olduğu tartışmalıdır. Knuth karıştırması (knuth shuffle) süresinde sonuca ulaşması beklenir.
Bozo Sıralaması	Bozo Sort	1	$n!$	sonsuz	Kararsız	Rastgele	Rastgele sıralamanın özel bir halidir. Rastgele olarak diziyi karıştırdıktan sonra, dizi sıralanmamışsa, yine rastgele iki sayının yeri değiştirilip denenir.
Goro Sıralaması	Goro Sort	$2^{(\log(d)/\log(2))}$	$2^{(\log(d)/\log(2))}$	$2^{(\log(d)/\log(2))}$	Kararsız	Rastgele	2011 Google kod yarışı (google code jam) sırasında ortaya çıkmıştır. Sıralanana kadar her alt küme permüte edilir. Buradaki performans değeri ispatlanmamıştır ve d dereceyi ifade eder.
Şanslı	Lucky	1	1	1	Kararsı	Rastgele	Algoritma

Sıralama	Sort				z		olarak kabul edilmemelidir.
Serseri Sıralama	Stooge Sort	$n^3$	$n^3$	$n^3$	Kararsız z	Yer değiştirme	e, doğal logaritma sayısıdır (2,71)
Şimşek Sıralaması	Partial Flash Sort	n	$n + d$	$n + d$	Kararsız z	Yer Değiştirme	d, kullanılan ikinci algoritmanın performansısıdır, bu algoritma bu listedekilerden herhangi birisi olabilir.
Permütasyon Sıralaması	Perm Sort	n	$n \cdot n!$	$n \cdot n!$	Kararsız	Yer Değiştirme	
Bazı Yegane Sıralaması	Several Unique Sort	n	$n^2$	$n^2$	Kararsız	Yer Değiştirme	Bir bilgisayar programı tarafından bulunmuştur.
Tarak Sıralaması	Comb Sort	$n \log(n)$	$n \log(n)$	$n \log(n)$	Kararsız	Yer Değiştirme	Kabarcık ve hızlı sıralama algoritmalarının birleşimi şeklinde düşünülebilir

Yukarıdaki yazıda geçen kararlılık kolonu ile, bir algoritmanın bitiş kontrolüne dayanmaktadır. Örneğin sıralı bir dizi verilse bile sıralama işlemi yapmaya çalışır mı?

Yukarıdaki tabloda bulunan algoritma analizi bilgisi için aşağıdaki yazıları okumanızda yarar olabilir:

### **SORU 65: Bağlayıcı (linker)**

Bir [derleyici](#) tarafından üretilmiş olan kodları bağlayarak işletim sisteminin çalıştırabileceği tek bir kod üreten programdır.

Günümüzde hızla gelişen programlama ihtiyaçları sonucunda programlamada modüler yaklaşıma geçilmiştir. Buna göre büyük bir yazılım küçük alt parçalara bölünmekte ve her parça ayrı ayrı işlenerek büyük program elde edilmektedir. Yapısal programlamanın da çıkış sebeplerinden birisi olan bu yaklaşıma göre dillerde fonksiyon desteği gelmiş ve değişik parametrelere göre aynı kodun farklı sonuçlar üretmesi sağlanmıştır. Daha sonradan gelişen nesne yönelimli programlama bu konuda bir sonraki nesil olarak kabul edilebilir. Nesne yönelimli programlamada, programlar nesnelere bölünerek farklı bir yaklaşım izlenmiştir.

Bu yaklaşımların [derleyici](#)lere yansması da uzun sürmemiş, daha yapısal programlamanın ilk geliştiği günlerde [derleyici](#)ler de farklı kütüphaneler ve bu kütüphaneleri birleştirmeye yarayan harici programlar kullanmaya başlamışlardır.

Kodun birden fazla parçaya bölünmesi ve her parçanın ayrı ayrı üretilmesi durumunda bu parçaların birleştirilmesi ve tek bir program halinde üretilmesinden sorumlu olan programlara bağlayıcı (linker) adı verilmektedir.

#### **SORU 66: sunucu (server)**

Bir hizmet yada kaynağı arz eden bilgisayara verilen isimdir. Buna göre hizmet veya kaynağı sunan bir sunucu bulunmakta ve [istemciler](#) bu sunucuya bağlanarak bu hizmetten faydalanmaktadır. Bu bağlantı sistemine istemci /sunucu ( arz /talep, client/server) ismi verilmektedir.

