

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Genel bilgiler

Değerlendirme

Arasınav : 40%
Ödevler : 30%
Final Sınavı : 30%

Ders kitabı

Operating System Concepts, A. Silberschatz, G. Gagne, P. B. Galvin, Wiley, 2013.

İletişim

E-posta : maakcayol@gmail.com
Web : <http://w3.gazi.edu.tr/~akcayol>

Genel bilgiler

Ders içeriği

1. İşletim Sistemlerine Giriş
2. İşletim Sistem Yapıları
3. Process'ler
4. Thread'ler
5. Process Senkronizasyonu
6. CPU Scheduling
7. Deadlocks
8. Ana Bellek
9. Sanal Bellek
10. Kütük İşlemleri
11. Dosya Sistemi Arayüzü ve Uygulaması
12. I/O Sistemleri
13. Koruma ve Güvenlik

3

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

4

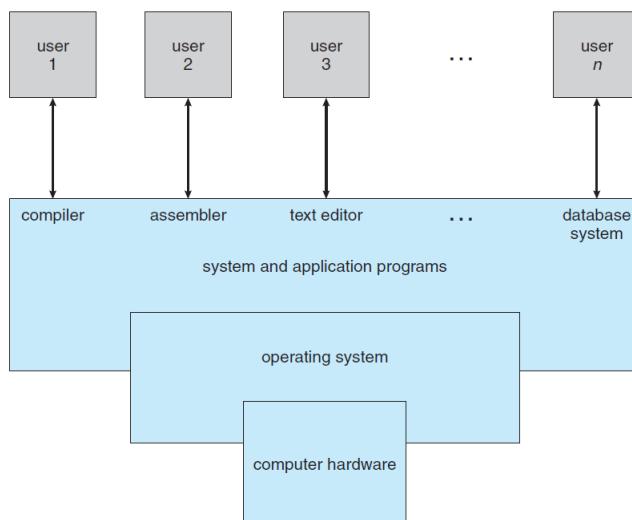
İşletim sistemi ne iş yapar?

- **İşletim sistemi bilgisayar donanımını yöneten bir programdır.**
- Kullanıcı ile bilgisayar donanımı arasında aracı olarak görev yapar.
- Bazı işletim sistemlerinde (server) **etkinlik**, bazlarında ise **kullanılabilirlik** (mobil, client) önemlidir.
- **Bir bilgisayar sistemi genel olarak 4 bileşene ayrılabilir:**
 - Donanım
 - İşletim sistemi
 - Uygulama programları
 - Kullanıcılar
- **Donanım** (CPU, memory, I/O cihazları) temel kaynakları sağlar.
- **Uygulama programları** (kelime işlemciler, derleyiciler, Web tarayıcıları) bu kaynakların kullanıcı problemlerinde nasıl kullanılacağını belirler.

5

İşletim sistemi ne iş yapar?

- **İşletim sistemi**, farklı kullanıcılar için farklı **kaynakların** uygun kullanımını koordine eder.



6



İşletim sistemi ne iş yapar?

Kullanıcı açısından bakış

- Kullanıcı bakışı **kullanılan arayüze bağlı olarak değişmektedir.**
- **Çoğu kullanıcı**, monitöre, klavyeye, fareye ve sistem birimine sahip bir **kişisel bilgisayar kullanır**.
- Bu durumda, **işletim sistemi** çoğunlukla **kolay kullanım için tasarlanır**, **kaynakların** nasıl paylaşıldığı ve **verimliliği**, çok kullanıcı yerine **tek kullanıcıya göre tasarlanır**.
- Diğer bir durumda ise, **çok sayıda kullanıcı bir ana bilgisayara bağlanır ve kaynakları paylaşırlar**.
- Burada, işletim sistemi **kaynak kullanım oranını maksimize edecek şekilde tasarlanır**.
- **Mobil cihazlar ve gömülü sistemler (ev cihazları, otomobil)** için tasarlanan işletim sistemlerinin de kendine özgü özellikleri vardır.



İşletim sistemi ne iş yapar?

Sistem açısından bakış

- Bilgisayar açısından işletim sistemi, **donanımla çok yakından ilişkiye sahip olan bir programdır**.
- Bu açıdan işletim sistemi, **kaynak** (CPU time, hafıza, depolama birimi, I/O cihazları, ...) **kullanımını planlayan programdır**.
- İşletim sistemi, **I/O cihazlarını ve kullanıcı programlarını kontrol eder** ve programların çalışması sırasında **hataları önlemeye yönelik işlemleri yönetir**.

İşletim sistemi ne iş yapar?

İşletim sistemi tanımı

- **Bilgisayarların temel amacı**, kullanıcı programlarının çalıştırılması ve kullanıcı problemlerinin kolay ve hızlı bir şekilde çözülmESİdir.
- Bilgisayar donanımlarının tek başına kullanımı çok zordur, bu yüzden uygulama yazılımları geliştirilir.
- Yaygın kabul edilen tanımlamada, **işletim sistemi bilgisayarda sürekli çalışan programdır ve kernel (çekirdek) olarak adlandırılır.**
- **Mobil işletim istemleri** sadece kernel'a sahip değildir, **middleware' e de sahiptirler.**
- **Middleware**, uygulama geliştiricilere ek servisler sağlayan **framework (platform) yazılımlarıdır.**
- Apple iOS ve Google Android, **middleware yazılımları ile veritabanı, multimedya ve grafik desteği** sağlayan mobil işletim sistemleridir.

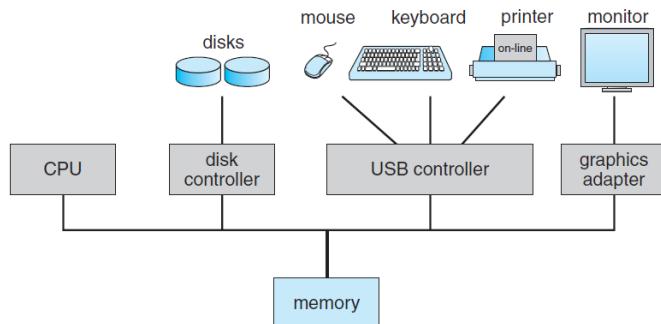
Konular

- İşletim sistemi ne iş yapar?
- **Bilgisayar sistemi organizasyonu**
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- Günüümüz genel amaçlı bilgisayarları **bir veya birden fazla CPU'ya**, ortak bus üzerinden kontrol edilen **cihazlara ve paylaşılmış hafızaya sahiptir.**



- CPU ile **cihaz denetleyicileri eş zamanlı çalışırlar** ve paylaşılmış hafızaya aynı anda erişmek isteyebilirler.
- Hafıza denetleyici cihazların hafızaya erişimini yönetir.

11

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- Bilgisayar çalışmaya başladığında, **başlangıç programı** olarak **bootstrap programını kullanır.**
- Bootstrap programı oldukça basittir ve **ROM** (Read-Only Memory) veya **EEPROM** (Electrically Erasable Programmable Read-Only Memory) içerisinde saklanır.
- **Bootstrap programlarına firmware adı verilir.**
- Firmware programı bilgisayarın tüm bileşenlerini (CPU register, cihaz denetleyicileri, hafıza içeriği) başlatır.
- Bootstrap programı, **işletim sistemi kernel'ının bulunduğu konumu bilmek ve hafızaya yüklemek zorundadır.**
- Kernel hafızaya yüklenikten sonra sisteme ve kullanıcılarla servis sağlamak için çalışır.

12

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

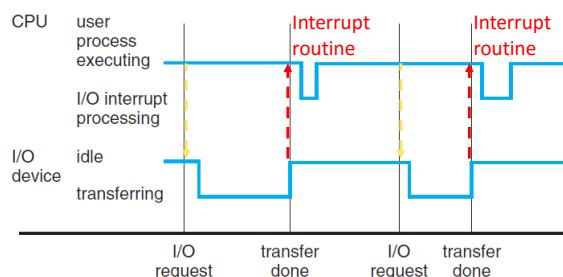
- Bazı servisler kernel dışındaki sistem programları (**system daemons, system processes**) tarafından sağlanır.
- UNIX üzerinde **init** ilk sistem prosesidir ve çok sayıda daemon başlatır. Bundan sonra sistem tümüyle **boot** edilmiş olur.
- İşletim sistemi **boot edildikten sonra bir olay (event) gerçekleşmesi için beklemeye başlar.**
- Bilgisayar sistemlerinde **bir olayın olduğu yazılım veya donanım tarafından interrupt kullanılarak bildirilir.**
- **Donanımlar**, CPU'ya bir sinyal ile interrupt bildirimi yapar.
- **Yazılımlar**, **system call** işlemlerini çalıştırarak interrupt başlatır.

13

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- CPU bir interrupt aldığında çalışmasını bulunduğu yerde keser ve **belirtilen diğer noktaya geçiş yapar.**
- Geçiş yaptığı yer, gelen kesmeyle ilişkilendirilmiş **service routine**'nin başlangıç adresidir.
- Interrupt'a ait **service routine bittiğinde ise önceki yere geçiş yaparak çalışmasına devam eder.**



14

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- **Interrupt routine adresleri pointer ile gösterilir.**
- Bu pointer'lar tablo halinde (**interrupt vector**) hafızanın 0-100 adresleri aralığında saklanır.
- **Interrupt pointer'ları her cihaz için ayrı adresi gösterir** ve interrupt gönderen cihazın routine'sinin adresini sağlarlar.
- Interrupt geldiğinde çalışan komutun (**instruction**) adresinin ve **CPU'nun konfigürasyonunun (register değerleri)** saklanması gereklidir (**context switch**).
- Komut adresi ve CPU konfigürasyonu **stack (yığın)** üzerinde saklanmaktadır.

15

Bilgisayar sistemi organizasyonu

Depolama yapısı

- **CPU**, programlara ait **komutları hafızadan yükler**. Bu yüzden, çalışacak programların önce hafızaya alınması gereklidir.
- **Genel amaçlı bilgisayarlar programları çalıştırmak için ana hafızayı (main memory) kullanır.**
- Ana hafıza, **RAM (random-access memory)** olarak da adlandırılır.
- Ana hafıza yarı iletken teknolojisi kullanılarak oluşturulur (**DRAM-dynamic RAM**).
- **von Neumann mimarisi'**ne sahip sistemlerde, komutlar **fetch** ile (**hafızadan CPU içerisindeki register'a alınması**) çalışmaya başlanır.
- Fetch aşaması sonucunda komut **instruction register'a alınmış** olur.
- Komut çözümlenir, çalıştırılır ve **sonucu (varsayı) hafızaya/reg aktarılır**.

16

Bilgisayar sistemi organizasyonu

Depolama yapısı

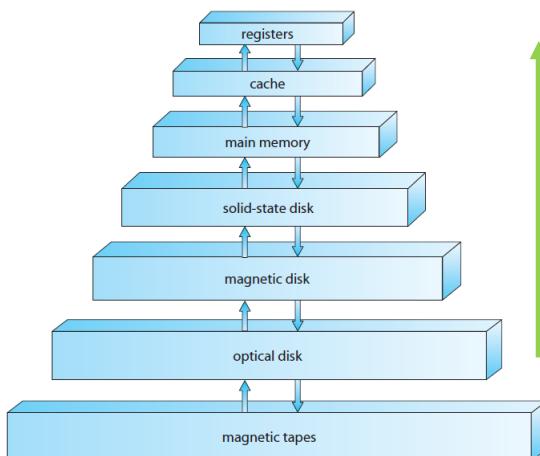
- Tüm programların hafızada saklanması istenir ancak iki neden dolayı mümkün değildir:
 - Hafıza sınırlı kapasiteye sahiptir. Veri ve programlar çok büyük boyuttadır.
 - Enerji kesildiğinde hafızadaki veri kaybolur (**volatile**).
- Tüm bilgisayar sistemleri, programları ve verileri kalıcı saklamak için ikincil depolama (**secondary storage**) birimlerine sahiptir.
- En yaygın kullanılan ikincil depolama birimi manyetik disklerdir (**magnetic disk, hdd**).
- Programlar hafızaya yüklenmeden önce manyetik disklerde tutulur.
- Veri saklama birimleri arasında, **hız, maliyet, boyut ve saklamanın kalıcılığı** açısından farklılıklar vardır.

17

Bilgisayar sistemi organizasyonu

Depolama yapısı

- Depolama birimleri arasında hiyerarşik bir ilişki vardır.



- Yukarı çıktıka erişim hızı artar.
- Yukarı çıktıka bit başına saklama maliyeti artar.
- Yukarı çıktıka toplam kapasite azalır.
- Yukarı çıktıka CPU tarafından kullanılma sıklığı artar.

18



Bilgisayar sistemi organizasyonu

Depolama yapısı

- **Main memory, cache ve register'lar** veriyi geçici saklama (**volatile**) birimleridir.
- **Solid-state disk, magnetic disk, optical disk ve magnetic tape** kalıcı saklama (**nonvolatile**) birimleridir.
- Solid-state disklerin farklı versiyonları vardır:
 - DRAM ile manyetik disk birlikte kullanılır. Normal işlem sırasında DRAM kullanılır daha sonra manyetik diske aktarma yapılır (dahili batarya kullanılır).
 - DRAM'den daha yavaş **flash memory** kullanılır (dahili batarya gerektirmez).
 - **NVRAM'de (Nonvolatile RAM)** ise **DRAM** ile **batarya** kullanılır ve kalıcı saklama yapılır.

19



Bilgisayar sistemi organizasyonu

I/O yapısı

- İşletim sistemleri kodunun büyük bir bölümü I/O yönetimine ayrılr.
- **I/O cihazlarında güvenilirlik (reliability)** ve **performans çok önemlidir.**
- Genel amaçlı bilgisayarlarda CPU ile çok sayıda I/O cihazı bus üzerinden bağlantıya sahiptir.
- Denetleyiciye bağlı olarak birden fazla cihaz bağlanabilir.
- **SCSI (small computer-systems interface)** denetleyiciye 7 veya daha fazla cihaz bağlanabilir.
- **İşletim sistemi her cihaz denetleyicisi için cihaz sürücüsüne (device driver)** sahiptir.

20

Bilgisayar sistemi organizasyonu

I/O yapısı

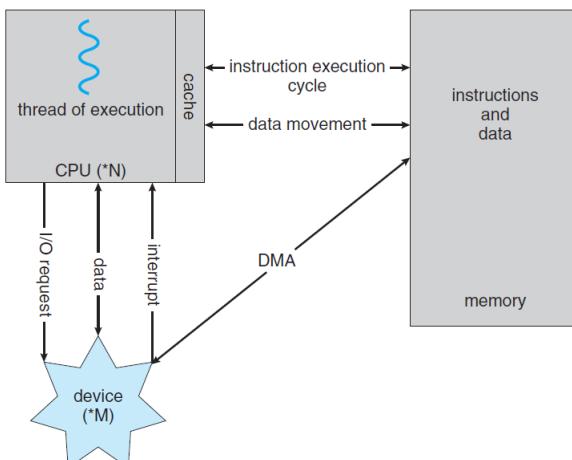
- I/O işlemini başlatmak için **device driver** uygun **register içeriğini device controller'a aktarır.**
- İşlemin tamamlandığı **device controller** tarafından **device driver'a interrupt ile bildirilir.**
- Bu şekilde veri aktarımında **overhead fazladır** ve aktarım işlemi yavaştır.
- **DMA (direct memory access)** ile **blok veri cihaz ile hafıza arasında doğrudan aktarılır.**
- DMA kullanıldığından I/O cihazı için buffer, pointer'lar ve sayıcılar oluşturulur ve device controller tüm aktarımı gerçekleştirir.
- İşlemin bittiği device driver'a interrupt ile bildirilir. **Bu işlem süresince CPU diğer işleri gerçekleştirir.**

21

Bilgisayar sistemi organizasyonu

I/O yapısı

- Modern bilgisayar sistemleri DMA ile veri aktarımı yapar ve paylaşılmış bus kullanılmaz.



22

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- **Bilgisayar sistemi mimarisi**
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

23

Bilgisayar sistemi mimarisi

Tek işlemcili sistemler

- Tek işlemciye sahip sistemde, **komut kümesindeki tüm komutlar bir işlemci tarafından çalıştırılmaktadır.**
- Bu sistemler disk, klavye, grafik denetleyici gibi bileşenlere sahiptir.
- **Tek işlemcili sistemlerin yönetimi** (sonraki görevin bildirilmesi, durumun izlenmesi) **işletim sistemi tarafından yapılmaktadır.**
- **Bu sistemler I/O cihazlarına özel işlemcilere de sahip olabilmektedir.**
- Örneğin, disk denetleyici işlemcisi, ana CPU'dan gelen isteklerin kuyruk yönetimi ve planlamasını gerçekleştirir.
- **Düzenli sistemlerde bu tür cihaz işlemcileri donanımların içerisinde yer almaktadır** ve işletim sistemi bu işlemcilerle haberleşmez.
- **Genel amaçlı tek işlemciye sahip olan sistemler tek işlemcili olarak adlandırılır.**

24

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

- Son birkaç yıldır **çok işlemcili sistemler** (*multiprocessor systems, parallel systems, multicore systems*) yoğun kullanılmaya başlanmıştır.
- **Çok işlemcili sistemler, iki veya daha fazla CPU'ya sahiptir ve bus, clock, memory ve çevre birimlerini paylaşırlar.**
- Çok işlemcili sistemler **önce sunucu sistemlerinde** kullanılmıştır.
- **Daha sonra masaüstü ve dizüstü bilgisayarlarda** kullanılmıştır.
- **Son yıllarda ise mobil cihazlarda** kullanılmaya başlanmıştır.

25

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

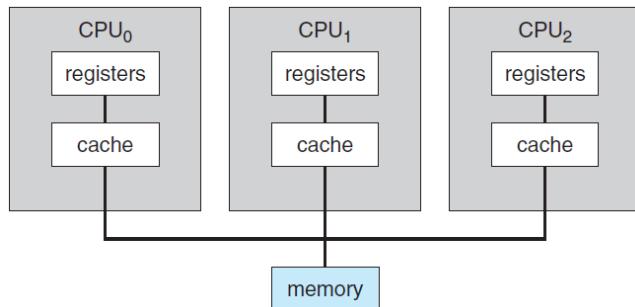
- Çok işlemcili sistemlerin temel olarak 3 avantajı vardır:
 - **Yüksek throughput:** İşlemci sayısı arttıkça daha kısa sürede daha fazla iş yapılır.
Hızlanma oranı işlemci sayısıyla doğru orantılı değildir!
 - **Ekonominik ölçeklendirme:** Aynı sayıda tek işlemcili sisteme göre **daha ekonomiktir**. Çevre birimlerini, depolama birimlerini ve güç birimlerini paylaşırlar.
 - **Yüksek güvenilirlik:** Bir işlemcide oluşan hata sistemin tümünü çalışmaz hale getirmez. **Performans azalır! (graceful degradation)**
- Asimetrik çok işlemcili sistemlerde (*AMP-asymmetric multiprocessing*), **her işlemci bir işe atanmıştır** ve tüm işlemciler başka bir işlemci tarafından denetlenir.
- Simetrik çok işlemcili sistemlerde (*SMP-symmetric multiprocessing*), **her işlemci** işletim sistemindeki **tüm işleri yapabilir**. Tüm işlemciler eş düzey (*peer*) olarak çalışır.

26

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

- SMP mimarisinde, **her CPU kendi register'larına ve lokal cache'e sahiptir** ancak **hafıza paylaşmaktadır**.



- **Windows, Mac OS X ve Linux** işletim sistemleri SMP mimarisini desteklemektedir.

27

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

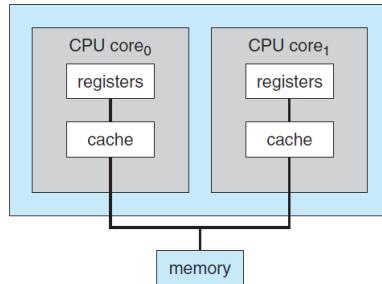
- RAM'e erişim süresi tüm işlemcilerde **aynı** olan modele **UMA (uniform memory access)** denilmektedir.
- RAM'e erişim süresi tüm işlemcilerde **farklı** olan modele **NUMA (nonuniform memory access)** denilmektedir.
- İşletim sistemi, kaynak yönetimi ile NUMA'nın erişim süresi dezavantajını ortadan kaldırabilir.

28

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

- Son yıllarda **bir chip üzerinde birden fazla işlemci (multicore)** kullanılmaktadır.
- Multicore sistemler birden fazla chip'e sahip çok işlemcili sistemlere göre **daha hızlıdır ve daha az enerji tüketirler**.
- **Her core kendi register'larına ve önbelleğine sahiptir, ancak hafızayı paylaşırlar.**



29

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

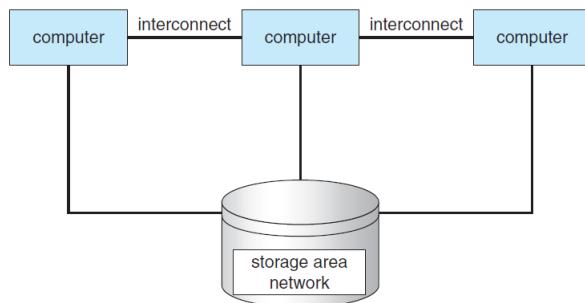
- **Blade sunucular**, çok işlemci board'ları, I/O board'ları ve ağ bağlantı board'larını **aynı kasada bulundururlar**.
- **Her blade işlemci board'u ayrı boot yapılır ve kendi işletim sistemini çalıştırır**.
- Bazı blade sunucularda, birden fazla çok işlemcili board kullanılabilir.

30

Bilgisayar sistemi mimarisi

Kümelenmiş (*clustered*) sistemler

- **Clustered sistemler** bağımsız iki veya daha fazla sistemden oluşurlar.
- Bu sistemler, depolama birimlerini paylaşırlar ve **LAN (local area network)** üzerinden haberleşirler.
- Bu sistemler **loosely coupled** (gevşek bağlı) olarak adlandırılırlar.



31

Bilgisayar sistemi mimarisi

Kümelenmiş (*clustered*) sistemler

- **Clustered sistemler**, **high-availability** sağlarlar.
- Her node, bir veya birkaç node'u izler hata oluşması durumunda o node'un görevlerini üstlenir.
- **Asymmetric clustering** yapısında, **bir sistem aktif çalışır diğer beklemeye modundadır (hot-standby mode)** ve çalışan sistemi izler. Hata olması halinde aktif çalışmaya başlar.
- **Symmetric clustering** yapısında, iki veya daha fazla sistem sktif olarak uygulamaları çalıştırır ve birbirlerini izlerler.

32

Bilgisayar sistemi mimarisi

Kümelenmiş (*clustered*) sistemler

- **Clustered sistemlerde**, bir program parçalara bölünerek eş zamanlı çalıştırılabilir (**parallelization**).
- Her sistemden elde edilen sonuçlar birleştirilerek sonuç çözüm elde edilir.
- Diğer **clustered** yapısında ise sistemler arasında iletişim **WAN (wide-area network)** üzerinden sağlanır.
- Bu sistemlerde işlem yapılan veride çakışmayı önlemek için **dağıtık kilitleme yönetimi (DLM-distributed lock manager)** yapılır.

33

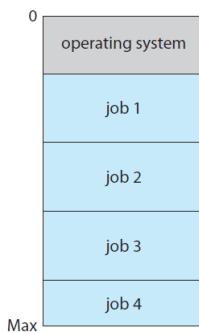
Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- **İşletim sistemi yapısı**
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

34

İşletim sistemi yapısı

- İşletim sistemi, programların çalıştırılması için ortam sağlamaktadır.
- İşletim sistemleri birden çok programı çalıştırabilir (**multiprogramming**).
- Multiprogramming çalışabilen işletim sistemi çok sayıda işi aynı anda hafızada tutar.
- Tüm işler disk üzerindeki job pool içinde tutulur.)



35

İşletim sistemi yapısı

- Multiprogramming işletim sistemi bir işi alır ve çalıştırılmaya başlar.
- Çalışan işte **bekleme olduğunda** başka bir işe geçiş yaparak çalışmaya devam eder.
- **Multitasking (time sharing)** işletim sistemlerinde CPU işler arasında çok hızlı geçişler yapar. (Geçiş için işte bekleme olması gereklidir.)
- Multitasking işletim sistemlerinde **kullanıcı** herhangi bir **iş ile etkileşime geçebilir**. **Tepki süresinin çok kısa olması gereklidir!**
- **Hafızaya yüklenen ve çalıştırılmakta olan programa process** denilir.
- Eğer hafızada ayrılan yerden daha çok sayıda iş hafızaya alınmak için hazır ise, **hafızaya alınacak olanı seçmeye job scheduling** denir.
- Aynı anda hafızada birden fazla iş hazır ise, **hangisinin ilk önce çalışacağına karar vermeye CPU scheduling** denilmektedir.

36

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- **İşletim sistemi işlemleri**
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

37

İşletim sistemi işlemleri

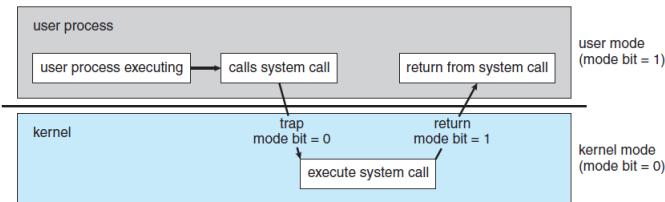
- **Modern işletim sistemleri, kesilmelerle yönetilirler (interrupt driven).**
- **Eğer çalışan process yoksa, hiçbir I/O cihazı servis sağlamıyorsa, kullanıcılarından etkileşim yoksa, işletim sistemi bekleme durumundadır ve hiçbir iş yapmaz.**
- Bir **trap** (veya **exception**), yazılım tarafından üretilen **interrupt'tır** ve işletim sisteminin iş gerçekleştirmesini sağlar.
- Bir işletim sisteminde çalışan programlardan birisi hata ürettiğinde sadece o programın etkilenmesi istenir.
- Ancak, bazı durumlarda **düger programların çalışma hızı etkilenebilir, verileri değiŞebilir veya işletim sisteminin kendisi bile çalışmaz hale gelebilir.**
- **İyi tasarılanmış işletim sistemleri** bu şekilde hatalı programların (**malicious**) diğerlerini etkilemesini engeller.

38

İşletim sistemi işlemleri

Dual mode ve multimode işlem

- İşletim sisteminin doğru çalışmasını sağlamak için, **işletim sistemi kodu ile kullanıcı programının kodunun ayrıt edilmesi gereklidir.**
- Mode bit** kullanılarak, kullanıcı modu (**user mode = 1**) ve kernel modu (**supervisor, system, privileged = 0**) ayrımı (**dual mode**) yapabilirler.



- Sistem boot edildiğinde kernel moddadır ve uygulama programı çalışmaya başlayınca user moda geçer.
- Birden fazla bit kullanılarak multimode** oluşturulabilir (test mode, ...).

39

İşletim sistemi işlemleri

Timer

- Bir kullanıcı programının **sonsuz döngüye girmesi** veya hata oluşması durumunda, **sistem servislerini çağrıramaması sonucunda işletim sistemine dönülemez**.
- Bu sorunu gidermek için **timer** kullanılır.
- Timer her programa geçildiğinde set edilir ve aşağıya doğru sayar.
- Timer 0 değerine ulaşınca kontrol işletim sistemine alınır.**
- İşletim sistemleri her program için belirlenen **timer süresini sabit veya değişken alabilmektedirler**.

40

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- **Process yönetimi**
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

41

Process yönetimi

- CPU tarafından çalıştırılmakta olan program **process** olarak adlandırılır.
- Bir process yapması gereken işi tamamlamak için, **CPU süresine, hafızaya, dosyalara, I/O cihazlarına ihtiyaç duyur.**
- Process çalıştığı sürece bu kaynaklardan ihtiyaç duyduğunu kullanır.
- Çalışması sonlanınca işletim sistemi ayrılmış kaynakları serbest bırakır.
- Bir **program pasif varlıktır (passive entity)**, bir **process ise aktif varlıktır (active entity)**.
- **Program counter (PC)**, CPU içerisinde register'dır ve sonraki çalıştırılacak komutun adresini tutar.
- **Single-threaded process** bir PC'ye sahiptir, **multithreaded process** birden çok PC'ye sahiptir.

42

Process yönetimi

- Bir işletim sistemi process yönetiminde aşağıdaki işlerden sorumludur:

- CPU üzerindeki process ve thread'lerin zamanlaması,
- Kullanıcı ve sistem process'lerinin oluşturulması ve silinmesi,
- Process'lerin askıya alınması ve devam ettirilmesi,
- Process'lerin senkronizasyonu,
- Process'lerin haberleşmesi.

43

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- **Memory yönetimi**
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

44

Memory yönetimi

- Hafıza, modern işletim sistemlerinde işlemlerin gerçekleşmesinde temel elemandır.
- **CPU, tüm programları hafıza üzerinden çalıştırır.**
- CPU, disk üzerindeki bir program parçasına ihtiyaç duyduğunda, I/O çağrısı ile önce hafızaya aktarır.
- Genel amaçlı bilgisayarlar **CPU verimliliğini artırmak** için birden çok programı hafızada tutarlar ve hafıza yönetimi gerçekleştirirler.
- İşletim sistemi **hafıza yönetiminde aşağıdaki işlerden sorumludur:**
 - **Hafızanın hangi kısmının kullanıldığı**n ve kimin tarafından kullanıldığın izlenmesi,
 - **Hangi process'in** (veya process parçasının) **hafızaya alınacağına** veya hafızadan atılacağına **karar verilmesi**,
 - Hafızadaki **boş alanların tahsis edilmesi** veya serbest bırakılması.

45

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- **Storage yönetimi**
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

45

Storage yönetimi

Dosya sistemi yönetimi

- Her depolama birimi, hız, kapasite, veri aktarım oranı ve erişim yöntemi gibi farklı özelliklere sahiptir.
- İşletim sistemi, depolama biriminin özelliklerini soyutlamak için mantıksal depolama birimi olarak **file (dosya)** tanımlar.
- Dosyalar, sayısal, alfabetik, alfanümerik veya binary veri bulundurabilir.
- Dosyaya birden fazla kullanıcı erişebilir. **Her kullanıcı için erişim denetiminin (okuma, yazma, ekleme) yapılması gereklidir.**
- **İşletim sistemi dosya yönetiminde aşağıdaki işlerden sorumludur:**
 - Dosya oluşturma ve silme,
 - Dizin oluşturma ve silme,
 - Dosya ve dizin manipülasyon işlemleri.

47

Storage yönetimi

Mass-storage yönetimi

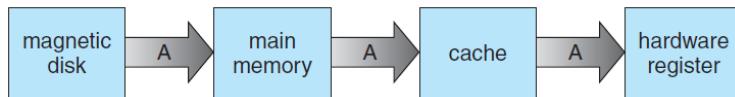
- Tüm programlar hafızaya alınmadan önce disk üzerinde saklanır.
- Disk yönetimi işletim sisteminin temel görevlerindendir.
- Manyetik tape, CD ve DVD sürücüler, üçüncü (tertiary) depolama cihazlarıdır.
- Bu cihazlar, **WORM (write-once, read-many-times)** veya **RW (read-write)** olarak farklı özelliklere sahip olabilirler.
- **İşletim sistemi disk yönetiminde aşağıdaki işlerden sorumludur:**
 - Boş alan yönetimi,
 - Depolama alanı tahsisı,
 - Disk kullanım zamanlaması.

48

Storage yönetimi

Cache bellek

- Cache bellek, hafızadan daha hızlı ve CPU'ya daha yakın saklama birimidir.
- CPU, bir veriye ihtiyaç duyduğunda hafızadan alır ve bir kopyasını cache bellek üzerine aktarır.
- Tekrarlı isteklerde cache bellek üzerindekini kullanır.



- Cache bellek kapasitesi çok küçük olduğundan yönetimi çok önemlidir.
- Cache bellekte tutulacak veya atılacak verilerin belirlenmesi için replacement algoritmaları kullanılır.

49

Storage yönetimi

Cache bellek

- Cache bellekler **hafızadan çok küçük kapasiteye** sahip olan ancak **register'lardan daha fazla kapasiteye** sahip olan depolama birimleridir.
- Cache belleklerde erişim **adrese göre, register'larda isme** göre yapılır.

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

- Birden fazla işlemcili sistemlerde cache tutarlılığının (**cache coherence**) sağlanması zorunludur.

50

Storage yönetimi

I/O sistemleri

- İşletim sistemlerinin amaçlarından birisi de **donanımların özelliklerinden kullanıcıyı soyutlamaktır.**
- Sadece **device driver** kendisine atanmış olan **cihazın özelliklerini bilir.**
- **I/O sistemi aşağıdaki bileşenlere sahiptir:**
 - Hafıza yönetim bileşeni (buffering, caching ve spooling),
 - Device driver arayüzü,
 - Donanımlar için driver.

51

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- **Koruma ve güvenlik**
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

52

Koruma ve güvenlik

- Bir bilgisayar sistemi **birden fazla kullanıcının erişimine açıksa ve birden çok işlemcinin eş zamanlı işlemine izin veriyorsa, verilere erişimin düzenlenmesi zorunludur.**
- İşlemciye, dosyalara, hafıza segment'lerine ve diğer kaynaklara sadece **yetkisi olan processlerin erişimine izin verilmelidir.**
- Bilgisayar sistemindeki kaynaklara kullanıcıların veya process'lerin **erişiminin denetlenmesine koruma (protection) denilmektedir.**
- **Protection** oluşabilecek hataları arayüzde iken algılar ve sistemin güvenilirliğini artırır.
- Koruma altındaki bir sistem yetkili ve yetkisiz kullanıcıları birbirinden ayırt eder.

53

Koruma ve güvenlik

- Bir sistem yeterli korumaya sahip olsa da **hatalara ve uygun olmayan erişimlere elverişli olabilir.**
- Örneğin, sisteme erişim yetkisi olan bir **kışının bilgileri çalınabilir ve bilgileri silinebilir, kopyalanabilir, dosyaları kalıcı hasara uğrayabilir.**
- **Güvenlik (security)**, bir sistemi dışarıdan veya içерiden **saldırılara karşı korumayı amaçlar.**
- Bu saldırılar, **virüsler, worm'lar, DoS, ...** gibi çok farklı şekillerde olabilir.
- **Bu saldırılardan korunmayı bazı işletim sistemleri görev** olarak düşünürken, **bazı işletim sistemleri** bu işleri **diğer yazılımlara bırakır.**
- Protection ve security, sistemindeki **tüm kullanıcıların birbirinden ayırt edilebilmesini gerektirir.**
- Çoğu işletim sistemi bunu **kullanıcı kimlikleri (user ID)** ile yapar.
- Grup ID veya sistem yönetici **(admin)** tanımlamaları da yapılabilir.

54

Konular

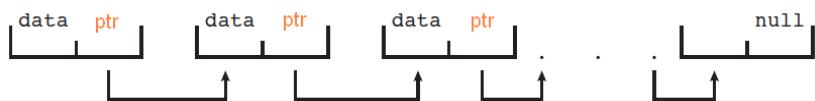
- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- **Kernel veri yapıları**
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

55

Kernel veri yapıları

Bağılı listeler

- Bir dizi (**array**) basit bir veri yapısıdır ve **elemanlara doğrudan erişim sağlar.**
- Dizilerde **elemanların boyutları sabittir** ve bir elemana doğrudan erişmek için öndeği eleman sayısı ile bir elemanın boyutu çarpılır.
- Ancak çoğu uygulamalarda **elemanlar farklı boyutlarda olabilir.**
- **Bu durumda, bağlı listeler (**linked lists**) kullanılır.**
- Bağlı listeler, bir grup veriyi art arda sıralı halde tutar ve doğrudan erişime olanak sağlar.

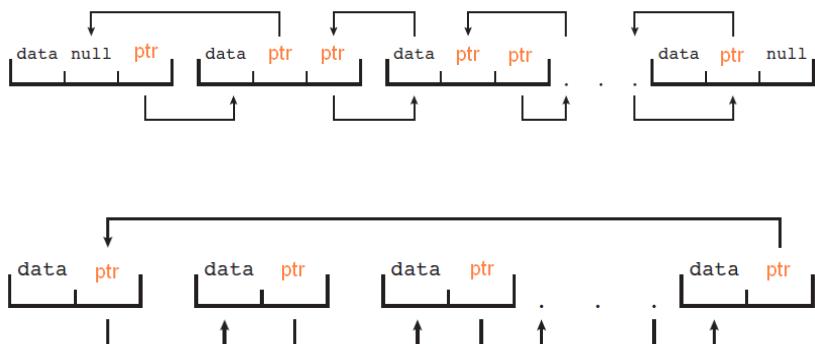


56

Kernel veri yapıları

Bağılı listeler

- Bağılı listeler bir bağlı (**singly linked list**), iki bağlı (**doubly linked list**) veya dairesel bağlı (**circularly linked list**) olabilir.

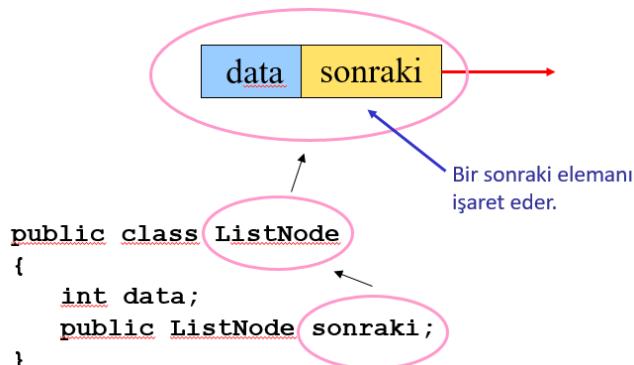


57

Kernel veri yapıları

Bağılı listeler

- Bağılı listelerde bir elemana erişim performansı $O(n)$ 'dır (**worst case**).
- Tek bağlı listenin tanımı aşağıdaki gibi yapılabilir.



58

Kernel veri yapıları

Yığın

- **Yığınlar (stack)** son gelen ilk çıkar (**last in first out - LIFO**) şeklinde çalışan veri yapılarıdır.
- Bir stack üzerinde yeni eleman eklemek için **push**, bir stack üzerinden son eklenen elemanı almak için **pop** işlevleri kullanılır.
- **İşletim sistemleri, iç içe fonksiyon çağrımlarında stack yapısını kullanır.**
- **Yeni fonksiyona geçiş yaparken ve geri dönerken, interrupt altyordamına geçiş yaparken ve geri dönerken**, parametreler, lokal değişkenler ve dönüş adresi stack üzerine saklanır.

59

Kernel veri yapıları

Kuyruk

- **Kuyruklar (queue)** ilk gelen ilk çıkar (**first in first out - FIFO**) şeklinde çalışan veri yapılarıdır.
- Bir kuyruk üzerinde yeni eleman eklendiğinde en sona kaydedilir, bir kuyruk üzerinden eleman alındığında en baştaki alınır.
- **İşletim sistemleri, yazıcıya iş gönderirken, işlemci tarafından çalıştırılmak için bekleyen görevlerin yönetiminde kuyruk yapısını kullanır.**

60

Kernel veri yapıları

Ağaçlar

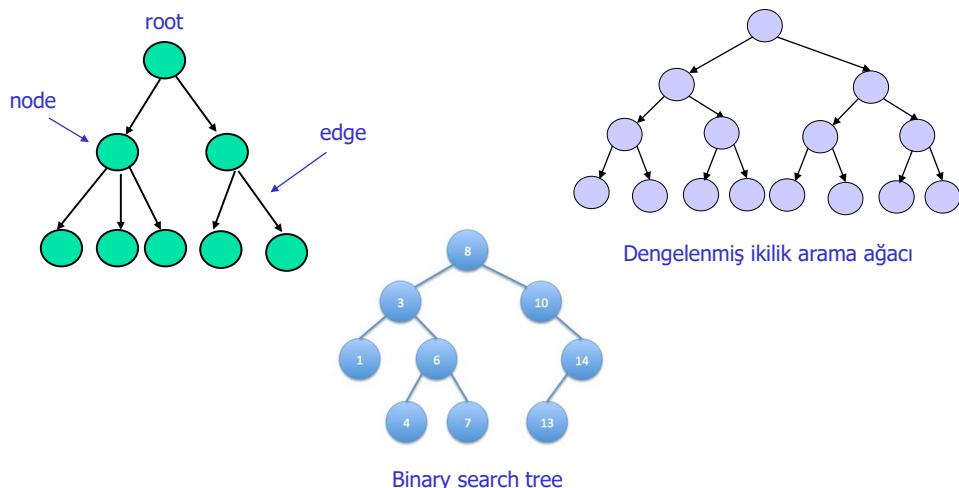
- Ağaçlar (**tree**) veriyi hiyerarşik şekilde göstermek için kullanılır.
- Ağaçlarda veriler **parent-child** ilişkisiyle tanımlanır.
- Genel olarak bir düğümde istenildiği kadar child olabilir.
- İkilik ağaçlarda (**binary tree**) ise **bir düğüm iki child düğüme sahip olabilir**.
- Binary arama ağaçlarında (**binary search tree**) bir elemana erişim performansı $O(n)$ 'dır (**worst case**).
- Dengelenmiş binary arama ağaçlarında (**balanced binary search tree**) bir elemana erişim performansı $O(\log n)$ 'dır (**worst case**).

61

Kernel veri yapıları

Ağaçlar

- Bir ağaç (**tree**) veriyi hiyerarşik şekilde göstermek için kullanılır.

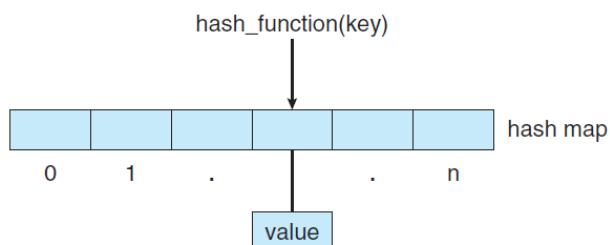


62

Kernel veri yapıları

Hash fonksiyonları

- Bir hash fonksiyonu (**hash function**) giriş olarak veri alır, bu veri üzerinde sayısal bir işlem yapar ve bir sayısal değer döndürür.
- Hash fonksiyonlarında veriye erişim performansı $O(1)$ 'dir.
- Hash map, bir anahtar ile değer eşleştirmesi yapar.
- Eşleştirme genellikle tekildir.



63

Kernel veri yapıları

Bitmap

- Bitmap'ler **n** adet binary bit ile oluşturulan dizgidir (**string**).
- Hash fonksiyonlarında veriye erişim performansı $O(1)$ 'dir.

001011101

- Çok sayıdaki kaynağın durumları ile ilgili bilgi (meşgul, kullanılabilir) bitlerle tutulabilir.
- Yukarıdaki bitler için 0, 1, 3 ve 7.kaynaklar kullanılabilir; 2, 4, 5, 6 ve 8.kaynaklar meşgul durumdadır.
- İşletim sistemi, disk bloklarının durumunu tutmak için bitmap kullanır.
- **İşletim sistemleri kernel algoritmalarında veri yapılarını sıkılıkla kullanır.**

64

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- **Hesaplama ortamları**
- Açık kaynak işletim sistemleri

65

Hesaplama ortamları

Geleneksel hesaplama

- Birkaç yıl öncesine kadar ofisteki bir bilgisayar ağa bağlanmakta, yazıcı veya diğer kaynakları kullanmaktadır.
- Günümüzde **Web teknolojileri** ve **WAN (Wide Area Network)** geleneksel hesaplama ortamlarının sınırlarını genişletmiştir.
- Firmalar **portal** oluşturmaktak ve Web erişimiyle istemcilere kaynaklara erişim sağlamaktadır.
- **Mobil cihazlar kablosuz ağlar (wireless networks)** ile **Web portal' e bağlanırlar.**
- Konut kullaclarının bant genişliği günümüzde hala yeterli düzeyde değildir, ayrıca ağlarının güvenliği için **firewall** kullanırlar.
- Firewall, **IP filtreleme, port filtreleme, Web filtreleme, içerik filtreleme** yapabilir.

66

Hesaplama ortamları

Mobil hesaplama

- **Mobil hesaplama (mobile computing)**, akıllı telefonlar ve **tablet bilgisayarlar ile yapılan işlemleri ifade eder.**
- Mobil cihazların özellikleri (ekran boyutu, hafıza kapasitesi ve performansı) son yıllarda önemli ölçüde gelişmiştir.
- Günümüzde **mobil cihazlar** sadece Web ve e-posta uygulamaları için değil, **tüm işlemler için kullanılır hale gelmiştir.**
- Mobil ortamlar için günümüzde **Apple iOS** ve **Google Android** işletim sistemleri yaygın olarak kullanılmaktadır.

67

Hesaplama ortamları

Dağıtık sistemler

- Bir dağıtık sistem, **fiziksel olarak ayrı, heterojen bilgisayar sistemidir.**
- Bir dağıtık sistem, **sahip olduğu çok sayıdaki kaynağı kullanıcıların erişimini sağlar.**
- **Bazı işletim sistemleri, sadece dosya erişimine yönelik** ve **FTP (File Transfer Protocol)** ve **NFS (Network File System)** protokollerini kullanırlar.
- **Bazı işletim sistemleri ise ağ fonksiyonlarını** (kullanıcı yönetimi, kaynak atama, ...) kullanıcıların kullanmasına izi verir.
- Bir ağ (**network**), iki veya daha fazla sistemin iletişimini sağlar.
- **Ağlar kullandıkları protokollere göre farklılık gösterirler.**
- Günümüzde **TCP/IP (Transmission Control Protocol/Internet Protocol)** en yaygın kullanılan protokol yığınıdır.

68

Hesaplama ortamları

Dağıtık sistemler

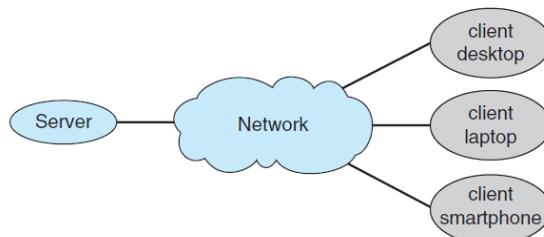
- Bilgisayar ağları kapsadıkları alana göre, **LAN (local-area network)**, **MAN (metropolitan-area network)** ve **WAN (wide-area network)** olarak üç gruba ayrılır.
- Bir bilgisayar ile laptop veya akıllı telefon arasında oluşturulan ağ, **PAN (personel-area network)** olarak adlandırılır.
- **WLAN (wireless LAN)** ise IEEE 802.11 ve **Bluetooth** teknolojileriyle oluşturulan yaklaşık 300 metre kapsama alanına sahip ağdır.
- Bir ağ işletim sistemi, ağdaki kaynakların yönetimini ve farklı bilgisayarlar üzerinde çalışan process'ler arasında iletişimini sağlar.

69

Hesaplama ortamları

İstemci-sunucu mimarisi

- Günümüzde **birçok sunucu (server) sistemi, istemci (client) bilgisayarların isteklerini karşılamak üzere tasarlanır.**
- **Compute-server** sistemlerinde, **istemciler bir işlemin yapılması için arayüz üzerinden istek gönderirler.**
- **File-server** sistemlerinde, **istemciler dosya oluşturma, silme veya okuma işlemlerini yapabilirler.**
- Sunucu, **yüksek konfigürasyona, istemci ise düşük konfigürasyona sahiptir.**

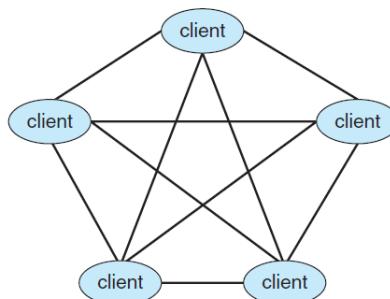


70

Hesaplama ortamları

Peer-to-peer mimarisi

- **Eş düzey (peer-to-peer, P2P) mimarisinde istemci ve sunucu ayrimı yoktur.** Tüm birimler aynı işlem kapasitesine ve yetkisine sahiptir.
- Kaynaklara erişim de dağıtık bir şekilde gerçekleştirilir.
- Napster, Gnutella gibi dosya paylaşım servisleri P2P mimarisine sahiptir.
- **VoIP (voice over IP)** teknolojisi kullanan **Skype**, P2P mimarisine sahiptir.



71

Hesaplama ortamları

Sanallaştırma

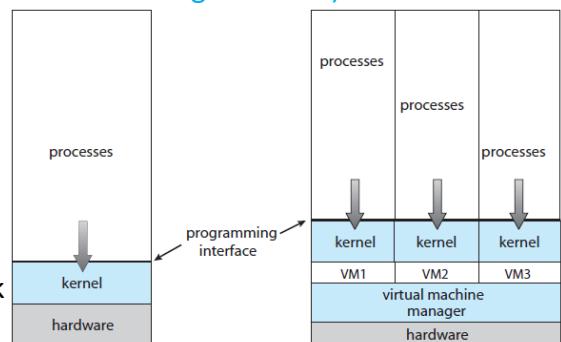
- **Sanallaştırma (virtualization)**, işletim sistemlerinin uygulamaları başka işletim sistemlerinde çalıştırmasına izin verir.
- Sanallaştırma, **emülatör** olarak adlandırılan yazılımı içerir.
- Emülatörler, kaynak CPU ile hedef CPU'nun farklı olduğu durumlarda kullanılır.
- Apple, **IBM CPU için derlenen bir programı Intel CPU'da çalıştırmak isterse**, Rosetta isimli emülatörü (dynamik binary çevirici) kullanır.
- **Yorumlayıcılar (interpreter)**, emülatör yazılımlarıdır ve yüksek seviyeli dilde编写的程序将被转换为低级的机器语言代码(native code)而无需进行编译。它们直接在运行时解释并执行代码。因此，它们通常比编译器慢，但提供了更快的开发和测试循环。
- Basic, derleme de yapabilir, yorumlama da yapabilir. Java, her zaman yorumlayıcıdır. **JVM (Java Virtual Machine)** bir emülatör yazılımıdır.