Öneğin internete bağlı bir kullanıcı, internet üzerinde bir web sayfasını görüntülemek istesin. Bu durumda sayfaya bağlanmakta ve sayfanın içeriğini kendi bilgisayarına çekerek internet görüntüleyicisi ile göstermektedir. Bu görüntüleme esnasında talep edilen web sayfası için [istemci](#) taraf ilgili sunucuya bağlanmakta ve sayfayı talep etmektedir. Sunucu ise bu talebe cevap olarak sayfayı [istemci](#)ye yollamaktadır. Görüldüğü üzere talep eden taraf [istemci](#) arz eden taraf ise sunucudur. Sonuçta bir sunucuya çok sayıda [istemci](#) bağlanarak taleplerde bulunabilmekte ve sunucu da bu talepleri cevaplamaktadır. Dolayısıyla [istemci](#)/sunucu mimarisi çoktan teke bir bağlantı şeklidir. Bu bağlantı şekli aynı zamanda özdeş olmayan uçlar arasında ([istemci](#)/sunucu) yapıldığı için asimetrik olarak da kabul edilir.

#### **SORU 67: istemci (client, talebe)**

bir hizmet yada kaynağı talep eden istekte bulunan taraftır. Buna göre hizmet veya kaynağı sunan bir sunucu bulunmakta ve istemciler bu sunucuya bağlanarak bu hizmetten faydalanmaktadır. Bu bağlantı sistemine istemci /sunucu ( arz /talep, client/server) ismi verilmektedir.

Öneğin internete bağlı bir kullanıcı, internet üzerinde bir web sayfasını görüntülemek istesin. Bu durumda sayfaya bağlanmakta ve sayfanın içeriğini kendi bilgisayarına çekerek internet görüntüleyicisi ile göstermektedir. Bu görüntüleme esnasında talep edilen web sayfası için istemci taraf ilgili sunucuya bağlanmakta ve sayfayı talep etmektedir. Sunucu ise bu talebe cevap olarak sayfayı istemciye yollamaktadır. Görüldüğü üzere talep eden taraf istemci arz eden taraf ise sunucudur. Sonuçta bir sunucuya çok sayıda istemci bağlanarak taleplerde bulunabilmekte ve sunucu da bu talepleri cevaplamaktadır. Dolayısıyla istemci/sunucu mimarisi çoktan teke bir bağlantı şeklidir. Bu bağlantı şekli aynı zamanda özdeş olmayan uçlar arasında (istemci/sunucu) yapıldığı için asimetrik olarak da kabul edilir.

### SORU 68: bit (ikil)

Bilgisayar dünyasında ikili tabandaki (binary) tek haneli bir sayıyı ifade eder. Yani bir bit değeri 1 veya 0 olabilir. Bu aslında elektronik sinyali olarak yüksek (1) veya düşük (0) gerilimde akım demektir.

bir bit, 1 veya 0 değeri alabildiğine göre her bit değerinin 2 farklı değer alması mümkündür. Bu durumda örneğin iki sistemde yazılmış 8 haneli bir sayı (8 bitlik bir sayı) 2 üzeri 8 farklı değer = 256 farklı değer alabilir. Örneğin 11010010 sayısı, 8 bitlik bir sayıdır.

8 bitlik sayılara bilgisayar dünyasında kısaca byte adı da verilmektedir.

Bit değerleri için kullanılan birimler aşağıda listelenmiştir:

Metrik Gösterim		İkili (binary) Gösterim	
İsim	Metrik Değeri	İsim	Değeri
<u>kilobit</u> (kbit)	$10^3$	<u>kibibit</u> (Kibit)	$2^{10}$
<u>megabit</u> (Mbit)	$10^6$	<u>mebibit</u> (Mibit)	$2^{20}$
<u>gigabit</u> (Gbit)	$10^9$	<u>gibibit</u> (Gibit)	$2^{30}$
<u>terabit</u> (Tbit)	$10^{12}$	<u>tebibit</u> (Tibit)	$2^{40}$
<u>petabit</u> (Pbit)	$10^{15}$	<u>pebibit</u> (Pibit)	$2^{50}$
<u>exabit</u> (Ebit)	$10^{18}$	<u>exbibit</u> (Eibit)	$2^{60}$
<u>zettabit</u> (Zbit)	$10^{21}$	<u>zebibit</u> (Zibit)	$2^{70}$
<u>yottabit</u> (Ybit)	$10^{24}$	<u>yobibit</u> (Yibit)	$2^{80}$

### SORU 69: Çok işlemlik (Multi processing)

Bir bilgisayarda aynı anda birden fazla işlemin(process) çalışmasına verilen isimdir. İşletim sistemlerinin gelişimi süreci incelendiğinde ilkel işletim sistemlerinde bu özellik bulunmuyordur. tek işlem (uniprocess) çalıştıran işletim sistemlerinde hafıza yönetimi bir işlemin kontrolünde yapıldığı için işletim sisteminin işlem üzerinde bir kontrolü bulunmuyor ve bir hafıza yönetimi yapılması gerekmiyordu. Benzer şekilde işlem yönetimi (process management) konusunda da oldukça rahatlık sağlayan tek işlemli işletim sistemlerinin en meşhur örneklerinden birisi de DOS (Disk Operating System)'dır. Bu işletim sisteminde aynı anda tek işlem çalışmaktaydı.

Zamanla gelişen birden fazla programın (dolayısıyla işlemin) aynı anda çalışması ihtiyacı ile hafıza yönetiminde ve MİB (merkezi işlem birimi (CPU Central processing unit)) planlaması üzerinde değişiklikler yapıldı. Yeni işletim sistemlerinde her işlem kendi adres alanında diğer işlemlerden habersiz bir şekilde çalışmaktadır. Çalışan her işlemin kendi adres bilgisine sahip olabilmesi için



mantıksal (logical) adres ataması yapılmaktadır. Bu sayede çalışan programlar her seferinde hafızanın farklı alanlarına yüklenseler bile aynı mantıksal alana erişebilmektedirler.

### SORU 70: İşlem (Process)

Bir işletim sistemi üzerinde herhangi bir dil ile kodlanmış ve bir compiler (derleyici) ile derlenmiş ve daha sonra [hafızaya](#) yüklenerek işlemcide çalıştırılan programlara verilen isimdir.

Genel anlamda her program bir process olarak düşünülebilir, ancak bir programın birden fazla processi olabileceği gibi her process, yeni başka processlerde üretebilir (fork) . İşletim sisteminin tasarımına göre değişmekle birlikte [işlemler \(process\)](#) kendi adres alanında (own adress space) çalışırlar ve hafıza koruması (memory protection) uygulanır. Bu sayede bir işlemin, başka işlemlerin bilgisine erişmesi engellenmiştir.

İşlemler arası iletişim (Inter process communication (IPC)), aynı bilgisayarda çalışan farklı programların haberleşmesini hedef alır, bu durum ağ üzerinde birbiriyle haberleşen bilgisayarlara benzetilebilir.

### SORU 71: İşlemler arası iletişim (Inter process communication (IPC))

Bir bilgisayarda çalışan birden fazla [işlemin \(process\)](#) bir biri ile haberleşmesini hedefleyen teknolojidir. Hız açısından düşünüldüğünde en hızlı iletişim yöntemi [hafıza \(RAM\)](#) üzerinde veri paylaşımıdır.

Dolayısıyla bir işlemin hafızaya yazdığı bilgi başka bir işlem tarafından okunarak bu iletişim sağlanmış olur.

Günümüz işletim sistemlerinde bu paylaşım işlemi aşağıdaki 4 işlemten birisi ile yapılmaktadır:

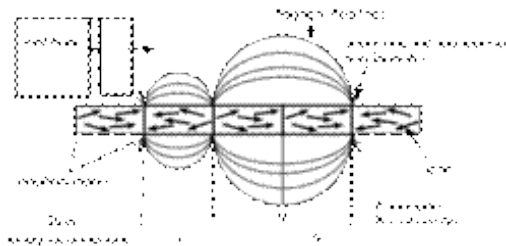
mesajlaşma	(message	passing)
senkronizasyon		(synchronization)
paylaşılmış hafıza	(shared	memory)
uzak prosedür çağırımı (remote procedure calls)		