72

Hesaplama ortamları

Sanallaştırma

- **VMware**, bir işletim sistemi üzerinde farklı işletim sistemlerinin misafir (**guest copy**) olarak çalışmasına ve kendi uygulamalarını çalıştırmasına izin verir.
- Şekilde Windows host işletim sistemi, VMware uygulaması ise sanal makine yöneticisidir (**virtual machine manager - VMM**).
- **VMM, farklı işletim sistemlerini çalıştırır**, kaynak kullanımlarını yönetir ve kullanıcıların birbirini etkilemesini önler.
- **VMware ESXi** ve **Citrix XenServer**, host olarak çalışır.



73

Hesaplama ortamları

Bulut bilişim

- **Bulut bilişim (cloud computing)**, hesaplama, depolama ve uygulamaları bir ağ aracılığıyla servis olarak dağıtır.
- Bulut hesaplama, sanallaştımanın mantıksal uzantısı olarak kabul edilebilir.
- **Amazon Elastic Compute Cloud (EC2)**, **binlerce sunucuya, milyonlarca sanal makineye ve petabyte depolama alanına sahiptir**.
- EC2 bu kaynakları internet üzerinden kullanıcılar sunar ve **kullanıcılar kullandıkları kaynak oranında aylık ücretlendirilirler**.

74

Hesaplama ortamları

Bulut bilişim

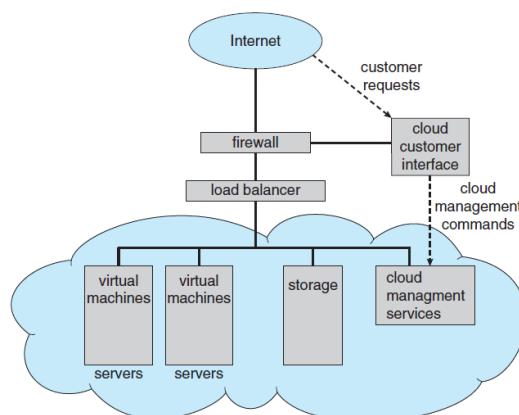
- Farklı bulut hesaplama türleri vardır:
 - **Public cloud** – İnternet üzerinden herkesin kullanımına açıktır.
 - **Private cloud** – bir firmanın sahibi olduğu buluttur.
 - **Hybrid cloud** – public ve private kullanılan bulut bileşenlerine sahiptir.
 - **Software as a service (SaaS)** – bir veya daha fazla uygulama (Word, Excel, ...) İnternet aracılığıyla kullanıma açıktır.
 - **Platform as a service (PaaS)** – Bir yazılım yığını (veritabanı sunucusu) uygulamalar için İnternet aracılığıyla kullanıma açıktır.
 - **Infrastructure as a service (IaaS)** – Sunucular veya depolama birimleri (üretilen verinin yedeklenmesi) İnternet aracılığıyla kullanıma açıktır.
- Bir bulut ortamı yukarıdaki türlerden birden fazlasını sağlayabilir.

75

Hesaplama ortamları

Bulut bilişim

- **Bulut hesaplama**, VMM yapısının ötesinde, kullanıcı process'lerinin çalıştığı **sanal makineleri yönetir**.
- VMM'ler bulut yönetim araçları (**Vware vCloud Director, Eucalyptus**) ile yönetilirler.



75

Hesaplama ortamları

Gerçek zamanlı gömülü sistemler

- Gömülü (**embedded**) sistemler, günümüzde **araç motorları**, **üretim robotları**, **mikro dalga fırınlar**, **uçaklar**, **teknolojik silahlar**, ..., gibi çok farklı yerlerde kullanılmaktadır.
- **Gömülü sistemler özel amaçlar için geliştirilirler ve sahip oldukları işletim sistemleri sınırlı özelliklere sahiptir.**
- Genellikle **arayüz gerektirmezler**, **donanımların izlenmesi** ve yönetimini gerçekleştirirler. **Farklı türleri vardır:**
 - İşletim sistemine sahiptirler ve özel amaçlı uygulamaları çalıştırırlar.
 - Özel amaçlı gömülü işletim sistemine sahiptirler ve istenen işlevleri yerine getirir.
 - Uygulamaya özel bütünsel devreye (**application specific integrated circuits - ASICs**) sahiptirler ve işletim sistemine sahip değildirler. Bu tür sistemler **hardwired systems** olarak da adlandırılırlar.

77

Hesaplama ortamları

Gerçek zamanlı gömülü sistemler

- **Gömülü sistemler, gerçek zamanlı işletim sistemlerini (**real-time operating systems**) çalıştırırlar.**
- **Gerçek zamanlı işletim sistemlerinde zaman gereksinimi çok hassastır.**
- **İstenen zaman aralıklarında veya belirlenen anda işlemlerin** gerçekleştirilmesi zorunludur.
- **Otomobil enjeksiyon sistemleri, medikal uygulamalar, silah sistemleri** başlıca uygulama alanlarıdır.

78

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- **Açık kaynak işletim sistemleri**

79

Açık kaynak işletim sistemleri

- Açık kaynak ([open-source](#)) işletim sistemlerinde, derlenmiş binary kod yerine **kaynak kodu da kullanılabilir durumdadır.**
- **Linux** açık kaynak işletim sistemlerine, **Windows** ise kapalı kaynak ([closed-source](#)) işletim sistemlerine örnek olarak verilebilir.
- **Apple Mac OS X** ve **iOS** hibrit işletim sistemleridir. Açık kaynak kernel' a ([Darwin](#)) sahiptir aynı zamanda kapalı kaynak bileşenlere de sahiptir.
- Kapalı kaynak işletim sistemlerinde **tersine mühendislik** ([reverse engineering](#)) kullanılarak binary kod oluşturulabilir.
- **Açık kaynak işletim sistemlerinde programcılar geliştirmeye katkı sağlayabilmektedirler.**
- **Açık kaynak**, kapalı kaynağa göre **daha güvenlidir**. Çünkü daha çok kişi tarafından kod görülmektedir.
- **Linux, BSD Unix** ve **Solaris** açık kaynak işletim sistemleridir.

80



Ödev

- İşletim sistemlerinin kernel fonksiyonları hakkında araştırma ödevi hazırlayınız.

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Genel bilgiler

Değerlendirme

Arasınav : 40%
Ödevler : 30%
Final Sınavı : 30%

Ders kitabı

Operating System Concepts, A. Silberschatz, G. Gagne, P. B. Galvin, Wiley, 2013.

İletişim

E-posta : maakcayol@gmail.com
Web : <http://w3.gazi.edu.tr/~akcayol>

Genel bilgiler

Ders içeriği

1. İşletim Sistemlerine Giriş
2. İşletim Sistem Yapıları
3. Process'ler
4. Thread'ler
5. Process Senkronizasyonu
6. CPU Scheduling
7. Deadlocks
8. Ana Bellek
9. Sanal Bellek
10. Kütük İşlemleri
11. Dosya Sistemi Arayüzü ve Uygulaması
12. I/O Sistemleri
13. Koruma ve Güvenlik

3

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

4

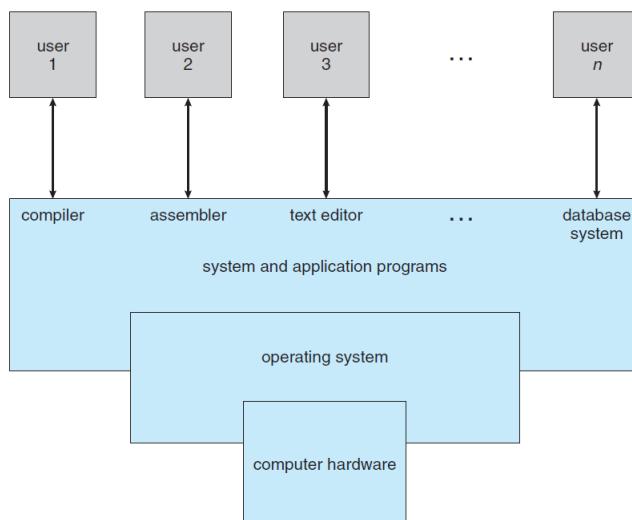
İşletim sistemi ne iş yapar?

- **İşletim sistemi bilgisayar donanımını yöneten bir programdır.**
- Kullanıcı ile bilgisayar donanımı arasında aracı olarak görev yapar.
- Bazı işletim sistemlerinde (server) **etkinlik**, bazlarında ise **kullanılabilirlik** (mobil, client) önemlidir.
- **Bir bilgisayar sistemi genel olarak 4 bileşene ayrılabilir:**
 - Donanım
 - İşletim sistemi
 - Uygulama programları
 - Kullanıcılar
- **Donanım** (CPU, memory, I/O cihazları) temel kaynakları sağlar.
- **Uygulama programları** (kelime işlemciler, derleyiciler, Web tarayıcıları) bu kaynakların kullanıcı problemlerinde nasıl kullanılacağını belirler.

5

İşletim sistemi ne iş yapar?

- **İşletim sistemi**, farklı kullanıcılar için farklı **kaynakların** uygun kullanımını koordine eder.



6



İşletim sistemi ne iş yapar?

Kullanıcı açısından bakış

- Kullanıcı bakışı **kullanılan arayüze bağlı olarak değişmektedir.**
- **Çoğu kullanıcı**, monitöre, klavyeye, fareye ve sistem birimine sahip bir **kişisel bilgisayar kullanır**.
- Bu durumda, **işletim sistemi** çoğunlukla **kolay kullanım için tasarlanır**, **kaynakların** nasıl paylaşıldığı ve **verimliliği**, çok kullanıcı yerine **tek kullanıcıya göre tasarlanır**.
- Diğer bir durumda ise, **çok sayıda kullanıcı bir ana bilgisayara bağlanır ve kaynakları paylaşırlar**.
- Burada, işletim sistemi **kaynak kullanım oranını maksimize edecek şekilde tasarlanır**.
- **Mobil cihazlar ve gömülü sistemler (ev cihazları, otomobil)** için tasarlanan işletim sistemlerinin de kendine özgü özellikleri vardır.



İşletim sistemi ne iş yapar?

Sistem açısından bakış

- Bilgisayar açısından işletim sistemi, **donanımla çok yakından ilişkiye sahip olan bir programdır**.
- Bu açıdan işletim sistemi, **kaynak** (CPU time, hafıza, depolama birimi, I/O cihazları, ...) **kullanımını planlayan programdır**.
- İşletim sistemi, **I/O cihazlarını ve kullanıcı programlarını kontrol eder** ve programların çalışması sırasında **hataları önlemeye yönelik işlemleri yönetir**.

İşletim sistemi ne iş yapar?

İşletim sistemi tanımı

- **Bilgisayarların temel amacı**, kullanıcı programlarının çalıştırılması ve kullanıcı problemlerinin kolay ve hızlı bir şekilde çözülmESİdir.
- Bilgisayar donanımlarının tek başına kullanımı çok zordur, bu yüzden uygulama yazılımları geliştirilir.
- Yaygın kabul edilen tanımlamada, **işletim sistemi bilgisayarda sürekli çalışan programdır ve kernel (çekirdek) olarak adlandırılır.**
- **Mobil işletim istemleri** sadece kernel'a sahip değildir, **middleware' e de sahiptirler.**
- **Middleware**, uygulama geliştiricilere ek servisler sağlayan **framework (platform) yazılımlarıdır.**
- Apple iOS ve Google Android, **middleware yazılımları ile veritabanı, multimedya ve grafik desteği** sağlayan mobil işletim sistemleridir.

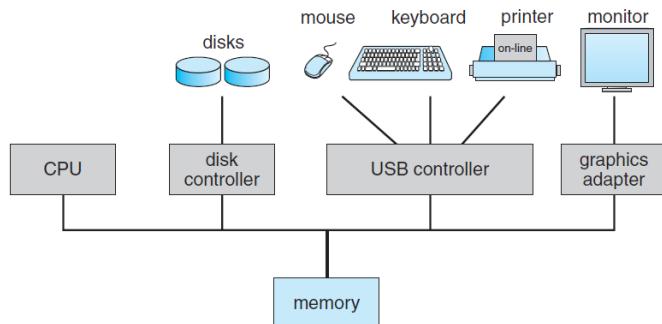
Konular

- İşletim sistemi ne iş yapar?
- **Bilgisayar sistemi organizasyonu**
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- Günüümüz genel amaçlı bilgisayarları **bir veya birden fazla CPU'ya**, ortak bus üzerinden kontrol edilen **cihazlara ve paylaşılmış hafızaya sahiptir.**



- CPU ile **cihaz denetleyicileri eş zamanlı çalışırlar** ve paylaşılmış hafızaya aynı anda erişmek isteyebilirler.
- Hafıza denetleyici cihazların hafızaya erişimini yönetir.

11

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- Bilgisayar çalışmaya başladığında, **başlangıç programı** olarak **bootstrap programını kullanır.**
- Bootstrap programı oldukça basittir ve **ROM** (Read-Only Memory) veya **EEPROM** (Electrically Erasable Programmable Read-Only Memory) içerisinde saklanır.
- **Bootstrap programlarına firmware adı verilir.**
- Firmware programı bilgisayarın tüm bileşenlerini (CPU register, cihaz denetleyicileri, hafıza içeriği) başlatır.
- Bootstrap programı, **işletim sistemi kernel'ının bulunduğu konumu bilmek ve hafızaya yüklemek zorundadır.**
- Kernel hafızaya yüklenikten sonra sisteme ve kullanıcılarla servis sağlamakaya başlar.

12

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

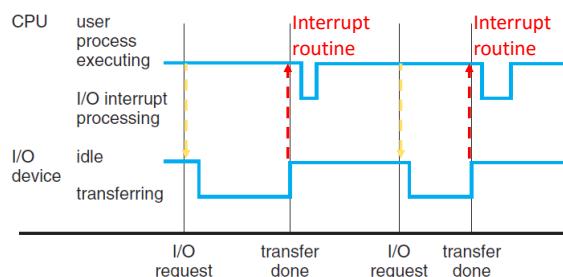
- Bazı servisler kernel dışındaki sistem programları (**system daemons, system processes**) tarafından sağlanır.
- UNIX üzerinde **init** ilk sistem prosesidir ve çok sayıda daemon başlatır. Bundan sonra sistem tümüyle **boot** edilmiş olur.
- İşletim sistemi **boot edildikten sonra bir olay (event) gerçekleşmesi için beklemeye başlar.**
- Bilgisayar sistemlerinde **bir olayın olduğu yazılım veya donanım tarafından interrupt kullanılarak bildirilir.**
- **Donanımlar**, CPU'ya bir sinyal ile interrupt bildirimi yapar.
- **Yazılımlar**, **system call** işlemlerini çalıştırarak interrupt başlatır.

13

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- CPU bir interrupt aldığında çalışmasını bulunduğu yerde keser ve **belirtilen diğer noktaya geçiş yapar.**
- Geçiş yaptığı yer, gelen kesmeyle ilişkilendirilmiş **service routine**'nin başlangıç adresidir.
- Interrupt'a ait **service routine bittiğinde ise önceki yere geçiş yaparak çalışmasına devam eder.**



14

Bilgisayar sistemi organizasyonu

Bilgisayar sisteminin çalışması

- **Interrupt routine adresleri pointer ile gösterilir.**
- Bu pointer'lar tablo halinde (**interrupt vector**) hafızanın 0-100 adresleri aralığında saklanır.
- **Interrupt pointer'ları her cihaz için ayrı adresi gösterir** ve interrupt gönderen cihazın routine'sinin adresini sağlarlar.
- Interrupt geldiğinde çalışan komutun (**instruction**) adresinin ve **CPU'nun konfigürasyonunun (register değerleri)** saklanması gereklidir (**context switch**).
- Komut adresi ve CPU konfigürasyonu **stack (yığın)** üzerinde saklanmaktadır.

15

Bilgisayar sistemi organizasyonu

Depolama yapısı

- **CPU**, programlara ait **komutları hafızadan yükler**. Bu yüzden, çalışacak programların önce hafızaya alınması gereklidir.
- **Genel amaçlı bilgisayarlar programları çalıştırmak için ana hafızayı (main memory) kullanır.**
- Ana hafıza, **RAM (random-access memory)** olarak da adlandırılır.
- Ana hafıza yarı iletken teknolojisi kullanılarak oluşturulur (**DRAM-dynamic RAM**).
- **von Neumann mimarisi'**ne sahip sistemlerde, komutlar **fetch** ile (**hafızadan CPU içerisindeki register'a alınması**) çalışmaya başlanır.
- Fetch aşaması sonucunda komut **instruction register'a alınmış** olur.
- Komut çözümlenir, çalıştırılır ve **sonucu (varsayı) hafızaya/reg aktarılır**.

16

Bilgisayar sistemi organizasyonu

Depolama yapısı

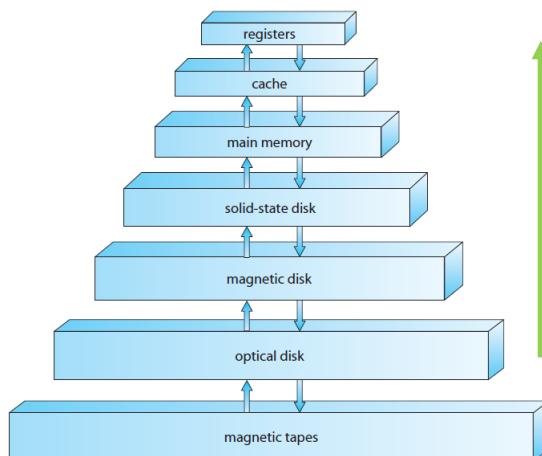
- Tüm programların hafızada saklanması istenir ancak iki neden dolayı mümkün değildir:
 - Hafıza sınırlı kapasiteye sahiptir. Veri ve programlar çok büyük boyuttadır.
 - Enerji kesildiğinde hafızadaki veri kaybolur (**volatile**).
- Tüm bilgisayar sistemleri, programları ve verileri kalıcı saklamak için ikincil depolama (**secondary storage**) birimlerine sahiptir.
- En yaygın kullanılan ikincil depolama birimi manyetik disklerdir (**magnetic disk, hdd**).
- Programlar hafızaya yüklenmeden önce manyetik disklerde tutulur.
- Veri saklama birimleri arasında, **hız, maliyet, boyut ve saklamanın kalıcılığı** açısından farklılıklar vardır.

17

Bilgisayar sistemi organizasyonu

Depolama yapısı

- Depolama birimleri arasında hiyerarşik bir ilişki vardır.



- Yukarı çıktıka erişim hızı artar.
- Yukarı çıktıka bit başına saklama maliyeti artar.
- Yukarı çıktıka toplam kapasite azalır.
- Yukarı çıktıka CPU tarafından kullanılma sıklığı artar.

18



Bilgisayar sistemi organizasyonu

Depolama yapısı

- **Main memory, cache ve register'lar** veriyi geçici saklama (**volatile**) birimleridir.
- **Solid-state disk, magnetic disk, optical disk ve magnetic tape** kalıcı saklama (**nonvolatile**) birimleridir.
- Solid-state disklerin farklı versiyonları vardır:
 - DRAM ile manyetik disk birlikte kullanılır. Normal işlem sırasında DRAM kullanılır daha sonra manyetik diske aktarma yapılır (dahili batarya kullanılır).
 - DRAM'den daha yavaş **flash memory** kullanılır (dahili batarya gerektirmez).
 - **NVRAM'de (Nonvolatile RAM)** ise **DRAM** ile **batarya** kullanılır ve kalıcı saklama yapılır.

19



Bilgisayar sistemi organizasyonu

I/O yapısı

- İşletim sistemleri kodunun büyük bir bölümü I/O yönetimine ayrılr.
- **I/O cihazlarında güvenilirlik (reliability)** ve **performans çok önemlidir.**
- Genel amaçlı bilgisayarlarda CPU ile çok sayıda I/O cihazı bus üzerinden bağlantıya sahiptir.
- Denetleyiciye bağlı olarak birden fazla cihaz bağlanabilir.
- **SCSI (small computer-systems interface)** denetleyiciye 7 veya daha fazla cihaz bağlanabilir.
- **İşletim sistemi her cihaz denetleyicisi için cihaz sürücüsüne (device driver)** sahiptir.

20

Bilgisayar sistemi organizasyonu

I/O yapısı

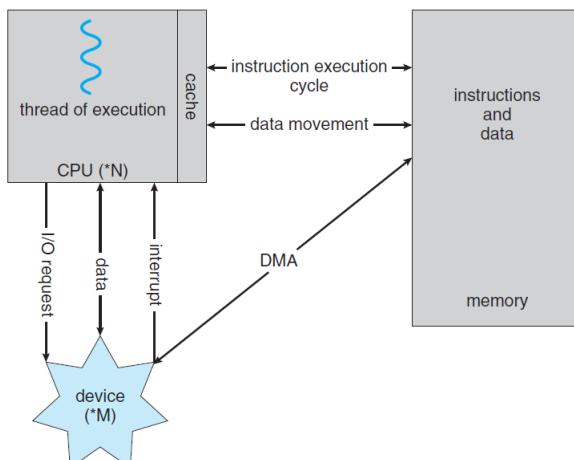
- I/O işlemini başlatmak için **device driver** uygun **register içeriğini device controller'a aktarır.**
- İşlemin tamamlandığı **device controller** tarafından **device driver'a interrupt ile bildirilir.**
- Bu şekilde veri aktarımında **overhead fazladır** ve aktarım işlemi yavaştır.
- **DMA (direct memory access)** ile **blok veri cihaz ile hafıza arasında doğrudan aktarılır.**
- DMA kullanıldığından I/O cihazı için buffer, pointer'lar ve sayıcılar oluşturulur ve device controller tüm aktarımı gerçekleştirir.
- İşlemin bittiği device driver'a interrupt ile bildirilir. **Bu işlem süresince CPU diğer işleri gerçekleştirir.**

21

Bilgisayar sistemi organizasyonu

I/O yapısı

- Modern bilgisayar sistemleri DMA ile veri aktarımı yapar ve paylaşılmış bus kullanılmaz.



22

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- **Bilgisayar sistemi mimarisi**
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

23

Bilgisayar sistemi mimarisi

Tek işlemcili sistemler

- Tek işlemciye sahip sistemde, **komut kümesindeki tüm komutlar bir işlemci tarafından çalıştırılmaktadır.**
- Bu sistemler disk, klavye, grafik denetleyici gibi bileşenlere sahiptir.
- **Tek işlemcili sistemlerin yönetimi** (sonraki görevin bildirilmesi, durumun izlenmesi) **işletim sistemi tarafından yapılmaktadır.**
- **Bu sistemler I/O cihazlarına özel işlemcilere de sahip olabilmektedir.**
- Örneğin, disk denetleyici işlemcisi, ana CPU'dan gelen isteklerin kuyruk yönetimi ve planlamasını gerçekleştirir.
- **Düzenli sistemlerde bu tür cihaz işlemcileri donanımların içerisinde yer almaktadır** ve işletim sistemi bu işlemcilerle haberleşmez.
- **Genel amaçlı tek işlemciye sahip olan sistemler tek işlemcili olarak adlandırılır.**

24

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

- Son birkaç yıldır **çok işlemcili sistemler** (*multiprocesser systems, parallel systems, multicore systems*) yoğun kullanılmaya başlanmıştır.
- **Çok işlemcili sistemler, iki veya daha fazla CPU'ya sahiptir ve bus, clock, memory ve çevre birimlerini paylaşırlar.**
- Çok işlemcili sistemler **önce sunucu sistemlerinde** kullanılmıştır.
- **Daha sonra masaüstü ve dizüstü bilgisayarlarda** kullanılmıştır.
- **Son yıllarda ise mobil cihazlarda** kullanılmaya başlanmıştır.

25

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

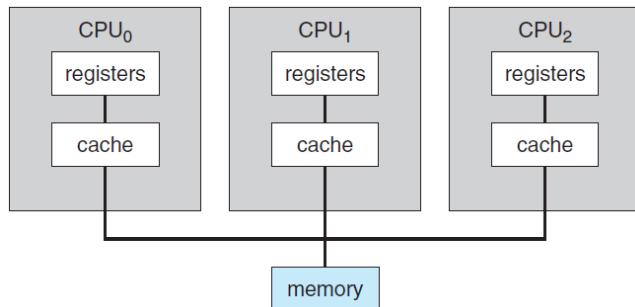
- Çok işlemcili sistemlerin temel olarak 3 avantajı vardır:
 - **Yüksek throughput:** İşlemci sayısı arttıkça daha kısa sürede daha fazla iş yapılır.
Hızlanma oranı işlemci sayısıyla doğru orantılı değildir!
 - **Ekonomik ölçeklendirme:** Aynı sayıda tek işlemcili sisteme göre **daha ekonomiktir**. Çevre birimlerini, depolama birimlerini ve güç birimlerini paylaşırlar.
 - **Yüksek güvenilirlik:** Bir işlemcide oluşan hata sistemin tümünü çalışmaz hale getirmez. **Performans azalır! (graceful degradation)**
- Asimetrik çok işlemcili sistemlerde (*AMP-asymmetric multiprocessing*), **her işlemci bir işe atanmıştır** ve tüm işlemciler başka bir işlemci tarafından denetlenir.
- Simetrik çok işlemcili sistemlerde (*SMP-symmetric multiprocessing*), **her işlemci** işletim sistemindeki **tüm işleri yapabilir**. Tüm işlemciler eş düzey (*peer*) olarak çalışır.

26

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

- SMP mimarisinde, **her CPU kendi register'larına ve lokal cache'e sahiptir** ancak **hafıza paylaşımaktadır**.



- **Windows, Mac OS X ve Linux** işletim sistemleri SMP mimarisini desteklemektedir.

27

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

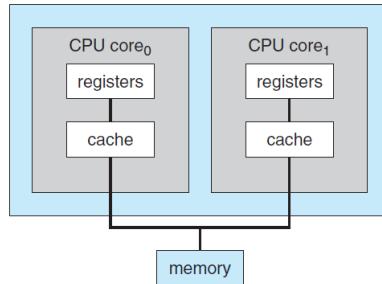
- RAM'e erişim süresi tüm işlemcilerde **aynı** olan modele **UMA (uniform memory access)** denilmektedir.
- RAM'e erişim süresi tüm işlemcilerde **farklı** olan modele **NUMA (nonuniform memory access)** denilmektedir.
- İşletim sistemi, kaynak yönetimi ile NUMA'nın erişim süresi dezavantajını ortadan kaldırabilir.

28

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

- Son yıllarda **bir chip üzerinde birden fazla işlemci (multicore)** kullanılmaktadır.
- Multicore sistemler birden fazla chip'e sahip çok işlemcili sistemlere göre **daha hızlıdır ve daha az enerji tüketirler**.
- **Her core kendi register'larına ve önbelleğine sahiptir, ancak hafızayı paylaşırlar.**



29

Bilgisayar sistemi mimarisi

Çok işlemcili sistemler

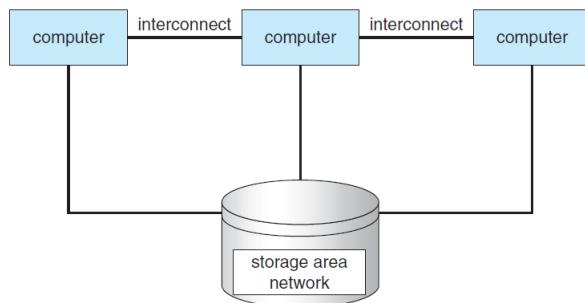
- **Blade sunucular**, çok işlemci board'ları, I/O board'ları ve ağ bağlantı board'larını **aynı kasada bulundururlar**.
- **Her blade işlemci board'u ayrı boot yapılır ve kendi işletim sistemini çalıştırır**.
- Bazı blade sunucularda, birden fazla çok işlemcili board kullanılabilir.

30

Bilgisayar sistemi mimarisi

Kümelenmiş (*clustered*) sistemler

- **Clustered sistemler** bağımsız iki veya daha fazla sistemden oluşurlar.
- Bu sistemler, depolama birimlerini paylaşırlar ve **LAN (local area network)** üzerinden haberleşirler.
- Bu sistemler **loosely coupled** (gevşek bağlı) olarak adlandırılırlar.



31

Bilgisayar sistemi mimarisi

Kümelenmiş (*clustered*) sistemler

- **Clustered sistemler**, **high-availability** sağlarlar.
- Her node, bir veya birkaç node'u izler hata oluşması durumunda o node'un görevlerini üstlenir.
- **Asymmetric clustering** yapısında, **bir sistem aktif çalışır diğer beklemeye modundadır (hot-standby mode)** ve çalışan sistemi izler. Hata olması halinde aktif çalışmaya başlar.
- **Symmetric clustering** yapısında, iki veya daha fazla sistem sktif olarak uygulamaları çalıştırır ve birbirlerini izlerler.

32

Bilgisayar sistemi mimarisi

Kümelenmiş (*clustered*) sistemler

- **Clustered sistemlerde**, bir program parçalara bölünerek eş zamanlı çalıştırılabilir (**parallelization**).
- Her sistemden elde edilen sonuçlar birleştirilerek sonuç çözüm elde edilir.
- Diğer **clustered** yapısında ise sistemler arasında iletişim **WAN (wide-area network)** üzerinden sağlanır.
- Bu sistemlerde işlem yapılan veride çakışmayı önlemek için **dağıtık kilitleme yönetimi (DLM-distributed lock manager)** yapılır.

33

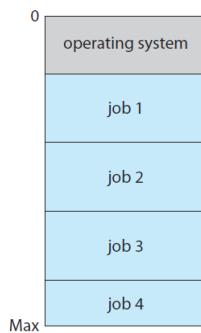
Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- **İşletim sistemi yapısı**
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

34

İşletim sistemi yapısı

- İşletim sistemi, programların çalıştırılması için ortam sağlamaktadır.
- İşletim sistemleri birden çok programı çalıştırabilir (**multiprogramming**).
- Multiprogramming çalışabilen işletim sistemi çok sayıda işi aynı anda hafızada tutar.
- Tüm işler disk üzerindeki job pool içinde tutulur.)



35

İşletim sistemi yapısı

- Multiprogramming işletim sistemi bir işi alır ve çalıştırılmaya başlar.
- Çalışan işte **bekleme olduğunda** başka bir işe geçiş yaparak çalışmaya devam eder.
- **Multitasking (time sharing)** işletim sistemlerinde CPU işler arasında çok hızlı geçişler yapar. (Geçiş için işte bekleme olması gereklidir.)
- Multitasking işletim sistemlerinde **kullanıcı** herhangi bir **iş ile etkileşime** **geçebilir**. **Tepki süresinin çok kısa olması** **gereklidir!**
- **Hafızaya yüklenen ve çalıştırılmakta olan programa process** denilir.
- Eğer hafızada ayrılan yerden daha çok sayıda iş hafızaya alınmak için hazır ise, **hafızaya alınacak olanı seçmeye job scheduling** denir.
- Aynı anda hafızada birden fazla iş hazır ise, **hangisinin ilk önce çalışacağına karar vermeye CPU scheduling** denilmektedir.

36

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- **İşletim sistemi işlemleri**
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

37

İşletim sistemi işlemleri

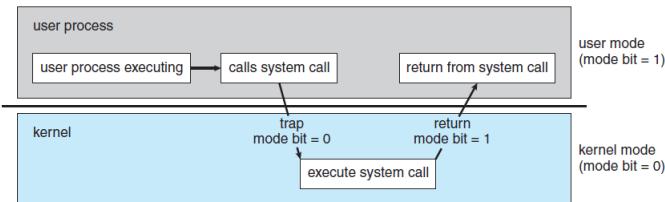
- **Modern işletim sistemleri, kesilmelerle yönetilirler (interrupt driven).**
- **Eğer çalışan process yoksa, hiçbir I/O cihazı servis sağlamıyorsa, kullanıcılarından etkileşim yoksa, işletim sistemi bekleme durumundadır ve hiçbir iş yapmaz.**
- Bir **trap** (veya **exception**), yazılım tarafından üretilen **interrupt'tır** ve işletim sisteminin iş gerçekleştirmesini sağlar.
- Bir işletim sisteminde çalışan programlardan birisi hata ürettiğinde sadece o programın etkilenmesi istenir.
- Ancak, bazı durumlarda **düger programların çalışma hızı etkilenebilir, verileri değiŞebilir veya işletim sisteminin kendisi bile çalışmaz hale gelebilir.**
- **İyi tasarılanmış işletim sistemleri** bu şekilde hatalı programların (**malicious**) diğerlerini etkilemesini engeller.

38

İşletim sistemi işlemleri

Dual mode ve multimode işlem

- İşletim sisteminin doğru çalışmasını sağlamak için, **işletim sistemi kodu ile kullanıcı programının kodunun ayrıt edilmesi gereklidir.**
- Mode bit** kullanılarak, kullanıcı modu (**user mode = 1**) ve kernel modu (**supervisor, system, privileged = 0**) ayrımı (**dual mode**) yapabilirler.



- Sistem boot edildiğinde kernel moddadır ve uygulama programı çalışmaya başlayınca user moda geçer.
- Birden fazla bit kullanılarak multimode** oluşturulabilir (test mode, ...).

39

İşletim sistemi işlemleri

Timer

- Bir kullanıcı programının **sonsuz döngüye girmesi** veya hata oluşması durumunda, **sistem servislerini çağrıramaması sonucunda işletim sistemine dönülemez**.
- Bu sorunu gidermek için **timer** kullanılır.
- Timer her programa geçildiğinde set edilir ve aşağıya doğru sayar.
- Timer 0 değerine ulaşınca kontrol işletim sistemine alınır.**
- İşletim sistemleri her program için belirlenen **timer süresini sabit veya değişken alabilmektedirler**.

40

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- **Process yönetimi**
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

41

Process yönetimi

- CPU tarafından çalıştırılmakta olan program **process** olarak adlandırılır.
- Bir process yapması gereken işi tamamlamak için, **CPU süresine, hafızaya, dosyalara, I/O cihazlarına ihtiyaç duyar.**
- Process çalıştığı sürece bu kaynaklardan ihtiyaç duyduğunu kullanır.
- Çalışması sonlanınca işletim sistemi ayrılmış kaynakları serbest bırakır.
- Bir **program pasif varlıktır (passive entity)**, bir **process ise aktif varlıktır (active entity)**.
- **Program counter (PC)**, CPU içerisinde register'dır ve sonraki çalıştırılacak komutun adresini tutar.
- **Single-threaded process** bir PC'ye sahiptir, **multithreaded process** birden çok PC'ye sahiptir.

42

Process yönetimi

- Bir işletim sistemi process yönetiminde aşağıdaki işlerden sorumludur:

- CPU üzerindeki process ve thread'lerin zamanlaması,
- Kullanıcı ve sistem process'lerinin oluşturulması ve silinmesi,
- Process'lerin askıya alınması ve devam ettirilmesi,
- Process'lerin senkronizasyonu,
- Process'lerin haberleşmesi.

43

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- **Memory yönetimi**
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

44

Memory yönetimi

- Hafıza, modern işletim sistemlerinde işlemlerin gerçekleşmesinde temel elemandır.
- **CPU, tüm programları hafıza üzerinden çalıştırır.**
- CPU, disk üzerindeki bir program parçasına ihtiyaç duyduğunda, I/O çağrısı ile önce hafızaya aktarır.
- Genel amaçlı bilgisayarlar **CPU verimliliğini artırmak** için birden çok programı hafızada tutarlar ve hafıza yönetimi gerçekleştirirler.
- İşletim sistemi **hafıza yönetiminde aşağıdaki işlerden sorumludur:**
 - **Hafızanın hangi kısmının kullanıldığı**n ve kimin tarafından kullanıldığın izlenmesi,
 - **Hangi process'in** (veya process parçasının) **hafızaya alınacağına** veya hafızadan atılacağına **karar verilmesi**,
 - Hafızadaki **boş alanların tahsis edilmesi** veya serbest bırakılması.

45

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- **Storage yönetimi**
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

45

Storage yönetimi

Dosya sistemi yönetimi

- Her depolama birimi, hız, kapasite, veri aktarım oranı ve erişim yöntemi gibi farklı özelliklere sahiptir.
- İşletim sistemi, depolama biriminin özelliklerini soyutlamak için mantıksal depolama birimi olarak **file (dosya)** tanımlar.
- Dosyalar, sayısal, alfabetik, alfanümerik veya binary veri bulundurabilir.
- Dosyaya birden fazla kullanıcı erişebilir. **Her kullanıcı için erişim denetiminin (okuma, yazma, ekleme) yapılması gereklidir.**
- **İşletim sistemi dosya yönetiminde aşağıdaki işlerden sorumludur:**
 - Dosya oluşturma ve silme,
 - Dizin oluşturma ve silme,
 - Dosya ve dizin manipülasyon işlemleri.

47

Storage yönetimi

Mass-storage yönetimi

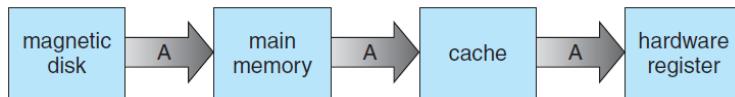
- Tüm programlar hafızaya alınmadan önce disk üzerinde saklanır.
- Disk yönetimi işletim sisteminin temel görevlerindendir.
- Manyetik tape, CD ve DVD sürücüler, üçüncü (tertiary) depolama cihazlarıdır.
- Bu cihazlar, **WORM (write-once, read-many-times)** veya **RW (read-write)** olarak farklı özelliklere sahip olabilirler.
- **İşletim sistemi disk yönetiminde aşağıdaki işlerden sorumludur:**
 - Boş alan yönetimi,
 - Depolama alanı tahsisı,
 - Disk kullanım zamanlaması.

48

Storage yönetimi

Cache bellek

- Cache bellek, hafızadan daha hızlı ve CPU'ya daha yakın saklama birimidir.
- CPU, bir veriye ihtiyaç duyduğunda hafızadan alır ve bir kopyasını cache bellek üzerine aktarır.
- Tekrarlı isteklerde cache bellek üzerindekini kullanır.



- Cache bellek kapasitesi çok küçük olduğundan yönetimi çok önemlidir.
- Cache bellekte tutulacak veya atılacak verilerin belirlenmesi için replacement algoritmaları kullanılır.

49

Storage yönetimi

Cache bellek

- Cache bellekler **hafızadan çok küçük kapasiteye** sahip olan ancak **register'lardan daha fazla kapasiteye** sahip olan depolama birimleridir.
- Cache belleklerde erişim **adrese göre, register'larda isme** göre yapılır.

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

- Birden fazla işlemcili sistemlerde cache tutarlılığının (**cache coherence**) sağlanması zorunludur.

50

Storage yönetimi

I/O sistemleri

- İşletim sistemlerinin amaçlarından birisi de **donanımların özelliklerinden kullanıcıyı soyutlamaktır.**
- Sadece **device driver** kendisine atanmış olan **cihazın özelliklerini bilir.**
- **I/O sistemi aşağıdaki bileşenlere sahiptir:**
 - Hafıza yönetim bileşeni (buffering, caching ve spooling),
 - Device driver arayüzü,
 - Donanımlar için driver.

51

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- **Koruma ve güvenlik**
- Kernel veri yapıları
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

52

Koruma ve güvenlik

- Bir bilgisayar sistemi **birden fazla kullanıcının erişimine açıksa ve birden çok işlemcinin eş zamanlı işlemine izin veriyorsa, verilere erişimin düzenlenmesi zorunludur.**
- İşlemciye, dosyalara, hafıza segment'lerine ve diğer kaynaklara sadece **yetkisi olan processlerin erişimine izin verilmelidir.**
- Bilgisayar sistemindeki kaynaklara kullanıcıların veya process'lerin **erişiminin denetlenmesine koruma (protection) denilmektedir.**
- **Protection** olusabilecek hataları arayüzde iken algılar ve sistemin güvenilirliğini artırır.
- Koruma altındaki bir sistem yetkili ve yetkisiz kullanıcıları birbirinden ayırt eder.

53

Koruma ve güvenlik

- Bir sistem yeterli korumaya sahip olsa da **hatalara ve uygun olmayan erişimlere elverişli olabilir.**
- Örneğin, sisteme erişim yetkisi olan bir **kışının bilgileri çalınabilir ve bilgileri silinebilir, kopyalanabilir, dosyaları kalıcı hasara uğrayabilir.**
- **Güvenlik (security)**, bir sistemi dışarıdan veya içерiden **saldırılara karşı korumayı amaçlar.**
- Bu saldırılar, **virüsler, worm'lar, DoS, ...** gibi çok farklı şekillerde olabilir.
- **Bu saldırılardan korunmayı bazı işletim sistemleri görev** olarak düşünürken, **bazı işletim sistemleri** bu işleri **diğer yazılımlara bırakır.**
- Protection ve security, sistemindeki **tüm kullanıcıların birbirinden ayırt edilebilmesini gerektirir.**
- Çoğu işletim sistemi bunu **kullanıcı kimlikleri (user ID)** ile yapar.
- Grup ID veya sistem yönetici **(admin)** tanımlamaları da yapılabilir.

54

Konular

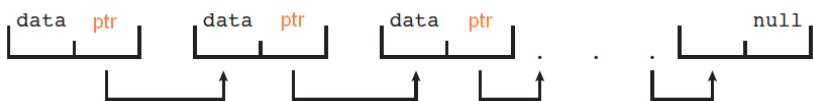
- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- **Kernel veri yapıları**
- Hesaplama ortamları
- Açık kaynak işletim sistemleri

55

Kernel veri yapıları

Bağılı listeler

- Bir dizi (**array**) basit bir veri yapısıdır ve **elemanlara doğrudan erişim sağlar.**
- Dizilerde **elemanların boyutları sabittir** ve bir elemana doğrudan erişmek için öndeği eleman sayısı ile bir elemanın boyutu çarpılır.
- Ancak çoğu uygulamalarda **elemanlar farklı boyutlarda olabilir.**
- **Bu durumda, bağlı listeler (**linked lists**) kullanılır.**
- Bağlı listeler, bir grup veriyi art arda sıralı halde tutar ve doğrudan erişime olanak sağlar.

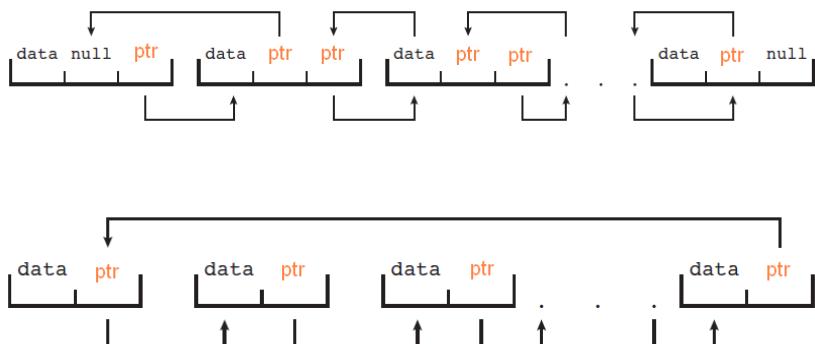


56

Kernel veri yapıları

Bağılı listeler

- Bağılı listeler bir bağlı (**singly linked list**), iki bağlı (**doubly linked list**) veya dairesel bağlı (**circularly linked list**) olabilir.

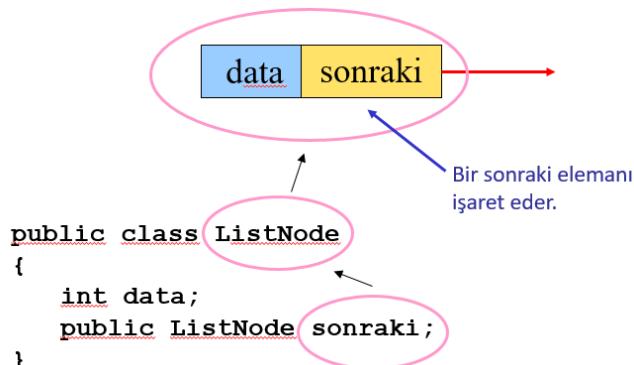


57

Kernel veri yapıları

Bağılı listeler

- Bağılı listelerde bir elemana erişim performansı $O(n)$ 'dır (**worst case**).
- Tek bağlı listenin tanımı aşağıdaki gibi yapılabilir.



58

Kernel veri yapıları

Yığın

- **Yığınlar (stack)** son gelen ilk çıkar (**last in first out - LIFO**) şeklinde çalışan veri yapılarıdır.
- Bir stack üzerinde yeni eleman eklemek için **push**, bir stack üzerinden son eklenen elemanı almak için **pop** işlevleri kullanılır.
- İşletim sistemleri, iç içe fonksiyon çağrımlarında stack yapısını kullanır.
- **Yeni fonksiyona geçiş yaparken ve geri dönerken, interrupt altyordamına geçiş yaparken ve geri dönerken**, parametreler, lokal değişkenler ve dönüş adresi stack üzerine saklanır.

59

Kernel veri yapıları

Kuyruk

- **Kuyruklar (queue)** ilk gelen ilk çıkar (**first in first out - FIFO**) şeklinde çalışan veri yapılarıdır.
- Bir kuyruk üzerinde yeni eleman eklendiğinde en sona kaydedilir, bir kuyruk üzerinden eleman alındığında en baştaki alınır.
- İşletim sistemleri, **yazıcıya iş gönderirken, işlemci tarafından çalıştırılmak için bekleyen görevlerin yönetiminde kuyruk yapısını kullanır.**

60

Kernel veri yapıları

Ağaçlar

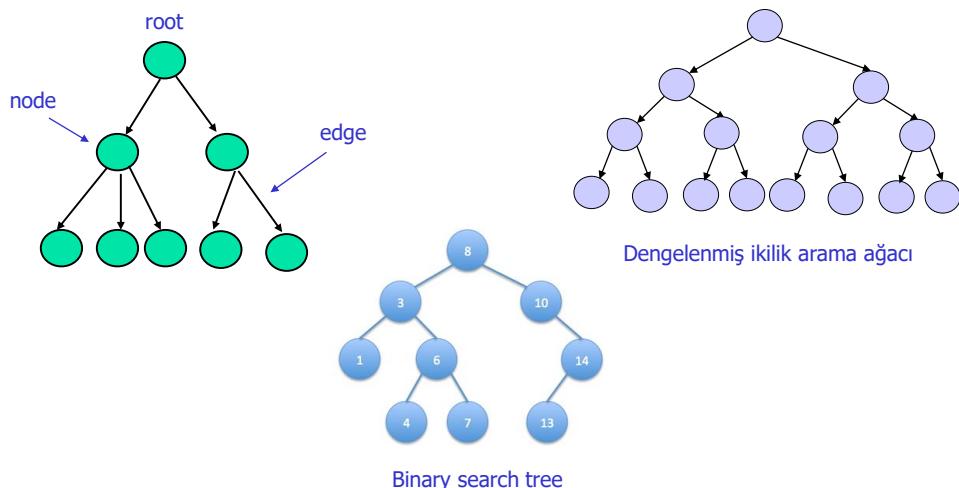
- Ağaçlar (**tree**) veriyi hiyerarşik şekilde göstermek için kullanılır.
- Ağaçlarda veriler **parent-child** ilişkisiyle tanımlanır.
- Genel olarak bir düğümde istenildiği kadar child olabilir.
- İkilik ağaçlarda (**binary tree**) ise **bir düğüm iki child düğüme sahip olabilir**.
- Binary arama ağaçlarında (**binary search tree**) bir elemana erişim performansı $O(n)$ 'dır (**worst case**).
- Dengelenmiş binary arama ağaçlarında (**balanced binary search tree**) bir elemana erişim performansı $O(\log n)$ 'dır (**worst case**).

61

Kernel veri yapıları

Ağaçlar

- Bir ağaç (**tree**) veriyi hiyerarşik şekilde göstermek için kullanılır.

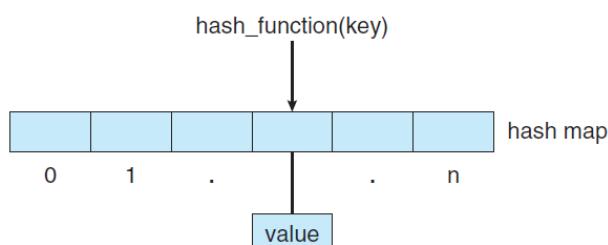


62

Kernel veri yapıları

Hash fonksiyonları

- Bir hash fonksiyonu (**hash function**) giriş olarak veri alır, bu veri üzerinde sayısal bir işlem yapar ve bir sayısal değer döndürür.
- Hash fonksiyonlarında veriye erişim performansı $O(1)$ 'dır.
- Hash map, bir anahtar ile değer eşleştirmesi yapar.
- Eşleştirme genellikle tekildir.



63

Kernel veri yapıları

Bitmap

- Bitmap'ler **n** adet binary bit ile oluşturulan dizgidir (**string**).
- Hash fonksiyonlarında veriye erişim performansı $O(1)$ 'dır.

001011101

- Çok sayıdaki kaynağın durumları ile ilgili bilgi (meşgul, kullanılabilir) bitlerle tutulabilir.
- Yukarıdaki bitler için 0, 1, 3 ve 7.kaynaklar kullanılabilir; 2, 4, 5, 6 ve 8.kaynaklar meşgul durumdadır.
- İşletim sistemi, disk bloklarının durumunu tutmak için bitmap kullanır.
- **İşletim sistemleri kernel algoritmalarında veri yapılarını sıkılıkla kullanır.**

64

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- **Hesaplama ortamları**
- Açık kaynak işletim sistemleri

65

Hesaplama ortamları

Geleneksel hesaplama

- Birkaç yıl öncesine kadar ofisteki bir bilgisayar ağa bağlanmakta, yazıcı veya diğer kaynakları kullanmaktadır.
- Günümüzde **Web teknolojileri** ve **WAN (Wide Area Network)** geleneksel hesaplama ortamlarının sınırlarını genişletmiştir.
- Firmalar **portal** oluşturmaktak ve Web erişimiyle istemcilere kaynaklara erişim sağlamaktadır.
- **Mobil cihazlar kablosuz ağlar (wireless networks)** ile **Web portal' e bağlanırlar.**
- Konut kullaclarının bant genişliği günümüzde hala yeterli düzeyde değildir, ayrıca ağlarının güvenliği için **firewall** kullanırlar.
- Firewall, **IP filtreleme, port filtreleme, Web filtreleme, içerik filtreleme** yapabilir.

66

Hesaplama ortamları

Mobil hesaplama

- **Mobil hesaplama (mobile computing)**, akıllı telefonlar ve **tablet bilgisayarlar ile yapılan işlemleri ifade eder.**
- Mobil cihazların özellikleri (ekran boyutu, hafıza kapasitesi ve performansı) son yıllarda önemli ölçüde gelişmiştir.
- Günümüzde **mobil cihazlar** sadece Web ve e-posta uygulamaları için değil, **tüm işlemler için kullanılır hale gelmiştir.**
- Mobil ortamlar için günümüzde **Apple iOS** ve **Google Android** işletim sistemleri yaygın olarak kullanılmaktadır.

67

Hesaplama ortamları

Dağıtık sistemler

- Bir dağıtık sistem, **fiziksel olarak ayrı, heterojen bilgisayar sistemidir.**
- Bir dağıtık sistem, **sahip olduğu çok sayıdaki kaynağı kullanıcıların erişimini sağlar.**
- **Bazı işletim sistemleri, sadece dosya erişimine yönelik** ve **FTP (File Transfer Protocol)** ve **NFS (Network File System)** protokollerini kullanırlar.
- **Bazı işletim sistemleri ise ağ fonksiyonlarını** (kullanıcı yönetimi, kaynak atama, ...) kullanıcıların kullanmasına izi verir.
- Bir ağ (**network**), iki veya daha fazla sistemin iletişimini sağlar.
- **Ağlar kullandıkları protokollere göre farklılık gösterirler.**
- Günümüzde **TCP/IP (Transmission Control Protocol/Internet Protocol)** en yaygın kullanılan protokol yığınıdır.

68

Hesaplama ortamları

Dağıtık sistemler

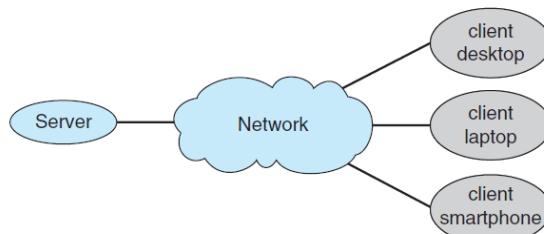
- Bilgisayar ağları kapsadıkları alana göre, **LAN (local-area network)**, **MAN (metropolitan-area network)** ve **WAN (wide-area network)** olarak üç gruba ayrılır.
- Bir bilgisayar ile laptop veya akıllı telefon arasında oluşturulan ağ, **PAN (personnel-area network)** olarak adlandırılır.
- **WLAN (wireless LAN)** ise IEEE 802.11 ve **Bluetooth** teknolojileriyle oluşturulan yaklaşık 300 metre kapsama alanına sahip ağdır.
- Bir ağ işletim sistemi, ağdaki kaynakların yönetimini ve farklı bilgisayarlar üzerinde çalışan process'ler arasında iletişimini sağlar.

69

Hesaplama ortamları

İstemci-sunucu mimarisi

- Günümüzde **birçok sunucu (server) sistemi, istemci (client) bilgisayarların isteklerini karşılamak üzere tasarlanır.**
- **Compute-server** sistemlerinde, **istemciler bir işlemin yapılması için arayüz üzerinden istek gönderirler.**
- **File-server** sistemlerinde, **istemciler dosya oluşturma, silme veya okuma işlemlerini yapabilirler.**
- Sunucu, **yüksek konfigürasyona, istemci ise düşük konfigürasyona sahiptir.**

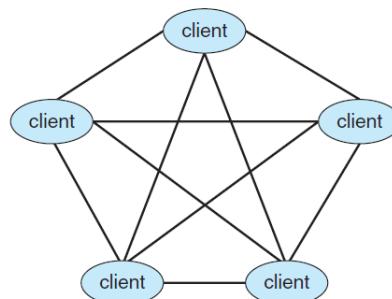


70

Hesaplama ortamları

Peer-to-peer mimarisi

- **Eş düzey (peer-to-peer, P2P) mimarisinde istemci ve sunucu ayrimı yoktur.** Tüm birimler aynı işlem kapasitesine ve yetkisine sahiptir.
- Kaynaklara erişim de dağıtık bir şekilde gerçekleştirilir.
- Napster, Gnutella gibi dosya paylaşım servisleri P2P mimarisine sahiptir.
- **VoIP (voice over IP)** teknolojisi kullanan **Skype**, P2P mimarisine sahiptir.



71

Hesaplama ortamları

Sanallaştırma

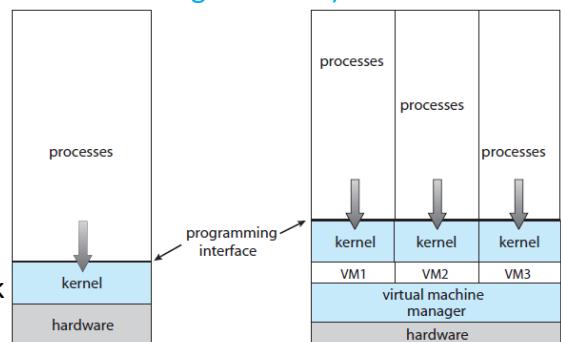
- **Sanallaştırma (virtualization)**, işletim sistemlerinin uygulamaları başka işletim sistemlerinde çalıştırmasına izin verir.
- Sanallaştırma, **emülatör** olarak adlandırılan yazılımı içerir.
- Emülatörler, kaynak CPU ile hedef CPU'nun farklı olduğu durumlarda kullanılır.
- Apple, **IBM CPU için derlenen bir programı Intel CPU'da çalıştırmak isterse**, Rosetta isimli emülatörü (dynamik binary çevirici) kullanır.
- **Yorumlayıcılar (interpreter)**, emülatör yazılımlarıdır ve yüksek seviyeli dilde编写的程序将被转换为低级的机器语言代码(native code)而无需进行编译。它们直接在运行时解释并执行代码。因此，它们通常比编译器慢，但更易于实现和维护。
- Basic, derleme de yapabilir, yorumlama da yapabilir. Java, her zaman yorumlayıcıdır. **JVM (Java Virtual Machine)** bir emülatör yazılımıdır.

72

Hesaplama ortamları

Sanallaştırma

- **VMware**, bir işletim sistemi üzerinde farklı işletim sistemlerinin misafir (**guest copy**) olarak çalışmasına ve kendi uygulamalarını çalıştırmasına izin verir.
- Şekilde Windows host işletim sistemi, VMware uygulaması ise sanal makine yöneticisidir (**virtual machine manager - VMM**).
- **VMM, farklı işletim sistemlerini çalıştırır**, kaynak kullanımlarını yönetir ve kullanıcıların birbirini etkilemesini önler.
- **VMware ESXi** ve **Citrix XenServer**, host olarak çalışır.



73

Hesaplama ortamları

Bulut bilişim

- **Bulut bilişim (cloud computing)**, hesaplama, depolama ve uygulamaları bir ağ aracılığıyla servis olarak dağıtır.
- Bulut hesaplama, sanallaştımanın mantıksal uzantısı olarak kabul edilebilir.
- **Amazon Elastic Compute Cloud (EC2)**, **binlerce sunucuya, milyonlarca sanal makineye ve petabyte depolama alanına sahiptir**.
- EC2 bu kaynakları internet üzerinden kullanıcılar sunar ve **kullanıcılar kullandıkları kaynak oranında aylık ücretlendirilirler**.

74

Hesaplama ortamları

Bulut bilişim

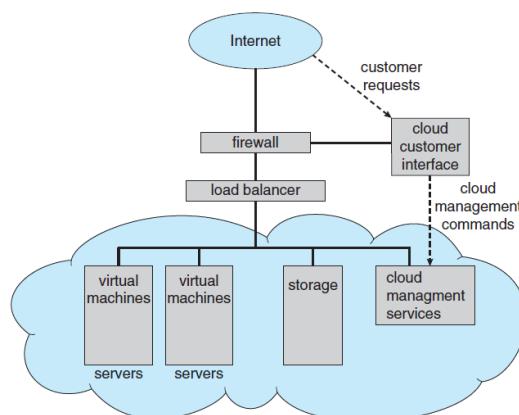
- Farklı bulut hesaplama türleri vardır:
 - **Public cloud** – İnternet üzerinden herkesin kullanımına açıktır.
 - **Private cloud** – bir firmanın sahibi olduğu buluttur.
 - **Hybrid cloud** – public ve private kullanılan bulut bileşenlerine sahiptir.
 - **Software as a service (SaaS)** – bir veya daha fazla uygulama (Word, Excel, ...) İnternet aracılığıyla kullanıma açıktır.
 - **Platform as a service (PaaS)** – Bir yazılım yığını (veritabanı sunucusu) uygulamalar için İnternet aracılığıyla kullanıma açıktır.
 - **Infrastructure as a service (IaaS)** – Sunucular veya depolama birimleri (üretilen verinin yedeklenmesi) İnternet aracılığıyla kullanıma açıktır.
- Bir bulut ortamı yukarıdaki türlerden birden fazlasını sağlayabilir.

75

Hesaplama ortamları

Bulut bilişim

- **Bulut hesaplama**, VMM yapısının ötesinde, kullanıcı process'lerinin çalıştığı **sanal makineleri yönetir**.
- VMM'ler bulut yönetim araçları (**Vware vCloud Director, Eucalyptus**) ile yönetilirler.



75

Hesaplama ortamları

Gerçek zamanlı gömülü sistemler

- Gömülü (**embedded**) sistemler, günümüzde **araç motorları**, **üretim robotları**, **mikro dalga fırınlar**, **uçaklar**, **teknolojik silahlar**, ..., gibi çok farklı yerlerde kullanılmaktadır.
- **Gömülü sistemler özel amaçlar için geliştirilirler ve sahip oldukları işletim sistemleri sınırlı özelliklere sahiptir.**
- Genellikle **arayüz gerektirmezler**, **donanımların izlenmesi** ve yönetimini gerçekleştirirler. **Farklı türleri vardır:**
 - İşletim sistemine sahiptirler ve özel amaçlı uygulamaları çalıştırırlar.
 - Özel amaçlı gömülü işletim sistemine sahiptirler ve istenen işlevleri yerine getirir.
 - Uygulamaya özel bütünsel devreye (**application specific integrated circuits - ASICs**) sahiptirler ve işletim sistemine sahip değildirler. Bu tür sistemler **hardwired systems** olarak da adlandırılırlar.

77

Hesaplama ortamları

Gerçek zamanlı gömülü sistemler

- **Gömülü sistemler, gerçek zamanlı işletim sistemlerini (**real-time operating systems**) çalıştırırlar.**
- **Gerçek zamanlı işletim sistemlerinde zaman gereksinimi çok hassastır.**
- **İstenen zaman aralıklarında veya belirlenen anda işlemlerin** gerçekleştirilmesi zorunludur.
- **Otomobil enjeksiyon sistemleri, medikal uygulamalar, silah sistemleri** başlıca uygulama alanlarıdır.

78

Konular

- İşletim sistemi ne iş yapar?
- Bilgisayar sistemi organizasyonu
- Bilgisayar sistemi mimarisi
- İşletim sistemi yapısı
- İşletim sistemi işlemleri
- Process yönetimi
- Memory yönetimi
- Storage yönetimi
- Koruma ve güvenlik
- Kernel veri yapıları
- Hesaplama ortamları
- **Açık kaynak işletim sistemleri**

79

Açık kaynak işletim sistemleri

- Açık kaynak ([open-source](#)) işletim sistemlerinde, derlenmiş binary kod yerine **kaynak kodu da kullanılabilir durumdadır.**
- **Linux** açık kaynak işletim sistemlerine, **Windows** ise kapalı kaynak ([closed-source](#)) işletim sistemlerine örnek olarak verilebilir.
- **Apple Mac OS X** ve **iOS** hibrit işletim sistemleridir. Açık kaynak kernel' a ([Darwin](#)) sahiptir aynı zamanda kapalı kaynak bileşenlere de sahiptir.
- Kapalı kaynak işletim sistemlerinde **tersine mühendislik** ([reverse engineering](#)) kullanılarak binary kod oluşturulabilir.
- **Açık kaynak işletim sistemlerinde programcılar geliştirmeye katkı sağlayabilmektedirler.**
- **Açık kaynak**, kapalı kaynağa göre **daha güvenlidir**. Çünkü daha çok kişi tarafından kod görülmektedir.
- **Linux, BSD Unix** ve **Solaris** açık kaynak işletim sistemleridir.

80



Ödev

- İşletim sistemlerinin kernel fonksiyonları hakkında araştırma ödevi hazırlayınız.

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- Sistem çağrıları
- Sistem çağrı türleri
- Sistem programları
- İşletim sistemi tasarıımı ve gerçekleştirmi
- İşletim sistemi yapısı
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- Sistem boot

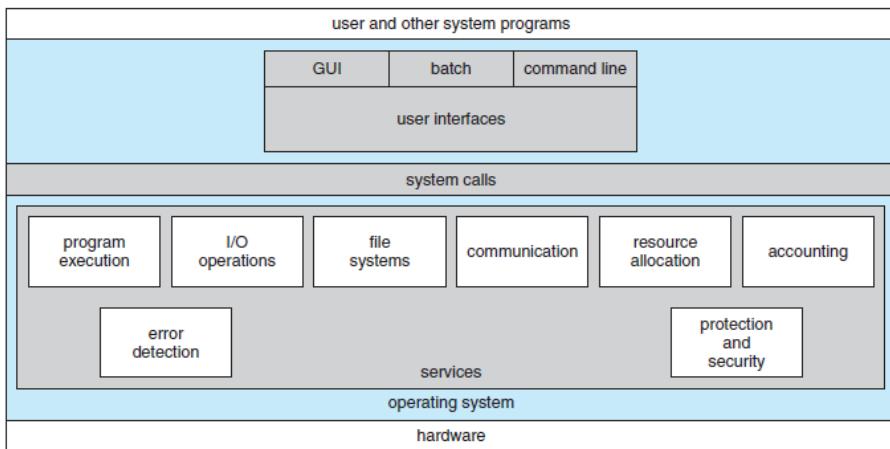
İşletim sistemi servisleri

- İşletim sistemi uygulama programlarının programlarının çalışması için ortam sağlar.
- İşletim sistemleri, sağladığı servisler, arayüzler , sunduğu bileşenler ve bileşenler arasındaki ilişkilerle değerlendirilirler.
- İşletim sistemleri, programcılar programları kolay bir şekilde geliştirilebilmeleri için servisleri ve bileşenleri sağlar.
- İşletim sistemlerinin sağladığı servisler farklılık gösterir.

3

İşletim sistemi servisleri

- İşletim sistemleri, ortak servislerle birlikte **kendine özgü servislere de sahiptir.**



4

İşletim sistemi servisleri

- Kullanıcıya yardımcı olmak için sağlanan fonksiyonlar:
 - **Kullanıcı arayüzü:** Tüm işletim sistemleri kullanıcı arayüzüne sahiptir. Komut satırı arayüzü ([command-line interface - CLI](#)), **metin** komutları alır. Batch arayüzü ([batch interface](#)), **komutlar** dosya içerisinde sağlanır. Grafik kullanıcı arayüzü ([graphical user interface - GUI](#)), en **esnek** ve yaygın kullanıldır.
 - **Program çalıştırma:** Sistem bir programı hafızaya **yükleme**, **çalıştırma** ve **sonlandırma** (hatalı veya hatasız) işlemlerini yapar.
 - **I/O işlemleri:** Programlar I/O cihazlarına ihtiyaç duyabilirler. Kullanıcılar koruma ve etkinlik için doğrudan I/O cihazlarını kullanamazlar. İşletim sistemleri, **I/O cihazları için gerekli işlevleri sağlarlar**.
 - **Dosya sistemi işlemi:** Programlar dosyalara veya dizinlere erişim, okuma ve yazma yapmak isteyebilirler. İşletim sistemleri **dosya erişimlerini düzenleyen işlevleri sağlarlar**.

5

İşletim sistemi servisleri

- Kullanıcıya yardımcı olmak için sağlanan fonksiyonlar:
 - **İletişim:** Bir process başka bir process'le haberleşmeye ihtiyaç duyabilir. Bu aynı bilgisayardaki process'ler arasında veya ağdaki bilgisayarlardaki process'ler arasında olabilir. **İletişim paylaşılmış hafıza** ([shared memory](#)) alanlarında mesaj göndererek yapılabilir.
 - **Hata denetimi:** İşletim sistemi oluşan **hataları sürekli izlemek, raporlamak ve mümkünse düzeltmek zorundadır**.

6

İşletim sistemi servisleri

- Sistemin etkin çalışması için kullanılan fonksiyonlar:
 - **Kaynak tahsis:** Çok sayıda kullanıcı veya iş aynı anda çalışır ve kaynakların bunlara tahsis edilmesi gereklidir.
Bu **kaynaklar, CPU time, hafıza, dosya, I/O cihazları** olabilir.
 - **Hesap oluşturma:** İşletim sistemi, kaynakların kullanımını kullanıcı bazında tutabilir.
Bu **ücretlendirme** veya **istatistiksel çalışma** için gerekli olabilir.
 - **Koruma ve güvenlik:** Sistem kaynaklarına **kullanıcıların erişimi denetlenir** ve kullanıcı bazında **yetkilendirilir**.

7

Konular

- İşletim sistemi servisleri
- **Kullanıcı ve işletim sistemi arayüzü**
- Sistem çağrıları
- Sistem çağrı türleri
- Sistem programları
- İşletim sistemi tasarıımı ve gerçekleştirmesi
- İşletim sistemi yapısı
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- Sistem boot

8



Kullanıcı ve işletim sistemi arayüzü

- İşletim sistemleri, kullanıcılarla iki farklı arayüz sağlarlar:
 - Command-line interface ([command interpreter](#))
 - Kullanıcı doğrudan komutları yazar.
 - Kullanımı ve öğrenmesi zordur.
 - Graphical user interface ([GUI](#))
 - Kullanıcı farklı bileşenlerle komutları oluşturabilir.
 - Kullanımı ve öğrenmesi kolaydır.

9



Kullanıcı ve işletim sistemi arayüzü

Command interpreter

- Bazı işletim sistemleri, **command interpreter’ı kernel kısmında bulundurur**.
- **Windows** ve **UNIX** gibi işletim sistemleri ise **command interpreter’ı ayrı bir program olarak bakar**.
- Bazı işletim sistemleri ise, **birden fazla command interpreter’ı sahiptir**.
- **Interpreter’lar shell olarak adlandırılır**.
- Kullanıcılar, UNIX ve Linux’da Bourne Shell, C Shell, Korn Shell, ... gibi farklı kabuklardan birisini seçebilirler.
- **Command interpreter** tarafından **create, delete, list, print, copy, execute, ...** işlemleri yapılır.
- Command interpreter, komutlara ait **kodlara sahip olabilir** veya her komut için **ayıri sistem programı** olabilir.

10



Kullanıcı ve işletim sistemi arayüzü

GUI

- Bazı işletim sistemlerinin arayüzü, menülere sahiptir ve mouse tabanlı işlemlere izin veren **desktop** şeklinde oluşturulmuştur.
- Programlara, dosyalara, dizinlere, sistem fonksiyonlarına **icon** kullanılarak erişilebilir veya programlar çalıştırılabilir.
- Herhangi bir dizin (**folder**) içerisindeyken mouse ile menüye ulaşılıp işlem yapılabilir.
- **İlk GUI, 1973 yılında Xerox Alto bilgisayarda kullanılmıştır.**
- İlk yaygın kullanılan GUI, Apple Macintosh bilgisayarlarda 1980'li yıllarda kullanılmaya başlanmıştır.
- Microsoft'un **Windows 1.0** işletim sistemi MS-DOS işletim sistemine arayüz eklenerek geliştirilmiştir.
- **Mobil cihazlar, parmak ile kullanılan arayzlere sahiptir.**

11



Konular

- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- **Sistem çağrıları**
- Sistem çağrı türleri
- Sistem programları
- İşletim sistemi tasarıımı ve gerçekleştirmesi
- İşletim sistemi yapısı
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- Sistem boot

12

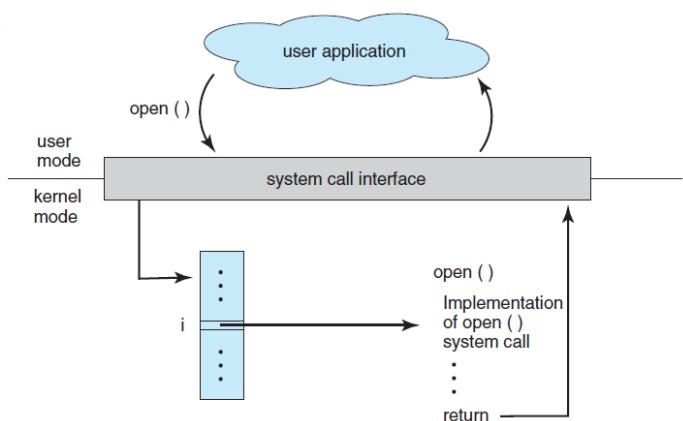
Sistem çağrıları

- Sistem çağrıları, servislerin işletim sistemi tarafından kullanılabilmesi için arayüz sağlarlar.
- Bu çağrıların kullandığı program blokları genellikle C veya C++ ile yazılır. Düşük seviyeli işlemler (**dolandırma erişimi**) için assembly dili kullanılır.
- Bir dosyayı açıp (sistem çağrısı), içeriğini okumak (sistem çağrısı) ve kapatmak (sistem çağrısı) ayrı ayrı sistem çağrıları gerektir.
- İşlemler sırasında oluşan hatalar (dosya bulunamadı, okuma hatası, ...) sistem çağrıları tarafından yönetilir.
- **Uygulama geliştirici, işletim sistemindeki API** (application programming interface) (Windows API, POSIX API, Java API) tarafından sağlanan **fonksiyonları** (dosya aç, oku, yaz, ...) kullanır.
- **Programcı**, bu fonksiyonlara programlama dilinin sağladığı **kütüphaneler** aracılığıyla erişir.

13

Sistem çağrıları

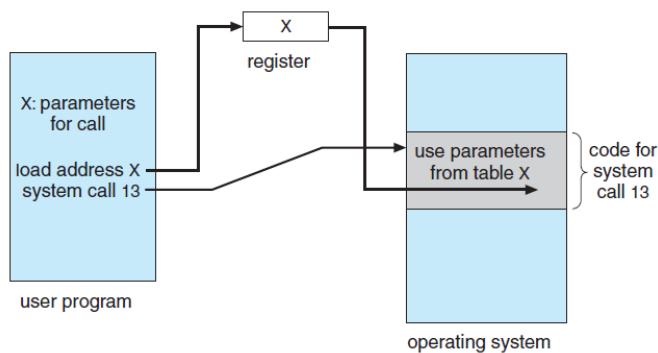
- İşletim sistemi fonksiyonları, kernel'daki sistem çağrılarını başlatır.
- Windows'daki CreateProcess() fonksiyonu, Windows kernel'daki NTCREATEPROCESS() sistem çağrımasını çalıştırır.
- Programlama dilleri, compiler tarafından sağlanan **kütüphaneler** ile sistem çağrılarına erişmeyi kolaylaştıran arayüz sağlarlar.



14

Sistem çağrıları

- Sistem çağrılarına veri girişi gerekebilir (dosya adı, giriş cihazı, ...).
- Üç farklı yöntemle işletim sistemine parametre gönderilebilir:
 - Register'lar kullanılabilir.
 - Hafızada blok veya tablo kullanılabilir.
 - Stack kullanılabilir.



15

Konular

- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- Sistem çağrıları
- **Sistem çağrı türleri**
- Sistem programları
- İşletim sistemi tasarıımı ve gerçekleştirmesi
- İşletim sistemi yapısı
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- Sistem boot

16

Sistem çağrı türleri

- Sistem çağrıları 6 grup altında sınıflandırılabilir:
 - Process control
 - File management
 - Device management
 - Information maintenance
 - Communications
 - Protection
- **İşletim sistemlerinde** desteklenen sistem çağrılarında farklılık olabilir, ancak **gerçekleştirilen işlevler aynıdır.**

17

Sistem çağrı türleri

Process control

- Bir program çalışmasını **normal bir şekilde** (`end()`) veya **beklenmeyen şekilde** (`abort()`) bitirebilir.
- Eğer bir sistem çağrısı programın çalışmasını sonlandırırsa, bir **hata mesajı** üretilir ve **hafızanın dökümü diske kaydedilir.**
- Hafıza dökümü **debugger** programı tarafından incelenir ve hataya neden olan **bug** düzeltılır.
- Bir programın çalıştırılması için **load()** ve **execute()** işlemleri yapılır.

18

Sistem çağrı türleri

- Sistem çağrı türleri ve gerçekleştirilen işlevler aşağıda verilmiştir.

<ul style="list-style-type: none">• Process control<ul style="list-style-type: none">◦ end, abort◦ load, execute◦ create process, terminate process◦ get process attributes, set process attributes◦ wait for time◦ wait event, signal event◦ allocate and free memory• File management<ul style="list-style-type: none">◦ create file, delete file◦ open, close◦ read, write, reposition◦ get file attributes, set file attributes	<ul style="list-style-type: none">• Device management<ul style="list-style-type: none">◦ request device, release device◦ read, write, reposition◦ get device attributes, set device attributes◦ logically attach or detach devices• Information maintenance<ul style="list-style-type: none">◦ get time or date, set time or date◦ get system data, set system data◦ get process, file, or device attributes◦ set process, file, or device attributes• Communications<ul style="list-style-type: none">◦ create, delete communication connection◦ send, receive messages◦ transfer status information◦ attach or detach remote devices
---	---

Sistem çağrı türleri

Process control

- Windows ve UNIX için örnek sistem çağrıları tabloda verilmiştir.

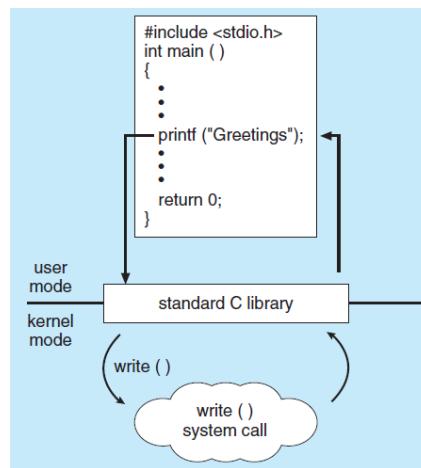
		Windows	Unix
■ Windows ve UNIX için örnek sistem çağrıları tabloda verilmiştir.	Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
	File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
	Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
	Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
	Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
	Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

20

Sistem çağrı türleri

Process control

- Bir C programının `printf()` deyimini standart C kütüphanesi ile çalıştırması için **sistem çağrısını kullanması gereklidir.**



21

Sistem çağrı türleri

Process control

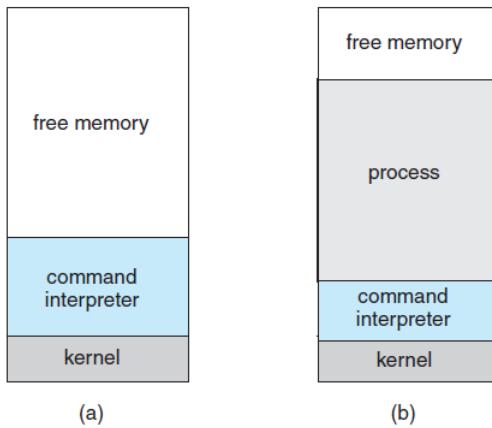
- Birden fazla program aynı veriyi paylaşabilir. Bu durumda kullanmakta olan **process veriyi kilitler (lock)** ve **diğer process'ler erişemez**.
- Kilitleme için `acquire lock()`, serbest bırakmak için `release lock()` sistem çağrıları kullanılır.
- Process kontrol işlemleri**, tek görevli (`single-tasking`) ve çok görevli (`multiple-tasking`) sistemlerde farklı gerçekleştirilir.
- MS-DOS (Microsoft-Disk Operating System)**, single-tasking işletim sistemidir ve **bir process çalışırken yeni process başlatılamaz**.
- MS-DOS'da bir program** hafızaya yerleştirilir ve instruction pointer (program counter register) ile **ilk komut gösterilerek çalıştırılır**.

22

Sistem çağrı türleri

Process control

- MS-DOS ile (a) başlangıç durumu ve (b) programın çalıştırılması şekilde görülmektedir.

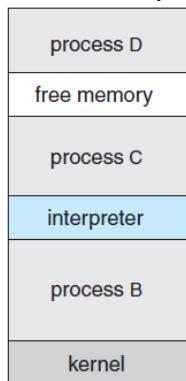


23

Sistem çağrı türleri

Process control

- [FreeBSD \(Berkeley Software Distribution\)](#), multi-tasking işletim sistemidir ve bir process çalışırken yeni process başlatılabilir.
- **Çok görevli işletim sistemlerinde, command interpreter** bir program çalışırken de **sürekli çalışmaktadır** ve yeni process'ler başlatabilir.



24

Sistem çağrı türleri

File management

- Dosya işlemleri de sistem çağrıları tarafından gerçekleştirilir.
- Dosya oluşturmak ve silmek için **create()** ve **delete()** sistem çağrıları kullanılır.
- Var olan dosyayı açmak için **open()**, dosyayı kapatmak için **close()** sistem çağrıları kullanılır.
- Dosyadan okuma yapmak veya dosyaya yazmak için **read()** ve **write()** sistem çağrıları kullanılır.
- Dosyaların özellikleri ve erişim haklarına erişmek ve değiştirmek için **get file attributes()** ve **set file attributes()** sistem çağrıları kullanılır.
- Dosyaların taşınması veya kopyalanması için **copy()** ve **move()** sistem çağrıları kullanılır.

25

Sistem çağrı türleri

Device management

- Bir **process**, çalışması sırasında **farklı kaynaklara ihtiyaç duyabilir**.
- İşletim sisteminin kontrol ettiği **tüm kaynaklar cihaz olarak düşünülebilir**. Bunlar **fiziksel** kaynaklar (disk sürücüler) veya **sanal** cihazlar (dosya) olabilir.
- Bir kaynağa erişim isteği **request()** ile işinin tamamlandığı ise **release()** sistem çağrıları ile bildirilir.
- Bir kaynağa erişim hakkı alındığında, **read()** veya **write()** sistem çağrıları ile okuma ve yazma işlemleri gerçekleştirilir.

26

Sistem çağrı türleri

Information maintenance

- Birçok sistem çağrısı **kullanıcı programı** ile **sistem çağrıları** arasında **veri transferi** yapmak için kullanılır.
- Örneğin **time()** ve **date()** sistem çağrıları kullanıcı programına anlık saat ve tarih bilgilerini aktarır.
- Diğer sistem çağrıları, **anlık kullanıcı sayısı**, **işletim sistemi versiyonu**, **boş hafıza** veya **disk alanı** gibi sisteme ait bilgileri aktarır.
- **dump()** gibi bazı sistem çağrıları ise bir programın debug aşamasında faydalıdır.
- Çoğu işletim sistemi, çalışan programları periyodik aralıklarla izler ve durumunu saklar. **Bunun için timer interrupt'ları kullanılır.**

27

Sistem çağrı türleri

Communication

- Process'ler arasında iletişim için iki yöntem kullanılır:
 - **Message-passing model**
 - Mesajlar process'ler arasında doğrudan veya dolaylı (mesaj kutusu) gönderilebilir.
 - Her process'in bir process adı ve ID değeri vardır.
 - **open connection()** and **close connection()** sistem çağrıları kullanılır.
 - Her process gelen mesajları kabul etmek için **accept connection()** sistem çağrımasını kullanır.
 - **Kaynak process istemci (client)**, **hedef process sunucusu (server)** olarak adlandırılır.
 - **Shared-memory model**
 - Hafıza alanı oluşturmak için **shared memory create()** ve erişmek için **shared memory attach()** sistem çağrıları kullanılır.
 - **İşletim sistemi**, paylaşılan alanın yönetimini yapar, veri içeriğini kontrol etmez.

28

Sistem çağrı türleri

Protection

- Protection, bilgisayar sistem kaynaklarına erişimi kontrol eden mekanizmaları sağlar.
- Protection, çok kullanıcılı ve multiprogramming sistemler için geliştirilmiştir.
- Günümüzde, mobil cihazlar da dahil tüm cihazlar için protection gereklidir.
- Sistemdeki kaynaklara erişim yetkilendirmesi için **set permission()** ve **get permission()** sistem çağrıları kullanılır.
- İzin vermek veya izni kaldırmak için **allow user()** ve **deny user()** sistem çağrıları kullanılır.

29

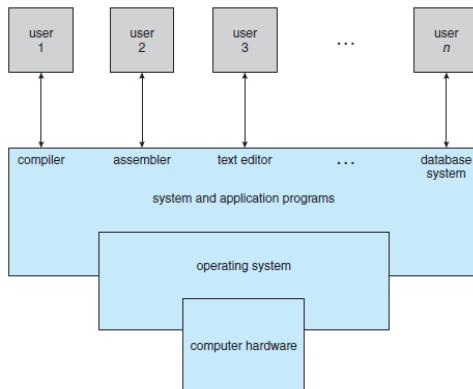
Konular

- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- Sistem çağrıları
- Sistem çağrı türleri
- **Sistem programları**
- İşletim sistemi tasarıımı ve gerçekleştirmi
- İşletim sistemi yapısı
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- Sistem boot

30

Sistem programları

- Modern bilgisayar sistemlerinde **en alt seviyede donanım** vardır.
- Donanımın üzerinde **işletim sistemi** çalışır.
- İşletim sisteminin üzerinde ise **sistem programları** vardır.
- En üstte ise **uygulama programları** vardır.



31

Sistem programları

- Sistem programları (**system utilities**), **program geliştirme** ve **çalıştırma** için kolay bir **platform** sağlar.
 - **File management**
 - Dosya ve dizinlerde, **create**, **delete**, **copy**, **rename**, **print**, **list**, ... gibi işlemleri gerçekleştirir.
 - **Status information**
 - Bazı programlar sistemden **tarih**, **saat**, **hafıza** ve **disk boş alanı**, **kullanıcı sayısı** gibi bilgileri ister.
 - Daha karmaşık programlar ise, **performans**, **log**, **debug** gibi bilgileri ister.
 - Bazı sistemler sistem konfigürasyonunu saklayan **registry** yapısına sahiptir.
 - **File modification**
 - Disk veya diğer saklama birimlerindeki **dosyaların içeriğini görüntüleme** ve **değiştirme** amacıyla kullanılan **metin editörleri** vardır.
 - **Programming-language support**
 - Programlama dilleri (C, C++, Java, Perl) için **compiler**, **assemblers**, **debugger**, **interpreter** sağlarlar veya ayrıca download edilerek kullanılmasına izin verirler.

32

Sistem programları

- **Program loading / execution**
 - Bir programı derledikten sonra **hafızaya yükleyip çalıştırmak için** kullanırlar.
 - Makine dilinde veya yüksek seviyeli dilde **debug amacıyla için programlar kullanılabilir.**
- **Communications**
 - Bu sistem programları, **process'ler, kullanıcılar ve bilgisayarlar arasında bağlantı kurulması** sağlarlar.
 - Kullanıcılar arasında **mesajla gönderme, Web sayfalarında gezinti, e-posta gönderme, uzak bağlantı** veya **dosya aktarımları** yapmayı sağlarlar.
- **Background services**
 - Tüm genel amaçlı **sistemler boot sırasında bazı sistem programlarını başlatırlar.**
 - Bu programların **bazları** sistem boot edildikten sonra **sonlandırılır, bazıları ise** sistem çalıştığı sürece **çalışmasını sürdürür.**
 - Bu tür sistem programları, **service, subsystem veya daemon** olarak adlandırılır.
 - **Ağ bağlantısı, yazıcı işlemleri** veya **sistem hata izleme** servislerini sistem programları sağlarlar.

33

Konular

- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- Sistem çağrıları
- Sistem çağrı türleri
- Sistem programları
- **İşletim sistemi tasarıımı ve gerçekleştirmesi**
- İşletim sistemi yapısı
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- Sistem boot

34



İşletim sistemi tasarıımı ve gerçekleştirmi

Tasarım hedefleri

- En üst düzeyde **bir işletim sisteminin tasarım hedefleri, donanım özellikleri ile sistem türünü** (single user, multiuser, gerçek zamanlı, dağıtık, özel amamçılı, genel amaçlı, ...) **belirler**.
- Gereksinimler, **kullanıcı hedefleri (user goals)** ve **sistem hedefleri (system goals)** olarak iki gruba ayrılır.
- Kullanıcı, sistemin kolay ve rahat kullanılmasını, kolay ve hızlı öğrenilmesini, güvenilir, güvenli ve hızlı olmasını ister.
- **Kullanıcı gereksinimlerine göre sistem gereksinimlerinin belirlenmesi ve bu hedeflerin gerçekleştirilmesi gereklidir.**
- Bir işletim sisteminin tasarıımı ile ilgili genel prensipler, **software engineering** alanındaki çalışmalarla belirlenmiştir.

35



İşletim sistemi tasarıımı ve gerçekleştirmi

Mekanizmalar ve kurallar

- Kurallar (**policies**) **ne yapılacağını**, mekanizmalar (**mechanisms**) **nasıl yapılacağını** belirler.
- Örneğin, **CPU'nun korunması için timer kullanımı mekanizmadır**, bir kullanıcı için **timer süresinin ne kadar olacağına karar vermek kuraldır**.
- Kurallar, kaynak ayrılmasına yönelik kararlarda oldukça önemlidir.

36



İşletim sistemi tasarımları ve gerçekleştirmeleri

Implementation

- Bir işletim sistemi tasarlandıktan sonra gerçekleştirilme aşamasına geçilir.
- İlk işletim sistemleri **assembly dili** ile yazılmıştır.
- Günümüzde işletim sistemleri C ve C++ gibi **yüksek seviyeli dillerle** yazılmaktadır.
- Kernel kısmı assembly, yüksek seviyeli işlemler C, sistem programları C, C++, Perl, Phyton gibi dillerle yazılmaktadır.
- Linux işletim sisteminde tüm dillerle yazılan kısımlar bulunmaktadır.

37



İşletim sistemi tasarımları ve gerçekleştirmeleri

Implementation

- Bir işletim sisteminin yüksek seviyeli dillerle yapılması, farklı mikroişlemcilerde çalışabilmesini kolaylaştırır.
- MS-DOS, 8088 assembly diliyle yazılmıştır ve sadece Intel X86 işlemcilerde çalışabilir.
- Linux, C ile yazılmıştır ve Intel X86, Oracle SPARC ve IBM PowerPC işlemcilerde çalışabilir.
- Yüksek seviyeli dillerde yazılan işletim sistemleri daha yavaştır ve daha fazla depolama alanına ihtiyaç duyarlar.
- Günümüzdeki **compiler'lar** derleme sırasında **optimizasyon** yapar ve mikroişlemciler **pipelining** gibi teknolojilerle çalışma performansı artırır.

38

Konular

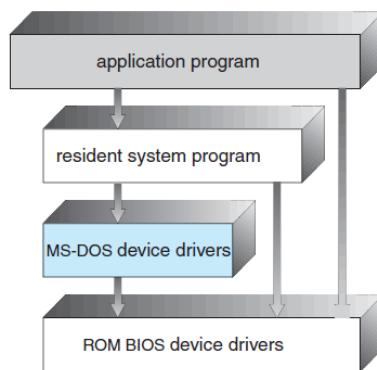
- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- Sistem çağrıları
- Sistem çağrı türleri
- Sistem programları
- İşletim sistemi tasarımları ve gerçekleştirmesi
- **İşletim sistemi yapısı**
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- Sistem boot

39

İşletim sistemi yapısı

Basit yapı

- **Çoğu işletim sistemi, küçük, basit ve sınırlı bir şekilde geliştirilmeye başlar, daha sonra gelişir.** MS-DOS bu şekilde bir işletim sistemidir.
- Başlangıçta az sayıda kişi tarafından geliştirilmiş ve **modüler yapı dikkatli** bir şekilde oluşturulmuştur.



40

İşletim sistemi yapısı

Basit yapı

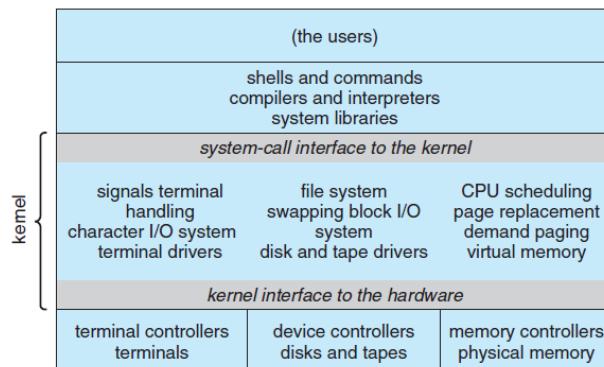
- MS-DOS işletim sisteminde, arayüzler ve işlev seviyeleri iyi bir şekilde oluşturulmamıştır.
- MS-DOS'ta bir **uygulama programı** temel I/O routinlerine erişip display veya **disk sürücülerini** kontrol edebilir.
- MS-DOS'un bu özelliği, kötücül programların sisteme zarar vermesine ve tümüyle çalışmaz hale gelmesine neden olabilmektedir.

41

İşletim sistemi yapısı

Basit yapı

- **UNIX işletim sistemi**, ilk geliştirildiğinde **kernel ve sistem programları** şeklinde **iki parçadan oluşmaktadır**.
- Kernel kısmı arayüzler ve cihaz sürücülerini şeklinde alt parçalara bölünerek gelişmesine devam etmiştir.

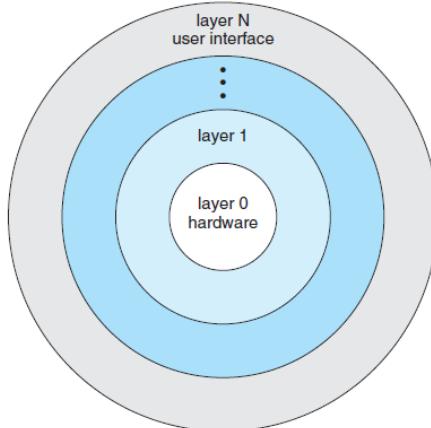


42

İşletim sistemi yapısı

Katmanlı yapı

- **İşletim sistemi, modüler yapıda birden fazla katman halinde geliştirilir.**
- En alt katman donanım, en üst katman ise kullanıcı arayüzüdür.



43

İşletim sistemi yapısı

Katmanlı yapı

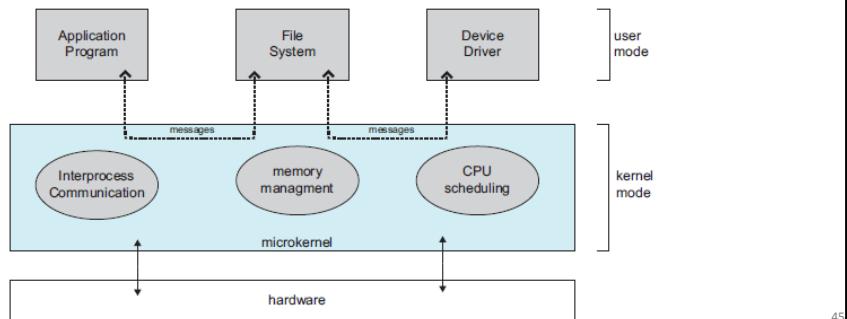
- Katmanlı yapıda, **tasarım ve implementation kolay yapılır.**
- **Her katman ayrı ayrı debug edilip doğrulama yapılabilir.**
- Her katman alt katman tarafından sağlanan işlemler kullanılarak oluşturulur.
- **Katmanlı yapıda her katmanda yapılacak işlevlerin çok iyi planlanması gereklidir.**
- **Katmanlı yapıda, her katman alt katmandaki fonksiyonu çalıştıracağından dolayı performans düşme eğilimindedir.**
- **Katman sayısını olabileceği kadar az olacak şekilde tasarım yapmak performans açısından gereklidir.**

44

İşletim sistemi yapısı

Mikrokernel

- **UNIX** gelişikçe, **kernel** kısmı büyümüş ve yönetimi zorlaşmıştır.
- 1980 yılında Carnegie Mellon Üniversitesinde **mikrokernel** yaklaşımıyla **Mach** adında işletim sistemini geliştirimişlerdir.
- Bu yaklaşımada, **kernel** içerisindeki gereksiz tüm bileşenler sistem programlarına aktarılmıştır.



45

İşletim sistemi yapısı

Mikrokernel

- **Mikrokernel** kısmında, **iletişim bileşenleri, hafıza yönetimi ve küçük process'ler** kalmıştır.
- Process'ler arasındaki iletişim mesaj gönderimi ile yapılmıştır.
- Mikrokernel yapısında **çoğu servis kullanıcı olarak çalıştığından** (kernel servisi olmadığından) **güvenlik daha iyi** sağlanabilmektedir.
- Mac OS X kernel'ı kısmen mikrokernel yapısını kullanmaktadır.
- Windows NT 4.0 kernel yapısına sahiptir.
- Windows XP ile birlikte ağırlıklı olarak **monolithic** yapıya (tek parça) dönülmüştür.

45

İşletim sistemi yapısı

Modül

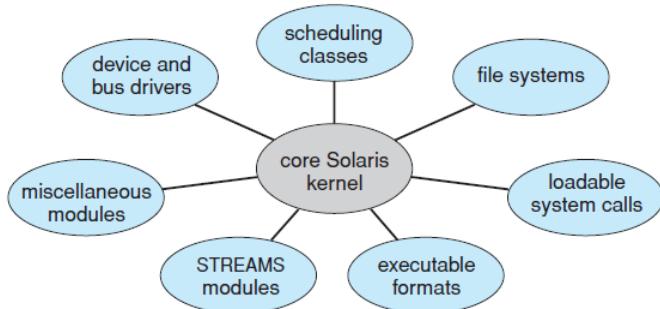
- İşletim sistemi tasarımında günümüzdeki **en iyi teknoloji yüklenebilir kernel modülleri kullanmaktadır.**
- Kernel, **boot sırasında** veya **sistemin çalışması devam ederken yüklenen bileşenlere sahiptir.**
- Modern işletim sistemlerinde, (UNIX, Linux, Mac OS X, Windows) bu tür çalışma yaygındır.
- **CPU yönetimi ve hafıza yönetimi doğrudan kernel kısmındadır**, farklı dosya sistemleri gibi destek bileşenleri yüklenebilir kernel modülleri ile sağlanmaktadır.

47

İşletim sistemi yapısı

Modül

- Bir modül kendisi yüklenikten sonra başka modülleri çağrılabılır veya iletişime geçebilir.
- Linux, **cihaz sürücülerini ve dosya sistemleri için** yüklenebilir kernel modüllerini kullanır.



48

İşletim sistemi yapısı

Hibrit

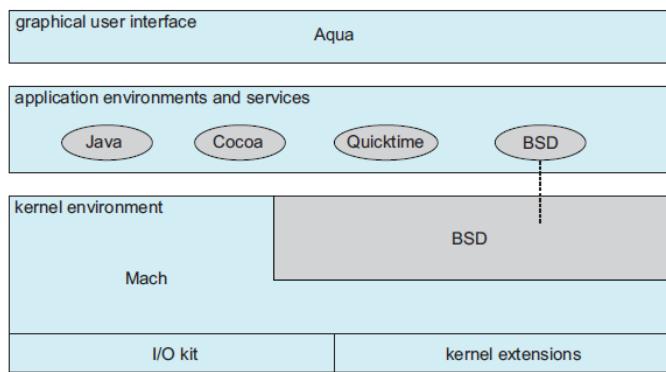
- Günümüzde işletim sistemleri, tek yapıya bağlı bir şekilde geliştirilmemektedir.
- **Farklı yapıları birleştirerek kullanıp; performans, güvenlik ve kullanılabilirliği artırmak amaçlanmıştır.**
- **Linux ve Solaris monolithic yapıdadır**, ancak kernel kısmına yeni fonksiyonlar dinamik olarak eklenebilmektedir.
- **Windows monolithic yapıdadır**, ancak bazı temel işlevleri mikrokernel yapısında destekler.

49

İşletim sistemi yapısı

Mac OS X

- Apple **Mac OS X** işletim sistemi **hibrit yapıya sahiptir**.
- En üst katmanda Aqua kullanıcı arayüzü vardır.
- İkinci katmanda, uygulama geliştirme platformları ve servisler vardır.



50

İşletim sistemi yapısı

Mac OS X

- **Cocoa, Objective C** programlama diline API sağlar.
- **Kernel**, BSD kernel ile Mach kernel'ına sahiptir.
- **Mach**, hafıza yönetimi, process'ler arası iletişim, çağrı başlatma, thread yönetimi gibi işlevleri sağlar.
- **BSD**, command-line, ağ ve dosya sistemleri gibi işlevleri sağlar.
- **I/O kit** cihaz sürücülerini sağlar.
- **Kernel extensions**, yüklenebilir modülleri sağlar.