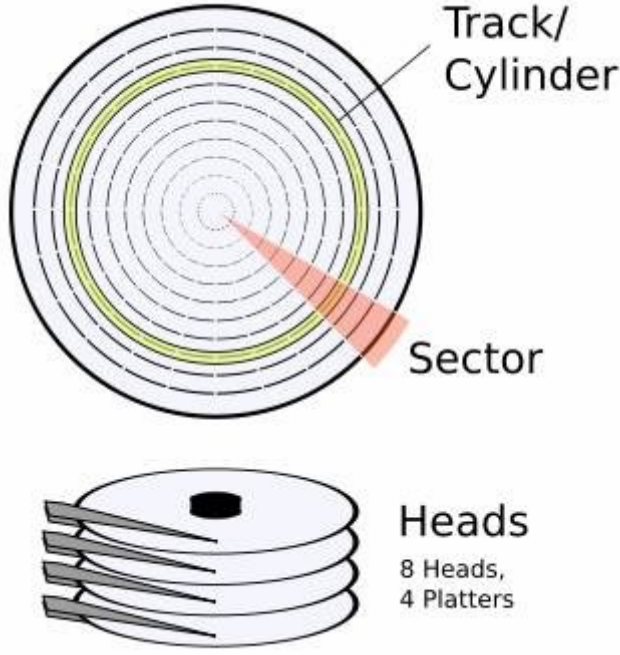
Yukarıdaki bu yöntemlerin hepsinin ortak yanı hafıza temelli iletişim yöntemleri olmalarıdır.

### SORU 72: Disk Yönetimi (Disk Management)

Bu yazının amacı bilgisayarın önemli donanım parçalarından birisi olan sabit diskin (hard disk) daha iyi anlaşılmasını sağlamaktır.

Aşağıdaki resimde klasik bir sabit diskin parçalarının isimleri gösterilmiştir:





Temel kavramlar:

Kafa (head) : Güncel sabit diskler genelde birden fazla disk içermektedirler. Her diskin üzerine işlem yapan (okuyan, yazan veya hareket eden) manyetik uca kafa denilir. Güncel disklerin iki yüzü de işlenebilir olduğu için bir disk için 2 kafa kullanılır.

İz (track) Disk üzerinde kafanın hareket ettiği ve manyetik olarak işlenebilen alanlardır.

Silindir (cylinder): Diskler daire şeklinde olduğu için, disk üzerindeki izler çembere benzetilebilir. Birden fazla disk olduğu için ve her disk üzerinde aynı koordinatlara çember olduğu düşünülürse, bu çemberlerin birleştirilmesi bir silindir görüntüsünde olur. Dolayısıyla silindir birden fazla diskte aynı yere düşen izler demektir.

Sektör (sector) Disk üzerindeki izlerin bölündüğü alt parçalardır. Bu parçalar sayesinde izler üzerinde bilgi gruplanabilmektedir. Yuvarlak bir pasta (disk) üzerindeki dilimler gibi düşünülebilir.

Disk üzerinde bilgilerin durması manyetik alan yönüne göre olur. Örneğin aşağıdaki resimde sağa doğru olan yükleme 1, sola doğru olan yükleme 0 olarak kabul edilmiştir:

Blok (Block) Sektör ve izlerin kesişimidir. Disk üzerindeki en küçük birimdir.

Yukarıdaki bilgiler ışığında aşağıdaki formüller elde edilebilir:

Disk bir yüzündeki blok sayısı: diskteki silindir sayısı \* diskteki sektör sayısı

Disk başına blok sayısı: Diskin bir yüzündeki blok sayısı \* diskin yüz sayısı

Disk başına blok sayısı: Diskin bir yüzündeki blok sayısı \* disk başına kafa sayısı

Disk başına blok sayısı: diskteki silindir sayısı \* diskteki sektör sayısı \* disk başına kafa sayısı

Toplam blok sayısı: Silindir sayısı \* disk başına kafa sayısı \* sektör sayısı \* disk sayısı

Toplam blok sayısı: silindir \* kafa \* sektör

Örnek: 100 silindiri 2 kafası ve 20 sektörü olan bir diskin kaç bloğu vardır?

Toplam blok sayısı:  $100 * 2 * 20$

Toplam blok sayısı: 4000 olarak bulunur.

Şayet bir blok boyutu 512 byte olarak kabul edilirse diskin kapasitesi:

$4000 * 512 = 2\text{MB}$  olarak bulunur.

LBA ( logical block addressing) mantıksal blok adreslemesi:

Disk üzerindeki blokların adreslenmesi için kullanılır. Diskte bulunan ilk blok lba0 , ikinci blok lba1 şeklinde adreslenir.

CHS (cylinder head sector) Silindir Kafa Sektör

Disk üzerinde blokların adreslenmesi için kullanılan ikinci yöntemdir. Diskte bulunan ilk blok 0 0 0 şeklinde ikinci blok 0 0 1 şeklinde adreslenir. 3 sayı tutar, silindir, kafa ve sektör bilgisi ayrı ayrı durur.