51

İşletim sistemi yapısı

iOS

- **iOS**, Mac OS X üzerine yapılandırılmıştır ve Apple iPhone, tablet ve iPad ürünlerinde çalışmak üzere tasarlanmıştır.
- **Cocoa Touch**, Objective C programlama dili için API sağlar.
- **Media services**, grafik, video ve ses servislerini sağlar.
- **Core services**, cloud computing ve veritabanı servislerini sağlar.
- **Core OS**, en alt katmandır ve kernel işlevlerini sağlar.

Cocoa Touch

Media Services

Core Services

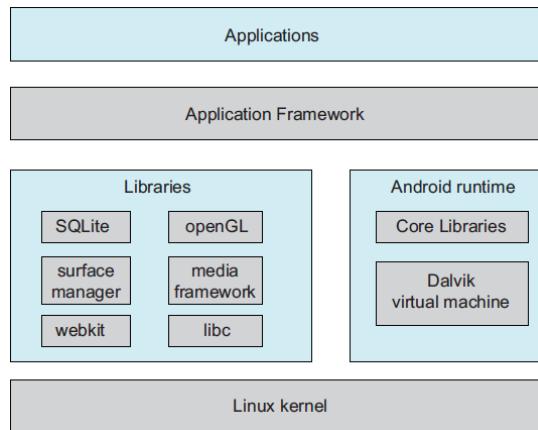
Core OS

52

İşletim sistemi yapısı

Android

- **Google Android**, akıllı telefonlar ve tablet bilgisayarlar için tasarlanmıştır.
- Android işletim sistemi açık kaynaklıdır ve farklı platformlarda çalışabilir.



53

İşletim sistemi yapısı

Android

- En alt katmanda, **Linux kernel** çalışır. **Hafıza yönetimi, process yönetimi, cihaz sürücülerini ve power management işlemleri** için kullanılır.
- **Anroid runtime**, Java programlama diliyle uygulama geliştirilmesi için gerekli kütüphaneleri sağlar.
- **Dalvik virtual machine**, Java executable dosyalarını çalıştırır.
- Kütüphaneler ise farklı uygulamaların geliştirilmesi için gerekli olabilecek bileşenleri bulundurur.
 - Web browser için **webkit**
 - veritabanı işlemleri için **SQLite**
 - multimedia ve oyunlar için **media framework** ve **openGL**
 - standart C kütüphanesine benzeyen **libc**
 - ekran erişimlerinin yönetimi için **surface manager**

54

Konular

- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- Sistem çağrıları
- Sistem çağrı türleri
- Sistem programları
- İşletim sistemi tasarımları ve gerçekleştirmesi
- İşletim sistemi yapısı
- **İşletim sistemi debugging**
- İşletim sistemi üretilmesi
- Sistem boot

55

İşletim sistemi debugging

- **Debugging**, genel olarak sistemdeki hataların bulunması ve giderilmesi işlemlerini ifade eder.
- **Debugging**, bug'lar ile ortaya çıkan performans sorunlarını gidermeyi amaçlar.
- Bu yüzden, **performans ayarı (performance tuning)**, konusunu da içerisine almaktadır.

56



İşletim sistemi debugging

Hata analizi

- Bir process hata verdiğiinde, işletim sistemleri [log](#) dosyasına hata bilgisini kaydedeler.
- İşletim sistemi aynı zamanda hafızanın anlık durumunu da ([memory dump](#)) kaydedebilir.
- Kernel'da ortaya çıkan hata [crash](#) olarak adlandırılır.
- **Debugging, bir kodu yazmaktan çok daha zordur.**

57

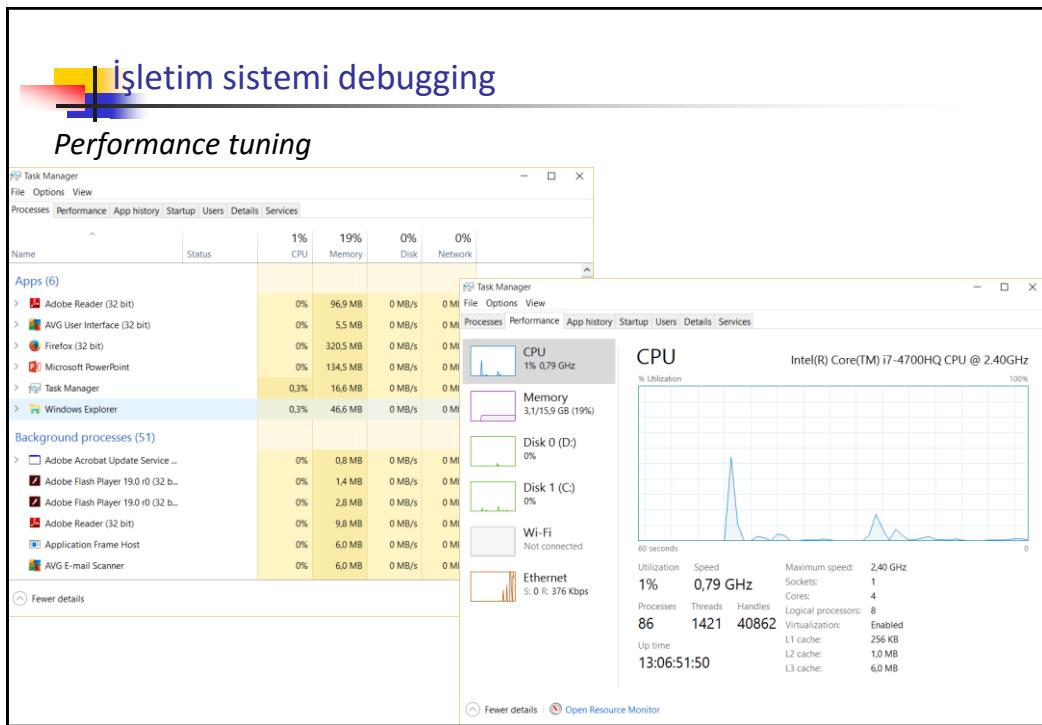


İşletim sistemi debugging

Performance tuning

- Sistem performansını artırmak için yapılan işlemlerdir.
- Sistemdeki tıkanıklıkların monitör edilerek belirlenmesi gereklidir.
- [Windows Task Manager](#), sistemin izlenmesi ve performans için kullanıcı etkileşiminin sağlanması amacıyla kullanılmaktadır.

58



- ## Konular
- İşletim sistemi servisleri
 - Kullanıcı ve işletim sistemi arayüzü
 - Sistem çağrıları
 - Sistem çağrı türleri
 - Sistem programları
 - İşletim sistemi tasarıımı ve gerçekleştirmesi
 - İşletim sistemi yapısı
 - İşletim sistemi debugging
 - **İşletim sistemi üretilmesi**
 - Sistem boot

İşletim sistemi üretilmesi

- **İşletim sistemleri**, farklı makinelerde ve farklı konfigürasyonlarda çalışmak üzere tasarılanırlar.
- İşletim sistemleri, **disklerde**, **CDROM'larda**, **DVD-ROM'larda** veya **ISO image** olarak dağıtırlırlar.
- **İşletim sisteminin yüklenmesi için** ayrı bir yazılım çalıştırılır ve **aşağıdaki bilgileri kullanıcının sağlaması gereklidir:**
 - Kullanılan CPU
 - Disk formatı ve partition'ları
 - Kullanılacak hafıza miktarı (Bazı sistemler hafıza erişimini denetler.)
 - Cihazların interrupt bilgileri (Günümüzdeki çoğu sistem bu bilgileri kendisi oluşturur.)
 - Hangi işletim sisteminin tercih edildiği (donanım kapasitesine göre belirlenebilir.)

61

Konular

- İşletim sistemi servisleri
- Kullanıcı ve işletim sistemi arayüzü
- Sistem çağrıları
- Sistem çağrı türleri
- Sistem programları
- İşletim sistemi tasarıımı ve gerçekleştirmesi
- İşletim sistemi yapısı
- İşletim sistemi debugging
- İşletim sistemi üretilmesi
- **Sistem boot**

62

Sistem boot

- Kernel'ın yüklenerek bilgisayarın başlatılmasına **booting** denilmektedir.
- **Bootstrap program, ROM (Read Only Memory)** üzerindedir ve işletim sisteminin kernel kısmını yükler.
- Bilgisayar reset yapıldığında veya yeni açıldığından **instruction register (program counter - PC)** önceden tanımlı noktaya geçer ve buradan itibaren çalışma başlar.
- **Mobil cihazlarda** işletim sisteminin tamamı **EPROM (erasable programmable read-only memory)** üzerinde saklanır.
- Bu tür **ROM'lar** **firmware** olarak adlandırılır.
- **Büyük işletim sistemlerinde**, **bootstrap** programı **ROM'da**, işletim sistemi ise **disk** üzerinde saklanır.
- Boot kısmına sahip olan disk, **boot disk** veya **system disk** olarak adlandırılır.

63

Ödev

- İşletim sistemlerinde kernel debugging hakkında araştırma ödevi hazırlayınız.

64

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- Process kavramı
- Process planlama
- Process işlemleri
- Process'ler arası iletişim
- İstemci-sunucu sistemlerde iletişim

Process kavramı

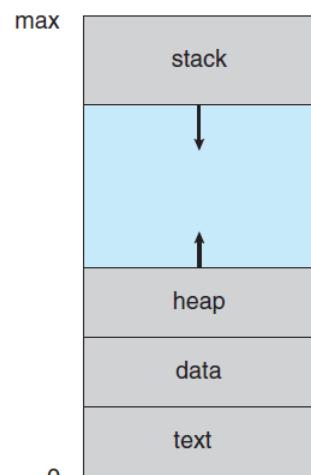
- Günümüz işletim sistemleri birden çok programın hafızaya yüklenmesine ve eşzamanlı çalıştırılmasına izin verir.
- Çalışmakta olan programa **process** denilmektedir.
- **Modern işletim sistemlerinde, process işin bir parçasıdır.**
- **CPU, aralarında geçiş yaparak tüm process'leri eş zamanlı çalıştırılabilir.**
- Bir sistem, tek kullanıcılı bile olsa, birden fazla uygulamayı (Word, Excel, Web Browser, ...) birlikte çalıştırabilir.
- İşletim sistemi multitasking desteklemese bile, işletim sisteminin kendi fonksiyonlarını çağırarak çalıştırır.

3

Process kavramı

Process

- Bir process, yürütme做的 işi, **program counter** değerini, **CPU register'larının değerlerini içermektedir.**
- Bir process aşağıdaki bileşenleri içermektedir:
 - **stack**, fonksiyon parametreleri, return adresleri ve lokal değişkenleri saklar.
 - **data section**, global değişkenleri saklar.
 - **heap**, process'e runtime'da dinamik olarak atanın bellektir.



4

Process kavramı

Process

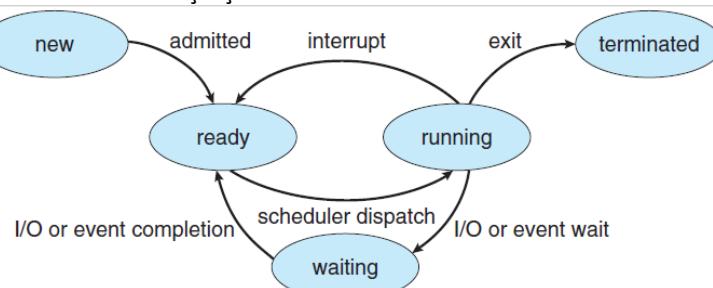
- **Bir program pasif varlıktır (executable file)**, disk üzerinde saklanan komut kümeleridir.
- **Bir process aktif varlıktır**, sonraki çalışacak komut adresini program counter saklar.
- **Aynı program birden fazla process ile ilişkili olabilir** (birden fazla eşzamanlı çalışan Web browser).
- **Her process'in stack, data section ve heap kısımları farklıdır.**

5

Process kavramı

Process state

- **Bir process çalıştığı sürece durum değişir.**
 - **New**: Process oluşturulmaktadır.
 - **Running**: Komutlar çalıştırılmaktadır.
 - **Waiting**: Process bir olayın gerçekleşmesini beklemektedir (I/O, bir cihazdan geribildirim).
 - **Ready**: Process çalışmak için CPU'ya atanmak üzere bekliyor.
 - **Terminated**: Process çalışmasını sonlandırır.

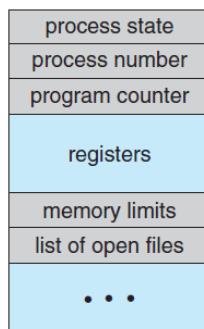


6

Process kavramı

Process control block

- Her process, işletim sisteminde **process control block (PCB)** (veya **task control block**) tarafından temsil edilir.



7

Process kavramı

Process control block

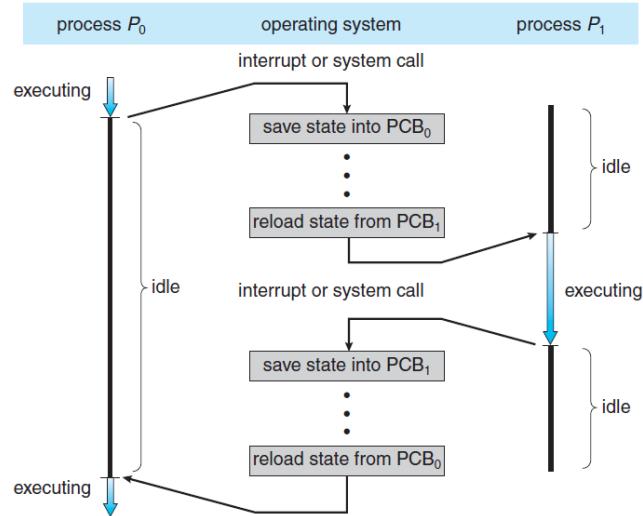
- Bir PCB aşağıdaki bilgilerle ilişkilendirilmiştir:
 - **Process state:** Durum, new, ready, running, waiting, halted olabilir.
 - **Program counter:** Bu process için sonraki komutun adresini gösterir.
 - **CPU register'ları:** CPU'ya göre değişen tür ve boyutta register'lar vardır.
(accumulators, index registers, stack pointers, general-purpose registers, condition codes, ...)
 - **CPU-scheduling information:** Process önceliğini içerir.
 - **Memory-management information:** Base ve limit register'ları, sayfa ve segment tabloları, ... içerir.
 - **Accounting information:** CPU kullanım oranları, account bilgileri ve process numaralarını içerir.
 - **I/O status information:** Process'lere tahsis edilmiş I/O cihazları ile açık durumdaki dosyaları içerir.

8

Process kavramı

Process control block

- CPU'nun process'ler arasında geçiş şekilde görülmektedir.



Process kavramı

Threads

- Tek thread ile bir process kontrol edilir ve birden fazla görev aynı anda yapılamaz** (karakter girişi ile spell check aynı anda yapılamaz.).
- Modern işletim sistemlerinde bir process ile birden fazla thread çalıştırılmasına izin verilir.**
- Bu özellik multicore işlemcilerde çok faydalıdır** ve çok thread eşzamanlı çalıştırılır.
- Çok thread ile çalışan sistemlerde, PCB ile her bir thread'e ait bilgiler saklanır.**

Konular

- Process kavramı
- **Process planlama**
- Process işlemleri
- Process'ler arası iletişim
- İstemci-sunucu sistemlerde iletişim

11

Process planlama

- Çok programlı sistemlerin temel amacı, **CPU kullanımını maksimuma çıkaracak şekilde process'leri çalıştırılmaktır.**
- Zaman paylaşımı sistemlerde CPU çok kısa aralıklarla process'ler arasında geçiş yapar.
- Bir process'i CPU'da çalışması için **process scheduler** seçer.

12

Process planlama

Scheduling queues

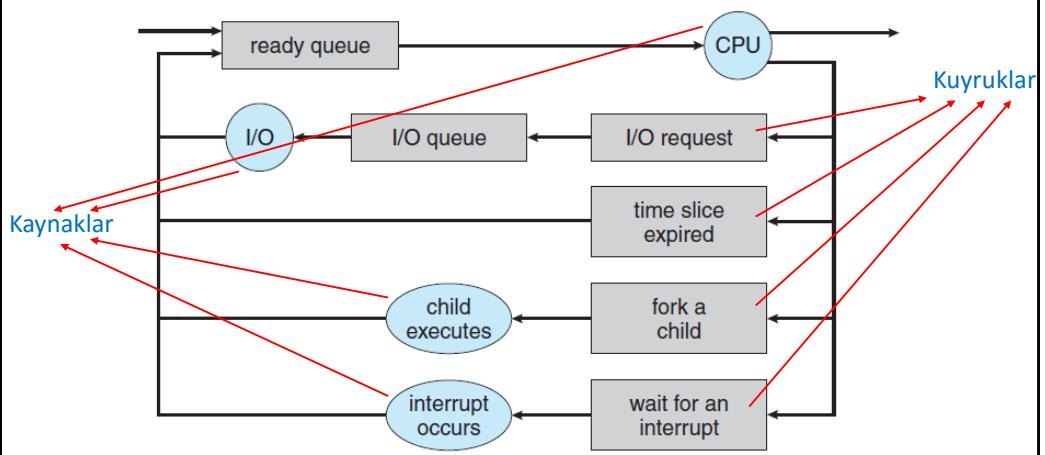
- Bir process sisteme girdiğinde, tüm işlerin bulunduğu iş kuyruğuna (**job queue**) alınır.
- Hafızaya alınmış ve çalışmayı bekleyen process'ler hazır kuyruğuna (**ready queue**) alınır.
- Kuyruk yapıları genellikle linked list veri yapısı ile gerçekleştirilir.
- Ready queue, listedeki PCB'lerin ilk ve son elemanlarını işaret eder.
- Bir sistemde hazır kuyruğu dışında I/O cihazları için de kuyruk (**device queue**) bulunur.

13

Process planlama

Scheduling queues

- Process planlama için aşağıdaki şekilde verilen **queueing diagram** yaygın kullanılan gösterimdir.



14

Process planlama

Scheduling queues

- Bir process **I/O isteğinde bulunursa, I/O kuyruğuna aktarılır.**
- Bir process başka bir process'i çalıştırırsa onun bitmesini bekler.
- Bir process çalışması için **ayrılan süre tamamlanırsa** CPU tarafından **tekrar hazır kuyruğunun sonuna alınır.**
- Bir process interrupt beklemeye başlarsa **interrupt kuyruğuna alınır.**

15

Process planlama

Schedulers

- Bir process, çalışma süresi boyunca farklı kuyruklara alınabilir.
- Kuyruktaki process'lerin seçilmesi **scheduler** tarafından gerçekleştirilir.
- **Genellikle batch sistemlerde çok sayıda process çalıştırılmak üzere sisteme gönderilir.**
- Bu process'ler disk üzerinde biriktirilir ve daha sonra çalıştırılır.
- **Long-term scheduler** (veya **job scheduler**) bu işleri seçerek çalıştırılmak üzere hafızaya yükler.
- **Short-term scheduler** (veya **CPU scheduler**) bu işlerden çalıştırılmak üzere hazır olanları seçerek CPU'yu onlara tahsis eder.
- Short-term scheduler çok kısa aralıklarla (<100ms) ve sıkılıkla çalıştırılır. Long-term scheduler ise dakika seviyesindeki aralıklarla çalıştırılır.

16

Process planlama

Schedulers

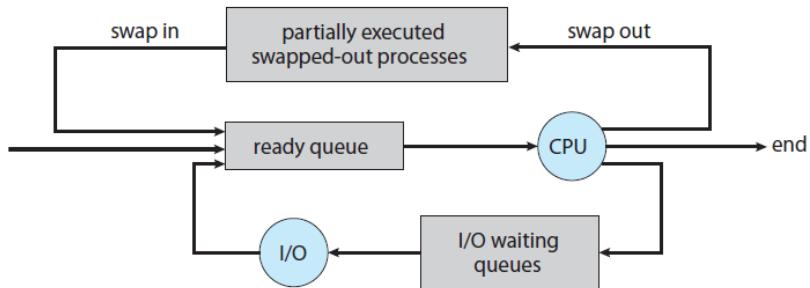
- Process'ler **I/O-bound** ve **CPU-bound** olarak iki gruba ayrılır.
- I/O-bound process'ler I/O işlemleri için daha fazla süre ayıırlar.
- CPU-bound process'ler CPU ile hesaplama işlemleri için daha fazla süre ayıırlar.
- Eğer tüm process'ler I/O-bound olursa, hafızadaki ready queue hemen hemen her zaman boş kalır ve **short-term scheduler** çok az çalışır.
- Eğer tüm process'ler CPU-bound olursa, I/O waiting queue hemen hemen her zaman boş kalır ve **cihazlar kullanılmadan boş kalır**.
- **Her iki durumda da sistem dengesiz iş dağılımına sahip olur.**
- Windows ve Unix işletim sistemlerinde long-term scheduler bulunmaz ve sadece short-term scheduler kullanılır.

17

Process planlama

Schedulers

- Bazı işletim sistemleri **medium-term scheduler** kullanır.



- Hafıza gereksiniminin değişmesi gibi bazı durumlarda, process'ler hafızadan atılır ve daha sonra tekrar hafızaya alınır (**swapping**).

18

Process planlama

Context switch

- **Interrupt'lar**, CPU'nun yürütülmekte olduğu **bir görevden** işletim sisteminin **kernel fonksiyonuna geçmesine neden olurlar**.
- Bir interrupt gerçekleştiğinde, mevcut konfigürasyon (**context**) saklanır ve geri dönüldüğünde yeniden aynı içeriğe dönülerek devam edilir.
- **Bir process için context, program control block (PCB) içerisinde saklanır**.
- Context, **CPU register'larının değerleri, hafıza yönetim bilgileri, process state bilgisini** içerir.
- **CPU'nun bir process'ten başka bir process'e geçmesine context switch denilmektedir**.
- **Context switch süresi**, bir iş üretilmediği için **overhead** olarak adlandırılır ve genellikle birkaç milisaniyedir.

19

Konular

- Process kavramı
- Process planlama
- **Process işlemleri**
- Process'ler arası iletişim
- İstemci-sunucu sistemlerde iletişim

20



Process İşlemleri

Process creation

- Process'ler dinamik olarak oluşturulurlar ve silinirler.
- Bir process çalışması sırasında birkaç tane başka process'i çalıştırabilir.
- Çağırılan process **parent**, yeni oluşturulan process **child** olarak adlandırılır.
- Unix, Linux ve Windows gibi işletim sistemleri, her process için **process identifier (pid)** değeri atarlar.
- Her process için atanmış değer tekildir (**unique**) ve process'e erişim için kullanılır.

21



Process İşlemleri

Process creation

- Bir parent process, yeni bir child process oluşturduğunda, **yeni child process CPU time, hafıza, dosyalar ve I/O cihazları gibi kaynaklara ihtiyaç duyar.**
- Parent process, kendi kaynaklarını child process'lere paylaşabilir veya işletim sistemi tarafından child process'e yeni kaynak tahsis edilebilir.
- Bir parent process, child process başlattığında sonlana kadar bekleyebilir veya eşzamanlı çalışmasını sürdürübiliir.

22

Process İşlemleri

Process creation - Unix

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

Yeni process için sistem çağrıları.

Yeni process için hata oluştu.

Yeni child process başlatıldı.

Dizin listesini ekrana yazan process.

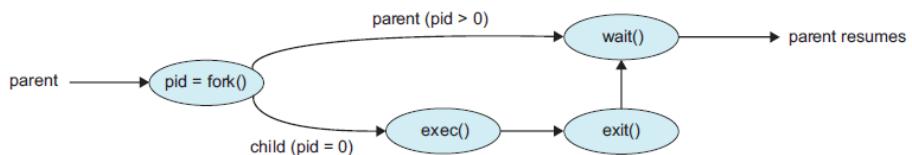
Parent process ($pid > 0$)

Parent process, child process'i bekliyor.

Process İşlemleri

Process creation - Unix

- Child process `exit()` ile sonlanıncaya kadar parent process bekler.
- Şekilde parent process'in child process'i beklemesi görülmektedir.



Process İşlemleri

Process creation - Windows

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

/* create child process */
if (!CreateProcess(NULL, /* use command line */
"C:\WINDOWS\system32\mspaint.exe", /* command */
NULL, /* don't inherit process handle */
NULL, /* don't inherit thread handle */
FALSE, /* disable handle inheritance */
0, /* no creation flags */
NULL, /* use parent's environment block */
NULL, /* use parent's existing directory */
&si,
&pi))
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

Yeni process için sistem çağrıları

Yeni process "mspaint.exe"

Yeni child process özellikleri
(Window size, giriş/çıkış
dosyaları, ...)

Process identifier

Process oluşturma hatası

Parent process, child process'i
bekliyor.

Process İşlemleri

Process termination

- Bir process son deyimini çalıştırıp tamamlandığında, exit() sistem çağrılarını çalıştırır ve hafızadan silinmesini ister.
- **Sonlanan process, parent process'e durum bilgisini gösteren değer (integer) döndürebilir.**
- Bir parent process, child process'in sonlanmasına da neden olabilir (Windows'ta TerminateProcess() sistem çağrısı).
 - Child process kaynak kullanım sınırını aştığında, parent process tarafından sonlandırılabilir.
 - Child process'in yaptığı işe gerek kalmayabilir.
 - İşletim sistemi parent process'i sonlandırdığında child process'lerin de sonlandırılmasını isteyebilir.
- exit(); doğrudan, return int; dolaylı sonlandırma yapar.

Konular

- Process kavramı
- Process planlama
- Process işlemleri
- **Process'ler arası iletişim**
- İstemci-sunucu sistemlerde iletişim

27

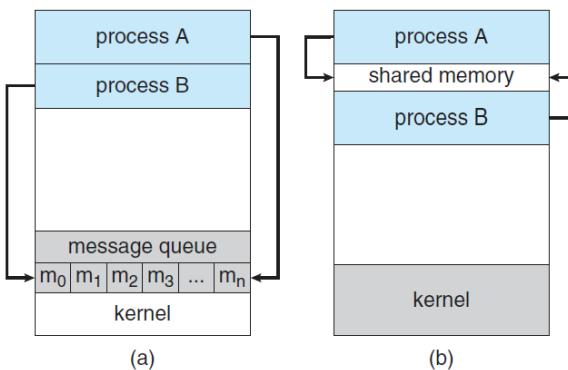
Process'ler arası iletişim

- Process'ler işletim sisteminde **independent process** veya **cooperating process** olarak çalışırlar.
- **Independent process'ler** diğer process'leri etkilemezler ve onlardan etkilenmezler.
- **Cooperating process'ler** diğer process'leri etkilerler ve onlardan etkilenirler (diğer process'lerle veri paylaşımı yaparlar).
 - **Information sharing:** Paylaşımımış dosyalar üzerinde işlem yapmak gerekebilir.
 - **Computation speedup:** Birden fazla core'a sahip işlemcili bilgisayarlarda, görevler parçalar halinde eşzamanlı yürütürlürler.
 - **Modularity:** Sistem parçalar (process'ler, thread'ler) halinde oluşturulabilir ve bu parçalar arasında iletişim yapılabılır.
 - **Convenience:** Bir kullanıcı farklı işleri (müzik dinleme, metin yazma, compile, ...) aynı anda gerçekleştirebilir.

28

Process'ler arası iletişim

- Cooperating process'ler **shared memory** ve **message passing** modelleri ile veri aktarımı yaparlar. **Shared memory modeli daha hızlıdır!!!**
 - Shared memory modelinde, **hafızada bir bölge process'ler arasında paylaştırılır.**
 - Message passing modelinde, **process'ler arasında mesaj ile veri gönderilir.**



Communications models. (a) Message passing. (b) Shared memory.

29

Process'ler arası iletişim

Shared memory

- Shared memory modelinde, **producer** veriyi oluşturur ve paylaşılmış hafıza alanına yazar, **consumer** ise veriyi okuyarak kullanır.
- Compiler**, bir programı derler ve assembly kod üretir, **assembler** bu kodu alır ve object kod üretir, **loader** ise bu kodu giriş olarak alır.
- Shared buffer aşağıdaki kod ile tanımlanır:

```
#define BUFFER_SIZE 10  
  
typedef struct {  
    . . .  
} item;  
  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;
```

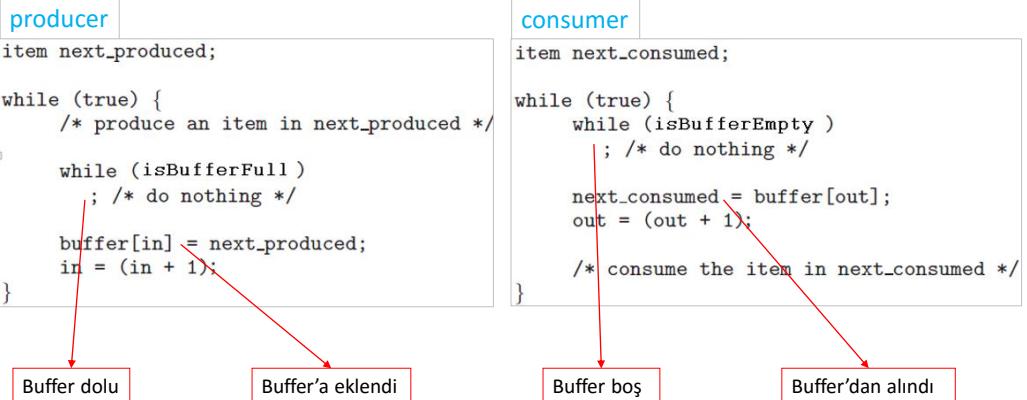
- Buffer dizi şeklinde oluşturulur (**dairesel bağlı liste kullanılabilir**) ve **in** değişkeni sonraki boş yeri, **out** ise ilk dolu yeri gösterir.

30

Process'ler arası iletişim

Shared memory

- Shared memory modelinde, **producer** veriyi oluşturur ve paylaşılmış hafıza alanına yazar, **consumer** ise veriyi okuyarak kullanır.



31

Process'ler arası iletişim

Message passing

- Message passing modeli, **dağıtık ortamlardaki process'lerin** (örn. chat programı) **iletişiminde faydalıdır.**
- Message passing modelinde en az iki işlem tanımlanır:
 - send(message)
 - receive(message)
- Mesaj boyutları sabit uzunlukta veya değişken uzunlukta olabilir.
- Process'ler birbirlerini doğrudan isimleriyle adresleyerek mesaj gönderirler:**
 - send(P, message) // P process'ine mesaj gönderilir.
 - receive(Q, message) // Q process'inden mesaj alınır.

32

Konular

- Process kavramı
- Process planlama
- Process işlemleri
- Process'ler arası iletişim
- **İstemci-sunucu sistemlerde iletişim**

33

İstemci-sunucu sistemde iletişim

Soketler

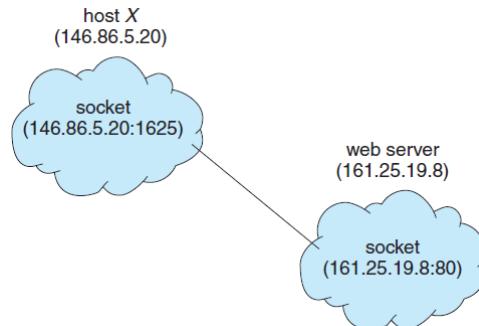
- **Bir soket iletişim için üç noktayı (process) tanımlar.**
- Bir ağ üzerinde haberleşen iki process'in her biri bir sokete sahiptir.
- **Bir soket, IP adresi ve port numarasıyla tanımlanır.**
- Sunucu, bir portu dinleyerek gelen istekleri bekler.
- Sunucuya bir istek geldiğinde alır ve gerekli işlemleri başlatır.
- FTP, HTTP gibi protokoller ayrılmış port numaralarına sahiptir (HTTP için 80, FTP için 21).
- **Bir host üzerindeki tüm process'ler için bağlantılarının tekil olması zorunludur.**
- **Tüm process'ler için işletim sisteminin atadığı port numaraları farklı olmak zorundadır.**

34

İstemci-sunucu sistemde iletişim

Soketler

- IP adresi **146.86.5.20** olan istemci host üzerindeki process port numarası olarak **1625**'e sahiptir.
- Web sunucu **161.25.19.8** IP adresine ve sunucu process **80** port numarasına sahiptir.
- **soket çiftleri (146.86.5.20:1625) ile (161.25.19.8:80)** olacaktır.



35

İstemci-sunucu sistemde iletişim

Soketler

- Soketler arasında iki tür bağlantı yapılmaktadır:
 - **Connection-oriented (reliable)**
 - **Connectionless (unreliable)**
- Reliable iletişim **TCP (Transmission Control Protocol)** ile, unreliable iletişim ise **UDP (User Datagram Protocol)** ile gerçekleştirilir.
- Java programlama dilinde TCP bağlantısı **Socket** sınıfı ile, UDP bağlantısı **DatagramSocket** sınıfı ile gerçekleştirilir.

36

İstemci-sunucu sistemde iletişim

Soketler – Örnek

- Sunucu 6013 portunu dinler.
- İstemcilerden gelen isteklere tarih ve saat bilgisini cevap olarak gönderir.
- Sunucu **ServerSocket** nesnesi oluşturarak **accept()** metodu ile 6013 portunu dinlemektedir.
- **PrintWriter** nesnesi bir sokete print() veya println() metodları ile yazma işlemi yapar.
- İstemci process, sunucu process ile belirlenmiş port üzerinden bağlantı yapar.
- İstemci Socket nesnesi oluşturur ve **127.0.0.1** IP adresinden **6013** portu ile bağlantı yapar.
- 127.0.0.1 IP adresi **loopback** olarak adlandırılır ve kendisini gösterir.

37

Sunucu process

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Sunucu ServerSocket nesnesi oluşturarak accept() metodu ile 6013 portunu dinlemektedir.

PrintWriter nesnesi bir sokete print() veya println() metodları ile yazma işlemi yapar.

İstemci ile bağlantı kapatılır.

İstemci process

İstemci **Socket** nesnesi
oluşturarak **127.0.0.1 IP**
adresinde **6013 portu**yla
bağlantı yapar.

BufferedReader nesnesi ile
nesnesi soketten okuma
bağlantısı tanımlanır.

İstemci gelen veriyi okur ve
ekrana yazar.

Bağlantı kapatılır.

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- Process kavramı
- Process planlama
- Process işlemleri
- Process'ler arası iletişim
- İstemci-sunucu sistemlerde iletişim

Process kavramı

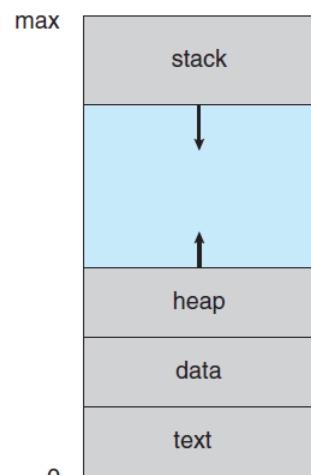
- Günümüz işletim sistemleri birden çok programın hafızaya yüklenmesine ve eşzamanlı çalıştırılmasına izin verir.
- Çalışmakta olan programa **process** denilmektedir.
- **Modern işletim sistemlerinde, process işin bir parçasıdır.**
- **CPU, aralarında geçiş yaparak tüm process'leri eş zamanlı çalıştırılabilir.**
- Bir sistem, tek kullanıcılı bile olsa, birden fazla uygulamayı (Word, Excel, Web Browser, ...) birlikte çalıştırabilir.
- İşletim sistemi multitasking desteklemese bile, işletim sisteminin kendi fonksiyonlarını çağırarak çalıştırır.

3

Process kavramı

Process

- Bir process, yürütme做的 işi, **program counter** değerini, **CPU register'larının değerlerini içermektedir.**
- Bir process aşağıdaki bileşenleri içermektedir:
 - **stack**, fonksiyon parametreleri, return adresleri ve lokal değişkenleri saklar.
 - **data section**, global değişkenleri saklar.
 - **heap**, process'e runtime'da dinamik olarak atanın bellektir.



0

4

Process kavramı

Process

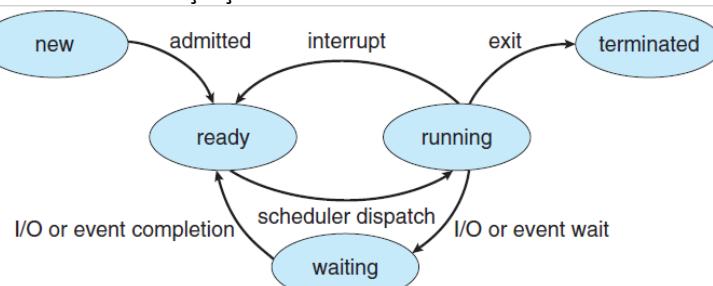
- **Bir program pasif varlıktır (executable file)**, disk üzerinde saklanan komut kümeleridir.
- **Bir process aktif varlıktır**, sonraki çalışacak komut adresini program counter saklar.
- **Aynı program birden fazla process ile ilişkili olabilir** (birden fazla eşzamanlı çalışan Web browser).
- **Her process'in stack, data section ve heap kısımları farklıdır.**

5

Process kavramı

Process state

- **Bir process çalıştığı sürece durum değişir.**
 - **New**: Process oluşturulmaktadır.
 - **Running**: Komutlar çalıştırılmaktadır.
 - **Waiting**: Process bir olayın gerçekleşmesini beklemektedir (I/O, bir cihazdan geribildirim).
 - **Ready**: Process çalışmak için CPU'ya atanmak üzere bekliyor.
 - **Terminated**: Process çalışmasını sonlandırır.

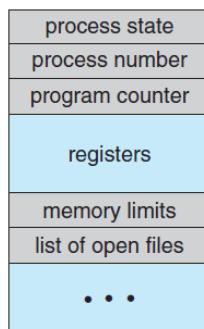


6

Process kavramı

Process control block

- Her process, işletim sisteminde **process control block (PCB)** (veya **task control block**) tarafından temsil edilir.



7

Process kavramı

Process control block

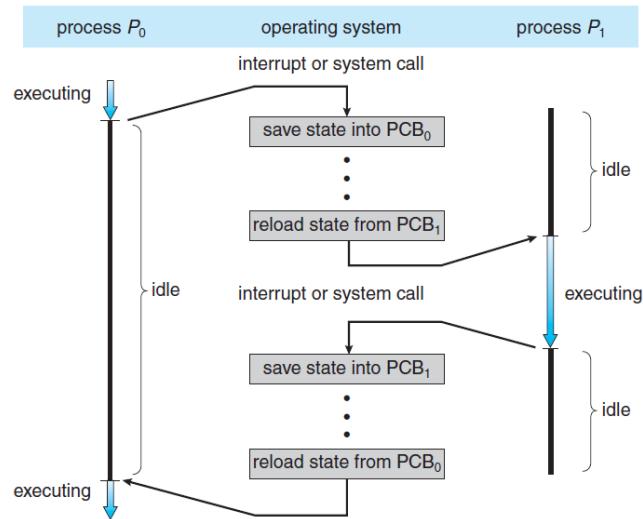
- Bir PCB aşağıdaki bilgilerle ilişkilendirilmiştir:
 - **Process state:** Durum, new, ready, running, waiting, halted olabilir.
 - **Program counter:** Bu process için sonraki komutun adresini gösterir.
 - **CPU register'ları:** CPU'ya göre değişen tür ve boyutta register'lar vardır.
(accumulators, index registers, stack pointers, general-purpose registers, condition codes, ...)
 - **CPU-scheduling information:** Process önceliğini içerir.
 - **Memory-management information:** Base ve limit register'ları, sayfa ve segment tabloları, ... içerir.
 - **Accounting information:** CPU kullanım oranları, account bilgileri ve process numaralarını içerir.
 - **I/O status information:** Process'lere tahsis edilmiş I/O cihazları ile açık durumdaki dosyaları içerir.

8

Process kavramı

Process control block

- CPU'nun process'ler arasında geçiş şekilde görülmektedir.



Process kavramı

Threads

- Tek thread ile bir process kontrol edilir ve birden fazla görev aynı anda yapılamaz** (karakter girişi ile spell check aynı anda yapılamaz.).
- Modern işletim sistemlerinde bir process ile birden fazla thread çalıştırılmasına izin verilir.**
- Bu özellik multicore işlemcilerde çok faydalıdır** ve çok thread eşzamanlı çalıştırılır.
- Çok thread ile çalışan sistemlerde, PCB ile her bir thread'e ait bilgiler saklanır.**

Konular

- Process kavramı
- **Process planlama**
- Process işlemleri
- Process'ler arası iletişim
- İstemci-sunucu sistemlerde iletişim

11

Process planlama

- Çok programlı sistemlerin temel amacı, **CPU kullanımını maksimuma çıkaracak şekilde process'leri çalıştırılmaktır.**
- Zaman paylaşımı sistemlerde CPU çok kısa aralıklarla process'ler arasında geçiş yapar.
- Bir process'i CPU'da çalışması için **process scheduler** seçer.

12

Process planlama

Scheduling queues

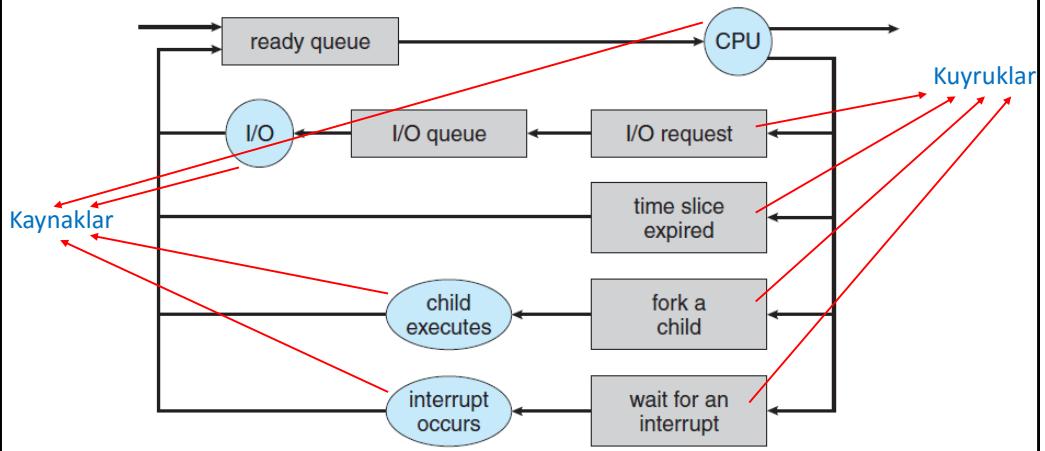
- Bir process sisteme girdiğinde, tüm işlerin bulunduğu iş kuyruğuna (**job queue**) alınır.
- Hafızaya alınmış ve çalışmayı bekleyen process'ler hazır kuyruğuna (**ready queue**) alınır.
- Kuyruk yapıları genellikle linked list veri yapısı ile gerçekleştirilir.
- Ready queue, listedeki PCB'lerin ilk ve son elemanlarını işaret eder.
- Bir sistemde hazır kuyruğu dışında I/O cihazları için de kuyruk (**device queue**) bulunur.

13

Process planlama

Scheduling queues

- Process planlama için aşağıdaki şekilde verilen **queueing diagram** yaygın kullanılan gösterimdir.



14

Process planlama

Scheduling queues

- Bir process **I/O isteğinde bulunursa, I/O kuyruğuna aktarılır.**
- Bir process başka bir process'i çalıştırırsa onun bitmesini bekler.
- Bir process çalışması için **ayrılan süre tamamlanırsa** CPU tarafından **tekrar hazır kuyruğunun sonuna alınır.**
- Bir process interrupt beklemeye başlarsa **interrupt kuyruğuna alınır.**

15

Process planlama

Schedulers

- Bir process, çalışma süresi boyunca farklı kuyruklara alınabilir.
- Kuyruktaki process'lerin seçilmesi **scheduler** tarafından gerçekleştirilir.
- **Genellikle batch sistemlerde çok sayıda process çalıştırılmak üzere sisteme gönderilir.**
- Bu process'ler disk üzerinde biriktirilir ve daha sonra çalıştırılır.
- **Long-term scheduler** (veya **job scheduler**) bu işleri seçerek çalıştırılmak üzere hafızaya yükler.
- **Short-term scheduler** (veya **CPU scheduler**) bu işlerden çalıştırılmak üzere hazır olanları seçerek CPU'yu onlara tahsis eder.
- Short-term scheduler çok kısa aralıklarla (<100ms) ve sıkılıkla çalıştırılır. Long-term scheduler ise dakika seviyesindeki aralıklarla çalıştırılır.

16

Process planlama

Schedulers

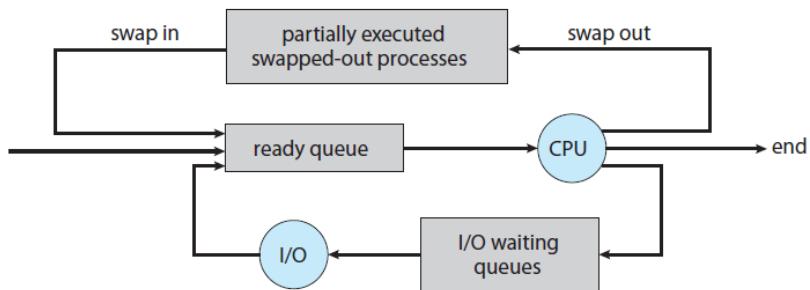
- Process'ler **I/O-bound** ve **CPU-bound** olarak iki gruba ayrılır.
- I/O-bound process'ler I/O işlemleri için daha fazla süre ayıırlar.
- CPU-bound process'ler CPU ile hesaplama işlemleri için daha fazla süre ayıırlar.
- Eğer tüm process'ler I/O-bound olursa, hafızadaki ready queue hemen hemen her zaman boş kalır ve **short-term scheduler** çok az çalışır.
- Eğer tüm process'ler CPU-bound olursa, I/O waiting queue hemen hemen her zaman boş kalır ve **cihazlar kullanılmadan boş kalır**.
- **Her iki durumda da sistem dengesiz iş dağılımına sahip olur.**
- Windows ve Unix işletim sistemlerinde long-term scheduler bulunmaz ve sadece short-term scheduler kullanılır.

17

Process planlama

Schedulers

- Bazı işletim sistemleri **medium-term scheduler** kullanır.



- Hafıza gereksiniminin değişmesi gibi bazı durumlarda, process'ler hafızadan atılır ve daha sonra tekrar hafızaya alınır (**swapping**).

18

Process planlama

Context switch

- **Interrupt'lar**, CPU'nun yürütülmekte olduğu **bir görevden** işletim sisteminin **kernel fonksiyonuna geçmesine neden olurlar**.
- Bir interrupt gerçekleştiğinde, mevcut konfigürasyon (**context**) saklanır ve geri dönüldüğünde yeniden aynı içeriğe dönülerek devam edilir.
- **Bir process için context, program control block (PCB) içerisinde saklanır**.
- Context, **CPU register'larının değerleri, hafıza yönetim bilgileri, process state bilgisini** içerir.
- **CPU'nun bir process'ten başka bir process'e geçmesine context switch denilmektedir**.
- **Context switch süresi**, bir iş üretilmediği için **overhead** olarak adlandırılır ve genellikle birkaç milisaniyedir.

19

Konular

- Process kavramı
- Process planlama
- **Process işlemleri**
- Process'ler arası iletişim
- İstemci-sunucu sistemlerde iletişim

20



Process İşlemleri

Process creation

- Process'ler dinamik olarak oluşturulurlar ve silinirler.
- Bir process çalışması sırasında birkaç tane başka process'i çalıştırabilir.
- Çağırılan process **parent**, yeni oluşturulan process **child** olarak adlandırılır.
- Unix, Linux ve Windows gibi işletim sistemleri, her process için **process identifier (pid)** değeri atarlar.
- Her process için atanmış değer tekildir (**unique**) ve process'e erişim için kullanılır.

21



Process İşlemleri

Process creation

- Bir parent process, yeni bir child process oluşturduğunda, **yeni child process CPU time, hafıza, dosyalar ve I/O cihazları gibi kaynaklara ihtiyaç duyar.**
- Parent process, kendi kaynaklarını child process'lere paylaşabilir veya işletim sistemi tarafından child process'e yeni kaynak tahsis edilebilir.
- Bir parent process, child process başlattığında sonlana kadar bekleyebilir veya eşzamanlı çalışmasını sürdürübiliir.

22

Process İşlemleri

Process creation - Unix

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

Yeni process için sistem çağrıları.

Yeni process için hata oluştu.

Yeni child process başlatıldı.

Dizin listesini ekrana yazan process.

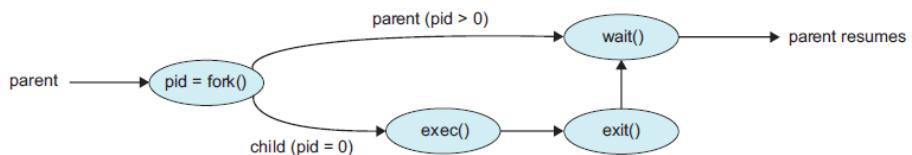
Parent process ($pid > 0$)

Parent process, child process'i bekliyor.

Process İşlemleri

Process creation - Unix

- Child process `exit()` ile sonlanıncaya kadar parent process bekler.
- Şekilde parent process'in child process'i beklemesi görülmektedir.



Process İşlemleri

Process creation - Windows

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

/* create child process */
if (!CreateProcess(NULL, /* use command line */
"C:\WINDOWS\system32\mspaint.exe", /* command */
NULL, /* don't inherit process handle */
NULL, /* don't inherit thread handle */
FALSE, /* disable handle inheritance */
0, /* no creation flags */
NULL, /* use parent's environment block */
NULL, /* use parent's existing directory */
&si,
&pi))
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

Yeni process için sistem çağrıları

Yeni process "mspaint.exe"

Yeni child process özellikleri
(Window size, giriş/çıkış
dosyaları, ...)

Process identifier

Process oluşturma hatası

Parent process, child process'i
bekliyor.

Process İşlemleri

Process termination

- Bir process son deyimini çalıştırıp tamamlandığında, exit() sistem çağrılarını çalıştırır ve hafızadan silinmesini ister.
- **Sonlanan process, parent process'e durum bilgisini gösteren değer (integer) döndürebilir.**
- Bir parent process, child process'in sonlanmasına da neden olabilir (Windows'ta TerminateProcess() sistem çağrısı).
 - Child process kaynak kullanım sınırını aştığında, parent process tarafından sonlandırılabilir.
 - Child process'in yaptığı işe gerek kalmayabilir.
 - İşletim sistemi parent process'i sonlandırdığında child process'lerin de sonlandırılmasını isteyebilir.
- exit(); doğrudan, return int; dolaylı sonlandırma yapar.

Konular

- Process kavramı
- Process planlama
- Process işlemleri
- **Process'ler arası iletişim**
- İstemci-sunucu sistemlerde iletişim

27

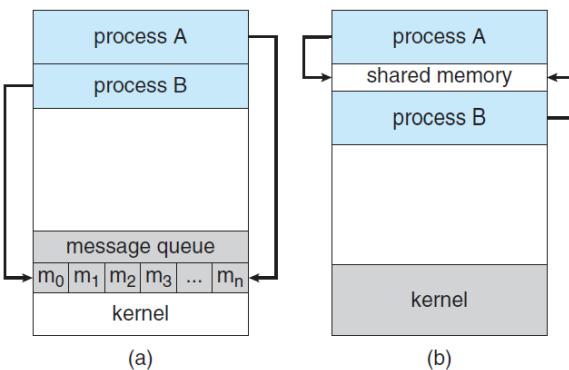
Process'ler arası iletişim

- Process'ler işletim sisteminde **independent process** veya **cooperating process** olarak çalışırlar.
- **Independent process'ler** diğer process'leri etkilemezler ve onlardan etkilenmezler.
- **Cooperating process'ler** diğer process'leri etkilerler ve onlardan etkilenirler (diğer process'lerle veri paylaşımı yaparlar).
 - **Information sharing:** Paylaşımımış dosyalar üzerinde işlem yapmak gerekebilir.
 - **Computation speedup:** Birden fazla core'a sahip işlemcili bilgisayarlarda, görevler parçalar halinde eşzamanlı yürütürlürler.
 - **Modularity:** Sistem parçalar (process'ler, thread'ler) halinde oluşturulabilir ve bu parçalar arasında iletişim yapılabılır.
 - **Convenience:** Bir kullanıcı farklı işleri (müzik dinleme, metin yazma, compile, ...) aynı anda gerçekleştirebilir.

28

Process'ler arası iletişim

- Cooperating process'ler **shared memory** ve **message passing** modelleri ile veri aktarımı yaparlar. **Shared memory modeli daha hızlıdır!!!**
 - Shared memory modelinde, **hafızada bir bölge process'ler arasında paylaştırılır.**
 - Message passing modelinde, **process'ler arasında mesaj ile veri gönderilir.**



Communications models. (a) Message passing. (b) Shared memory.

29

Process'ler arası iletişim

Shared memory

- Shared memory modelinde, **producer** veriyi oluşturur ve paylaşılmış hafıza alanına yazar, **consumer** ise veriyi okuyarak kullanır.
- Compiler**, bir programı derler ve assembly kod üretir, **assembler** bu kodu alır ve object kod üretir, **loader** ise bu kodu giriş olarak alır.
- Shared buffer aşağıdaki kod ile tanımlanır:

```
#define BUFFER_SIZE 10  
  
typedef struct {  
    . . .  
} item;  
  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;
```

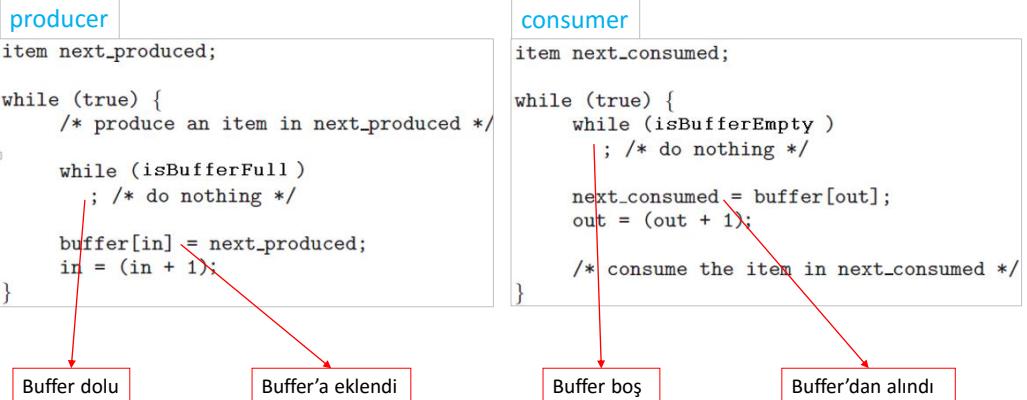
- Buffer dizi şeklinde oluşturulur (**dairesel bağlı liste kullanılabilir**) ve **in** değişkeni sonraki boş yeri, **out** ise ilk dolu yeri gösterir.

30

Process'ler arası iletişim

Shared memory

- Shared memory modelinde, **producer** veriyi oluşturur ve paylaşılmış hafıza alanına yazar, **consumer** ise veriyi okuyarak kullanır.



31

Process'ler arası iletişim

Message passing

- Message passing modeli, **dağıtık ortamlardaki process'lerin** (örn. chat programı) **iletişiminde faydalıdır.**
- Message passing modelinde en az iki işlem tanımlanır:
 - send(message)
 - receive(message)
- Mesaj boyutları sabit uzunlukta veya değişken uzunlukta olabilir.
- Process'ler birbirlerini doğrudan isimleriyle adresleyerek mesaj gönderirler:**
 - send(P, message) // P process'ine mesaj gönderilir.
 - receive(Q, message) // Q process'inden mesaj alınır.

32

Konular

- Process kavramı
- Process planlama
- Process işlemleri
- Process'ler arası iletişim
- **İstemci-sunucu sistemlerde iletişim**

33

İstemci-sunucu sistemde iletişim

Soketler

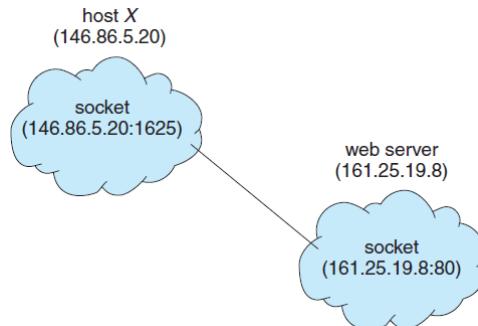
- **Bir soket iletişim için üç noktayı (process) tanımlar.**
- Bir ağ üzerinde haberleşen iki process'in her biri bir sokete sahiptir.
- **Bir soket, IP adresi ve port numarasıyla tanımlanır.**
- Sunucu, bir portu dinleyerek gelen istekleri bekler.
- Sunucuya bir istek geldiğinde alır ve gerekli işlemleri başlatır.
- FTP, HTTP gibi protokoller ayrılmış port numaralarına sahiptir (HTTP için 80, FTP için 21).
- **Bir host üzerindeki tüm process'ler için bağlantılarının tekil olması zorunludur.**
- **Tüm process'ler için işletim sisteminin atadığı port numaraları farklı olmak zorundadır.**

34

İstemci-sunucu sistemde iletişim

Soketler

- IP adresi **146.86.5.20** olan istemci host üzerindeki process port numarası olarak **1625**'e sahiptir.
- Web sunucu **161.25.19.8** IP adresine ve sunucu process **80** port numarasına sahiptir.
- **soket çiftleri (146.86.5.20:1625) ile (161.25.19.8:80)** olacaktır.



35

İstemci-sunucu sistemde iletişim

Soketler

- Soketler arasında iki tür bağlantı yapılmaktadır:
 - **Connection-oriented (reliable)**
 - **Connectionless (unreliable)**
- Reliable iletişim **TCP (Transmission Control Protocol)** ile, unreliable iletişim ise **UDP (User Datagram Protocol)** ile gerçekleştirilir.
- Java programlama dilinde TCP bağlantısı **Socket** sınıfı ile, UDP bağlantısı **DatagramSocket** sınıfı ile gerçekleştirilir.

36

İstemci-sunucu sistemde iletişim

Soketler – Örnek

- Sunucu 6013 portunu dinler.
- İstemcilerden gelen isteklere tarih ve saat bilgisini cevap olarak gönderir.
- Sunucu **ServerSocket** nesnesi oluşturarak **accept()** metodu ile 6013 portunu dinlemektedir.
- **PrintWriter** nesnesi bir sokete print() veya println() metodları ile yazma işlemi yapar.
- İstemci process, sunucu process ile belirlenmiş port üzerinden bağlantı yapar.
- İstemci Socket nesnesi oluşturur ve **127.0.0.1** IP adresinden **6013** portu ile bağlantı yapar.
- 127.0.0.1 IP adresi **loopback** olarak adlandırılır ve kendisini gösterir.

37

Sunucu process

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Sunucu ServerSocket nesnesi oluşturarak accept() metodu ile 6013 portunu dinlemektedir.

PrintWriter nesnesi bir sokete print() veya println() metodları ile yazma işlemi yapar.

İstemci ile bağlantı kapatılır.

İstemci process

İstemci **Socket** nesnesi
oluşturarak **127.0.0.1 IP**
adresinde **6013 portu**yla
bağlantı yapar.

BufferedReader nesnesi ile
nesnesi soketten okuma
bağlantısı tanımlanır.

İstemci gelen veriyi okur ve
ekrana yazar.

Bağlantı kapatılır.

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);

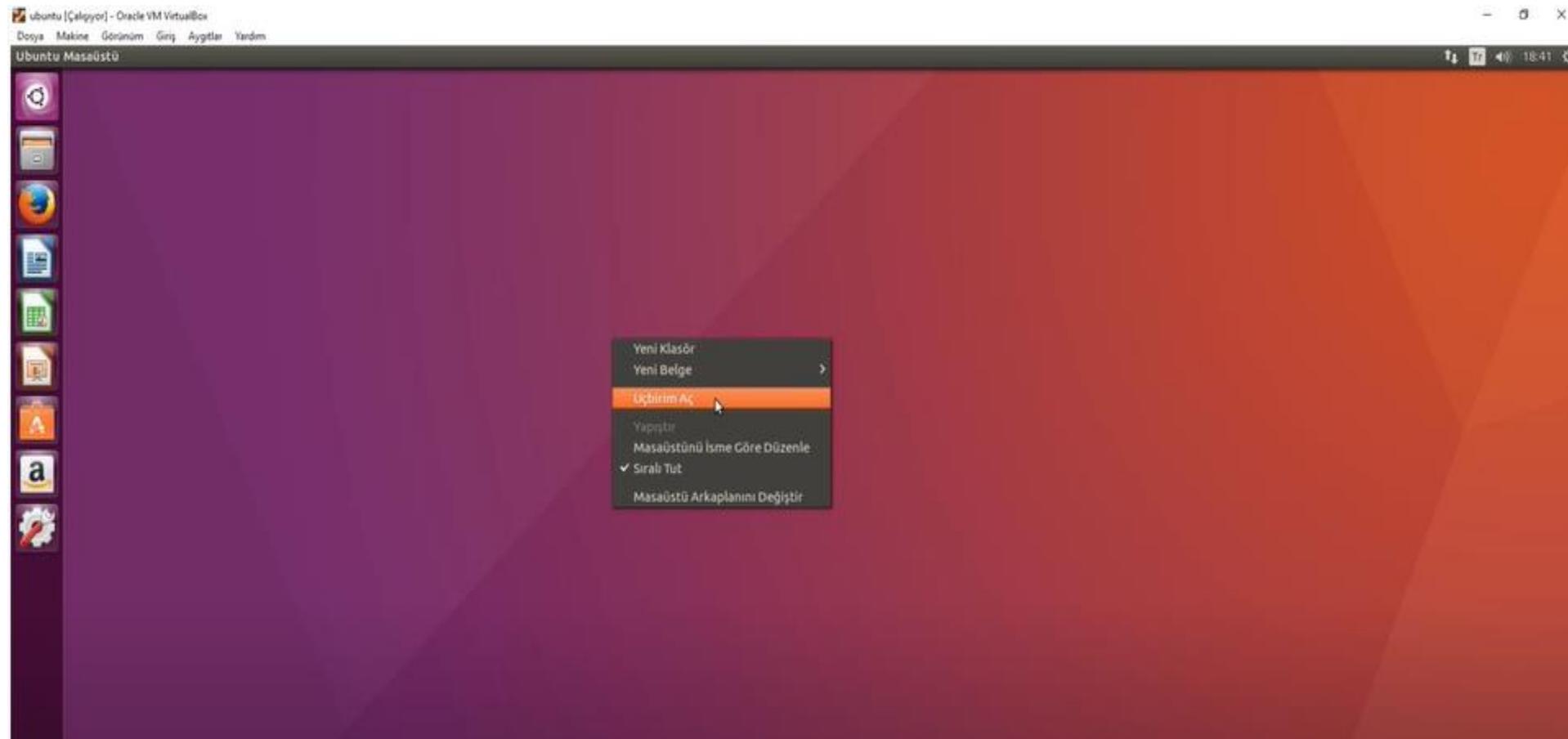
            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

İşletim Sistemleri 2.Uygulama Process(İşlem)Komutları

Hazırlayan : Özlem ÖZMEN AKALIN



Öncelikle Terminalde yazacağımız komutların bir yönetici olarak hareket edebilmesi için şu komutları çalıştırılmamız gerekiyor.

~\$ sudo su komutunu vererek kurulum kısmında verdığımız *şifreyi* yazarak ilerliyoruz.

root@tecmint_VirtualBox:/home-tecmint#

Buraya kendi şifrenizi yazarak devam ediyorsunuz.

Ben bu işlemi slayttan göstermek zorundayım çünkü daha önce bu işlemi yaptığım için tekrar yapmama ve şifreyi girmeme izin vermiyor.

- Bundan sonra yönetici olarak oturum açmış bulunuyoruz.
- Yine ben daha önceden güncelleştirmeleri, yükseltmeleri ve paket indirmelerini yaptığım için sizlere sunum üzerinden göstermek zorundayım.
- Sizler
- **apt-get update** komutunu verdiğimizde paketleri bizim için listeleyeceğ .Şimdi tekrar
- **apt-get upgrade** komutunu veriyorum ve bundan sonra evet anlamında e yazarak onaylıyorum.
- Sonrasında internetten indirip kurulumları tamamlayacak.
- Upgrade ile sistemimiz için uygun olan güncelleştirmeleri internetten indirecek ,paketleri açacak ve ayarlayacak.

- Daha sonra
- **apt-get install** diyerek kurulumu başlatalım.
- **exit** diyerek root dizininden çıkışıyoruz. Bundan sonra sanal bilgisayarı
- yeniden başlatmamız gerekiyor.

Process(İşlem)

- Linux'ta belirli komutları kullanarak süreçlerin nasıl yönetileceğine kısaca bakacağız.
- **İşlem Türleri**
- Linux'ta temelde iki tür işlem vardır:
 - **Ön plan işlemleri(süreçleri)** (etkileşimli süreçler olarak da adlandırılır)
- bunlar bir terminal oturumu aracılığıyla başlatılır ve kontrol edilir. Diğer bir deyişle, bu tür işlemlerin başlatılması için sisteme bağlı bir kullanıcının olması gereklidir; sistem işlevlerinin / hizmetlerinin bir parçası olarak otomatik olarak başlamamışlardır.
 - **Arka plan işlemleri(süreçleri)** (etkileşimli olmayan / otomatik süreçler olarak da adlandırılır) - bir terminale bağlı olmayan süreçlerdir; herhangi bir kullanıcı girdisi beklemiyorlar.

- **System() işlevini kullanma :**
- Bu yöntem nispeten basittir fakat verimsizdir ve önemli ölçüde belirli güvenlik risklerine sahiptir.
- **fork() ve exec() işlevini kullanma :**
- Bu teknik biraz ileri düzeydedir fakat güvenlik ile birlikte daha fazla esneklik,hız sunar.
- **Linux Süreçleri(islemleri, processleri) nasıl tanımlar?**
- Linux çok kullanıcılı bir sistem olduğundan, yani farklı kullanıcılar sistem üzerinde çeşitli programlar çalıştırılabilir, bir programın çalışan her örneği çekirdek tarafından benzersiz bir şekilde tanımlanmalıdır.
- Ve bir program, kendi süreç kimliği yani(process id) ve parent process id(üst süreç kimliği) ile tanımlanır.
- Bu nedenle süreçler ayrıca şu şekilde kategorize edilebilir.

- Parent process(Ana işlemler) :
- Bunlar çalışma zamanı sırasında diğer işlemleri oluşturan işlemlerdir.
- Child processes(alt süreçler) :
- Bu işlemler çalışma zamanı sırasında diğer işlemler tarafından oluşturulur.
- The Init Processes(Başlatma Süreci,İşlemi) :
 - Başlatma süreci, sistemdeki tüm işlemlerin anasıdır.
 - Linux sistemi açıldığında çalıştırılan ilk programdır ;
 - Sistemdeki diğer süreçleri yönetir. Çekirdeğin kendisi tarafından başlatılır,bu nedenle prensipte bir ana süreç sahip değildir.
 - Başlatma sürecinin her zaman bir işlem kimliği vardır.

- Tüm orphaned processes(öksüz süreçler) için evlat edinen bir ebeveyn işlevi görür.
- Bir işlemin kimliğini bulmak için **pidof** komutunu kullanabiliriz.
- # pidof systemd
- # pidof top
- # pidof httpd

```
[root@tecmint ~]# pidof systemd
1
[root@tecmint ~]# pidof top
2160
[root@tecmint ~]# pidof httpd
2103 2102 2101 2100 2099 1076
[root@tecmint ~]#
```

Linux parent processid (üst süreç id 'sini bulma komutları)

- Mevcut işlemin süreç kimliğini ve üst süreç kimliğini bulmak için şunu çalıştırılmamız gerekiyor :

```
[root@tecmint ~]# echo $$  
2109  
[root@tecmint ~]# echo $PPID  
2106  
[root@tecmint ~]#
```

- Linux 'ta İşlem Başlatma :
- Bir komutu veya programı çalıştırıldığınızda(örneğin cloucmd-yani CloudCommander), sistemde bir işlem başlatacaktır.
- Etkileşimli bir önplan işlemini şöyle başlatabiliriz.
- Terminale bağlanacak ve bir kullanıcı girdiyi gönderebilecektir.
- # cloucmd
- Bu komutla etkileşimli bir süreç başlatabilirsiniz.

```
[root@tecmint ~]# cloucmd  
url: http://localhost:8000/
```

- Linux Arka Plan İşleri
 - Arka planda (etkileşimli olmayan) bir işlem başlatmak için & simgesini kullanalım, burada işlem, ön plana taşınana kadar bir kullanıcıdan gelen girdiyi okumaz.
 - # cloucmd &
 - # jobs

- Ayrıca [Ctrl + Z] kullanarak işlemi askıya alarak arka plana bir işlem gönderebilirsiniz, bu işlem SIGSTOP sinyalini gönderecek ve böylece işlemlerini durduracaktır; boşta kalır:
- # tar -cf backup.tar / backups/* #press Ctrl+Z
- # jobs
- Arka planda yukarıda askıya alınan komutu çalışmaya devam etmek için bg komutunu kullanalım :
- # bg
- Ön plana bir arka plan işlemi göndermek için, fg komutunu aşağıdaki gibi iş kimliği ile birlikte kullanalım :
- # jobs
- # fg %1

```
[root@tecmint ~]# tar -cf backup.tar /backups/*
tar: Removing leading '/' from member names
^Z
[1]+  Stopped                  tar -cf backup.tar /backups/*
[root@tecmint ~]# jobs
[1]+  Stopped                  tar -cf backup.tar /backups/*
[root@tecmint ~]# bg
[1]+ tar -cf backup.tar /backups/* &
[root@tecmint ~]# jobs
[1]+ Running                   tar -cf backup.tar /backups/* &
[root@tecmint ~]# fg %1
tar -cf backup.tar /backups/*

```

Linux Arka Plan Süreç İşleri

- **Linux'ta Bir Sürecin Durumları**
- Yürütme sırasında bir süreç, ortamına / koşullarına bağlı olarak bir durumdan diğerine değişir. Linux'ta bir işlem aşağıdaki olası durumlara sahiptir:
 - **Çalışıyor** - burada ya çalışıyor (sisteme mevcut işlem) ya da çalışmaya hazır (CPU'lardan birine atanmayı bekliyor).
 - **Bekliyor** - bu durumda, bir işlem, bir olayın gerçekleşmesini veya bir sistem kaynağını bekler. Ek olarak, çekirdek ayrıca iki tür bekleme işlemi arasında ayırm yapar; Kesintisiz bekleme süreçleri - sinyaller ve kesintisiz bekleme süreçleri ile kesintiye uğrayabilir - doğrudan donanım koşullarında bekler ve herhangi bir olay / sinyal tarafından kesintiye uğratılamaz.
 - **Durduruldu** - bu durumda, genellikle bir sinyal alınarak bir işlem durduruldu. Örneğin, hata ayıklanmakta olan bir süreç.
 - **Zombie** - burada bir işlem öldü, durduruldu, ancak işlem tablosunda hala bir girişi var.

- Linux'ta Etkin İşlemler Nasıl Görüntülenir ?
- Sistemde çalışan işlemleri görüntülemek / listelemek için birkaç Linux aracı vardır, iki geleneksel ve iyi bilinen ps ve top komutlarıdır:
- 1. ps Komutu
- Aşağıda gösterildiği gibi, sistemdeki aktif işlemlerin bir seçimi hakkında bilgi görüntüler:
 - # ps
 - # ps -e \ had

```
[root@tecmint ~]# ps
  PID TTY          TIME CMD
 2109 pts/0        00:00:00 bash
 2200 pts/0        00:00:01 node
 2321 pts/0        00:00:00 ps
[root@tecmint ~]# ps -e | head
  PID TTY          TIME CMD
    1 ?        00:00:01 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 kworker/u2:0
    7 ?        00:00:00 migration/0
    8 ?        00:00:00 rcu_bh
    9 ?        00:00:00 rcuob/0
   10 ?        00:00:00 rcu_sched
[root@tecmint ~]# █
```

Linux Etkin İşlemlerini Listeleme

- 2. top - Sistem İzleme Aracı
- top, aşağıdaki gösterildiği gibi çalışan bir sistemin dinamik gerçek zamanlı görünümünü sunan güçlü bir araçtır:
- # top

```
top - 08:53:06 up 16 min,  1 user,  load average: 0.20, 0.28, 0.35
Tasks: 238 total,  1 running, 237 sleeping,  0 stopped,  0 zombie
%Cpu(s): 5.4 us, 1.0 sy, 0.0 ni, 93.0 id, 0.6 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 3742792 total, 1144416 free, 1236544 used, 1361832 buff/cache
KiB Swap: 5631996 total, 5631996 free,          0 used. 2249948 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2469	aaronki+	20	0	1757760	168424	56580	S	11.6	4.5	0:37.61	cinnamon
3691	aaronki+	20	0	479484	35208	26268	S	6.3	0.9	0:00.42	gnome-scre+
1946	root	20	0	463000	96932	85920	S	4.3	2.6	0:21.98	Xorg
170	root	20	0	0	0	0	S	1.3	0.0	0:00.69	kworker/u1+
6	root	20	0	0	0	0	S	1.0	0.0	0:00.85	kworker/u1+
921	root	20	0	449740	19428	14048	S	0.7	0.5	0:01.05	NetworkMan+
1743	shinken	20	0	1557824	31040	6504	S	0.7	0.8	0:06.95	shinken-sc+
1817	shinken	20	0	1631460	32172	7856	S	0.7	0.9	0:04.32	shinken-re+
7	root	20	0	0	0	0	S	0.3	0.0	0:01.17	rcu_sched
1865	shinken	20	0	1632116	32604	7284	S	0.3	0.9	0:05.92	shinken-br+
1908	shinken	20	0	1557024	30232	6556	S	0.3	0.8	0:03.24	shinken-re+
1953	root	20	0	1633896	34552	5712	S	0.3	0.9	0:04.19	shinken-ar+
2082	shinken	20	0	1631232	29112	4728	S	0.3	0.8	0:00.20	shinken-po+
3684	aaronki+	20	0	41908	3808	3104	R	0.3	0.1	0:00.04	top
1	root	20	0	119696	5924	4040	S	0.0	0.2	0:01.45	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0

- 3. glances - Sistem İzleme Aracı
- glances, gelişmiş özelliklere sahip nispeten yeni bir sistem izleme aracıdır:

```
# glances
```

tecmint (CentOS Linux 7.0.1406 64bit / Linux 3.10.0-123.el7.x86_64)								Uptime: 0:52:45	
CPU	0.0%	Load	1-core	Mem	26.0%	active:	369M	Swap	0.0%
user:	0.0%	1 min:	0.19	total:	994M	inactive:	377M	total:	2.70G
system:	0.0%	5 min:	0.14	used:	258M	buffers:	51.8M	used:	0
idle:	100.0%	15 min:	0.14	free:	736M	cached:	551M	free:	2.70G
Network	Rx/s	Tx/s	Tasks	81 (107 thr), 1 run, 80 slp, 0 oth					
enp0s3	176b	720b		VIRT	RES	CPU%	MEM%	PID	USER
enp0s8	160b	376b		229M	14M	2.9	1.4	2275	root
lo	0b	0b		142M	5M	0.3	0.5	2106	root
Disk I/O	In/s	Out/s		45M	6M	0.0	0.6	1	root
sdal	0	0		0	0	0.0	0.0	2	root
sda2	0	0		0	0	0.0	0.0	3	root
sr0	0	0		0	0	0.0	0.0	5	root
				0	0	0.0	0.0	6	root
				0	0	0.0	0.0	7	root
Mount	Used	Total		0	0	0.0	0.0	8	root
/	3.22G	28.7G		0	0	0.0	0.0	9	root
/run	6.59M	497M		0	0	0.0	0.0	10	root
/user/0	0	99.4M		0	0	0.0	0.0	11	root
/var/1003	0	99.4M		0	0	0.0	0.0	12	root
/selinux	0	0		0	0	0.0	0.0	13	root
				0	0	0.0	0.0		khelper

Press 'h' for help

2017-03-28 10:06:10

- \$ pgrep -u tecmint top
- \$ kill 2308
- \$ pgrep -u tecmint top
- \$ pgrep -u tecmint glances
- \$ pkill glances
- \$ pgrep -u tecmint glances

```
[root@tecmint ~]# pgrep -u tecmint top  
2308  
[root@tecmint ~]#  
[root@tecmint ~]# kill 2308  
[root@tecmint ~]#  
[root@tecmint ~]# pgrep -u tecmint top  
[root@tecmint ~]#  
[root@tecmint ~]# pgrep -u tecmint glances  
2311  
[root@tecmint ~]#  
[root@tecmint ~]# pkill glances  
[root@tecmint ~]#  
[root@tecmint ~]# pgrep -u tecmint glances  
[root@tecmint ~]#  
[root@tecmint ~]#
```

- İşlemlere Sinyal Gönderme
- Linux'ta süreçleri kontrol etmenin temel yolu, onlara sinyal göndermektir. Çalışmakta olan tüm sinyalleri görüntülemek için bir işleme gönderebileceğiniz birden fazla sinyal vardır:
 - \$ kill -l

```
[root@tecmint ~]# kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
 11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
 16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
 21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
 26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
 31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
 38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
 43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
 58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
 63) SIGRTMAX-1  64) SIGRTMAX
[root@tecmint ~]#
```

- Bir işleme sinyal göndermek için, daha önce bahsettiğimiz kill, pkill veya pgrep komutlarını kullanmamız gerekiyor. Ancak programlar, sinyallere yalnızca bu sinyalleri tanıyacak şekilde programlanıslarsa yanıt verebilir.
- Ve çoğu sinyal, sistem tarafından dahili kullanım veya kod yazarken programcılar içindir. Şimdi de bir sistem kullanıcısı için yararlı olan sinyallere bakalım :

- SIGHUP 1 - kontrol terminali kapatıldığında bir işleme gönderilir.
- SIGINT 2 - bir kullanıcı [Ctrl + C] tuşlarına basarak işlemi durdurduğunda kontrol terminali tarafından bir işleme gönderilir.
- SIGQUIT 3 - kullanıcı bir çıkış sinyali [Ctrl + D] gönderirse bir işleme gönderilir.
- SIGKILL 9 - bu sinyal bir işlemi hemen sonlandırır (öldürür) ve işlem herhangi bir temizleme işlemi gerçekleştirmez.
- SIGTERM 15 - bu bir program sonlandırma sinyali (kill bunu varsayılan olarak gönderecektir).
- SIGTSTP 20 - durmasını talep etmek için kontrol terminali tarafından bir işleme gönderilir (terminal durdurma); [Ctrl + Z] 'ye basarak kullanıcı tarafından başlatılır.

- Aşağıdakiler, Firefox uygulamasını donduğunda PID'sini kullanarak öldürmek için kill komutları örnekleridir:
- \$ pidof firefox
- \$ kill 9 2687
- OR
- \$ kill –KILL 2687
- OR
- \$ kill –SIGKILL 2687
- Bir uygulamayı adını kullanarak öldürmek için, pkill veya killall kullanalım
- \$ pkill firefox
- \$ killall firefox

- Linux İşlem Önceliğini Değiştirme
- Linux sisteminde, tüm aktif işlemlerin bir önceliği ve belirli bir iyi değeri vardır. Daha yüksek önceliğe sahip işlemler, normalde daha düşük öncelikli işlemlere göre daha fazla CPU zamanı alır.
- Ancak, kök ayrıcalıklarına sahip bir sistem kullanıcısı bunu nice ve renice komutlarıyla etkileyebilir.
- Üst komutun çıktısından NI, işlem güzel değerini gösterir:
- \$ top

```

top - 08:53:06 up 16 min,  1 user,  load average: 0.20, 0.28, 0.35
Tasks: 238 total,   1 running, 237 sleeping,   0 stopped,   0 zombie
%Cpu(s): 5.4 us, 1.0 sy, 0.0 ni, 93.0 id, 0.6 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 3742792 total, 1144416 free, 1236544 used, 1361832 buff/cache
KiB Swap: 5631996 total, 5631996 free,      0 used. 2249948 avail Mem

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2469	aaronki+	20	0	1757760	168424	56580	S	11.6	4.5	0:37.61	cinnamon
3691	aaronki+	20	0	479484	35208	26268	S	6.3	0.9	0:00.42	gnome-scre+
1946	root	20	0	463000	96932	85920	S	4.3	2.6	0:21.98	Xorg
170	root	20	0	0	0	0	S	1.3	0.0	0:00.69	kworker/u1+
6	root	20	0	0	0	0	S	1.0	0.0	0:00.85	kworker/u1+
921	root	20	0	449740	19428	14048	S	0.7	0.5	0:01.05	NetworkMan+
1743	shinken	20	0	1557824	31040	6504	S	0.7	0.8	0:06.95	shinken-sc+
1817	shinken	20	0	1631460	32172	7856	S	0.7	0.9	0:04.32	shinken-re+
7	root	20	0	0	0	0	S	0.3	0.0	0:01.17	rcu_sched
1865	shinken	20	0	1632116	32604	7284	S	0.3	0.9	0:05.92	shinken-br+
1908	shinken	20	0	1557024	30232	6556	S	0.3	0.8	0:03.24	shinken-re+
1953	root	20	0	1633896	34552	5712	S	0.3	0.9	0:04.19	shinken-ar+
2082	shinken	20	0	1631232	29112	4728	S	0.3	0.8	0:00.20	shinken-po+
3684	aaronki+	20	0	41908	3808	3104	R	0.3	0.1	0:00.04	top
1	root	20	0	119696	5924	4040	S	0.0	0.2	0:01.45	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0

- Bir işlem için iyi bir değer ayarlamak için nice komutunu kullanırız. Normal kullanıcıların sahip oldukları işlemelere sıfırdan 20'ye kadar iyi bir değer atayabileceğini unutmayın.
- Yalnızca kök(root) kullanıcı negatif iyi değerleri kullanabilir.
- Bir işlemin önceliğini yeniden belirlemek için, renice komutunu aşağıdaki gibi kullanalım :
- \$ renice +8 2687
- \$ renice +8 2103

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- Thread'ler
 - Thread'lerin sağladığı faydalar
 - Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
 - Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
 - Thread kütüphaneleri
 - Dolaylı thread oluşturma
 - Thread çalışma kuralları
 - Windows ve Linux thread'leri

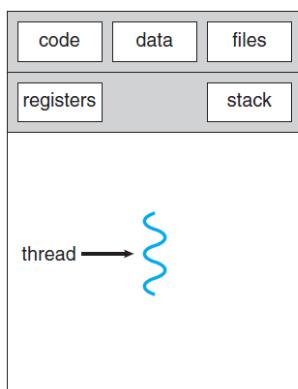
Thread'ler

- Bir **thread**, program counter, bir grup register ve bir stack yapısına sahiptir.
- Thread'ler, program kodunu, data kısmını, dosyalar gibi işletim sistemi kaynaklarını ortak kullanır.
- Klasik process'ler tek thread'e sahiptirler.
- Eğer bir process, birden fazla thread'e sahipse birden fazla görevi eşzamanlı yapabilir.
- Günümüzdeki modern bilgisayarlarda çalışan yazılım uygulamalarının çoğu multithread çalışmaları.
- Uygulamalar, çok sayıda thread'e sahip tek process şeklinde geliştirilirler.
- Bir Web browser, bir thread ile veri aktarımı yapabilir, başka thread ile verileri ekranда görüntüleyebilir.

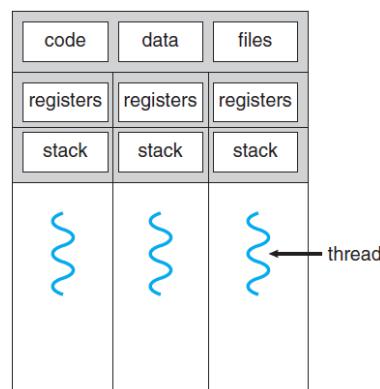
3

Thread'ler

- Bir kelime işlemci uygulaması, **bir thread ile klavyeden giriş alabilir, bir thread ile spell check yapabilir ve başka bir thread ile ekran görüntüsünü düzenleyebilir.**
- Her thread, **paylaşmadan kullandığı** kendisine ait bileşenlere sahiptir.



single-threaded process

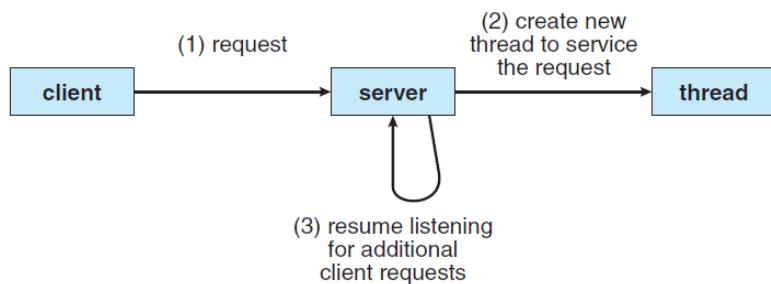


multithreaded process

4

Thread'ler

- Uygulamalar, multicore sistemlerin **kapasitesini maksimum kullanacak şekilde tasarlanabilir**.
- Bir Web sunucu process'i multithreaded çalışırsa, her gelen istek için ayrı bir thread oluşturulur ve process portu dinlemeye devam eder.
- **Çoğu işletim sisteminin kernel'ı multithreaded yapıdadır** ve cihazların yönetimi, hafıza yönetimi veya **interrupt işlemi** aynı anda yapılabilir.



5

Konular

- Thread'ler
- **Thread'lerin sağladığı faydalar**
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

6



Thread'lerin sağladığı faydalar

- Thread'lerin sağladığı faydalar 4 kategori halinde ifade edilebilir:
 1. **Responsiveness:** Kullanıcı etkileşimli uygulamalarda, bir kısım bloklanmış, kilitlenmiş veya uzun süren işlem yürütüyorSA, kullanıcı ile etkileşim yapan başka bir kısım çalışmasını sürdürür.
Sistemin cevap verebilirlik özelliği artmış olur.
 2. **Resource sharing:** Process'ler kaynaklarını shared memory veya message passing teknikleri aracılığıyla paylaşabilirler.
Thread'ler ait oldukları process'in sahip olduğu hafıza alanını ve diğer kaynakları paylaşabilirler.

7



Thread'lerin sağladığı faydalar

- Thread'lerin sağladığı faydalar 4 kategori halinde ifade edilebilir:
 3. **Economy:** Bir process oluştururken **hafıza ve kaynak tahsis edilmesi** maliyeti yüksek bir iştir.
Thread'ler ait oldukları process'in kaynaklarını paylaştıklarından dolayı context switch daha düşük maliyetle yapılır.
(Solaris işletim sisteminde, **thread oluşturma 30 kat daha hızlıdır** ve **thread'lerde context switch 5 kat daha hızlıdır.**)
 4. **Scalability:** **Çok işlemcili mimarilerde thread'ler farklı core'lar üzerinde eşzamanlı çalışabilir.**
Ancak, tek thread yapısına sahip process sadece bir işlemci üzerinde çalışabilir.

8

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- **Multicore programlama**
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

9

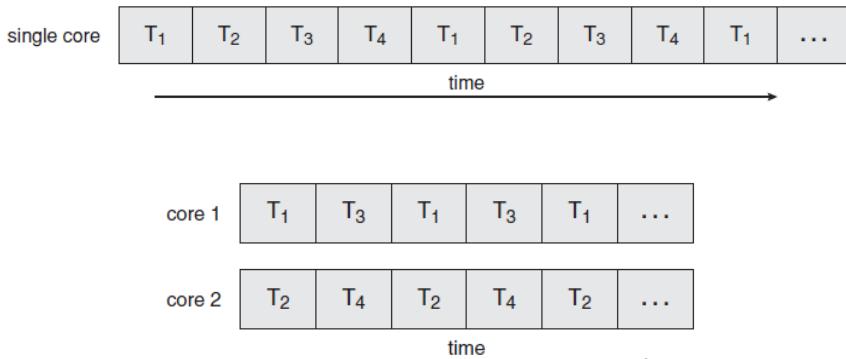
Multicore programlama

- Bilgisayar tasarımindaki en önemli gelişmelerden birisi, çok işlemcili sistemlerin geliştirilmesidir.
- Son zamanlarda, **tek chip içeresine birden fazla core yerleştirilmektedir**. Bu tür sistemler **multicore** veya **multiprocessor** olarak adlandırılır.
- **Her bir core işletim sistemi için ayrı bir işlemci olarak görünür**.
- **Bir core** üzerinde çalışan 4 thread'e sahip bir uygulama için **eşzamanlı çalışma**, **thread'lerin belirli aralıklarla çalıştırılmasını ifade eder**.
- **Çok core'a sahip sistemlerde eşzamanlı çalışma, her core'a bir thread atanarak thread'lerin paralel çalışmasını ifade eder**.
- **Parallelism**, birden fazla görevin **eşzamanlı** yapılmasını ifade eder.
- **Concurrency**, birden fazla görev arasında kısa aralıklarla geçiş yaparak **birlikte ilerletilmesini** ifade eder.

10

Multicore programlama

- Sistemdeki core sayısı arttıkça eşzamanlı gerçekleştirilen görev sayısı da artacaktır.



11

Multicore programlama

- Amdahl kuralı** core sayısına göre bir sistemdeki performans artışını aşağıdaki gibi ifade etmektedir:

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- Burada, S uygulamada seri çalışması zorunlu olan kısmın oranını, N ise core sayısını ifade eder.
- Bir uygulamada, %75 paralel ve %25 seri çalışıyorsa (S=0,25), **2 core'a** (N=2) sahip sistemde bu uygulamayı çalıştırınca **1,6 kat hız artar**.
- Core sayısı 4** olduğunda, **2.28 kat hız artışı sağlanır**.
- Core sayısı sonsuza giderken hız artışı (1/S)** 'e doğru gider.
- Intel CPU'lar** her core için 2 thread, **Oracle T4 CPU** ise 4 thread destekler.

12

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

13

Multicore programlamanın zorlukları

- **İşletim sistemi tasarımcıları** multicore sistemlerin performansını artırmak için **scheduling algoritmaları yazmak zorundadır.**
- Uygulama geliştiricilerin mevcut programları değiştirmeleri ve **yeni programları multihreaded şekilde tasarlamaları gerekmektedir.**
- Multicore programlamada 5 önemli zorluk vardır:
 - **Identifying tasks:** Uygulamalarda eşzamanlı çalışabilecek ayrı alanların bulunması gereklidir. Bu alanlar farklı core'lar üzerinde paralel çalışacaktır.
 - **Balance:** Programcılar görevleri ayırtırırken **iş yükünün eşit dağıtılması gereklidir.**
 - **Data splitting:** Verilerin farklı core'lar üzerinde çalışan görevler tarafından erişilecek ve işlem yapılacak şekilde ayrıştırılması gereklidir.
 - **Data dependency:** Bir görevin erişeceği verinin diğer görevlerle bağımlılığının incelenmesi gereklidir.
 - **Testing and debugging:** Multihreaded çalışan programların **test ve debug işlemi daha zordur.**

14

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - **Paralel çalışma türleri**
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

15

Paralel çalışma türleri

- Genel olarak **data parallelism** ve **task parallelism** olarak iki tür paralel çalışma türü vardır.
- **Data parallelism**, aynı **veri kümesine ait parçaların core'lara dağıtılması** ve aynı tür işlemin eşzamanlı yürütülmesine odaklanır.
- N elemanlı bir **dizinin toplamı** için **iki core** kullanılacaksa, **[0]..[(N/2)-1]** eleman 1.core'da, **[N/2]..[N-1]** eleman 2.core'da toplanır.
- **Task parallelism**, core'lara **görevlerin (thread'ler) dağıtımasına** odaklanır.
- Her thread **ayıri bir işlemi gerçekleştirir**. Farklı thread'ler aynı veride veya farklı veride çalışabilir.
- **Aynı dizi** elemanları üzerinde **farklı istatistiksel hesaplamalar** yapan thread'ler aynı veriyi kullanır farklı core'larda çalışır.

16

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- **Multithreading modelleri**
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

17

Multithreading modelleri

- Thread desteği kullanıcı seviyesinde **user thread'**ler için veya kernel seviyesinde **kernel thread'**ler için sağlanabilir.
- **User thread'leri kullanıcı uygulamaları tarafından, kernel thread'leri ise işletim sistemi tarafından gerçekleştirilir.**
- Windows, Linux, Unix, Mac OS X ve Solaris gibi işletim sistemleri **kernel thread'leri destekler**.
- Kernel thread'leri ile user thread'leri arasında aşağıdaki **ilişkilendirme modellerinden** birisinin oluşturulması zorundadır.
 - Many-to-one model
 - One-to-one model
 - Many-to-many model

18

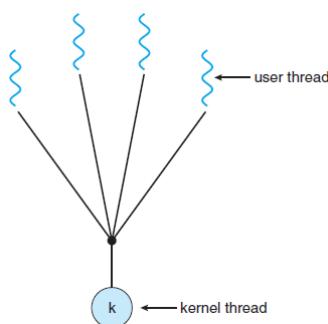
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

19

Many-to-one

- Many-to-one modelinde, **çok sayıda kullanıcı thread'i bir tane kernel thread'i ile eşleştirilir (Solaris işletim sistemi kullanır.)**.



- Eğer **bir thread sistem çağrısını bloklarsa** tüm process bloklanmış olur.
- Aynı anda sadece bir tane kullanıcı thread'i kernel thread'e erişebilir.
- Sadece bir kernel thread'i kullanıldığı için **multicore sistemlerde birden fazla thread için eşzamanlı çalışma yapılamaz**.

20

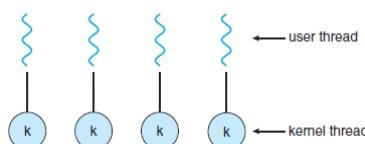
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

21

One-to-one

- One-to-one modelinde, **bir kullanıcı thread'i bir kernel thread'i ile eşleştirilir (Linux, Windows işletim sistemleri kullanır.)**.



- Eğer **bir thread sistem çağrımasını bloklarsa** diğer thread'ler çalışmasına devam eder.
- Birden fazla kernel thread'inin **multicore sistemlerde eşzamanlı çalışmasına izin verir**.
- **Bir user thread için bir kernel thread oluşturulması gereklidir.**

22

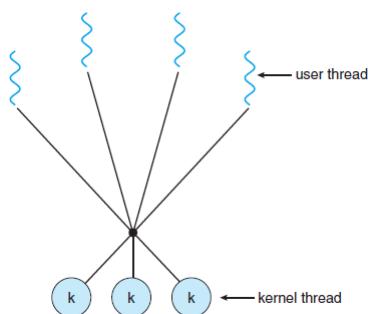
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - **Many-to-many**
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

23

Many-to-many

- Many-to-many modelinde, **çok sayıda kullanıcı thread'i ile aynı sayıdaki veya daha az sayıdaki kernel thread'i eşleştirilir (Solaris 9, Unix işletim sistemleri kullanır.)**.



- **Bir thread sistem çağrısını bloklarsa, kernel başka bir thread'i çalıştırır.**

24

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- **Thread kütüphaneleri**
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

25

Thread kütüphaneleri

- Thread kütüphanesi, **programcıya thread oluşturmak ve yönetmek için API sağlar.**
- Thread kütüphanesi oluşturulurken **iki farklı yaklaşım kullanılır:**
 - Tüm thread kütüphanesi **kullanıcı alanında** oluşturulur ve **kernel desteği yoktur.**
 - İşletim sisteminin doğrudan desteklediği **kernel seviyesinde kütüphane oluşturulur.**
- Çoklu thread oluşturmak için iki farklı strateji kullanılmaktadır:
 - **Asenkron threading:** Parent, yeni bir child thread oluşturduğunda **eşzamanlı olarak çalışmasını sürdürür.**
 - **Senkron threading:** Parent, child process oluşturduğunda **çalışmasını durdurur** ve tüm child process'ler sonlandığında çalışmasına devam eder (**fork-join strategy**).
- **Asenkron threading**, thread'ler arasında **veri paylaşımı az olduğunda**, **senkron threading** ise threadler arasında **veri paylaşımı çok olduğunda** **kullanılır.**

26



Thread kütüphaneleri

- Günümüzde 3 temel thread kütüphanesi kullanılmaktadır:
 - POSIX Pthreads
 - Windows
 - Java
- **Pthreads**, user-level veya kernel-level thread kütüphanesi sağlar.
- **Windows threads**, kernel-level thread kütüphanesi sağlar.
- **Java threads**, user-level thread kütüphanesi sağlar.

27



Thread kütüphaneleri

Pthreads

- **Pthreads, IEEE1003.1c standarıyla thread oluşturma ve yönetmek için tanımlanan API'dir.**
- **Linux, Unix, Mac OS X ve Solaris** işletim sistemleri Pthreads standartını kullanır.
- **Windows** Pthreads standartını **desteklemez**.
- **Pthreads standartında thread'lerin hepsi ayrı fonksiyonlar halinde oluşturulur.**
- **Tüm thread'ler global scope'ta tanımlanan verileri paylaşırlar.**

28

Thread kütüphaneleri

Pthreads - Örnek

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param),
        sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```

Pthreads programları için kullanılan header file

Yeni thread için ID tanımlar.

Yeni thread için özellikleri (stack size, ...) belirler.

Varsayılan özellikler (senkron thread, sistem stack addr, ...)

Yeni thread başlatıldı.

Yeni thread için başlama noktası.

Komut satırında girilen parametre

Komut satırında girilen parametre

fork-join stratejisi

Dönen değer

Thread kütüphaneleri

Pthreads - Örnek

- Önceki örnekte bir thread oluşturulmuştur. Çok sayıda thread aşağıdaki örnekteki gibi oluşturulabilir.

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Oluşturulacak thread sayısı

10 thread tanımlandı.

10 thread için fork-join yapıldı.

Thread kütüphaneleri

Windows threads

- Windows thread kütüphanesi ile thread oluşturma Pthreads ile birçok açıdan benzerlik gösterir.
- **Thread'lerin hepsi ayrı fonksiyonlar halinde oluşturulur.**
- **Tüm thread'ler global scope'ta tanımlanan verileri paylaşırlar.**

31

Thread kütüphaneleri

Windows threads - Örnek

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }
}
```

* create the thread */
ThreadHandle = CreateThread(
 NULL, /* default security attributes */
 0, /* default stack size */
 Summation, /* thread function */
 &Param, /* parameter to thread function */
 0, /* default creation flags */
 &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
 /* now wait for the thread to finish */
 WaitForSingleObject(ThreadHandle, INFINITE);
}
/* close the thread handle */
CloseHandle(ThreadHandle);
printf("sum = %d\n",Sum);

fork-join stratejisi
WaitForMultipleObjects() ile birden fazla
thread senkron çalıştırılabilir.

32

Thread kütüphaneleri

Java threads

- Java thread'leri, JVM (Java Virtual Machine) kullanılabilen tüm sistemlerde çalışır.
- Java thread API, Windows, Linux, Unix, Mac OS X ve Android için kullanılabilir.
- Java thread'leri arasında **veri paylaşımı parameter passing** ile yapılır.
- Java ile iki farklı teknik kullanılarak thread oluşturulabilir:
 - Thread sınıfından yeni bir sınıf türetilir ve run() metodu override yapılır.
 - Runnable arayüzüünü kullanan bir sınıf oluşturulur. (yaygın kullanılır.)