Örnek dönüşüm tablosu:

LBA Değeri

CHS Satırı

0

0, 0, 1

1

0, 0, 2

2

0, 0, 3

62

0, 0, 63

63

0, 1, 1

64

0, 1, 2

65

0, 1, 3

125

0, 1, 63

126

0, 2, 1

127

0, 2, 2

188

0, 2, 63

189

0, 3, 1

190

0, 3, 2

16,063

0, 254, 62

16,064

0, 254, 63

16,065

1, 0, 1

16,066

1, 0, 2

16,127

1, 0, 63

16,128

1, 1, 1

16,450,497

1023, 254, 1

16,450,558

1023, 254, 62

16,450,559

1023, 254, 63

Yukarıdaki tabloya göre bir disk CHS kullanılarak en fazla:

$1024 * 255 * 64 * 512$  ( byte) = 8.4GB olabilir.

CHS sistemi MS-DOS ve NT4.0 versiyonu bilgisayarlarda kullanılan bir sistemdi ancak yukarıda da görüldüğü üzere adreslemedeki yetersizlik yüzünden ilerletilmesi gereken bir sistemdir.

ATA sürücülerinde kullanılan yeni teknoloji sayesinde ECHS (enhanced CHS) silindir sayısı arttırılabilmektedir:

LBA Değeri

CHS Satırı

0

0, 0, 1

1

0, 0, 2

2

0, 0, 3

62

0, 0, 63

945

0, 15, 1

1007

0, 15, 63

1008

1, 0, 1

1070

1, 0, 63

1071

1, 1, 1

1133

1, 1, 63

1134

1, 2, 1

2015

1, 15, 63

2016

2, 0, 1

16,127

15, 15, 63

16,128

16, 0, 1

32,255

31, 15, 63

32,256

32, 0, 1

16,450,559

16319, 15, 63

16,514,063

16382, 15, 63

Yukarıdaki dönüşüm tablosunu kullanan bir disk:

16384 \* 16\* 42 blok işleyebilmektedir bu değer de 128 GB alan karşılık gelmektedir. ATA-6 standardı ile bu değer 28 [bitlik](#) bilgiden 48 [bitlik](#) bilgiye çıkartılmıştır. Dolayısıyla adreslene bilen bilg 128PB olabilmektedir. 64 [bit](#)'e çıkan son adresleme yöntemi ile de bu bilgi 9 tirilyon GB'a kadar çıkabilmektedir. Burada BIOS'un bu adresleme şeklini desteklemesi gerektiği unutulmamalıdır.

Arayüz

Standart CHS

Extended CHS (ECHS) / Large

Logical Block Addressing

Entegre disk kontrolüne eklenen basit diskler

Fiziksel geometri

Fiziksel geometri

Fiziksel geometri

BIOS'a entegre edilmiş disk kontrolü

Mantıksal Geometri

Mantıksal Geometri

Mantıksal Blok Adresleme(LBA)

BIOS'tan işletim sistemi ve uygulamalara geçiş ( Int 13h ile)