```
public interface Runnable  
{  
    public abstract void run();  
}
```

Aynı thread için override yapmak gereklidir.

33

Thread kütüphaneleri

Java threads – Örnek

```
class Sum  
{  
    private int sum;  
  
    public int getSum() {  
        return sum;  
    }  
  
    public void setSum(int sum) {  
        this.sum = sum;  
    }  
}  
  
class Summation implements Runnable  
{  
    private int upper;  
    private Sum sumValue;  
  
    public Summation(int upper, Sum sumValue) {  
        this.upper = upper;  
        this.sumValue = sumValue;  
    }  
  
    public void run() {  
        int sum = 0;  
        for (int i = 0; i <= upper; i++)  
            sum += i;  
        sumValue.setSum(sum);  
    }  
}  
  
public class Driver  
{  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            if (Integer.parseInt(args[0]) < 0)  
                System.err.println(args[0] + " must be >= 0.");  
            else {  
                Sum sumObject = new Sum();  
                int upper = Integer.parseInt(args[0]);  
                Thread thrd = new Thread(new Summation(upper, sumObject));  
                thrd.start();  
                try {  
                    thrd.join();  
                    System.out.println("The sum of "+upper+" is "+sumObject.getSum());  
                } catch (InterruptedException ie) {}  
            }  
        } else  
            System.err.println("Usage: Summation <integer value>");  
    }  
}
```

Aynı thread için gereklidir.

Yeni thread oluşturuldu.

Yeni thread başlıyor.
run() çağırılır.

fork-join stratejisi
Thread'ler senkron çalışıyor.

34

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- **Dolaylı thread oluşturma**
- Thread çalışma kuralları
- Windows ve Linux thread'leri

35

Dolaylı thread oluşturma

- Multicore işlemcilerdeki gelişmelerle birlikte, uygulamalar yüzlerce hatta binlerce thread içermektedirler.
- Çok sayıda thread ile uygulama geliştirmek oldukça zordur ve hata olma olasılığı vardır.
- Thread oluşturma işinin **uygulama geliştiriciler yerine, compiler tarafından yapılması** günümüzde giderek popüler hale gelmektedir.
- Bu stratejiye **implicit threading** denilmektedir.

36

Dolaylı thread oluşturma

Thread pools

- Multithreaded bir Web sunucusu, gelen isteklerin her birisi için yeni thread oluşturur.
- Multihreaded bir Web sunucusu, eşzamanlı çok sayıda istemciye servis sağlayabilir.
- Gelen istek sayısı çok artarsa** sistem kaynakları (CPU time, memory, ...) tükenir.
- Multithreaded sistemlerde belirli sayıda thread oluşturulmasına izin vermek için thread pool oluşturulur.**
- Yeni istek geldiğinde thread pool içerisinde kullanılabilir **thread varsa cevaplanır**, yoksa bir thread'in serbest hale gelmesi beklenir.
- Varolan thread'in kullanımı yeni thread oluşturmaya göre daha hızlıdır.**

37

Dolaylı thread oluşturma

Thread pools - Windows

- Örnekte `PoolFunction()` fonksiyonu thread olarak çalıştırılmaktadır.

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```

- Thread pool API içerisindeki `QueueUserWorkItem()` fonksiyonu, **pool içerisindeki bir thread ile PoolFunction() fonksiyonunu çalıştırır.**

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

- Parametreler:**

- LPTHREAD_START_ROUTINE, thread olarak çalışacak fonksiyonun pointer'i
- PVOID, Gönderilecek parametre
- ULONG, Bayrak bitleri (bekleme süresi, I/O gerekliliği, ...)

38

Dolaylı thread oluşturma

OpenMP

- OpenMP, C, C++ ve Fortran için yazılmış bir grup compiler direktifidir.
- Shared memory yaklaşımını kullanılır.
- **OpenMP ile paralel çalışacak kod blokları tanımlanır.**
- OpenMP ile `#pragma omp parallel` direktifi paralel çalışacak bloğun hemen başında kullanılır.
- OpenMP farklı türdeki deyimler için ayrı direktifler kullanır.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

- OpenMP, Linux, Windows ve Mac OS X sistemlerdeki çok sayıda açık kaynak ve ticari compiler'larda kullanılabilir.

39

Dolaylı thread oluşturma

OpenMP - Örnek

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```

printf() deyimi paralel çalışacaktır.

40

Dolaylı thread oluşturma

Grand central dispatch

- Grand central dispatch (GCD), **Apple Mac OS X ve iOS işletim sistemlerinde paralel çalışacak kısımları belirlemek için kullanılır.**
- Paralel çalışacak bloğun belirtilmesi için ^ simbolü kullanılır.

```
^{ printf("I am a block"); }
```
- GCD blokları **dispatch queue** içerisine yerleştirir.
- Bir blok kuyruktan atılırsa, tekrar thread havuzundaki bir thread'e atanabilir.
- Dispatch queue, **serial** veya **concurrent** şeklinde oluşturulabilir.
- **Serial queue**, FIFO çalışır ve **sadece bir blok kuyruktan alınabilir**.
- **Concurrent queue**, FIFO çalışır ve kuyruktan **birden fazla blok aynı anda alınabilir**.

41

Dolaylı thread oluşturma

Grand central dispatch

- Concurrent queue, önceliklendirilmiş 3 tane dispatch kuyruğa sahiptir: **low**, **default** ve **high**.
- Önceliklendirme ile blokların önem derecesi belirlenmektedir.
- Aşağıdaki örnekte, default önceliğe sahip concurrent kuyruğa bir blok eklenmektedir.

```
dispatch_queue_t queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```

Kuyruğa blok eklendi.

default öncelikli concurrent kuyruk

42

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- **Thread çalışma kuralları**
- Windows ve Linux thread'leri

43

Thread çalışma kuralları

fork() ve exec() sistem çağrıları

- **Bazı Unix sistemlerde, iki tür fork() çağrısı vardır.**
- Birisi ile **tüm thread'ler duplicate yapılır**, diğeri ile **sadece fork() ile başlatılan thread duplicate yapılır**.
- **exec()** sistem çağrısı ile **tüm thread'leri ile birlikte process duplicate yapılır**.

44

Thread çalışma kuralları

Signal handling

- Unix sistemlerde bir **signal** belirli bir olayın gerçekleştiğini gösterir.
- **Oluşan olaya karşılık gelen sinyal bir process'e iletilir.**
- Oluşan sinyal, **senkron** veya **asenkron** alınabilir.
- **Senkron sinyal, sinyalin oluşmasına neden olan olayı gerçekleştiren process'e iletilir.**
- Senkron sinyale illegal hafıza erişimi veya 0'a bölme verilebilir.
- **Asenkron sinyal, sinyali oluşturan process'ten başka bir process'e iletilir.**
- Asenkron sinyale <ctrl><C> tuşlarına birlikte basmak verilebilir.

45

Thread çalışma kuralları

Signal handling

- İşletim sistemlerinde **sinyaller farklı hedeflere gönderebilir**:
 - Process içerisindeki **sadece bir thread'e gönderebilir** (Senkron).
 - Process içerisindeki **tüm thread'lere gönderebilir** <ctrl><C>.
 - Process içerisindeki **bazı thread'lere gönderebilir**.
 - Bir process için **tüm sinyalleri almak üzere bir thread atanabilir**.
- **Senkron sinyaller** sadece **oluşturan thread'e** gönderilir.
- Aşağıdaki UNİX fonksiyonu ile ID değeri verilen process'e iletilir.

```
kill(pid_t pid, int signal)
```
- POSIX Pthreads ile aşağıdaki fonksiyon kullanılır.

```
pthread_kill(pthread_t tid, int signal)
```

45

Thread çalışma kuralları

Thread iptal etme

- Bir thread'in çalışması tamamlanmadan iptal edilebilir.
- İstenen bir sonucun bir thread tarafından bulunması halinde diğerleri iptal edilebilir.
- Bir Web sayfası yüklenirken stop butonuna basıldığında process içerisindeki tüm thread'ler iptal edilir.
- Bir thread başka bir thread'i aniden sonlandırabilir (**Asenkron cancellation**).
- Bir thread başka bir thread'in kendi kendisini sonlandırmasını sağlayabilir (**Deferred cancellation**).

47

Thread çalışma kuralları

Thread iptal etme

- Pthreads aşağıdaki tabloda verilen üç farklı iptal etme modunu destekler.

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
...  
  
/* cancel the thread */  
pthread_cancel(tid);
```

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

48

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- **Windows ve Linux thread'leri**

49

Windows ve Linux thread'leri

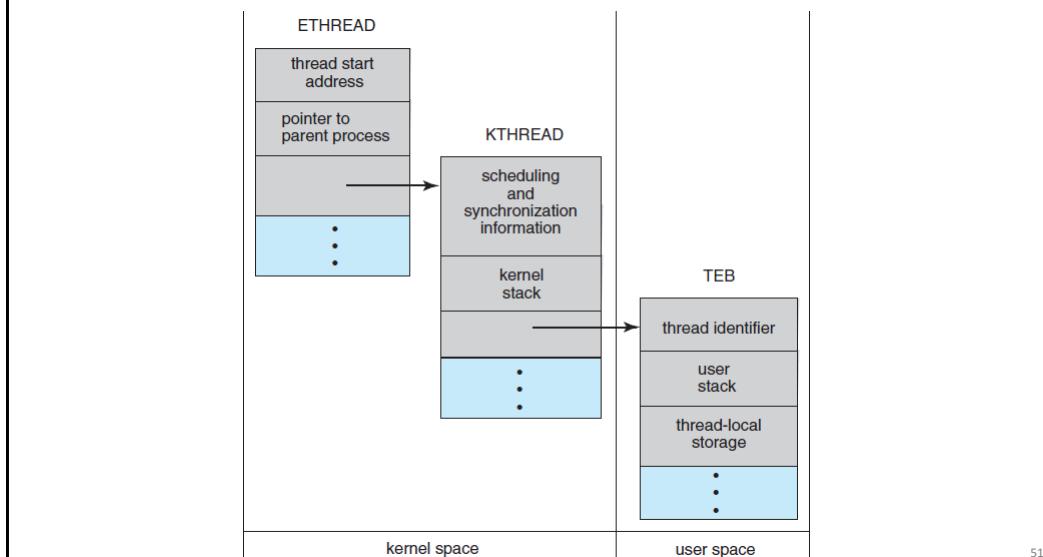
Windows thread'leri

- Microsoft işletim sistemleri için temel API olarak **Windows API** kullanılır.
- Bir Windows uygulaması ayrı bir process olarak çalışır ve **her process bir veya daha fazla thread içerebilir.**
- **Windows, kullanıcı thread'leri ile kernel thread'leri arasında one-to-one eşleştirme yapar.**
- Bir thread için ayrılan **register kümesi, stack, depolama alanı, context** olarak adlandırılır.
- Windows bir thread için aşağıdaki veri yapılarını kullanır:
 - **ETHREAD:** Yürüttü thread blok
 - **KTHREAD:** Kernel thread blok
 - **TEB:** Thread environment (ortam) blok

50

Windows ve Linux thread'leri

Windows thread'leri



51

Windows ve Linux thread'leri

Linux thread'leri

- Linux, `fork()` sistem çağrısının yanı sıra `clone()` sistem çağrıları ile thread oluşturabilir.
- Linux, `clone()` ile yeni bir görev başlattığında, parent task ile child task arasında paylaşım miktarını da gönderir.
- **Dosya sistemi, hafıza aralığı, sinyaller veya açık olan dosyalar paylaşılabilir.**
- Görevler, Linux kernel içerisinde bir veri yapısına sahiptir (`struct task_struct`) ve açık dosyalar, virtual memory ve sinyal bilgilerini gösterir.
- Linux, `fork()` ile **yeni bir görev başlattığında, parent task veri yapısı kopyalanır.**

52

Windows ve Linux thread'leri

Linux thread'leri

- Linux, clone() ile yeni bir görev başlattığında bayrak bitlerine göre veri yapısı oluşturulur.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- Thread'ler
 - Thread'lerin sağladığı faydalar
 - Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
 - Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
 - Thread kütüphaneleri
 - Dolaylı thread oluşturma
 - Thread çalışma kuralları
 - Windows ve Linux thread'leri

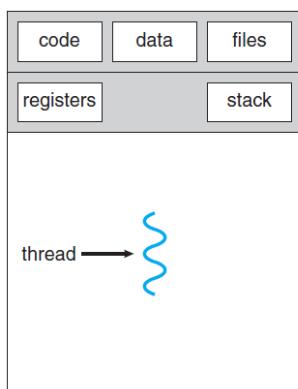
Thread'ler

- Bir **thread**, program counter, bir grup register ve bir stack yapısına sahiptir.
- Thread'ler, program kodunu, data kısmını, dosyalar gibi işletim sistemi kaynaklarını ortak kullanır.
- Klasik process'ler tek thread'e sahiptirler.
- Eğer bir process, birden fazla thread'e sahipse birden fazla görevi eşzamanlı yapabilir.
- Günümüzdeki modern bilgisayarlarda çalışan yazılım uygulamalarının çoğu multithread çalışmaları.
- Uygulamalar, çok sayıda thread'e sahip tek process şeklinde geliştirilirler.
- Bir Web browser, bir thread ile veri aktarımı yapabilir, başka thread ile verileri ekranда görüntüleyebilir.

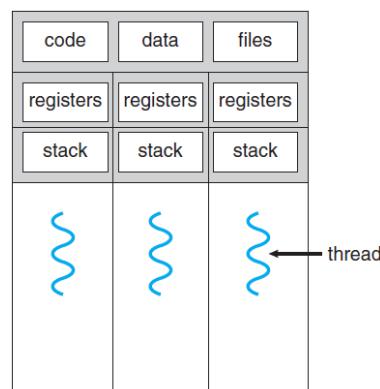
3

Thread'ler

- Bir kelime işlemci uygulaması, **bir thread ile klavyeden giriş alabilir, bir thread ile spell check yapabilir ve başka bir thread ile ekran görüntüsünü düzenleyebilir.**
- Her thread, **paylaşmadan kullandığı** kendisine ait bileşenlere sahiptir.



single-threaded process

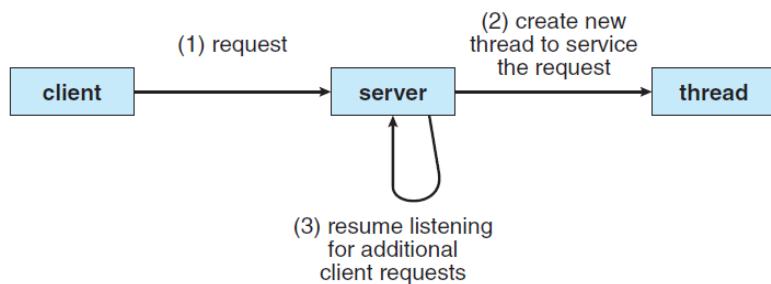


multithreaded process

4

Thread'ler

- Uygulamalar, multicore sistemlerin **kapasitesini maksimum kullanacak şekilde tasarlanabilir**.
- Bir Web sunucu process'i multithreaded çalışırsa, her gelen istek için ayrı bir thread oluşturulur ve process portu dinlemeye devam eder.
- **Çoğu işletim sisteminin kernel'ı multithreaded yapıdadır ve cihazların yönetimi, hafıza yönetimi veya interrupt işlemi aynı anda yapılabilir.**



5

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

6



Thread'lerin sağladığı faydalar

- Thread'lerin sağladığı faydalar 4 kategori halinde ifade edilebilir:
 1. **Responsiveness:** Kullanıcı etkileşimli uygulamalarda, bir kısım bloklanmış, kilitlenmiş veya uzun süren işlem yürütüyorSA, kullanıcı ile etkileşim yapan başka bir kısım çalışmasını sürdürür.
Sistemin cevap verebilirlik özelliği artmış olur.
 2. **Resource sharing:** Process'ler kaynaklarını shared memory veya message passing teknikleri aracılığıyla paylaşabilirler.
Thread'ler ait oldukları process'in sahip olduğu hafıza alanını ve diğer kaynakları paylaşabilirler.

7



Thread'lerin sağladığı faydalar

- Thread'lerin sağladığı faydalar 4 kategori halinde ifade edilebilir:
 3. **Economy:** Bir process oluştururken **hafıza ve kaynak tahsis edilmesi** maliyeti yüksek bir iştir.
Thread'ler ait oldukları process'in kaynaklarını paylaştıklarından dolayı context switch daha düşük maliyetle yapılır.
(Solaris işletim sisteminde, **thread oluşturma 30 kat daha hızlıdır** ve **thread'lerde context switch 5 kat daha hızlıdır.**)
 4. **Scalability:** **Çok işlemcili mimarilerde thread'ler farklı core'lar üzerinde eşzamanlı çalışabilir.**
Ancak, tek thread yapısına sahip process sadece bir işlemci üzerinde çalışabilir.

8

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- **Multicore programlama**
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

9

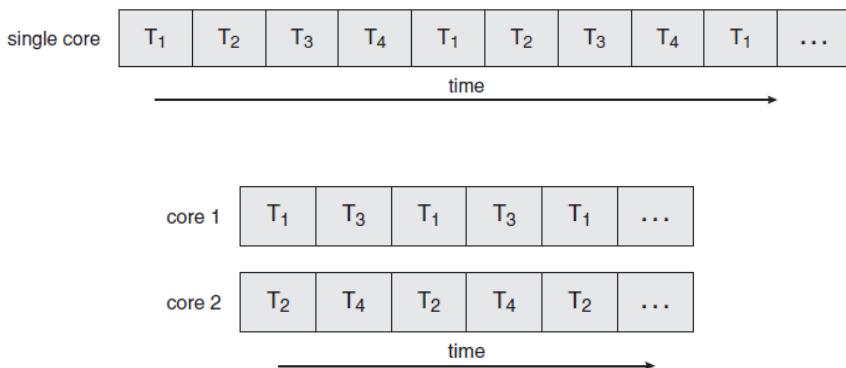
Multicore programlama

- Bilgisayar tasarımindaki en önemli gelişmelerden birisi, çok işlemcili sistemlerin geliştirilmesidir.
- Son zamanlarda, **tek chip içeresine birden fazla core yerleştirilmektedir**. Bu tür sistemler **multicore** veya **multiprocessor** olarak adlandırılır.
- **Her bir core işletim sistemi için ayrı bir işlemci olarak görünür**.
- **Bir core** üzerinde çalışan 4 thread'e sahip bir uygulama için **eşzamanlı çalışma**, **thread'lerin belirli aralıklarla çalıştırılmasını ifade eder**.
- **Çok core'a sahip sistemlerde eşzamanlı çalışma, her core'a bir thread atanarak thread'lerin paralel çalışmasını ifade eder**.
- **Parallelism**, birden fazla görevin **eşzamanlı** yapılmasını ifade eder.
- **Concurrency**, birden fazla görev arasında kısa aralıklarla geçiş yaparak **birlikte ilerletilmesini** ifade eder.

10

Multicore programlama

- Sistemdeki core sayısı arttıkça eşzamanlı gerçekleştirilen görev sayısı da artacaktır.



11

Multicore programlama

- Amdahl kuralı** core sayısına göre bir sistemdeki performans artışını aşağıdaki gibi ifade etmektedir:

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- Burada, S uygulamada seri çalışması zorunlu olan kısmın oranını, N ise core sayısını ifade eder.
- Bir uygulamada, %75 paralel ve %25 seri çalışıyorsa ($S=0,25$), **2 core'a** ($N=2$) sahip sistemde bu uygulamayı çalıştırınca **1,6 kat hız artar**.
- Core sayısı 4** olduğunda, **2.28 kat hız artışı sağlanır**.
- Core sayısı sonsuza giderken hız artışı ($1/S$) 'e doğru gider**.
- Intel CPU'lar** her core için 2 thread, **Oracle T4 CPU** ise 4 thread destekler.

12

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

13

Multicore programlamanın zorlukları

- **İşletim sistemi tasarımcıları** multicore sistemlerin performansını artırmak için **scheduling algoritmaları yazmak zorundadır.**
- Uygulama geliştiricilerin mevcut programları değiştirmeleri ve **yeni programları multihreaded şekilde tasarlamaları gerekmektedir.**
- Multicore programlamada 5 önemli zorluk vardır:
 - **Identifying tasks:** Uygulamalarda eşzamanlı çalışabilecek ayrı alanların bulunması gereklidir. Bu alanlar farklı core'lar üzerinde paralel çalışacaktır.
 - **Balance:** Programcılar görevleri ayırtırırken **iş yükünün eşit dağıtılması gereklidir.**
 - **Data splitting:** Verilerin farklı core'lar üzerinde çalışan görevler tarafından erişilecek ve işlem yapılacak şekilde ayrıştırılması gereklidir.
 - **Data dependency:** Bir görevin erişeceği verinin diğer görevlerle bağımlılığının incelenmesi gereklidir.
 - **Testing and debugging:** Multihreaded çalışan programların **test ve debug işlemi daha zordur.**

14

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - **Paralel çalışma türleri**
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

15

Paralel çalışma türleri

- Genel olarak **data parallelism** ve **task parallelism** olarak iki tür paralel çalışma türü vardır.
- **Data parallelism**, aynı **veri kümesine ait parçaların core'lara dağıtılması** ve aynı tür işlemin eşzamanlı yürütülmesine odaklanır.
- N elemanlı bir **dizinin toplamı** için **iki core** kullanılacaksa, **[0]..[(N/2)-1]** eleman 1.core'da, **[N/2]..[N-1]** eleman 2.core'da toplanır.
- **Task parallelism**, core'lara **görevlerin (thread'ler) dağıtımasına** odaklanır.
- Her thread **ayıri bir işlemi gerçekleştirir**. Farklı thread'ler aynı veride veya farklı veride çalışabilir.
- **Aynı dizi** elemanları üzerinde **farklı istatistiksel hesaplamalar** yapan thread'ler aynı veriyi kullanır farklı core'larda çalışır.

16

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- **Multithreading modelleri**
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

17

Multithreading modelleri

- Thread desteği kullanıcı seviyesinde **user thread'**ler için veya kernel seviyesinde **kernel thread'**ler için sağlanabilir.
- **User thread'leri kullanıcı uygulamaları tarafından, kernel thread'leri ise işletim sistemi tarafından gerçekleştirilir.**
- Windows, Linux, Unix, Mac OS X ve Solaris gibi işletim sistemleri **kernel thread'leri destekler**.
- Kernel thread'leri ile user thread'leri arasında aşağıdaki **ilişkilendirme modellerinden** birisinin oluşturulması zorundadır.
 - Many-to-one model
 - One-to-one model
 - Many-to-many model

18

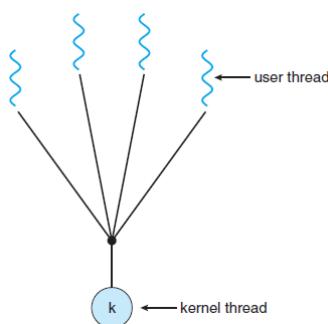
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

19

Many-to-one

- Many-to-one modelinde, **çok sayıda kullanıcı thread'i bir tane kernel thread'i ile eşleştirilir (Solaris işletim sistemi kullanır.)**.



- Eğer **bir thread sistem çağrısını bloklarsa** tüm process bloklanmış olur.
- Aynı anda sadece bir tane kullanıcı thread'i kernel thread'e erişebilir.
- Sadece bir kernel thread'i kullanıldığı için **multicore sistemlerde birden fazla thread için eşzamanlı çalışma yapılamaz**.

20

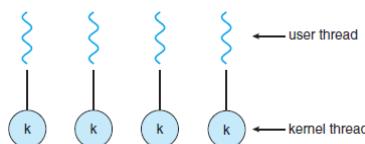
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

21

One-to-one

- One-to-one modelinde, **bir kullanıcı thread'i bir kernel thread'i ile eşleştirilir (Linux, Windows işletim sistemleri kullanır.)**.



- Eğer **bir thread sistem çağrımasını bloklarsa** diğer thread'ler çalışmasına devam eder.
- Birden fazla kernel thread'inin **multicore sistemlerde eşzamanlı çalışmasına izin verir**.
- **Bir user thread için bir kernel thread oluşturulması gereklidir.**

22

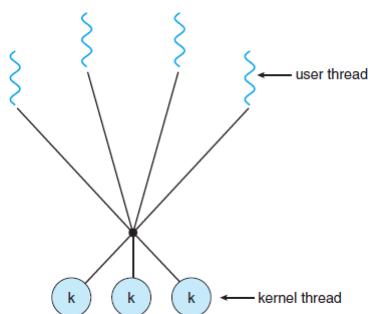
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - **Many-to-many**
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

23

Many-to-many

- Many-to-many modelinde, **çok sayıda kullanıcı thread'i ile aynı sayıdaki veya daha az sayıdaki kernel thread'i eşleştirilir (Solaris 9, Unix işletim sistemleri kullanır.)**.



- **Bir thread sistem çağrısını bloklarsa, kernel başka bir thread'i çalıştırır.**

24

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- **Thread kütüphaneleri**
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

25

Thread kütüphaneleri

- Thread kütüphanesi, **programcıya thread oluşturmak ve yönetmek için API sağlar.**
- Thread kütüphanesi oluşturulurken **iki farklı yaklaşım kullanılır:**
 - Tüm thread kütüphanesi **kullanıcı alanında** oluşturulur ve **kernel desteği yoktur.**
 - İşletim sisteminin doğrudan desteklediği **kernel seviyesinde kütüphane oluşturulur.**
- Çoklu thread oluşturmak için iki farklı strateji kullanılmaktadır:
 - **Asenkron threading:** Parent, yeni bir child thread oluşturduğunda **eşzamanlı olarak çalışmasını sürdürür.**
 - **Senkron threading:** Parent, child process oluşturduğunda **çalışmasını durdurur** ve tüm child process'ler sonlandığında çalışmasına devam eder (**fork-join strategy**).
- **Asenkron threading**, thread'ler arasında **veri paylaşımı az olduğunda**, **senkron threading** ise threadler arasında **veri paylaşımı çok olduğunda** **kullanılır.**

26



Thread kütüphaneleri

- Günümüzde 3 temel thread kütüphanesi kullanılmaktadır:
 - POSIX Pthreads
 - Windows
 - Java
- **Pthreads**, user-level veya kernel-level thread kütüphanesi sağlar.
- **Windows threads**, kernel-level thread kütüphanesi sağlar.
- **Java threads**, user-level thread kütüphanesi sağlar.

27



Thread kütüphaneleri

Pthreads

- **Pthreads, IEEE1003.1c standarıyla thread oluşturma ve yönetmek için tanımlanan API'dir.**
- **Linux, Unix, Mac OS X ve Solaris** işletim sistemleri Pthreads standartını kullanır.
- **Windows** Pthreads standartını **desteklemez**.
- **Pthreads standartında thread'lerin hepsi ayrı fonksiyonlar halinde oluşturulur.**
- **Tüm thread'ler global scope'ta tanımlanan verileri paylaşırlar.**

28

Thread kütüphaneleri

Pthreads - Örnek

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param),
        sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```

Pthreads programları için kullanılan header file

Yeni thread için ID tanımlar.

Yeni thread için özellikleri (stack size, ...) belirler.

Varsayılan özellikler (senkron thread, sistem stack addr, ...)

Yeni thread başlatıldı.

Yeni thread için başlama noktası.

Komut satırında girilen parametre

Komut satırında girilen parametre

fork-join stratejisi

Dönen değer

Thread kütüphaneleri

Pthreads - Örnek

- Önceki örnekte bir thread oluşturulmuştur. Çok sayıda thread aşağıdaki örnekteki gibi oluşturulabilir.

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Oluşturulacak thread sayısı

10 thread tanımlandı.

10 thread için fork-join yapıldı.

Thread kütüphaneleri

Windows threads

- Windows thread kütüphanesi ile thread oluşturma Pthreads ile birçok açıdan benzerlik gösterir.
- **Thread'lerin hepsi ayrı fonksiyonlar halinde oluşturulur.**
- **Tüm thread'ler global scope'ta tanımlanan verileri paylaşırlar.**

31

Thread kütüphaneleri

Windows threads - Örnek

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }
}
```

* create the thread */
ThreadHandle = CreateThread(
 NULL, /* default security attributes */
 0, /* default stack size */
 Summation, /* thread function */
 &Param, /* parameter to thread function */
 0, /* default creation flags */
 &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
 /* now wait for the thread to finish */
 WaitForSingleObject(ThreadHandle, INFINITE);
}
/* close the thread handle */
CloseHandle(ThreadHandle);
printf("sum = %d\n",Sum);

fork-join stratejisi
WaitForMultipleObjects() ile birden fazla
thread senkron çalıştırılabilir.

32

Thread kütüphaneleri

Java threads

- Java thread'leri, JVM (Java Virtual Machine) kullanılabilen tüm sistemlerde çalışır.
- Java thread API, Windows, Linux, Unix, Mac OS X ve Android için kullanılabilir.
- Java thread'leri arasında **veri paylaşımı parameter passing** ile yapılır.
- Java ile iki farklı teknik kullanılarak thread oluşturulabilir:
 - Thread sınıfından yeni bir sınıf türetilir ve run() metodu override yapılır.
 - Runnable arayüzüünü kullanan bir sınıf oluşturulur. (yaygın kullanılır.)

```
public interface Runnable  
{  
    public abstract void run();  
}
```

Aynı thread için override yapmak gereklidir.

33

Thread kütüphaneleri

Java threads – Örnek

```
class Sum  
{  
    private int sum;  
  
    public int getSum() {  
        return sum;  
    }  
  
    public void setSum(int sum) {  
        this.sum = sum;  
    }  
}  
  
class Summation implements Runnable  
{  
    private int upper;  
    private Sum sumValue;  
  
    public Summation(int upper, Sum sumValue) {  
        this.upper = upper;  
        this.sumValue = sumValue;  
    }  
  
    public void run() {  
        int sum = 0;  
        for (int i = 0; i <= upper; i++)  
            sum += i;  
        sumValue.setSum(sum);  
    }  
}  
  
public class Driver  
{  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            if (Integer.parseInt(args[0]) < 0)  
                System.err.println(args[0] + " must be >= 0.");  
            else {  
                Sum sumObject = new Sum();  
                int upper = Integer.parseInt(args[0]);  
                Thread thrd = new Thread(new Summation(upper, sumObject));  
                thrd.start();  
                try {  
                    thrd.join();  
                    System.out.println  
                        ("The sum of "+upper+" is "+sumObject.getSum());  
                } catch (InterruptedException ie) { }  
            }  
        } else  
            System.err.println("Usage: Summation <integer value>");  
    }  
}
```

Aynı thread için gereklidir.

Yeni thread oluşturuldu.

Yeni thread başlıyor.
run() çağırılır.

fork-join stratejisi
Thread'ler senkron çalışıyor.

34

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- **Dolaylı thread oluşturma**
- Thread çalışma kuralları
- Windows ve Linux thread'leri

35

Dolaylı thread oluşturma

- Multicore işlemcilerdeki gelişmelerle birlikte, uygulamalar yüzlerce hatta binlerce thread içermektedirler.
- Çok sayıda thread ile uygulama geliştirmek oldukça zordur ve hata olma olasılığı vardır.
- Thread oluşturma işinin **uygulama geliştiriciler yerine, compiler tarafından yapılması** günümüzde giderek popüler hale gelmektedir.
- Bu stratejiye **implicit threading** denilmektedir.

36

Dolaylı thread oluşturma

Thread pools

- Multithreaded bir Web sunucusu, gelen isteklerin her birisi için yeni thread oluşturur.
- Multihreaded bir Web sunucusu, eşzamanlı çok sayıda istemciye servis sağlayabilir.
- Gelen istek sayısı çok artarsa** sistem kaynakları (CPU time, memory, ...) tükenir.
- Multithreaded sistemlerde belirli sayıda thread oluşturulmasına izin vermek için thread pool oluşturulur.**
- Yeni istek geldiğinde thread pool içerisinde kullanılabilir **thread varsa cevaplanır**, yoksa bir thread'in serbest hale gelmesi beklenir.
- Varolan thread'in kullanımı yeni thread oluşturmaya göre daha hızlıdır.**

37

Dolaylı thread oluşturma

Thread pools - Windows

- Örnekte `PoolFunction()` fonksiyonu thread olarak çalıştırılmaktadır.

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```

- Thread pool API içerisindeki `QueueUserWorkItem()` fonksiyonu, **pool içerisindeki bir thread ile PoolFunction() fonksiyonunu çalıştırır.**

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

- Parametreler:**

- LPTHREAD_START_ROUTINE, thread olarak çalışacak fonksiyonun pointer'i
- PVOID, Gönderilecek parametre
- ULONG, Bayrak bitleri (bekleme süresi, I/O gerekliliği, ...)

38

Dolaylı thread oluşturma

OpenMP

- OpenMP, C, C++ ve Fortran için yazılmış bir grup compiler direktifidir.
- Shared memory yaklaşımını kullanılır.
- **OpenMP ile paralel çalışacak kod blokları tanımlanır.**
- OpenMP ile `#pragma omp parallel` direktifi paralel çalışacak bloğun hemen başında kullanılır.
- OpenMP farklı türdeki deyimler için ayrı direktifler kullanır.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

- OpenMP, Linux, Windows ve Mac OS X sistemlerdeki çok sayıda açık kaynak ve ticari compiler'larda kullanılabilir.

39

Dolaylı thread oluşturma

OpenMP - Örnek

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```

printf() deyimi paralel çalışacaktır.

40

Dolaylı thread oluşturma

Grand central dispatch

- Grand central dispatch (GCD), **Apple Mac OS X ve iOS işletim sistemlerinde paralel çalışacak kısımları belirlemek için kullanılır.**
- Paralel çalışacak bloğun belirtilmesi için ^ simbolü kullanılır.

```
^{ printf("I am a block"); }
```
- GCD blokları **dispatch queue** içerisinde yerleştirir.
- Bir blok kuyruktan atılırsa, tekrar thread havuzundaki bir thread'e atanabilir.
- Dispatch queue, **serial** veya **concurrent** şeklinde oluşturulabilir.
- **Serial queue**, FIFO çalışır ve **sadece bir blok kuyruktan alınabilir**.
- **Concurrent queue**, FIFO çalışır ve kuyruktan **birden fazla blok aynı anda alınabilir**.

41

Dolaylı thread oluşturma

Grand central dispatch

- Concurrent queue, önceliklendirilmiş 3 tane dispatch kuyruğa sahiptir: **low**, **default** ve **high**.
- Önceliklendirme ile blokların önem derecesi belirlenmektedir.
- Aşağıdaki örnekte, default önceliğe sahip concurrent kuyruğa bir blok eklenmektedir.

```
dispatch_queue_t queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```

Kuyruğa blok eklendi.

default öncelikli concurrent kuyruk

42

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- **Thread çalışma kuralları**
- Windows ve Linux thread'leri

43

Thread çalışma kuralları

fork() ve exec() sistem çağrıları

- **Bazı Unix sistemlerde, iki tür fork() çağrısı vardır.**
- Birisi ile **tüm thread'ler duplicate yapılır**, diğeri ile **sadece fork() ile başlatılan thread duplicate yapılır**.
- **exec()** sistem çağrısı ile **tüm thread'leri ile birlikte process duplicate yapılır**.

44

Thread çalışma kuralları

Signal handling

- Unix sistemlerde bir **signal** belirli bir olayın gerçekleştiğini gösterir.
- **Oluşan olaya karşılık gelen sinyal bir process'e iletilir.**
- Oluşan sinyal, **senkron** veya **asenkron** alınabilir.
- **Senkron sinyal, sinyalin oluşmasına neden olan olayı gerçekleştiren process'e iletilir.**
- Senkron sinyale illegal hafıza erişimi veya 0'a bölme verilebilir.
- **Asenkron sinyal, sinyali oluşturan process'ten başka bir process'e iletilir.**
- Asenkron sinyale <ctrl><C> tuşlarına birlikte basmak verilebilir.

45

Thread çalışma kuralları

Signal handling

- İşletim sistemlerinde **sinyaller farklı hedeflere gönderebilir**:
 - Process içerisindeki **sadece bir thread'e gönderebilir** (Senkron).
 - Process içerisindeki **tüm thread'lere gönderebilir** <ctrl><C>.
 - Process içerisindeki **bazı thread'lere gönderebilir**.
 - Bir process için **tüm sinyalleri almak üzere bir thread atanabilir**.
- **Senkron sinyaller** sadece **oluşturan thread'e** gönderilir.
- Aşağıdaki UNİX fonksiyonu ile ID değeri verilen process'e iletilir.

```
kill(pid_t pid, int signal)
```
- POSIX Pthreads ile aşağıdaki fonksiyon kullanılır.

```
pthread_kill(pthread_t tid, int signal)
```

45

Thread çalışma kuralları

Thread iptal etme

- Bir thread'in çalışması tamamlanmadan iptal edilebilir.
- İstenen bir sonucun bir thread tarafından bulunması halinde diğerleri iptal edilebilir.
- Bir Web sayfası yüklenirken stop butonuna basıldığında process içerisindeki tüm thread'ler iptal edilir.
- Bir thread başka bir thread'i aniden sonlandırabilir (**Asenkron cancellation**).
- Bir thread başka bir thread'in kendi kendisini sonlandırmasını sağlayabilir (**Deferred cancellation**).

47

Thread çalışma kuralları

Thread iptal etme

- Pthreads aşağıdaki tabloda verilen üç farklı iptal etme modunu destekler.

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
...  
  
/* cancel the thread */  
pthread_cancel(tid);
```

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

48

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- **Windows ve Linux thread'leri**

49

Windows ve Linux thread'leri

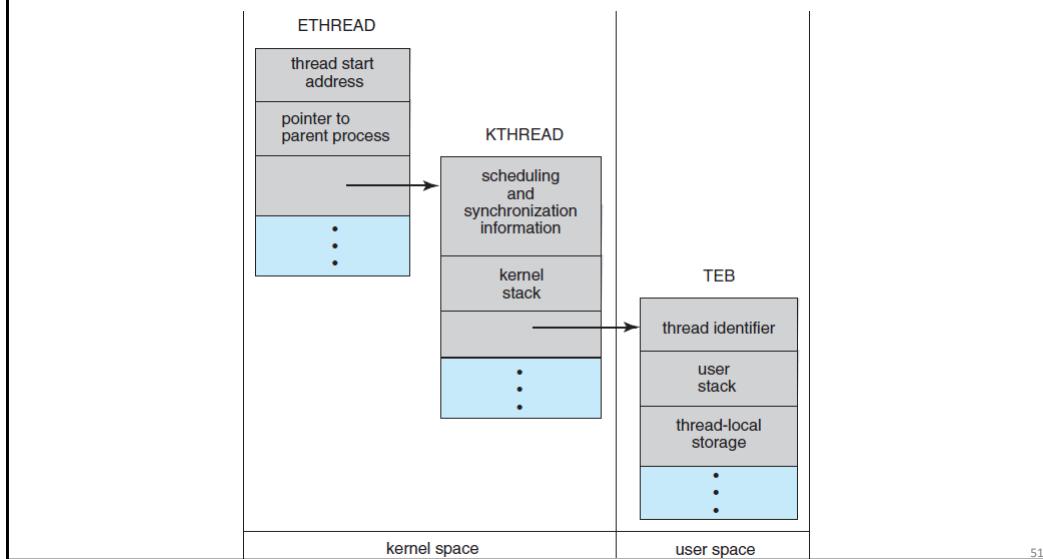
Windows thread'leri

- Microsoft işletim sistemleri için temel API olarak **Windows API** kullanılır.
- Bir Windows uygulaması ayrı bir process olarak çalışır ve **her process bir veya daha fazla thread içerebilir.**
- **Windows, kullanıcı thread'leri ile kernel thread'leri arasında one-to-one eşleştirme yapar.**
- Bir thread için ayrılan **register kümesi, stack, depolama alanı, context** olarak adlandırılır.
- Windows bir thread için aşağıdaki veri yapılarını kullanır:
 - **ETHREAD:** Yürüttü thread blok
 - **KTHREAD:** Kernel thread blok
 - **TEB:** Thread environment (ortam) blok

50

Windows ve Linux thread'leri

Windows thread'leri



51

Windows ve Linux thread'leri

Linux thread'leri

- Linux, `fork()` sistem çağrısının yanı sıra `clone()` sistem çağrıları ile thread oluşturabilir.
- Linux, `clone()` ile yeni bir görev başlattığında, parent task ile child task arasında paylaşım miktarını da gönderir.
- **Dosya sistemi, hafıza aralığı, sinyaller veya açık olan dosyalar paylaşılabilir.**
- Görevler, Linux kernel içerisinde bir veri yapısına sahiptir (`struct task_struct`) ve açık dosyalar, virtual memory ve sinyal bilgilerini gösterir.
- Linux, `fork()` ile **yeni bir görev başlattığında, parent task veri yapısı kopyalanır.**

52

Windows ve Linux thread'leri

Linux thread'leri

- Linux, clone() ile yeni bir görev başlattığında bayrak bitlerine göre veri yapısı oluşturulur.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

İşletim Sistemleri 3.Uygulama

Thread Komutları

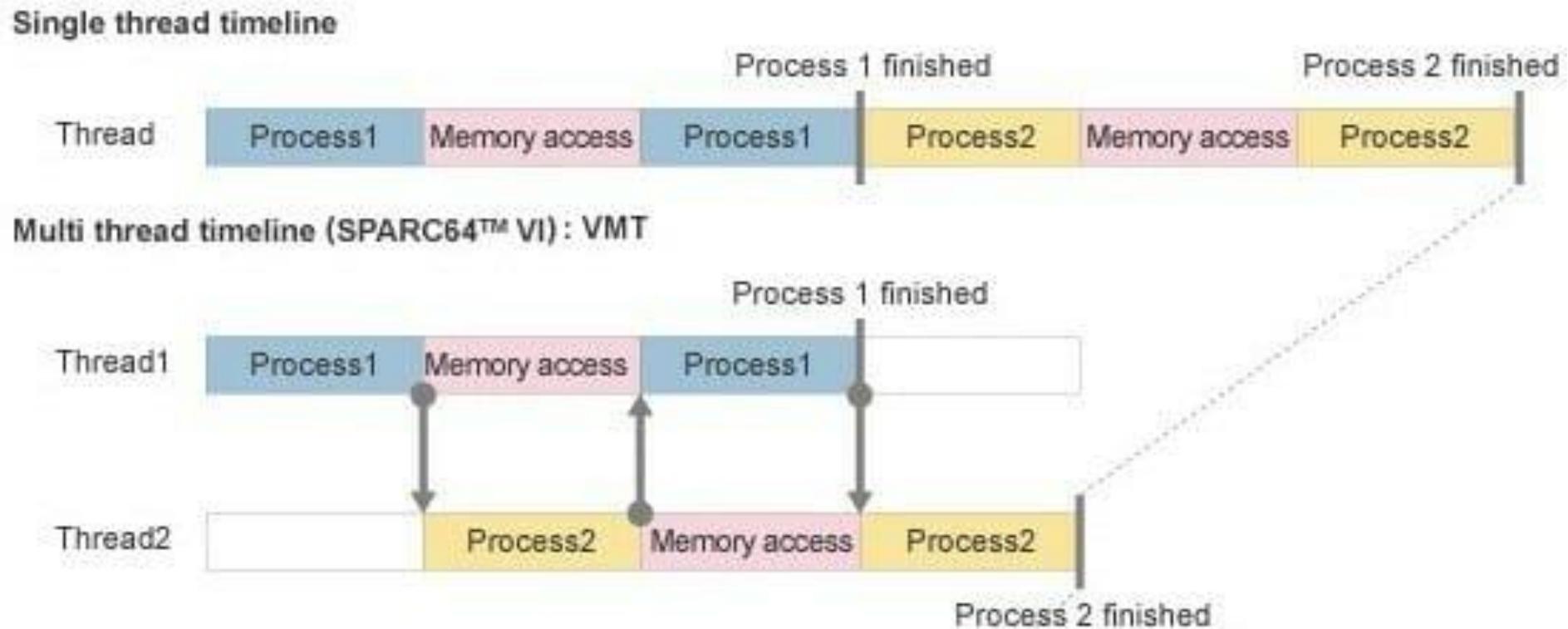
Hazırlayan : Özlem ÖZMEN AKALIN

- Linux altında thread kullanımı C kütüphanesinin bir parçası olan pthread kütüphanesi ile yapılmaktadır.
- Diğer işletim sistemlerinin aksine, Linux'te thread ile process arasında çok az fark vardır.
- Bu nedenle thread'ler, LightWeightProcess olarak adlandırılmaktadır.
- Linux versiyon **2.6** öncesindeki thread implementasyonu **LinuxThreads** şeklinde adlandırılıyordu. Bu implementasyon performans ve senkronizasyon işlemleri açısından önemli limitlere sahipti. Maksimum çalışabilecek thread sayısı 1000'li rakamlarla sınırlıydı.
- 2003 yılında özellikle **IBM** ve **RedHat** firmasındaki geliştiricilerin başını çektiği ekip, **Native POSIX Thread Library** (NPTL) projesini kullanılabılır duruma getirmeyi başardı ve ilk olarak enterprise dünyanın Linux üzerinde Java Virtual Machine performansı şikayetlerini çözüme kavuşturmak için RedHat Enterprise versiyon 3'te kullanılmaya başlandı. Ardından GNU C kütüphanesindeki yeni NPTL implementasyonu gelmeye başladı. Günümüzde de her iki implementasyon glibc içerisinde yer almaktır, kullanılan çekirdek versiyonu tarafından uygun implementasyon seçilmektedir.

Thread Nedir ?

- İş parçacıkları olarak da bilinir ve projelerde olmazsa olmaz terimler arasındadır. Thread, programın kendini senkronizasyonla birden çok iş bölümüne ayırabilmesinin farklı bir yoludur. Bilgisayarımızda tek işlemci kullanıyorsak thread kullandığımızda uygulamalar zaman dilimleme ile gerçekleştirilir. Yani tek işlemci iki ya da daha çok Thread arasında çok hızlı geçiş yapar. Bu olay sanki eş zamanlı olarak işlemlerin gerçekleştirildiği izlenimini yansıtır.
- İş parçacıkları tek başına kullanıldığı gibi çoklu yani multithreading olarak da kullanılır. Genel kullanımı bu şekildedir. Bunun amacı da eş zamanlı olarak -paralel – yapılması istenilen işlemlerde kullanılmaktadır.
- Örneğin saat gösteren bir uygulamada aynı zamanda başka bir işlem de yapılması gerekebilir. Bu durumda iş parçacıkları kullanılır ve ikisi de birbirinden bağımsız olarak işlemiş olurlar.

Threading, aynı ortamda aynı anda birden fazla işi yapmaya denir. Thread'ler ise bu işlerin her biridir. Thread'ler aynı anda çalıştığında işlemciye process'ler olarak gider ve sıraya alınır. Tek çekirdekli işlemcilerde birden fazla thread çalıştığı zaman sırayla threadlerin processlerini işleme alır ve bir thread diğer thread'in bitişini beklemeden processleri işlemcide işleme girer. Çok çekirdekli işlemcilerde ise farklı thread'ler farklı çekirdeklerde aynı anda çalışabilme durumuna sahiptir.



Threadleri neden kullanıyoruz?

- Thread'ler genel olarak, oyunlar, paralel çalışan task'ler, event handling gibi durumlarda kullanılmaktadır. Ayrıca çok çekirdekli işlemcilerde tam performans yararlanmak için de thread'ler kullanılır. İnsan üzerinden thread'leri açıklamak gerekirse, bir insan aynı anda konuşurken gözleri başka bir yere bakıp elleriyle ayrı işler yapabilmektedir. Bu tam olarak multithreading örneğidir.
- Thread'lerin kullanıldığı bir başka alan ise server-client uygulamalar. Günümüzde popüler olarak Whatsapp, forum siteleri, online oyunlar bu mimariyi kullanmaktadır. Birden fazla kullanıcı birbirini beklemeden işlerini halletmek durumunda olduğundan her bir kullanıcının işlemleri ayrı threadlerde yönetilmektedir.

Bazı terimler

- **ThreadPool:** İşletim sistemi seviyesinde threadlerin bir arada bulunduğu bir havuzdur. Çok fazla thread kullanan uygulamalar her bir thread işlemciye yük oluşturacağından thread ihtiyaçlarını bu havuzdan alarak çözmektedir. Örneğin IIS bu şekilde threadpool kullanmaktadır.
- **WorkerThread:** ThreadPool'daki threadlerin her biri bir worker thread'dır
- **BackgroundWorker:** Arkaplanda çalışan thread'lere verilen isimdir. Bir örnekle açıklamak gerekirse,
- Bir masaüstü uygulaması düşünelim. Kullanıcı bir indirme işlemi başlatmak istiyor ancak bu işlem büyük dosya boyutlarında çok uzun süreceğinden backgroundworker kullanılmazsa kullanıcı programı indirme işlemi bitinceye kadar kullanamayacaktır. Bunun önüne geçmek için bir backgroundworker oluşturulup indirme işlemine burdan devam etmesi ve kullanıcının da programı kesintiye uğramadan kullanması sağlanmaktadır.

- C# uygulamalarında ana programın işlerinin yürüdüğü bir main thread bulunmaktadır. Her Thread objesi main thread'e paralel olarak çalışan bir worker thread yaratmaktadır.
- Örnek Uygulama:

```
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread (WriteY);
        t.Start();

        // Simultaneously, do something on the main thread.
        for (int i = 0; i < 1000; i++) Console.Write ("x");
    }

    static void WriteY()
    {
        for (int i = 0; i < 1000; i++) Console.Write ("y");
    }
}
```

Ekran çıktısı:

```
xxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyy  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyy  
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyy  
yyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
...
```

- Threading işlemlerini kontrol eden tipler **System.Threading** kütüphanesidir. Thread in sınıfından birden fazla metodu olduğu için en temel Start() ve Sleep() metodlarından bahsedelim.
- **Start():** Thread classının en ana metodudur ve threadlerin çalışmasını başlatır.
- // Nesne oluşturup ThreadStart(çalıştırılacak methodun adı) methoduyla çalıştırılacak olan method belirlenir.
- Thread thread1 = new Thread(new ThreadStart(function1));
- // thread1 çalıştırılır
- thread1.Start();
- **Sleep():** Sleep metodunu kullanarak threadin çalışmasını bir süreliğine bekletebiliriz. Örneğin; belli bir zamanda fonksiyonu çalıştırıp 0.20 saniye dinlendirdikten sonra işleme devam ettirebiliriz ya da başka fonksiyonları çalıştırabiliriz.

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//Thread kullanabilmek için öncelikle aşağıdaki 2 satırı eklemeliyiz.
using System.Threading;
using System;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        //Button Click olayı altında Thread lerimizi oluşturalım ve başlatalım.
        static void button1_Click(object sender, EventArgs e)
        {
            Thread thread1 = new Thread(new ThreadStart(ThreadFuncEven));
            Thread thread2 = new Thread(new ThreadStart(ThreadFuncOdd));
            //Threadleri başlatalım bakalım sırasına göremi yoksa paralelmi işleyecek.
            //Sırasına göre işlenseydi önce çift sayılar, sonra tek sayılar basılmalıydı.
```

```
thread1.Start();
thread2.Start();
}
//Output her saniye 2 sayı gelecek şekilde 1 2 3 4 5 ... 100 şeklinde olacaktır.
static void ThreadFuncEven()
{
    for (int i = 0; i < 100; i += 2)
    {
        //Her çift Sayı için 1 saniye bekleyelim ve sayıyı yazdıralım
        Thread.Sleep(1000);
        Console.WriteLine(i);
    }
}
static void ThreadFuncOdd()
{
    for (int i = 1; i < 100; i += 2)
    {
        //Her Sayı için bir saniye bekleyelim ve sayıyı yazdırın.
        Thread.Sleep(1000);
        Console.WriteLine(i);
    }
}
```

Threadlerde Öncelik (Thread Priority)

- Eşzamanlı olarak çalışabilen iş parçacıkları olduğundan bahsetmiştik. Fakat bazı durumlarda öncelik değişimlebilir. Aynı anda başlatılan iki parçacığın kullanımında birisini önce başlatmak durumunda kalabiliriz. **Thread.Priority** özelliği ile bunu yapabiliriz.
- *Highest, AboveNormal, Normal, BelowNormal, Lowest* gibi türleri vardır.
- `thread_1.Priority = ThreadPriority.Lowest ;`

Thread.Join İşlemi

- Thread işleminde bazı durumların da birbirini beklemesi gerekmektedir. Bu durumlarda join işlemini uyguluyoruz. Böylece bir önceki iş bitmeden diğerine geçiş yapılmamış oluyor. Ayrıca bu metodun belirlenen bir süre kadar işlem yapılması gibi overload özellikleri de bulunur.
- İş parçacıkları için bazı genel kullanılan özellikler şunlardır ;
- **IsAlive** : Bu özellik iş parçacıklarının durumunu sorgulamaya yarar. Dönüş olarak bool değer tipi döner. Fakat Thread.ThreadState değerinden farklı olarak true veya false değer döner.
- **ThreadState**: Bu özellik ise durumu çalışıyor, iptal edildi, beklemede gibi daha detaylı özelliklerine ayırmamıza yarar.
- **Suspend**: Adından da anlaşılacağı gibi suspend yani işlemimizi askıya almamıza yarayan fonksiyondur.

Thread Nasıl Kullanılır?

- using System.Collections.Generic ;
- using System.ComponentModel ;
- using System.Data ;
- using System.Drawing ;
- using System.Text ;
- //Thread kullanabilmek için öncelikle aşağıdaki 2 satırı eklemeliyiz.
- using System.Threading ;
- using System;

Thread Oluşturma Komutları

- Yeni bir thread oluşturmak için `pthread_create` fonksiyonu kullanılır.
- Thread fonksiyonlarının kullanımı için `pthread.h` başlık dosyası dahil edilmelidir.
- Threadler ana program ile aynı adresleme alanını ve aynı dosya tanımlayıcılarını kullanırlar.
- Pthread kütüphanesi aynı zamanda senkronizasyon işlemleri için gerekli **mutex** ve **conditional** işlemleri için gerekli desteği de içermektedir.
- Pthread kütüphanesi fonksiyonları kullanıldığında uygulama `pthread` kütüphanesi ile birlikte çağrırmalıdır.

- \$ gcc -o example example_thread.c -lpthread
- int pthread_create(pthread_t *thread, const pthread_attr_t *attr ,
- void *(*start_routine) (void *), void *arg)
- Başarılı durumda 0 döner. Hata durumunda ise geriye 1 hata kodu dönecektir. İş parçası kimliği "pthread_t" türüyle temsil edilir. Bir işlem kimliği bir tamsayı değeridir, ancak iş parçası kimliği mutlaka bir tamsayı değeri değildir. Bir yapı olabilir
- Thread parametresi **pthread_t** türünde olup önceden tanımlanması gereklidir. Oluşan thread' e bu referansla her zaman erişilebilecektir.
- attr parametresi thread spesifik olarak **pthread_attr** ile başlayan fonksiyonlarla ayarlanmış, scheduling policy , stack büyüklüğü,
- detach policy gibi kuralları gösterir.
- start_routine thread tarafından çalıştırılacak olan fonksiyonu gösterir.
- arg ise thread tarafından çalıştırılacak fonksiyona geçirilecek void* 'a cast edilmiş genel bir veri yapısını göstermektedir.

- **Örnek Uygulama**

- # include <stdio.h>
- # include <stdlib.h>
- # include <unistd.h>
- # include <phtread.h>
- void *worker (void *data)
- {
- char *name=(char*)data
- for (int i=0 ; i<120; i++)
- {
- usleep(50000) ;
- printf('Hello from thread %s \n ',name);
- }
- printf ('Thread %s done...\n ,name);
- return NULL ;
- }

```
int main (void) {  
    pthread_t th1,th2 ;  
    pthread_create (&th1,NULL,worker, 'A ') ;  
    pthread_create (&th2,NULL,worker, 'B ') ;  
    sleep(5) ;  
    printf('Exiting from main program\n') ;  
    return 0 ;  
}
```

Birleştirilebilir ve Çıkarılabilir Thread Türleri

- Thread kullanılan bir uygulamada main() fonksiyonundan return edilirse, tüm thread'lerde sonlandırılır ve kullanılan tüm kaynaklar sisteme geri verilir.
- Aynı şekilde herhangi bir thread içerisinde exit() benzeri bir komutla çıkış yapılması halinde gene tüm thread'ler sonlandırılacaktır.
- `pthread_join` fonksiyonu ile bir thread'in sonlanması bekleyebiliriz. Bu fonksiyonun kullanıldığı thread, sonlanması beklenen thread sonlanana kadar bloklanacaktır.
- Normal (*joinable*) thread'ler, sonlanmış olsa dahi `pthread_join` ile *join* işlemine tabi tutulmazlar ise, CPU tarafından tekrar programlanamasalar da sistemden kullandığı kaynaklar **geri verilmez**.

Çıkarılabilir Thread

- Bazen pthread_join ile join işlemi yapmanın anlamlı olmadığı, thread'in ne zaman sonlanacağının öngörülemediği durumlar olabilir.
- Bu durumda thread return ettiği noktada tüm kaynakların sisteme otomatik olarak geri verilmesini sağlayabiliriz.
- Bunu sağlamak için ise, ilgili thread'leri, **DETACHED** durumu ile başlatmamız gerekmektedir.
- Bir thread başlatılırken thread attribute değerleri üzerinden veya pthread_detach fonksiyonu ile DETACH durumu belirtilebilir :
- `int pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);`
- `int pthread_detach(pthread_t thread) ;`

Thread Sonlandırma

- Bir thread, başka bir thread tarafından, ilgili p_thread_t id değeri verilmek suretiyle iptal edilebilir.
- `int p_thread_cancel (pthread_t thread) ;`

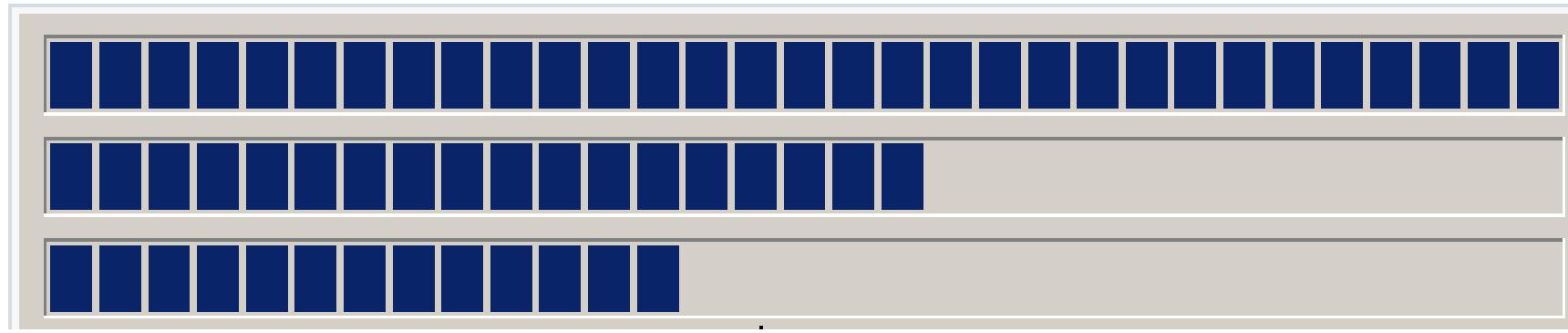
Multi Threading

- public void DegerArttir1()
- {
 - for (int i = 1; i <= 100; ++i)
 - {
 - progressBar1.Value += 1;
 - lbThreads.Items.Add("Thread 1");
 - Thread.Sleep(10);
- } public void DegerArttir2()
- {
 - for (int j = 1; j <= 100; ++j)
 - {
 - progressBar2.Value += 1;
 - lbThreads.Items.Add("Thread 2");
 - Thread.Sleep(100);
- }

- public void DegerArttir3()
 - {
 - for (int k = 1; k <= 100; ++k)
 - {
 - progressBar3.Value += 1;
- lbThreads.Items.Add("Thread 3");
- Thread.Sleep(150);

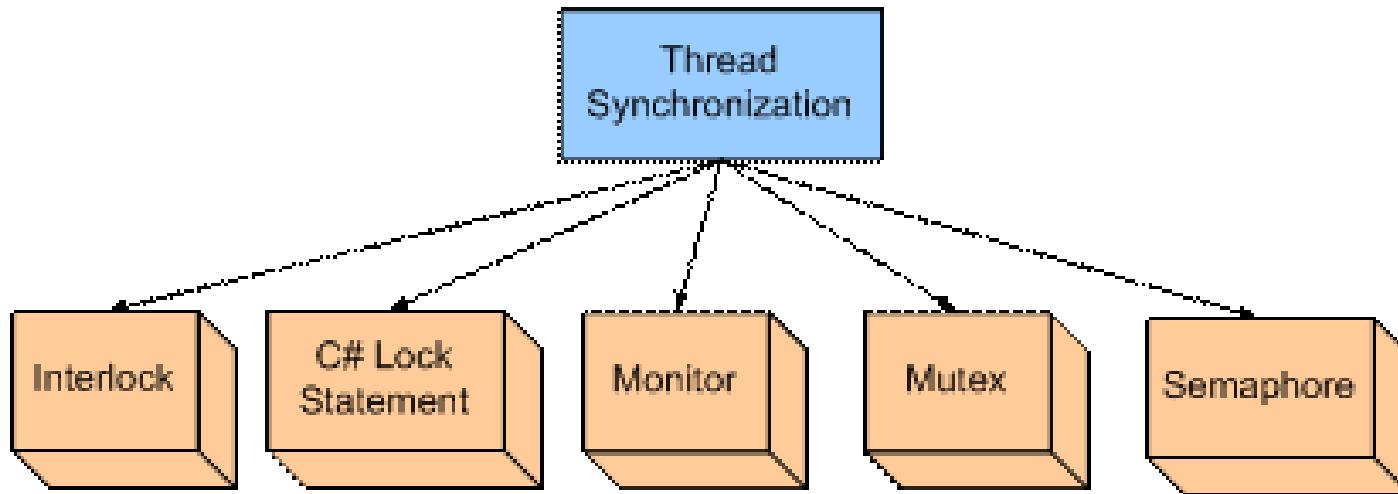
}

```
private void btnStart_Click(object sender, EventArgs e)
{
    th1 = new Thread(new ThreadStart(DegerArttir1));
    th2 = new Thread(new ThreadStart(DegerArttir2));
    th3 = new Thread(new ThreadStart(DegerArttir3));
    th1.Start();
    th2.Start();
    th3.Start();
}
```



Asenkron thread lerle doldurulan üç progressbar.

- .NET'te thread senkronizasyonu için dört yöntem kullanılabilir.



.NET'te thread senkronizasyon teknikleri.

- **Interlocked :**

Interlocked sınıfı birden fazla kanal tarafından kullanılan değişkenler üzerinde çeşitli işlemlerin yapılmasına olanak sağlayan bir sınıfır. Buradaki değişken değer tipinde olabileceği gibi herhangi bir nesne de olabilir. **Interlocked** sınıfının şimdilik sadece Exchange metodunu referans tipleri ile çalıştırılabilir. Exchange'in generic metod olarak tasarlanmış bir overload'u da mevcuttur. Increment, decrement metodları ise Int32, Int64 tipi değişkenleri parametre olarak kabul eder. **Interlocked** sınıfını şöyle kullanabiliriz:

- public void DegerArttirInterlocked1()
{
 try
 {
 while (counter <= 300)
 {
 Interlocked.Increment(ref counter);
 progressBar1.Value += 1;
 lbThreads.Items.Add("Thread 1");
 Thread.Sleep(80);
 }
 }
 catch { }
}

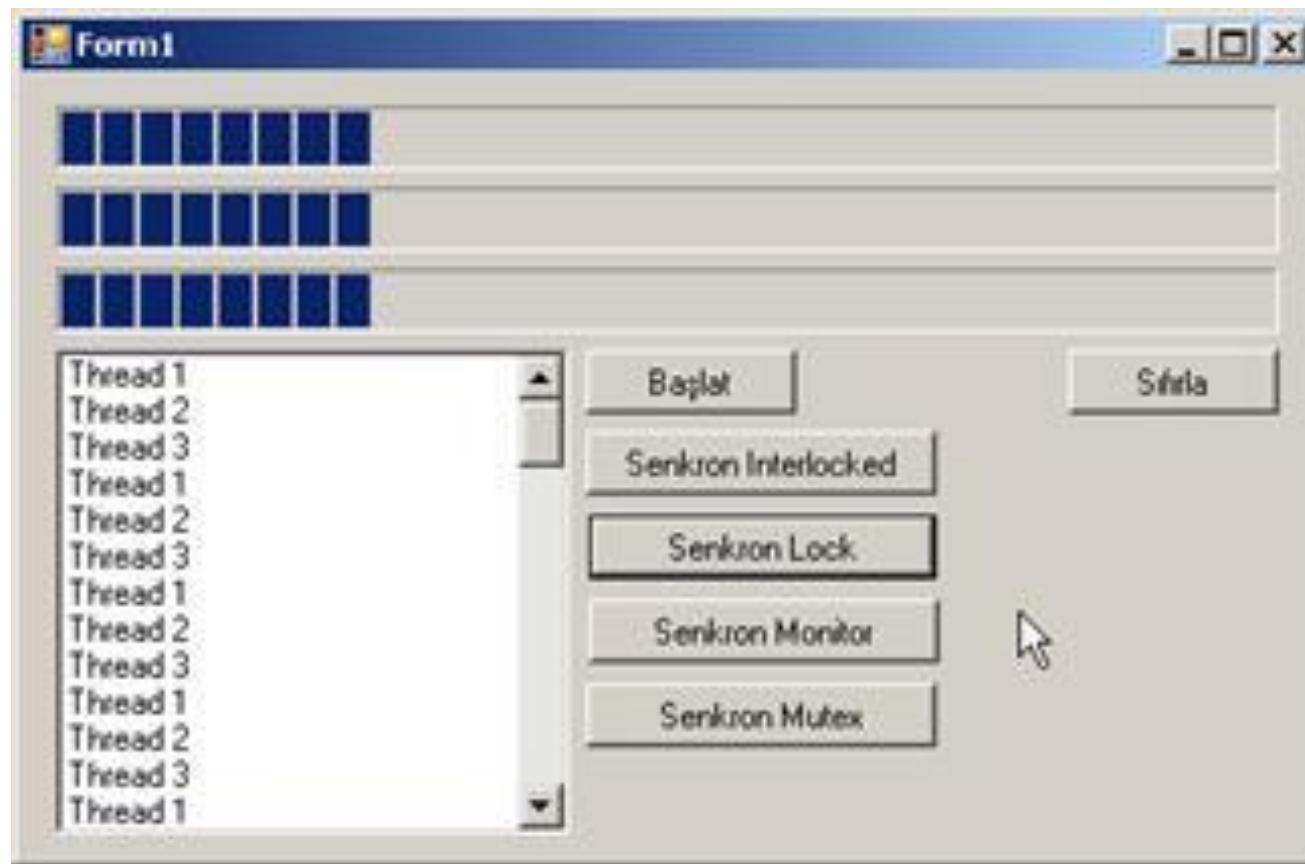
- Burada counter global olarak tanımlanmış olan int değişkendir. Increment fonksiyonu ile counter değişkeni kontrollü arttırılmaktadır. Bir thread altında counter üzerinde işlem yapılırken diğer bir thread içerisinde counter üzerinde aynı anda işlem yapılmaz.

- **Lock :**

Bir thread içersindeki işlemlerin, diğer bir thread tarafından müdahale edilmeden çalışabilmesi için **lock** anahtar kelimesi kullanılır. **Lock** ile bloklanmış olan işlemler bir thread içerisinde tamamlanıncaya kadar çalışırlar. Bu süre zarfında başka thread **lock** ile bloklanmış işlemlerin bitmesini bekler. İşlem bitince diğer threadler kendi metodlarını çalıştırırmaya devam eder. Bu şekilde senkronizasyon sağlanabilir. **Lock** parametre olarak herhangi bir nesne alabilir. Örneğimizde form nesnemizi parametre olarak kullandık.

- public void DegerArttirLock1()
{
 for (int i = 1; i <= 100; ++i)
 {
 lock (this)
 {
 progressBar1.Value += 1;
 lbThreads.Items.Add("Thread 1");
 Thread.Sleep(10);
 }
 }
}

- Genellikle **lock**'a aktarılan parametrenin public olmamasına dikkat edilir. **Lock** ile kilitlenen nesne public ise bu nesne üzerinde işlem yapmak isteyen metodlar (bir thread'e bağlı olmayan) da kilitlenir. Bu tür kilitlenmelere deadlocks (kısır döngü) denir. Bu istenmeyen bir durumdur. Public yerine protected veya private kullanmak bu sorunu çözer. Şekil 'de DegerArttirLock metodlarını çağıran kanallar çalıştırıldığında progressbar'ların ve kanalların durumu sergilenmiştir. Dikkate edilirse kanallar 123, 123 şeklinde art arda düzgün bir şekilde çağrılmaktadır. Asenkron kanallarda ise 132, 131, 111 şeklinde düzensiz çağrılmalar gerçekleşir.



Senkron thread leri doldurulan üç progressbar

- **Monitor :**

Monitor, nesnelere kanalların senkronize bir şekilde ulaşmasını sağlayan bir sınıfıdır. **Monitor** sınıfı referans tipindeki değişkenleri senkronize etmek için kullanılır. Değer tipindeki değişkenler için **monitor** sınıfı kullanılmaz. Monitor'ün kullanımı lock'un kullanımına benzer. **Monitor** blokladığı kodların başka kanallar tarafından erişilmesini engeller. **Monitor** sınıfının bazı metodları aşağıdaki tabloda özetlenmiştir.

Metod	Görevi
Enter, TryEnter	<p>Bloklanmak istenen nesne Enter ile Monitor nesnesine kayıt ettirilir. Diğer kanallar bu bloklanan nesneye müdahale edemez. Nesne artık kilitlenmiştir.</p> <p>Monitor.Enter ile bloklanmak istenen kodlar için lock'a benzer şekilde başlangıç verilebilir.</p>
Wait	<p>Bloklanmış nesne üzerindeki kilidi açar. Diğer kanallar nesneye ulaşabilir ve onu kilitleyebilir.</p>
Pulse, PulseAll	<p>Bir kanaldaki kilitlenmiş nesnenin, üzerindeki kilidin kaldırılacağını bekleyen kanallara duyurmak için kullanılır. Diğer kanallar kendi kuyruklarına kilidi açılabilecek olan nesneyi kayıt ederler. Pulse, Exit metodu öncesi kullanılan bir metottur ve ana amacı diğer kanallara kilidin açılacağını önceden duyurmaktır.</p>
Exit	<p>Nesne üzerindeki kilidi açar.</p> <p>Monitor.Exit ile bloklanmak istenen kodlar için lock'a benzer şekilde sonlanma noktası verilebilir.</p>

Monitor sınıfının bazı metodları

- **Monitor** sınıfının DegerArttir fonksiyonunda nasıl kullanılacağı aşağıda gösterilmiştir. Kısır döngüler (deadlock) ile karşılaşmamak için monitorObject nesnesinin üzerindeki kilit finally bloğunda herzaman kaldırılmaktadır.
- ```
private object monitorObject = new object();
public void DegerArttirMonitor1()
{
 for (int i = 1; i <= 100; ++i)
 {
 try
 {
 Monitor.Enter(monitorObject);
 progressBar1.Value += 1;
 lbThreads.Items.Add("Thread 1");
 Thread.Sleep(10);
 Monitor.Pulse(monitorObject);
 }
 finally
 {
 Monitor.Exit(monitorObject);
 }
 }
}
```

- **Mutex :**
- **Mutex** sınıfı monitor sınıfına benzer ancak System.Threading.WaitHandle sınıfından türetilmiştir ve türediğin sınıfın daha kolay kullanılabilir bir genelleştirilmesidir. **Mutex** sınıfı threadler tarafından ortak kullanılan nesnelere aynı anda ulaşılıp, işlem yapılmasını engellemek için kullanılır. **Mutex** sınıfı ortak kullanılan kaynaklara bir t zamanında sadece bir kanalın ulaşılmasını garanti eder. Mutex kendisini kullanan thread'in tekillliğini (identity) kontrol eder. Bir mutex'e sahip olan thread WaitOne metodu ile onu kilitler ve ReleaseMutex metodu ile mutex'i serbest bırakır. Bir **mutex** kullanan kanal sadece kendi mutex'ini ReleaseMutex metodu ile açabilir. Kanallar birbirlerinin mutex'lerini serbest bırakamaz. Bizim örneğimizde progressbar'ların tam anlamı ile aynı anda ilerlemediğini fark edeceksiniz. Burada mutex'i sadece ortak kaynaklara güvenli erişim amacı ile kullandık.

- private object mutexObject = new object();  
public void DegerArttirMutex1()  
{  
    for (int i = 1; i <= 100; ++i)  
    {  
        mutexObject.WaitOne();  
        progressBar1.Value += 1;  
        lbThreads.Items.Add(>"Thread 1");  
        Thread.Sleep(10);  
        mutexObject.ReleaseMutex();  
    }  
}

- **Semaphore** :
- **Semaphore** sınıfı .NET 2.0 framework ile gelen yeni bir sınıfır. Bu sınıf da System.Threading.WaitHandle sınıfından türetilmiştir. **Semaphore** sınıfının Mutex sınıfından farkı, farklı kanalların birbirlerinin Semaphore'larının kilitlerini Release metodu ile açabilmeleridir. Bir kanal semaphore'un WaitOne metodunu birçok kez çağırabilir. Bu kilitleri açmak için art arda Release metodunu çağrıabilecegi gibi, Release(int) overload'unu da kullanabilir. **Semaphore** kendisini kullanan kanalın identity'sine bakmaz. Bu yüzden farklı kanallar birbirlerinin semaphore'larının WaitOne ve Release metodlarını çağrıabilir. Herbir WaitOne metodu çağrııldığında semaphore'un sayacı bir azaltılır. Herbir release metodu çağrııldığında ise sayıç bir arttırılır. Semaphore'un yapılandırıcısında (constructor) sayacın minimum ve maksimum değerleri belirlenebilir

- ```
private Semaphore semaphoreObject = new Semaphore(0,2);
public void DegerArttirSemaphore1()
{
    for (int i = 1; i <= 100; ++i)
    {
        semaphoreObject.WaitOne();
        progressBar1.Value += 1;
        lbThreads.Items.Add("Thread 1");
        Thread.Sleep(10);
        semaphoreObject.Release();
    }
}

private void btnSemaphore_Click(object sender, EventArgs e)
{
    th1 = new Thread(new ThreadStart(DegerArttirSemaphore1));
    th2 = new Thread(new ThreadStart(DegerArttirSemaphore2));
    th3 = new Thread(new ThreadStart(DegerArttirSemaphore3));
    th1.Start();
    th2.Start();
    th3.Start();
    semaphoreObject.Release(2);
}
```

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- **Process senkronizasyonu**
 - Kritik bölüm problemi
 - Peterson çözümü
 - Senkronizasyon donanımı
 - Mutex kilitlenmeleri
 - Semaforlar
 - İzleyiciler
 - Alternatif yaklaşımlar

Process senkronizasyonu

- Cooperating process'ler diğer process'leri etkilerler veya diğer process'lerden etkilenirler.
- Cooperating process'ler paylaşılmış hafıza alanı veya dosya sistemleri ile veri paylaşımı yaparlar.
- Paylaşılmış veriye eşzamanlı erişim tutarsızlık problemlerine yol açabilir.
- Paylaşılmış veri üzerinde işlem yapan process'ler arasında veriye erişimin yönetilmesi gereklidir.
- Paylaşılan veriye erişim üretici-tüketici (**producer-consumer**) problemi olarak modellenebilir.

3

Process senkronizasyonu

- Üretici ve tüketici processler için örnek kod aşağıda verilmiştir.

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}  
  
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
  
    /* consume the item in next_consumed */  
}
```

Yeni eleman eklendi.

Bir eleman alındı.

- **counter** değişkeninin değeri buffer'a yeni eleman eklendiğinde artmakta, eleman alındığında azalmaktadır.

4

Process senkronizasyonu

```
while (true) {  
    /* produce an item in next_produced */  
  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}  
  
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
  
    /* consume the item in next_consumed */  
}
```

- İki örnek ayrı ayrı doğru olsa da eşzamanlı doğru çalışmamayabilirler.
- `counter=5` iken `counter++` ve `counter--` deyimlerinin aynı anda çalıştığını düşünelim.
- Farklı zaman aralıklarında çalışmış olsalar da `counter=5` olacaktır.

5

Process senkronizasyonu

- `counter++` için makine komutları aşağıdaki gibi olabilir.

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- `counter--` için makine komutları aşağıdaki gibi olabilir.

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- `register1` ve `register2` aynı (AC) veya farklı register olabilir.

6

Process senkronizasyonu

- `counter++` ve `counter--` için sıralı zaman aralıklarında yapılan mikroişlemler aşağıdaki gibi olabilir.

$T_0:$	<i>producer</i>	execute	$register_1 = \text{counter}$	{ $register_1 = 5$ }
$T_1:$	<i>producer</i>	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
$T_2:$	<i>consumer</i>	execute	$register_2 = \text{counter}$	{ $register_2 = 5$ }
$T_3:$	<i>consumer</i>	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
$T_4:$	<i>producer</i>	execute	$\text{counter} = register_1$	{ $\text{counter} = 6$ }
$T_5:$	<i>consumer</i>	execute	$\text{counter} = register_2$	{ $\text{counter} = 4$ }

- Yukarıdaki sırada buffer'daki eleman sayısı **4 olarak görülür**, ancak **gerçekte buffer'daki eleman 5 tanedir**.
- **T_4 ile T_5 yer değiştirirse** buffer'daki eleman sayısı **6 olarak görülecektir**.
- **İki process aynı anda counter değişkeni üzerinde işlem yaptığından** sonuç yanlış olmaktadır.

Process senkronizasyonu

- Aynı değişkene **çok sayıda process'in erişmesi** durumunda sonuç değer erişim sırasına bağlı olarak değişecektir (**race condition**).
- Paylaşılan bir değişkene **aynı anda sadece bir process'in erişimi** sağlanmak zorundadır (**process synchronization**).
- Günümüzde işletim sistemlerinin farklı kısımlarındaki **process'lerin aynı veriye erişiminde senkronizasyon yapılmak zorundadır**.
- Multicore işlemcilerde çalışan multithread uygulamalarda da **process senkronizasyonu yapılmak zorundadır**.

Konular

- Process senkronizasyonu
- **Kritik bölüm problemi**
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

9

Kritik bölüm problemi

- Bir sistem, $\{P_0, P_1, \dots, P_{n-1}\}$ şeklinde n tane process'e sahip olsun.
- **Her process**, ortak değişkenler, tablolar veya dosyalar üzerinde işlem yapan **kritik bölüme sahip olabilir**.
- Bir process kendi kritik bölümünü çalıştırırken diğer process'lerin kendi kritik bölümlerini çalışmamaları zorunludur.
- **Aynı anda iki process kritik bölümünü çalışmamalıdır**.
- Kullanılan protokoller ile **her process kritik bölümne girmek için izin istemektedir**.
- Kritik bölümden sonra çıkış bölümü de yer alabilir.
- Örnekte, **entry section** giriş izni için kullanılır.

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

10

Kritik bölüm problemi

- Kritik bölüm probleminin çözümü aşağıdaki üç gereksinimi sağlamak zorundadır:
 - **Mutual exclusion (karşılıklı dışlama):** Bir P_i process'i kritik bölümünü çalıştırıyorsa diğer process'lerin hiç birisi kritik bölümlerini çalıştırılamazlar.
 - **Progress:** **Hiçbir process kritik bölümünü çalıştırımıyorsa**, kritik bölüme girmek isteyenlerden (remainder section çalışmayanlar arasından) bir tanesinin kritik bölüme girmesine izin verilir.
 - **Bounded waiting (sınırlı bekleme):** Bir process kritik bölüme giriş izni istedikten sonra veizin verildikten önceki aralıkta, kritik bölüme giriş izni verilen process sayısının sınır değeri vardır.
- İşletim sisteminde **açık durumdaki tüm dosya listesi için kernel veri yapısı oluşturulur.**
- İki process aynı anda dosya açma veya kapatma işlemi yaptığında bu listeye erişmeleri gerekir (race condition).

11

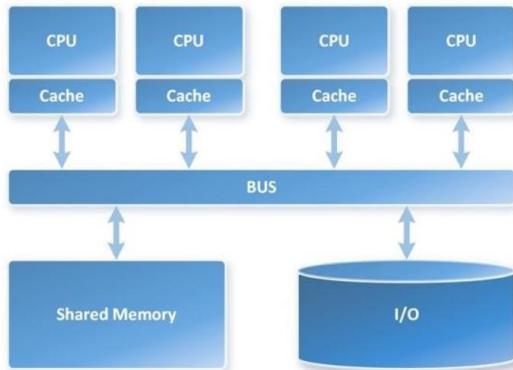
Kritik bölüm problemi

- **Hafıza tahsis edilmesi** ve **interrupt işlemleri** gibi işlemler **race condition** içeren örneklerdir.
- **Kritik bölüm yönetimi için** iki yaklaşım vardır:
 - **Preemptive kernel:** Bir process **kernel mode'da** çalışırken sonsuz öncelikli (preemptive) olabilir.
 - **Nonpreemptive kernel:** Bir process **kernel mode'da** çalışırken sonsuz öncelikli olamaz, onun yerine bir **kernel-function** çalışır.
- Nonpreemptive kernel, **kernel veri yapıları üzerinde race condition oluşturmaz**, aynı anda bir process kernel içinde aktif durumdadır.

12

Kritik bölüm problemi

- SMP (Symmetric Multiprocessing) mimarisinde (her işlemci eş düzey işlem kapasitesine sahiptir.) preemptive kernel tasarımları daha zordur.
- Birden fazla kernel mode process, aynı anda farklı işlemcilerde çalışabilir.



- Preemptive kernel'in cevap verebilirliği (responsiveness) daha iyidir ve gerçek zamanlı uygulamalar için daha uygunudur.

13

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

14

Peterson çözümü

- Peterson çözümü, **yazılım tabanlı** kritik bölüm çözümüdür.
- P_i ve P_j process'leri için kritik bölüm çözümü aşağıdaki gibidir.

Bu process kritik kesime hazır.

Diğer process kritik kesime hazırlırsa öncelik ona verilir.

Sıra kendisine gelene kadar bekler.

Kritik kesimden çıkış bildirimi ($\text{flag}[i] = \text{false}$).

```
do {
```

```
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);
```

critical section

```
    flag[i] = false;
```

remainder section

```
} while (true);
```

- Peterson çözümü, P_i ve P_j process'leri için **iki veriyi paylaşarak kullanır**.

```
int turn;  
boolean flag[2];
```

15

Peterson çözümü

- Eğer $\text{turn} = i$ ise, kritik bölüme i process'i girecektir.
- $\text{flag}[j]$ bitleri ise process'lerin kritik bölüme girmeye hazır durumunu gösterir.
- Eğer $\text{flag}[i] = \text{true}$ ise, i .process kritik bölümünü girmeye hazırlıdır.
- i .process $\text{flag}[i] = \text{true}$ VE $\text{turn} = i$ olunca kritik bölümünü girer.
- turn değişkenini iki process'te aynı anda değiştirse bile, son değer alınır ve o process kritik bölüme girer (**mutual exclusion**).
- Kritik bölümü tamamlayan process kritik bölüme giriş isteğini iptal eder ve diğer process kritik bölüme girer (**progress**).
- Bir process kritik bölüme bir kez girdikten sonra sırayı diğerine aktarır (**bounded waiting**).

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    critical section  
    flag[i] = false;  
    remainder section  
} while (true);
```

16

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- **Senkronizasyon donanımı**
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

17

Senkronizasyon donanımı

- Kritik bölüm problemi için çok sayıda donanım ve yazılım tabanlı çözüm vardır. **Bunlar, temel olarak kilitleme (**locking**) tabanlı yaklaşılardır.**
- **Tek işlemcili sistemlerde, interrupt'ların paylaşılmış veriye erişimi engellenirse**, kritik bölüm problemi basit bir şekilde çözülebilir.
- Bu sistemlerde, **komut sırası değiştirilmeden ve önceliklendirme yapmadan çalışma sağlanırsa kritik bölüm problemi ortaya çıkmaz.**
- Nonpreemptive kernel'ların sıklıkla kullandığı yaklaşımındır.
- **Bu yöntem çok işlemcili sistemlerde uygun çözüm değildir.**
- Çok işlemcili sistemlerde, **interrupt'ların disable/enable yapılması için tüm işlemcilere mesaj göndermek için zaman gereklidir** ve sistem performansı düşer.

18

Senkronizasyon donanımı

- Modern bilgisayar sistemlerinde, bir word içeriğini **test edip değiştirme** veya iki word içeriğini **yer değiştirme (swap)** işlemlerini **atomik** olarak yapan özel donanım komutları vardır.
- Test et ve değiştir komutu (atomik çalışır)** ile **karşılıklı dışlama (mutual exclusion)** yapan P_i process'i aşağıda verilmiştir.

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}

do {
    while (test_and_set(&lock))
        ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

Değer true yapıldı.

Mevcut değeri alındı ve geri döndürüldü.

lock = true

Değer początkta false.

pass by reference

19

Senkronizasyon donanımı

- Karşılaştır ve yer değiştir komutu (compare_and_swap)** ile **karşılıklı dışlama (mutual exclusion)** yapan P_i process'i aşağıda verilmiştir.

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}

do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```

Değer początkta 0.

Değer değiştirildi.

lock = 1

Mevcut değeri alındı ve geri döndürüldü.

pass by reference

Senkronizasyon donanımı

- Önceki algoritmalar karşılıklı dışlamayı sağlar, ancak **bounded-waiting** gereksinimini sağlamaz.
- Yandaki algoritma (P_i için çalışır) ile bounded-waiting karşılanır.
- Her process en fazla $(n-1)$ çalışma sonrasında sırayı alır.

Kendisinden sonraki bekleyen process dairesel dizide aranıyor.

Bekleyen yok

Bekleyen var

P_j kritik bölüme girer.

```
boolean waiting[n];
boolean lock;

do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */
} while (true);
```

Döngüden çıkışmasını sağlar.

Konular

- Process senkronizasyonu
 - Kritik bölüm problemi
 - Peterson çözümü
 - Senkronizasyon donanımı
 - Mutex kilitlenmeleri**
 - Semaforlar
 - İzleyiciler
 - Alternatif yaklaşımlar

Mutex kilitlenmeleri

- Donanım tabanlı kritik bölüm çözümleri **karmaşıktır** ve **uygulama geliştirici tarafından erişilemez**.
- İşletim sistemi tasarımcıları, kritik bölüm problemi için yazılım araçları geliştirmiştir.
- En basit yazılım aracı **mutex (mutual exclusion) lock** aracıdır.
- Her process kritik bölüme girmek ve lock yapmak için izin ister (**acquire() function**).
- Kritik bölümünden çıktıktan sonra da lock durumu sonlandırılır (**release() function**).
- Lock durumunun uygun olup olmadığına karar vermek için bir boolean değişken kullanılır (**available**).

23

Mutex kilitlenmeleri

- Aşağıda, **acquire()**, **release()** fonksiyonları ile **mutex lock** kullanılan **kritik bölüm çözümü verilmiştir**.

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false; ;  
}
```

```
release() {  
    available = true;  
}
```

```
do {  
    acquire lock  
  
    critical section  
  
    release lock  
  
    remainder section  
} while (true);
```

- Bir process, kritik bölümünde iken diğer tüm process'ler **acquire()** fonksiyonunda sürekli döngüdedirler (**spinlock**).

24

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- **Semaforlar**
- İzleyiciler
- Alternatif yaklaşımlar

25

Semaforlar

- S semaforu bir tamsayıdır ve sadece **wait()** ve **signal()** **atomik** işlemleri tarafından erişilebilir.
- Literatürde **wait()** işlemi P ile, **signal()** işlemi ise V ile gösterilir.
- **wait()** ve **signal()** işlemleri aşağıda verilmiştir.

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

- **wait()** ile S'nin değeri azaltılır, **signal()** ile S'nin değeri artırılır.
- S üzerindeki wait() ve signal() işlemleri **kesintisiz (atomik)** bir şekilde gerçekleştirilir.

26

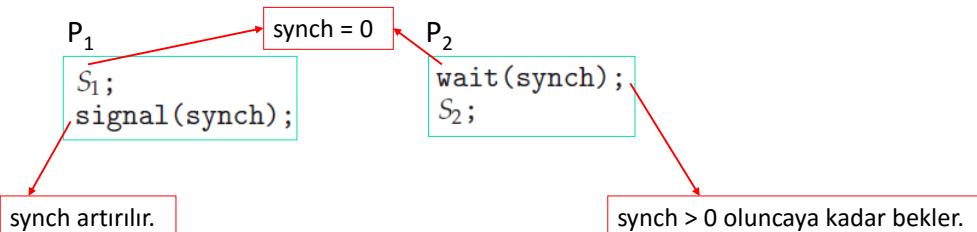
Semaforlar

- İşletim sistemleri, **sayan semafor (counting semaphore)** ve **ikilik semafor (binary semaphore)** kullanırlar.
- Sayan semaforların değeri kısıtlı değildir.**
- İkilik semaforların değeri 0 veya 1 olabilir.**
- İkilik semafor mutex lock gibi davranışır.
- Sayan semaforlar, belirli sayıdaki kaynağa erişimi denetlemek için kullanılır.** Sayan semafor kaynak sayısı ile başlatılır.
- Kaynağı kullanmak isteyen her **process semafor üzerinde wait() işlemi** gerçekleştirir (**sayaç azaltılır**).
- Bir process kaynağı serbest bıraktığında ise **signal()** işlemi gerçekleştirir (**sayaç artırılır**).
- Semafor değeri = 0** olduğunda tüm kaynaklar kullanılır durumdadır.

27

Semaforlar

- Semaforlar, **İşlem bağımlılığı** gibi farklı senkronizasyon problemlerinde de kullanılabilir.
- Örneğin, P_2 process'indeki S_2 deyimi P_1 process'indeki S_1 deyiminden sonra çalışmak zorunda olsun.
- Örnekte P_1 ve P_2 için paylaşılan **synch semaforu** tanımlanmıştır.
- synch semaforu başlangıçta 0** değerine sahiptir.
- P_1 ve P_2 içeresine eklenen deyimler aşağıda verilmiştir.**



28

Semaforlar

Semafor oluşturulması

- Mutex lock gibi semafordaki wait() ve signal() tanımları da süresiz beklemeye neden olabilir.
- Aşağıda örnek semafor tanımı verilmiştir:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- Her semafor bir tamsayı değeri ve process listesine sahiptir.
- Bir process semaforu bekliyorsa process listesine eklenir.
- Listeden bir process signal() ile alınır ve çalıştırılır.

29

Semaforlar

Semafor oluşturulması

- Semafor için wait() ve signal() işlemleri aşağıdaki gibi tanımlanabilir:
- **block()** process'i beklemeye alır, **wakeup()** ise çalışmaya devam ettirir.

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

Negatif değer büyüklüğü bekleyen process sayısını gösterir.

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

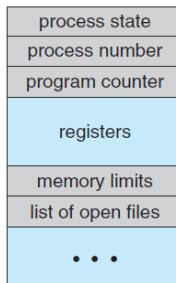
```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

30

Semaforlar

Semafor oluşturulması

- Bekleyen process listesi, her process'in PCB (Process Control Block) linkiyle oluşturulabilir.



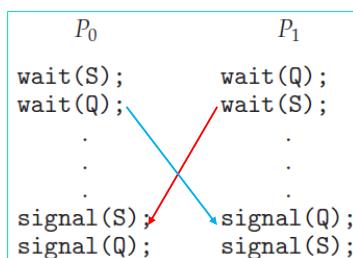
- Her semafor bir tamsayı ile PCB pointer'ına sahiptir.
- Bounded waiting için FIFO kuyruk oluşturulur.
- FIFO kuyruk yapısının dışında priority queue yapısı da oluşturulabilir.
- Semafor işlemlerinin atomik olarak çalıştırılması gereklidir.

31

Semaforlar

Kilitlenme (Deadlock)

- Bir process'in beklemesine bağlı olarak, iki veya daha çok process'in sonsuza kadar beklemesine kilitlenme (deadlock) denir.
- Aşağıdaki P_0 ve P_1 process'leri, S ve Q semaforlarına erişmektedir.



- P_0 wait(S) ve P_1 wait(Q) işlemlerini aynı anda çalıştırınsın.
- P_0 wait(Q)'yu çalıştırırken, P_1 wait(S)'yı çalıştırır.
- P_0 wait(Q)'da, P_1 wait(S)'de kilitlenir.

32

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- **İzleyiciler**
- Alternatif yaklaşımlar

33

İzleyiciler

- Semaforlar kullanıldığında da senkronizasyon hataları olabilmektedir.
- Tüm process'lerin **kritik bölüme girmeden önce wait(), girdikten sonra ise signal() işlemlerini** yapmaları gereklidir.
- **Program geliştirici bu sırada dikkat etmezse, iki veya daha fazla process aynı anda kritik bölüme girebilir.**
- Bu durumlar programcılar arasında yeterli işbirliği olmadığı durumlarda da olabilmektedir.
- Kritik bölüm problemine ilişkin tasarımında oluşan sorunlardan dolayı **kilitlenmeler veya eşzamanlı erişimden dolayı yanlış sonuçlar** ortaya çıkabilmektedir.

34

Izleyiciler

- **wait() ile signal() yer değiştirse**, aşağıdaki çalışma ortaya çıkar.

```
signal(mutex);
...
critical section
...
wait(mutex);
```

- Örnekte **birden fazla process kritik bölüme aynı anda girebilir**.
- signal() yerine wait() yazılırsa aşağıdaki çalışma ortaya çıkar.

```
wait(mutex);
...
critical section
...
wait(mutex);
```

- Bu durumda da **deadlock oluşur**.
- wait() veya signal() unutulursa, karşılıklı dışlama yapılamaz veya deadlock oluşur.

35

Izleyiciler

- Programçıdan kaynaklanabilecek bu hataların giderilmesi için **izleyici (monitor)** kullanılır.
- **monitor içinde tanımlanan bir fonksiyon**, sadece **monitor içinde tanımlanan değişkenlere ve kendi parametrelerine erişebilir**.
- **Monitor içindeki fonksiyonlardan sadece bir tanesi aynı anda aktif olabilir**.

```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . . .
    }

    function P2 ( . . . ) {
        . . .
    }

    .
    .

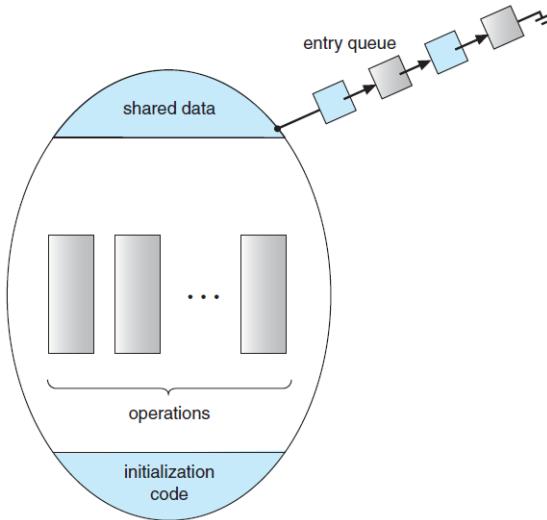
    function Pn ( . . . ) {
        . . .
    }

    initialization_code ( . . . ) {
        . . .
    }
}
```

36

İzleyiciler

- Programcının senkronizasyon kısıtlarını yazması gereklidir.



37

İzleyiciler

- Programcı işe veya **değişkene özel senkronizasyon** oluşturmak için durum değişkenleri oluşturabilir.

condition x, y

- Sadece **bir durum değişkenine bağlı çalışan** wait() ve signal() işlemleri tanımlanabilir.

x.wait();

ile x durum değişkenine bağlı bir process beklemeye alınır.

x.signal();

ile x durum değişkenine bağlı beklemekte olan bir process çalışmaya devam eder.

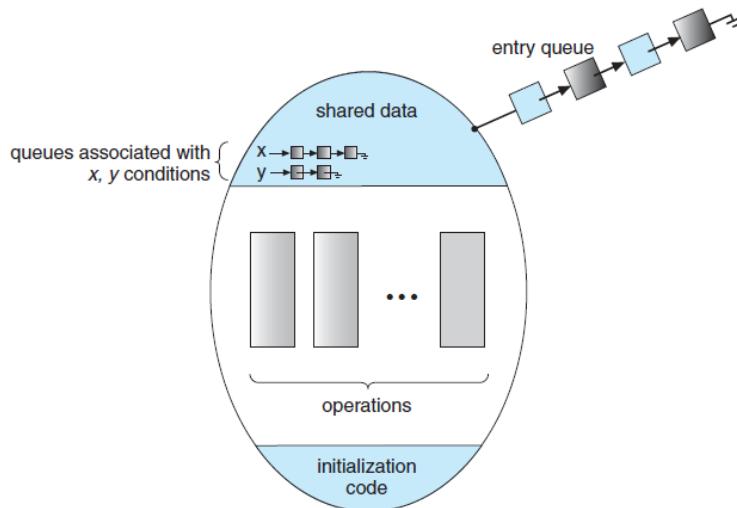
38

İzleyiciler

- **x.signal ()** işlemini bir **P** process'i başlatmış olsun. Aynı anda, **x** durumuna bağlı **beklemekte olan bir Q process'i olsun.**
- **Q process'i çalışmaya başladığında**, tekrar işlem yapmak isterse **P** process'i beklemek zorundadır.
- Aksi durumda, monitör içindeki P ve Q aynı anda aktif olur.
- Bu durumda iki olasılık vardır:
 - **Signal and wait:** **P process'i, Q process'inin monitör'den ayrılmmasını veya başka bir duruma geçmesini bekler.**
 - **Signal and continue:** **Q process'i, P process'inin monitör'den ayrılmmasını veya başka bir duruma geçmesini bekler.**

İzleyiciler

- **x** ve **y** durum değişkenlerine bağlı process'lerin monitör içinde çalışması.



Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

41

Alternatif yaklaşımlar

Transactional memory

- Multicore sistemlerde, mutex lock, semafor gibi mekanizmalarda deadlock gibi problemlerin oluşma riski bulunmaktadır.
- Bunun yanı sıra, thread sayısı arttıkça deadlock problemlerinin ortaya çıkma olasılığı artmaktadır.
- Klasik mutex lock (veya semafor) kullanılarak paylaşılmış veride güncelleme yapan update() fonksiyonu aşağıdaki gibi yazılabılır.