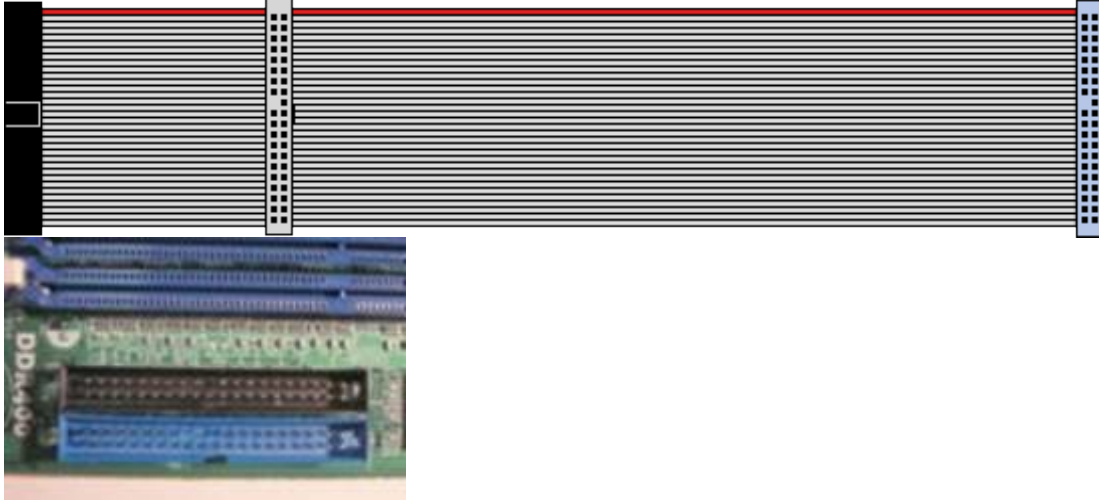
Mantıksal Geometri

Çevirilmiş Geometri

## Çevirilmiş Geometri

IDE (Integrated Drive Electronics) sabit disk bağlantılarını (ana kart ile) standartlaştırmak için Western Digital firması tarafından geliştirilmiş olan 40 [bit](#)lik standarttır. Daha sonraları SATA (serial ATA) ile ayırmak için PATA (paralel ATA) isminde de kullanılmıştır.

EIDE: IDE teknolojisinin adreslemede yetersiz kalması üzerine (max. 520mb adreslenebilmekteydi) 8.4 GB'a kadar adresleme yapabilen ve yine western digital firması tarafından geliştirilen standarttır.



Yukarıdaki resimde bir ide kablosu ve ana kart üzerindeki bağlantı yuvası gösterilmiştir.

Standardın ismi

Diğer ismi

Transfer şekli (MB/s)

Maximum disk boyutu

Diğer yeni özellikler

pre-ATA

IDE

PIO 0

2.1 GB

22-[bit](#) logical block addressing (LBA)

ATA-1

ATA, IDE



PIO 0, 1, 2 (3.3, 5.2, 8.3)  
Single-word DMA 0, 1, 2 (2.1, 4.2, 8.3)  
Multi-word DMA 0 (4.2)

137 GB

28-bit logical block addressing (LBA)

ATA-2

EIDE, Fast IDE, Ultra ATA Fast ATA,

PIO 3, 4: (11.1, 16.6)  
Multi-word DMA 1, 2 (13.3, 16.6)

ATA-3

EIDE

S.M.A.R.T., Security

ATA/ATAPI-4

ATA-4, Ultra ATA/33

Ultra DMA 0, 1, 2 (16.7, 25.0, 33.3)  
aka UDMA/33

AT Attachment Packet Interface (ATAPI), i.e. support for CD-ROM, tape drives etc.,  
Optional overlapped and queued command set features,  
Host Protected Area (HPA)

ATA/ATAPI-5

ATA-5, Ultra ATA/66

Ultra DMA 3, 4 (44.4, 66.7)  
aka UDMA/66

80-wire cables

ATA/ATAPI-6

ATA-6, Ultra ATA/100

UDMA 5 (100)  
aka UDMA/100

144 PB

48-bit LBA, Device Configuration Overlay (DCO),  
Automatic Acoustic Management

ATA/ATAPI-7

ATA-7, Ultra ATA/133

UDMA 6 (133)  
aka UDMA/133  
SATA/150

SATA 1.0, Streaming feature set, long logical/physical sector feature set for non-packet devices

ATA/ATAPI-8

ATA-8

—

Hybrid drive featuring non-volatile cache to speed up critical OS files

Disk Erişim Hızları.

Diskin üzerindeki bir veriye okumak veya yazmak için erişilmesi için belirli bir vakit geçmektedir. Bu geçen vakit aşağıdaki parçalara ayrılabilir:

Rotational Delay (döme gecikmesi)

Diskin ilgili sektörünün kafanın altına getirilmesi için disklerin döndürülmesi süresidir.

İki türlü olabilir :

Constant Angular Velocity: Sabit Açısal Hız. Bu çeşit disklerde açısal hız sabittir yani disk her zaman aynı hızda döner.

Constant Linear Velocity: Sabit Doğrusal Hız: Bu çeşit disklerde kafanın okuma hızına göre disk hızı yavaşlayıp artar. Örneğin disklin merkezine yakın bölgelerinde daha çevre uzunluğu kısaldığı için, bu bölgelerden okuma yapıldığı sırada doğrusal hız sabit tutulmak için disklin hızı azaltılır. Veya disklin dışına doğru uzunluk arttığı için disklin hızı arttırılır.

Seek Time: Arama Zamanı

Kafanın bir track (silindir, iz) üzerine gelmesi için kafanın hareket etmesi süresidir.

Latency Time (erişim süresi) verinin manyetik ortamdan okunup işlenecek hale çevrilmesi süresidir.

Bir bilginin diskten okunması veya yazılması için Dönme gecikmesi + arama zamanı + latency kadar vakit geçmelidir.