```
void update ()  
{  
    acquire();  
  
    /* modify shared data */  
  
    release();  
}
```

42

Alternatif yaklaşımlar

Transactional memory

- Klasik kilitleme yöntemlerine alternatif olarak **programlama dillerine yeni özellikler eklenmiştir.**
- Örneğin, **atomic(S)** kullanılarak **S işlemlerinin tümünün transaction olarak gerçekleştirilmesi sağlanır.**

```
void update ()  
{  
    atomic {  
        /* modify shared data */  
    }  
}
```

- Lock işlemine gerek olmadan ve kilitlenme olmadan işlem tamamlanır.

43

Alternatif yaklaşımlar

OpenMP (Open Multi-Processing)

- OpenMP, C, C++ ve Fortran için compiler direktiflerinden oluşan API'dir.
- OpenMP, paylaşılmış hafızada eşzamanlı çalışmayı destekler.
- OpenMP, **#pragma omp critical** komutu ile kritik bölümü belirler ve aynı anda sadece bir thread çalışmasına izin verir.

```
void update(int value)  
{  
    #pragma omp critical  
    {  
        counter += value;  
    }  
}
```

44

İşletim Sistemleri 3.Uygulama

Thread Komutları

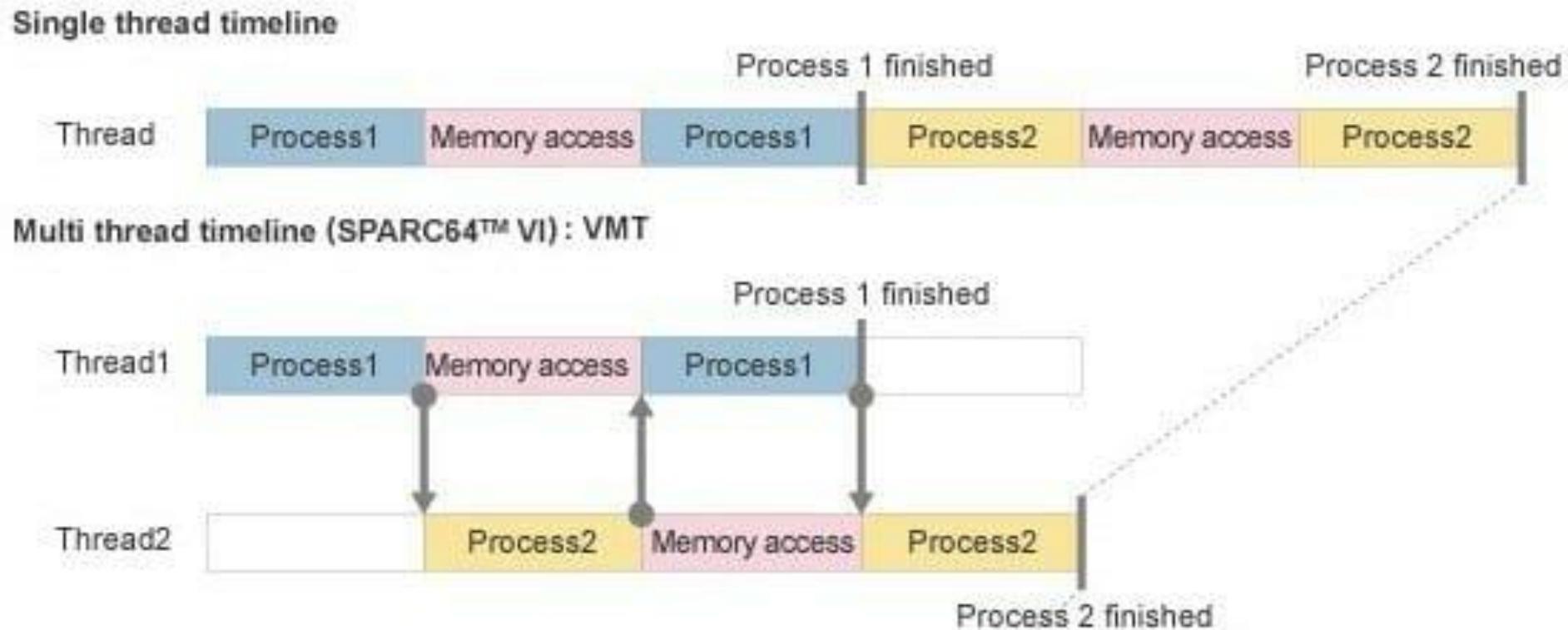
Hazırlayan : Özlem ÖZMEN AKALIN

- Linux altında thread kullanımı C kütüphanesinin bir parçası olan pthread kütüphanesi ile yapılmaktadır.
- Diğer işletim sistemlerinin aksine, Linux'te thread ile process arasında çok az fark vardır.
- Bu nedenle thread'ler, LightWeightProcess olarak adlandırılmaktadır.
- Linux versiyon **2.6** öncesindeki thread implementasyonu **LinuxThreads** şeklinde adlandırılıyordu. Bu implementasyon performans ve senkronizasyon işlemleri açısından önemli limitlere sahipti. Maksimum çalışabilecek thread sayısı 1000'li rakamlarla sınırlıydı.
- 2003 yılında özellikle **IBM** ve **RedHat** firmasındaki geliştiricilerin başını çektiği ekip, **Native POSIX Thread Library** (NPTL) projesini kullanılabılır duruma getirmeyi başardı ve ilk olarak enterprise dünyanın Linux üzerinde Java Virtual Machine performansı şikayetlerini çözüme kavuşturmak için RedHat Enterprise versiyon 3'te kullanılmaya başlandı. Ardından GNU C kütüphanesindeki yeni NPTL implementasyonu gelmeye başladı. Günümüzde de her iki implementasyon glibc içerisinde yer almaktır, kullanılan çekirdek versiyonu tarafından uygun implementasyon seçilmektedir.

Thread Nedir ?

- İş parçacıkları olarak da bilinir ve projelerde olmazsa olmaz terimler arasındadır. Thread, programın kendini senkronizasyonla birden çok iş bölümüne ayırabilmesinin farklı bir yoludur. Bilgisayarımızda tek işlemci kullanıyorsak thread kullandığımızda uygulamalar zaman dilimleme ile gerçekleştirilir. Yani tek işlemci iki ya da daha çok Thread arasında çok hızlı geçiş yapar. Bu olay sanki eş zamanlı olarak işlemlerin gerçekleştirildiği izlenimini yansıtır.
- İş parçacıkları tek başına kullanıldığı gibi çoklu yani multithreading olarak da kullanılır. Genel kullanımı bu şekildedir. Bunun amacı da eş zamanlı olarak -paralel – yapılması istenilen işlemlerde kullanılmaktadır.
- Örneğin saat gösteren bir uygulamada aynı zamanda başka bir işlem de yapılması gerekebilir. Bu durumda iş parçacıkları kullanılır ve ikisi de birbirinden bağımsız olarak işlemiş olurlar.

Threading, aynı ortamda aynı anda birden fazla işi yapmaya denir. Thread'ler ise bu işlerin her biridir. Thread'ler aynı anda çalıştığında işlemciye process'ler olarak gider ve sıraya alınır. Tek çekirdekli işlemcilerde birden fazla thread çalıştığı zaman sırayla threadlerin processlerini işleme alır ve bir thread diğer thread'in bitişini beklemeden processleri işlemcide işleme girer. Çok çekirdekli işlemcilerde ise farklı thread'ler farklı çekirdeklerde aynı anda çalışabilme durumuna sahiptir.



Threadleri neden kullanıyoruz?

- Thread'ler genel olarak, oyunlar, paralel çalışan task'ler, event handling gibi durumlarda kullanılmaktadır. Ayrıca çok çekirdekli işlemcilerde tam performans yararlanmak için de thread'ler kullanılır. İnsan üzerinden thread'leri açıklamak gerekirse, bir insan aynı anda konuşurken gözleri başka bir yere bakıp elleriyle ayrı işler yapabilmektedir. Bu tam olarak multithreading örneğidir.
- Thread'lerin kullanıldığı bir başka alan ise server-client uygulamalar. Günümüzde popüler olarak Whatsapp, forum siteleri, online oyunlar bu mimariyi kullanmaktadır. Birden fazla kullanıcı birbirini beklemeden işlerini halletmek durumunda olduğundan her bir kullanıcının işlemleri ayrı threadlerde yönetilmektedir.

Bazı terimler

- **ThreadPool:** İşletim sistemi seviyesinde threadlerin bir arada bulunduğu bir havuzdur. Çok fazla thread kullanan uygulamalar her bir thread işlemciye yük oluşturacağından thread ihtiyaçlarını bu havuzdan alarak çözmektedir. Örneğin IIS bu şekilde threadpool kullanmaktadır.
- **WorkerThread:** ThreadPool'daki threadlerin her biri bir worker thread'dır
- **BackgroundWorker:** Arkaplanda çalışan thread'lere verilen isimdir. Bir örnekle açıklamak gerekirse,
- Bir masaüstü uygulaması düşünelim. Kullanıcı bir indirme işlemi başlatmak istiyor ancak bu işlem büyük dosya boyutlarında çok uzun süreceğinden backgroundworker kullanılmazsa kullanıcı programı indirme işlemi bitinceye kadar kullanamayacaktır. Bunun önüne geçmek için bir backgroundworker oluşturulup indirme işlemine burdan devam etmesi ve kullanıcının da programı kesintiye uğramadan kullanması sağlanmaktadır.

- C# uygulamalarında ana programın işlerinin yürüdüğü bir main thread bulunmaktadır. Her Thread objesi main thread'e paralel olarak çalışan bir worker thread yaratmaktadır.
- Örnek Uygulama:

```
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread (WriteY);
        t.Start();

        // Simultaneously, do something on the main thread.
        for (int i = 0; i < 1000; i++) Console.Write ("x");
    }

    static void WriteY()
    {
        for (int i = 0; i < 1000; i++) Console.Write ("y");
    }
}
```

Ekran çıktısı:

```
xxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyy  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyy  
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyy  
yyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
...
```

- Threading işlemlerini kontrol eden tipler **System.Threading** kütüphanesidir. Thread in sınıfından birden fazla metodu olduğu için en temel Start() ve Sleep() metodlarından bahsedelim.
- **Start():** Thread classının en ana metodudur ve threadlerin çalışmasını başlatır.
- // Nesne oluşturup ThreadStart(çalıştırılacak methodun adı) methoduyla çalıştırılacak olan method belirlenir.
- Thread thread1 = new Thread(new ThreadStart(function1));
- // thread1 çalıştırılır
- thread1.Start();
- **Sleep():** Sleep metodunu kullanarak threadin çalışmasını bir süreliğine bekletebiliriz. Örneğin; belli bir zamanda fonksiyonu çalıştırıp 0.20 saniye dinlendirdikten sonra işleme devam ettirebiliriz ya da başka fonksiyonları çalıştırabiliriz.

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
//Thread kullanabilmek için öncelikle aşağıdaki 2 satırı eklemeliyiz.
using System.Threading;
using System;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        //Button Click olayı altında Thread lerimizi oluşturalım ve başlatalım.
        static void button1_Click(object sender, EventArgs e)
        {
            Thread thread1 = new Thread(new ThreadStart(ThreadFuncEven));
            Thread thread2 = new Thread(new ThreadStart(ThreadFuncOdd));
            //Threadleri başlatalım bakalım sırasına göremi yoksa paralelmi işleyecek.
            //Sırasına göre işlenseydi önce çift sayılar, sonra tek sayılar basılmalıydı.
```

```
thread1.Start();
thread2.Start();
}
//Output her saniye 2 sayı gelecek şekilde 1 2 3 4 5 ... 100 şeklinde olacaktır.
static void ThreadFuncEven()
{
    for (int i = 0; i < 100; i += 2)
    {
        //Her çift Sayı için 1 saniye bekleyelim ve sayıyı yazdıralım
        Thread.Sleep(1000);
        Console.WriteLine(i);
    }
}
static void ThreadFuncOdd()
{
    for (int i = 1; i < 100; i += 2)
    {
        //Her Sayı için bir saniye bekleyelim ve sayıyı yazdırın.
        Thread.Sleep(1000);
        Console.WriteLine(i);
    }
}
```

Threadlerde Öncelik (Thread Priority)

- Eşzamanlı olarak çalışabilen iş parçacıkları olduğundan bahsetmiştik. Fakat bazı durumlarda öncelik değişim mümkündür. Aynı anda başlatılan iki parçacığın kullanımında birisini önce başlatmak durumunda kalabiliriz. **Thread.Priority** özelliği ile bunu yapabiliriz.
- *Highest, AboveNormal, Normal, BelowNormal, Lowest* gibi türleri vardır.
- `thread_1.Priority = ThreadPriority.Lowest ;`

Thread.Join İşlemi

- Thread işleminde bazı durumların da birbirini beklemesi gerekmektedir. Bu durumlarda join işlemini uyguluyoruz. Böylece bir önceki iş bitmeden diğerine geçiş yapılmamış oluyor. Ayrıca bu metodun belirlenen bir süre kadar işlem yapılması gibi overload özellikleri de bulunur.
- İş parçacıkları için bazı genel kullanılan özellikler şunlardır ;
- **IsAlive** : Bu özellik iş parçacıklarının durumunu sorgulamaya yarar. Dönüş olarak bool değer tipi döner. Fakat Thread.ThreadState değerinden farklı olarak true veya false değer döner.
- **ThreadState**: Bu özellik ise durumu çalışıyor, iptal edildi, beklemede gibi daha detaylı özelliklerine ayırmamıza yarar.
- **Suspend**: Adından da anlaşılacağı gibi suspend yani işlemimizi askıya almamıza yarayan fonksiyondur.

Thread Nasıl Kullanılır?

- using System.Collections.Generic ;
- using System.ComponentModel ;
- using System.Data ;
- using System.Drawing ;
- using System.Text ;
- //Thread kullanabilmek için öncelikle aşağıdaki 2 satırı eklemeliyiz.
- using System.Threading ;
- using System;

Thread Oluşturma Komutları

- Yeni bir thread oluşturmak için **pthread_create** fonksiyonu kullanılır.
- Thread fonksiyonlarının kullanımı için **pthread.h** başlık dosyası dahil edilmelidir.
- Threadler ana program ile aynı adresleme alanını ve aynı dosya tanımlayıcılarını kullanırlar.
- Pthread kütüphanesi aynı zamanda senkronizasyon işlemleri için gerekli **mutex** ve **conditional** işlemleri için gerekli desteği de içermektedir.
- Pthread kütüphanesi fonksiyonları kullanıldığında uygulama **pthread kütüphanesi** ile birlikte çağrırmalıdır.

- \$ gcc -o example example_thread.c -lpthread
- int pthread_create(pthread_t *thread, const pthread_attr_t *attr ,
- void *(*start_routine) (void *), void *arg)
- Başarılı durumda 0 döner. Hata durumunda ise geriye 1 hata kodu dönecektir. İş parçası kimliği "pthread_t" türüyle temsil edilir. Bir işlem kimliği bir tamsayı değeridir, ancak iş parçası kimliği mutlaka bir tamsayı değeri değildir. Bir yapı olabilir
- Thread parametresi **pthread_t** türünde olup önceden tanımlanması gereklidir. Oluşan thread' e bu referansla her zaman erişilebilecektir.
- attr parametresi thread spesifik olarak **pthread_attr** ile başlayan fonksiyonlarla ayarlanmış, scheduling policy , stack büyüklüğü,
- detach policy gibi kuralları gösterir.
- start_routine thread tarafından çalıştırılacak olan fonksiyonu gösterir.
- arg ise thread tarafından çalıştırılacak fonksiyona geçirilecek void* 'a cast edilmiş genel bir veri yapısını göstermektedir.

- **Örnek Uygulama**

- # include <stdio.h>
- # include <stdlib.h>
- # include <unistd.h>
- # include <phtread.h>
- void *worker (void *data)
- {
- char *name=(char*)data
- for (int i=0 ; i<120; i++)
- {
- usleep(50000) ;
- printf('Hello from thread %s \n ',name);
- }
- printf ('Thread %s done...\n ,name);
- return NULL ;
- }

```
int main (void) {  
    pthread_t th1,th2 ;  
    pthread_create (&th1,NULL,worker, 'A ') ;  
    pthread_create (&th2,NULL,worker, 'B ') ;  
    sleep(5) ;  
    printf('Exiting from main program\n') ;  
    return 0 ;  
}
```

Birleştirilebilir ve Çıkarılabilir Thread Türleri

- Thread kullanılan bir uygulamada main() fonksiyonundan return edilirse, tüm thread'lerde sonlandırılır ve kullanılan tüm kaynaklar sisteme geri verilir.
- Aynı şekilde herhangi bir thread içerisinde exit() benzeri bir komutla çıkış yapılması halinde gene tüm thread'ler sonlandırılacaktır.
- `pthread_join` fonksiyonu ile bir thread'in sonlanması bekleyebiliriz. Bu fonksiyonun kullanıldığı thread, sonlanması beklenen thread sonlanana kadar bloklanacaktır.
- Normal (*joinable*) thread'ler, sonlanmış olsa dahi `pthread_join` ile *join* işlemine tabi tutulmazlar ise, CPU tarafından tekrar programlanamasalar da sistemden kullandığı kaynaklar **geri verilmez**.

Çıkarılabilir Thread

- Bazen pthread_join ile join işlemi yapmanın anlamlı olmadığı, thread'in ne zaman sonlanacağının öngörülemediği durumlar olabilir.
- Bu durumda thread return ettiği noktada tüm kaynakların sisteme otomatik olarak geri verilmesini sağlayabiliriz.
- Bunu sağlamak için ise, ilgili thread'leri, **DETACHED** durumu ile başlatmamız gerekmektedir.
- Bir thread başlatılırken thread attribute değerleri üzerinden veya pthread_detach fonksiyonu ile DETACH durumu belirtilebilir :
- `int pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);`
- `int pthread_detach(pthread_t thread) ;`

Thread Sonlandırma

- Bir thread, başka bir thread tarafından, ilgili p_thread_t id değeri verilmek suretiyle iptal edilebilir.
- `int p_thread_cancel (pthread_t thread) ;`

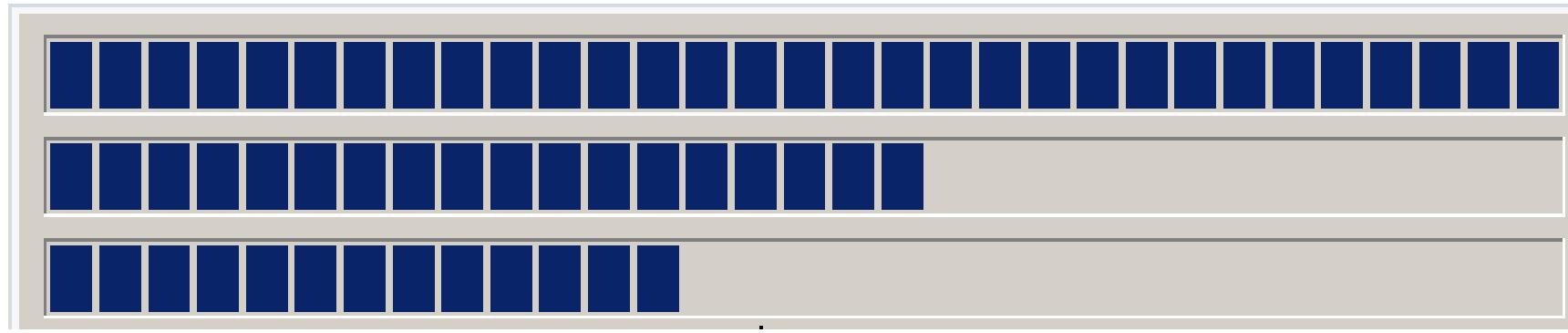
Multi Threading

- public void DegerArttir1()
- {
 - for (int i = 1; i <= 100; ++i)
 - {
 - progressBar1.Value += 1;
 - lbThreads.Items.Add("Thread 1");
 - Thread.Sleep(10);
- } public void DegerArttir2()
- {
 - for (int j = 1; j <= 100; ++j)
 - {
 - progressBar2.Value += 1;
 - lbThreads.Items.Add("Thread 2");
 - Thread.Sleep(100);
- }

- public void DegerArttir3()
 - {
 - for (int k = 1; k <= 100; ++k)
 - {
 - progressBar3.Value += 1;
- lbThreads.Items.Add("Thread 3");
- Thread.Sleep(150);

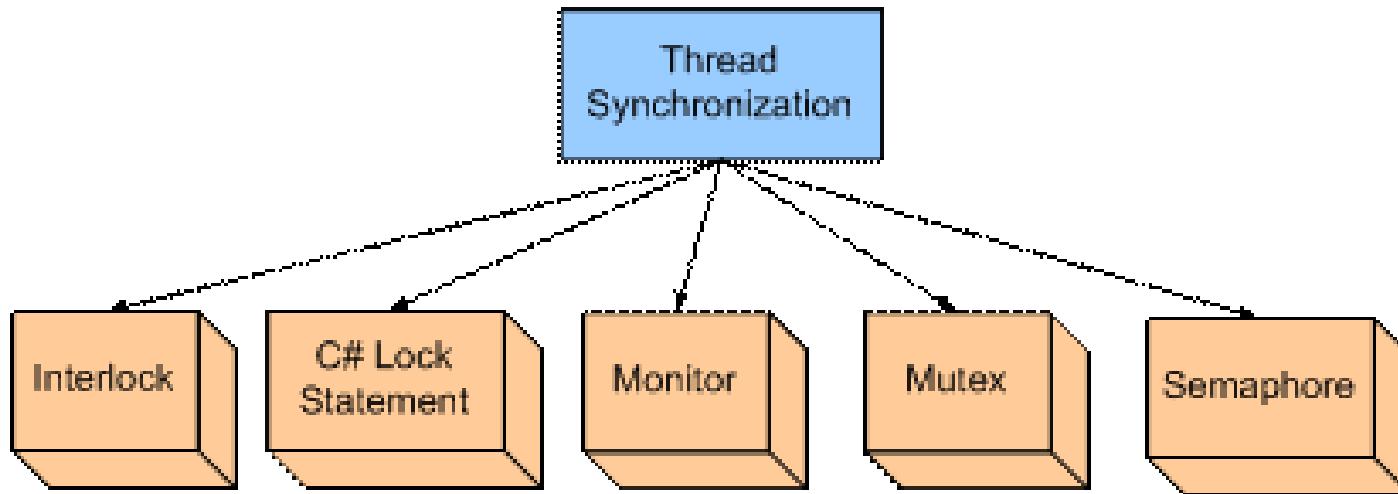
}

```
private void btnStart_Click(object sender, EventArgs e)
{
    th1 = new Thread(new ThreadStart(DegerArttir1));
    th2 = new Thread(new ThreadStart(DegerArttir2));
    th3 = new Thread(new ThreadStart(DegerArttir3));
    th1.Start();
    th2.Start();
    th3.Start();
}
```



Asenkron thread lerle doldurulan üç progressbar.

- .NET'te thread senkronizasyonu için dört yöntem kullanılabilir.



.NET'te thread senkronizasyon teknikleri.

- **Interlocked :**

Interlocked sınıfı birden fazla kanal tarafından kullanılan değişkenler üzerinde çeşitli işlemlerin yapılmasına olanak sağlayan bir sınıfır. Buradaki değişken değer tipinde olabileceği gibi herhangi bir nesne de olabilir. **Interlocked** sınıfının şimdilik sadece Exchange metodunu referans tipleri ile çalıştırır. Exchange'in generic metod olarak tasarlanmış bir overload'u da mevcuttur. Increment, decrement metodları ise Int32, Int64 tipi değişkenleri parametre olarak kabul eder. **Interlocked** sınıfını şöyle kullanabiliriz:

- public void DegerArttirInterlocked1()
{
 try
 {
 while (counter <= 300)
 {
 Interlocked.Increment(ref counter);
 progressBar1.Value += 1;
 lbThreads.Items.Add("Thread 1");
 Thread.Sleep(80);
 }
 }
 catch { }
}

- Burada counter global olarak tanımlanmış olan int değişkendir. Increment fonksiyonu ile counter değişkeni kontrollü arttırılmaktadır. Bir thread altında counter üzerinde işlem yapılırken diğer bir thread içerisinde counter üzerinde aynı anda işlem yapılmaz.

- **Lock :**

Bir thread içersindeki işlemlerin, diğer bir thread tarafından müdahale edilmeden çalışabilmesi için **lock** anahtar kelimesi kullanılır. **Lock** ile bloklanmış olan işlemler bir thread içerisinde tamamlanıncaya kadar çalışırlar. Bu süre zarfında başka thread **lock** ile bloklanmış işlemlerin bitmesini bekler. İşlem bitince diğer threadler kendi metodlarını çalıştırırmaya devam eder. Bu şekilde senkronizasyon sağlanabilir. **Lock** parametre olarak herhangi bir nesne alabilir. Örneğimizde form nesnemizi parametre olarak kullandık.

- public void DegerArttirLock1()
{
 for (int i = 1; i <= 100; ++i)
 {
 lock (this)
 {
 progressBar1.Value += 1;
 lbThreads.Items.Add("Thread 1");
 Thread.Sleep(10);
 }
 }
}

- Genellikle **lock**'a aktarılan parametrenin public olmamasına dikkat edilir. **Lock** ile kilitlenen nesne public ise bu nesne üzerinde işlem yapmak isteyen metodlar (bir thread'e bağlı olmayan) da kilitlenir. Bu tür kilitlenmelere deadlocks (kısır döngü) denir. Bu istenmeyen bir durumdur. Public yerine protected veya private kullanmak bu sorunu çözer. Şekil 'de DegerArttirLock metodlarını çağrıran kanallar çalıştırıldığında progressbar'ların ve kanalların durumu sergilenmiştir. Dikkate edilirse kanallar 123, 123 şeklinde art arda düzgün bir şekilde çağrılmaktadır. Asenkron kanallarda ise 132, 131, 111 şeklinde düzensiz çağrılmalar gerçekleşir.



Senkron thread leri doldurulan üç progressbar

- **Monitor :**

Monitor, nesnelere kanalların senkronize bir şekilde ulaşmasını sağlayan bir sınıfıdır. **Monitor** sınıfı referans tipindeki değişkenleri senkronize etmek için kullanılır. Değer tipindeki değişkenler için **monitor** sınıfı kullanılmaz. Monitor'ün kullanımı lock'un kullanımına benzer. **Monitor** blokladığı kodların başka kanallar tarafından erişilmesini engeller. **Monitor** sınıfının bazı metodları aşağıdaki tabloda özetlenmiştir.

Metod	Görevi
Enter, TryEnter	<p>Bloklanmak istenen nesne Enter ile Monitor nesnesine kayıt ettirilir. Diğer kanallar bu bloklanan nesneye müdahale edemez. Nesne artık kilitlenmiştir.</p> <p>Monitor.Enter ile bloklanmak istenen kodlar için lock'a benzer şekilde başlangıç verilebilir.</p>
Wait	<p>Bloklanmış nesne üzerindeki kilidi açar. Diğer kanallar nesneye ulaşabilir ve onu kilitleyebilir.</p>
Pulse, PulseAll	<p>Bir kanaldaki kilitlenmiş nesnenin, üzerindeki kilidin kaldırılacağını bekleyen kanallara duyurmak için kullanılır. Diğer kanallar kendi kuyruklarına kilidi açılabilecek olan nesneyi kayıt ederler. Pulse, Exit metodu öncesi kullanılan bir metottur ve ana amacı diğer kanallara kilidin açılacağını önceden duyurmaktır.</p>
Exit	<p>Nesne üzerindeki kilidi açar.</p> <p>Monitor.Exit ile bloklanmak istenen kodlar için lock'a benzer şekilde sonlanma noktası verilebilir.</p>

Monitor sınıfının bazı metodları

- **Monitor** sınıfının DegerArttir fonksiyonunda nasıl kullanılacağı aşağıda gösterilmiştir. Kısır döngüler (deadlock) ile karşılaşmamak için monitorObject nesnesinin üzerindeki kilit finally bloğunda herzaman kaldırılmaktadır.
- ```
private object monitorObject = new object();
public void DegerArttirMonitor1()
{
 for (int i = 1; i <= 100; ++i)
 {
 try
 {
 Monitor.Enter(monitorObject);
 progressBar1.Value += 1;
 lbThreads.Items.Add("Thread 1");
 Thread.Sleep(10);
 Monitor.Pulse(monitorObject);
 }
 finally
 {
 Monitor.Exit(monitorObject);
 }
 }
}
```

- **Mutex :**
- **Mutex** sınıfı monitor sınıfına benzer ancak System.Threading.WaitHandle sınıfından türetilmiştir ve türediğin sınıfın daha kolay kullanılabilir bir genelleştirilmesidir. **Mutex** sınıfı threadler tarafından ortak kullanılan nesnelere aynı anda ulaşılıp, işlem yapılmasını engellemek için kullanılır. **Mutex** sınıfı ortak kullanılan kaynaklara bir t zamanında sadece bir kanalın ulaşılmasını garanti eder. Mutex kendisini kullanan thread'in tekillliğini (identity) kontrol eder. Bir mutex'e sahip olan thread WaitOne metodu ile onu kilitler ve ReleaseMutex metodu ile mutex'i serbest bırakır. Bir **mutex** kullanan kanal sadece kendi mutex'ini ReleaseMutex metodu ile açabilir. Kanallar birbirlerinin mutex'lerini serbest bırakamaz. Bizim örneğimizde progressbar'ların tam anlamı ile aynı anda ilerlemediğini fark edeceksiniz. Burada mutex'i sadece ortak kaynaklara güvenli erişim amacı ile kullandık.

- private object mutexObject = new object();  
public void DegerArttirMutex1()  
{  
    for (int i = 1; i <= 100; ++i)  
    {  
        mutexObject.WaitOne();  
        progressBar1.Value += 1;  
        lbThreads.Items.Add(>"Thread 1");  
        Thread.Sleep(10);  
        mutexObject.ReleaseMutex();  
    }  
}

- **Semaphore** :
- **Semaphore** sınıfı .NET 2.0 framework ile gelen yeni bir sınıfır. Bu sınıf da System.Threading.WaitHandle sınıfından türetilmiştir. **Semaphore** sınıfının Mutex sınıfından farkı, farklı kanalların birbirlerinin Semaphore'larının kilitlerini Release metodu ile açabilmeleridir. Bir kanal semaphore'un WaitOne metodunu birçok kez çağırabilir. Bu kilitleri açmak için art arda Release metodunu çağrıabilecegi gibi, Release(int) overload'unu da kullanabilir. **Semaphore** kendisini kullanan kanalın identity'sine bakmaz. Bu yüzden farklı kanallar birbirlerinin semaphore'larının WaitOne ve Release metodlarını çağrıabilir. Herbir WaitOne metodu çağrııldığında semaphore'un sayacı bir azaltılır. Herbir release metodu çağrııldığında ise sayıç bir arttırılır. Semaphore'un yapılandırıcısında (constructor) sayacın minimum ve maksimum değerleri belirlenebilir

- ```
private Semaphore semaphoreObject = new Semaphore(0,2);
public void DegerArttirSemaphore1()
{
    for (int i = 1; i <= 100; ++i)
    {
        semaphoreObject.WaitOne();
        progressBar1.Value += 1;
        lbThreads.Items.Add("Thread 1");
        Thread.Sleep(10);
        semaphoreObject.Release();
    }
}

private void btnSemaphore_Click(object sender, EventArgs e)
{
    th1 = new Thread(new ThreadStart(DegerArttirSemaphore1));
    th2 = new Thread(new ThreadStart(DegerArttirSemaphore2));
    th3 = new Thread(new ThreadStart(DegerArttirSemaphore3));
    th1.Start();
    th2.Start();
    th3.Start();
    semaphoreObject.Release(2);
}
```

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- Thread'ler
 - Thread'lerin sağladığı faydalar
 - Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
 - Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
 - Thread kütüphaneleri
 - Dolaylı thread oluşturma
 - Thread çalışma kuralları
 - Windows ve Linux thread'leri

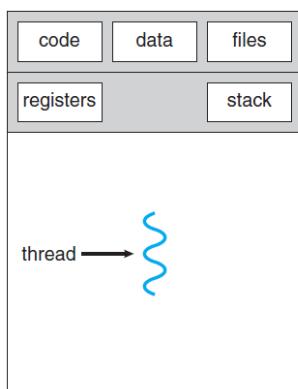
Thread'ler

- Bir **thread**, program counter, bir grup register ve bir stack yapısına sahiptir.
- Thread'ler, program kodunu, data kısmını, dosyalar gibi işletim sistemi kaynaklarını ortak kullanır.
- Klasik process'ler tek thread'e sahiptirler.
- Eğer bir process, birden fazla thread'e sahipse birden fazla görevi eşzamanlı yapabilir.
- Günümüzdeki modern bilgisayarlarda çalışan yazılım uygulamalarının çoğu multithread çalışmaları.
- Uygulamalar, çok sayıda thread'e sahip tek process şeklinde geliştirilirler.
- Bir Web browser, bir thread ile veri aktarımı yapabilir, başka thread ile verileri ekranда görüntüleyebilir.

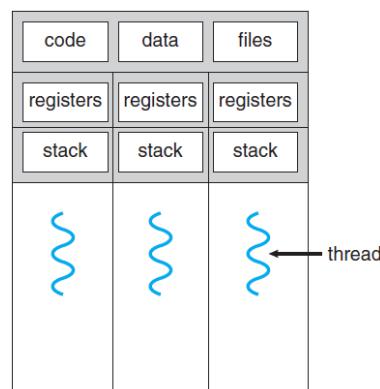
3

Thread'ler

- Bir kelime işlemci uygulaması, **bir thread ile klavyeden giriş alabilir, bir thread ile spell check yapabilir ve başka bir thread ile ekran görüntüsünü düzenleyebilir.**
- Her thread, **paylaşmadan kullandığı** kendisine ait bileşenlere sahiptir.



single-threaded process

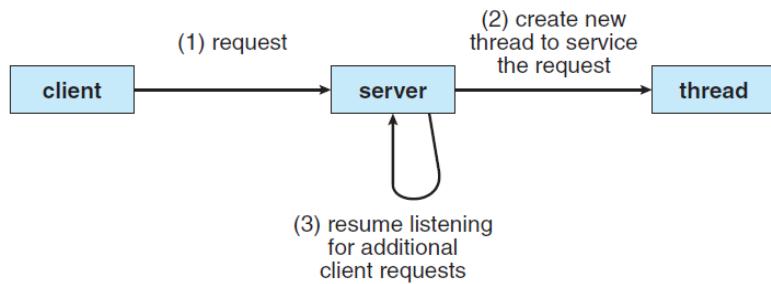


multithreaded process

4

Thread'ler

- Uygulamalar, multicore sistemlerin **kapasitesini maksimum kullanacak şekilde tasarlanabilir**.
- Bir Web sunucu process'i multithreaded çalışırsa, her gelen istek için ayrı bir thread oluşturulur ve process portu dinlemeye devam eder.
- **Çoğu işletim sisteminin kernel'ı multithreaded yapıdadır ve cihazların yönetimi, hafıza yönetimi veya interrupt işlemi aynı anda yapılabilir.**



5

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

6



Thread'lerin sağladığı faydalar

- Thread'lerin sağladığı faydalar 4 kategori halinde ifade edilebilir:
 1. **Responsiveness:** Kullanıcı etkileşimli uygulamalarda, bir kısım bloklanmış, kilitlenmiş veya uzun süren işlem yürütüyorSA, kullanıcı ile etkileşim yapan başka bir kısım çalışmasını sürdürür.
Sistemin cevap verebilirlik özelliği artmış olur.
 2. **Resource sharing:** Process'ler kaynaklarını shared memory veya message passing teknikleri aracılığıyla paylaşabilirler.
Thread'ler ait oldukları process'in sahip olduğu hafıza alanını ve diğer kaynakları paylaşabilirler.

7



Thread'lerin sağladığı faydalar

- Thread'lerin sağladığı faydalar 4 kategori halinde ifade edilebilir:
 3. **Economy:** Bir process oluştururken **hafıza ve kaynak tahsis edilmesi** maliyeti yüksek bir iştir.
Thread'ler ait oldukları process'in kaynaklarını paylaştıklarından dolayı context switch daha düşük maliyetle yapılır.
(Solaris işletim sisteminde, **thread oluşturma 30 kat daha hızlıdır** ve **thread'lerde context switch 5 kat daha hızlıdır.**)
 4. **Scalability:** **Çok işlemcili mimarilerde thread'ler farklı core'lar üzerinde eşzamanlı çalışabilir.**
Ancak, tek thread yapısına sahip process sadece bir işlemci üzerinde çalışabilir.

8

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- **Multicore programlama**
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

9

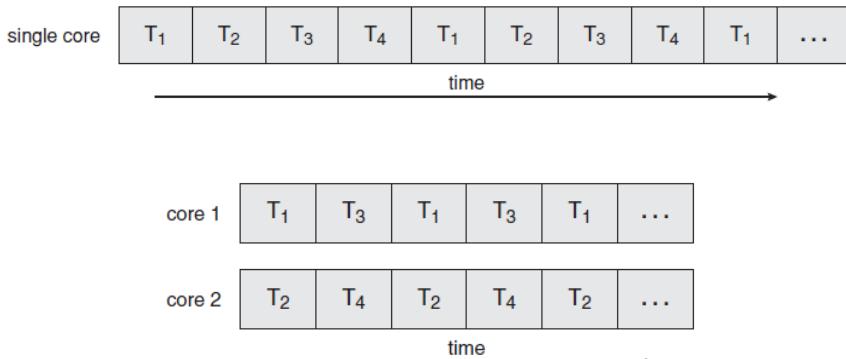
Multicore programlama

- Bilgisayar tasarımindaki en önemli gelişmelerden birisi, çok işlemcili sistemlerin geliştirilmesidir.
- Son zamanlarda, **tek chip içeresine birden fazla core yerleştirilmektedir**. Bu tür sistemler **multicore** veya **multiprocessor** olarak adlandırılır.
- **Her bir core işletim sistemi için ayrı bir işlemci olarak görünür**.
- **Bir core** üzerinde çalışan 4 thread'e sahip bir uygulama için **eşzamanlı çalışma**, **thread'lerin belirli aralıklarla çalıştırılmasını ifade eder**.
- **Çok core'a sahip sistemlerde eşzamanlı çalışma, her core'a bir thread atanarak thread'lerin paralel çalışmasını ifade eder**.
- **Parallelism**, birden fazla görevin **eşzamanlı** yapılmasını ifade eder.
- **Concurrency**, birden fazla görev arasında kısa aralıklarla geçiş yaparak **birlikte ilerletilmesini** ifade eder.

10

Multicore programlama

- Sistemdeki core sayısı arttıkça eşzamanlı gerçekleştirilen görev sayısı da artacaktır.



11

Multicore programlama

- Amdahl kuralı** core sayısına göre bir sistemdeki performans artışını aşağıdaki gibi ifade etmektedir:

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- Burada, S uygulamada seri çalışması zorunlu olan kısmın oranını, N ise core sayısını ifade eder.
- Bir uygulamada, %75 paralel ve %25 seri çalışıyorsa (S=0,25), **2 core'a** (N=2) sahip sistemde bu uygulamayı çalıştırınca **1,6 kat hız artar**.
- Core sayısı 4** olduğunda, **2.28 kat hız artışı sağlanır**.
- Core sayısı sonsuza giderken hız artışı (1/S)** 'e doğru gider.
- Intel CPU'lar** her core için 2 thread, **Oracle T4 CPU** ise 4 thread destekler.

12

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

13

Multicore programlamanın zorlukları

- **İşletim sistemi tasarımcıları** multicore sistemlerin performansını artırmak için **scheduling algoritmaları yazmak zorundadır.**
- Uygulama geliştiricilerin mevcut programları değiştirmeleri ve **yeni programları multihreaded şekilde tasarlamaları gerekmektedir.**
- Multicore programlamada 5 önemli zorluk vardır:
 - **Identifying tasks:** Uygulamalarda eşzamanlı çalışabilecek ayrı alanların bulunması gereklidir. Bu alanlar farklı core'lar üzerinde paralel çalışacaktır.
 - **Balance:** Programcılar görevleri ayırtırırken **iş yükünün eşit dağıtılması gereklidir.**
 - **Data splitting:** Verilerin farklı core'lar üzerinde çalışan görevler tarafından erişilecek ve işlem yapılacak şekilde ayrıştırılması gereklidir.
 - **Data dependency:** Bir görevin erişeceği verinin diğer görevlerle bağımlılığının incelenmesi gereklidir.
 - **Testing and debugging:** Multihreaded çalışan programların **test ve debug işlemi daha zordur.**

14

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - **Paralel çalışma türleri**
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

15

Paralel çalışma türleri

- Genel olarak **data parallelism** ve **task parallelism** olarak iki tür paralel çalışma türü vardır.
- **Data parallelism**, aynı **veri kümesine ait parçaların core'lara dağıtılması** ve aynı tür işlemin eşzamanlı yürütülmesine odaklanır.
- N elemanlı bir **dizinin toplamı** için **iki core** kullanılacaksa, **[0]..[(N/2)-1]** eleman 1.core'da, **[N/2]..[N-1]** eleman 2.core'da toplanır.
- **Task parallelism**, core'lara **görevlerin (thread'ler) dağıtımasına** odaklanır.
- Her thread **ayıri bir işlemi gerçekleştirir**. Farklı thread'ler aynı veride veya farklı veride çalışabilir.
- **Aynı dizi** elemanları üzerinde **farklı istatistiksel hesaplamalar** yapan thread'ler aynı veriyi kullanır farklı core'larda çalışır.

16

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- **Multithreading modelleri**
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

17

Multithreading modelleri

- Thread desteği kullanıcı seviyesinde **user thread'**ler için veya kernel seviyesinde **kernel thread'**ler için sağlanabilir.
- **User thread'leri kullanıcı uygulamaları tarafından, kernel thread'leri ise işletim sistemi tarafından gerçekleştirilir.**
- Windows, Linux, Unix, Mac OS X ve Solaris gibi işletim sistemleri **kernel thread'leri destekler**.
- Kernel thread'leri ile user thread'leri arasında aşağıdaki **ilişkilendirme modellerinden** birisinin oluşturulması zorundadır.
 - Many-to-one model
 - One-to-one model
 - Many-to-many model

18

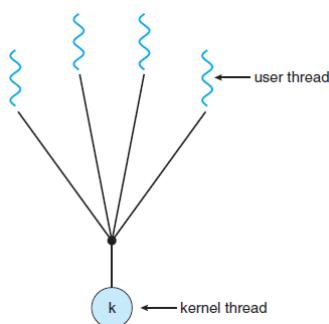
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

19

Many-to-one

- Many-to-one modelinde, **çok sayıda kullanıcı thread'i bir tane kernel thread'i ile eşleştirilir (Solaris işletim sistemi kullanır.)**.



- Eğer **bir thread sistem çağrısını bloklarsa** tüm process bloklanmış olur.
- Aynı anda sadece bir tane kullanıcı thread'i kernel thread'e erişebilir.
- Sadece bir kernel thread'i kullanıldığı için **multicore sistemlerde birden fazla thread için eşzamanlı çalışma yapılamaz**.

20

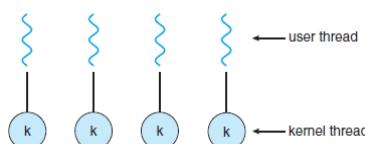
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

21

One-to-one

- One-to-one modelinde, **bir kullanıcı thread'i bir kernel thread'i ile eşleştirilir (Linux, Windows işletim sistemleri kullanır.)**.



- Eğer **bir thread sistem çağrımasını bloklarsa** diğer thread'ler çalışmasına devam eder.
- Birden fazla kernel thread'inin **multicore sistemlerde eşzamanlı çalışmasına izin verir**.
- **Bir user thread için bir kernel thread oluşturulması gereklidir.**

22

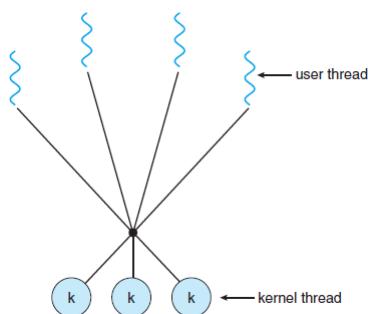
Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - **Many-to-many**
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

23

Many-to-many

- Many-to-many modelinde, **çok sayıda kullanıcı thread'i ile aynı sayıdaki veya daha az sayıdaki kernel thread'i eşleştirilir (Solaris 9, Unix işletim sistemleri kullanır.)**.



- **Bir thread sistem çağrısını bloklarsa, kernel başka bir thread'i çalıştırır.**

24

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- **Thread kütüphaneleri**
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- Windows ve Linux thread'leri

25

Thread kütüphaneleri

- Thread kütüphanesi, **programcıya thread oluşturmak ve yönetmek için API sağlar.**
- Thread kütüphanesi oluşturulurken **iki farklı yaklaşım kullanılır:**
 - Tüm thread kütüphanesi **kullanıcı alanında** oluşturulur ve **kernel desteği yoktur.**
 - İşletim sisteminin doğrudan desteklediği **kernel seviyesinde kütüphane oluşturulur.**
- Çoklu thread oluşturmak için iki farklı strateji kullanılmaktadır:
 - **Asenkron threading:** Parent, yeni bir child thread oluşturduğunda **eşzamanlı olarak çalışmasını sürdürür.**
 - **Senkron threading:** Parent, child process oluşturduğunda **çalışmasını durdurur** ve tüm child process'ler sonlandığında çalışmasına devam eder (**fork-join strategy**).
- **Asenkron threading**, thread'ler arasında **veri paylaşımı az olduğunda**, **senkron threading** ise threadler arasında **veri paylaşımı çok olduğunda** **kullanılır.**

26



Thread kütüphaneleri

- Günümüzde 3 temel thread kütüphanesi kullanılmaktadır:
 - POSIX Pthreads
 - Windows
 - Java
- **Pthreads**, user-level veya kernel-level thread kütüphanesi sağlar.
- **Windows threads**, kernel-level thread kütüphanesi sağlar.
- **Java threads**, user-level thread kütüphanesi sağlar.

27



Thread kütüphaneleri

Pthreads

- **Pthreads, IEEE1003.1c standarıyla thread oluşturma ve yönetmek için tanımlanan API'dir.**
- **Linux, Unix, Mac OS X ve Solaris** işletim sistemleri Pthreads standartını kullanır.
- **Windows** Pthreads standartını **desteklemez**.
- **Pthreads standartında thread'lerin hepsi ayrı fonksiyonlar halinde oluşturulur.**
- **Tüm thread'ler global scope'ta tanımlanan verileri paylaşırlar.**

28

Thread kütüphaneleri

Pthreads - Örnek

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param),
        sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```

Pthreads programları için kullanılan header file

Yeni thread için ID tanımlar.

Yeni thread için özellikleri (stack size, ...) belirler.

Varsayılan özellikler (senkron thread, sistem stack addr, ...)

Yeni thread başlatıldı.

Yeni thread için başlama noktası.

Komut satırında girilen parametre

Komut satırında girilen parametre

fork-join stratejisi

Dönen değer

Thread kütüphaneleri

Pthreads - Örnek

- Önceki örnekte bir thread oluşturulmuştur. Çok sayıda thread aşağıdaki örnekteki gibi oluşturulabilir.

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Oluşturulacak thread sayısı

10 thread tanımlandı.

10 thread için fork-join yapıldı.

Thread kütüphaneleri

Windows threads

- Windows thread kütüphanesi ile thread oluşturma Pthreads ile birçok açıdan benzerlik gösterir.
- **Thread'lerin hepsi ayrı fonksiyonlar halinde oluşturulur.**
- **Tüm thread'ler global scope'ta tanımlanan verileri paylaşırlar.**

31

Thread kütüphaneleri

Windows threads - Örnek

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }
}
```

* create the thread */
ThreadHandle = CreateThread(
 NULL, /* default security attributes */
 0, /* default stack size */
 Summation, /* thread function */
 &Param, /* parameter to thread function */
 0, /* default creation flags */
 &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
 /* now wait for the thread to finish */
 WaitForSingleObject(ThreadHandle, INFINITE);
}
/* close the thread handle */
CloseHandle(ThreadHandle);
printf("sum = %d\n",Sum);

fork-join stratejisi
WaitForMultipleObjects() ile birden fazla
thread senkron çalıştırılabilir.

32

Thread kütüphaneleri

Java threads

- Java thread'leri, JVM (Java Virtual Machine) kullanılabilen tüm sistemlerde çalışır.
- Java thread API, Windows, Linux, Unix, Mac OS X ve Android için kullanılabilir.
- Java thread'leri arasında **veri paylaşımı parameter passing** ile yapılır.
- Java ile iki farklı teknik kullanılarak thread oluşturulabilir:
 - Thread sınıfından yeni bir sınıf türetilir ve run() metodu override yapılır.
 - Runnable arayüzüünü kullanan bir sınıf oluşturulur. (yaygın kullanılır.)

```
public interface Runnable  
{  
    public abstract void run();  
}
```

Aynı thread için override yapmak gereklidir.

33

Thread kütüphaneleri

Java threads – Örnek

```
class Sum  
{  
    private int sum;  
  
    public int getSum() {  
        return sum;  
    }  
  
    public void setSum(int sum) {  
        this.sum = sum;  
    }  
}  
  
class Summation implements Runnable  
{  
    private int upper;  
    private Sum sumValue;  
  
    public Summation(int upper, Sum sumValue) {  
        this.upper = upper;  
        this.sumValue = sumValue;  
    }  
  
    public void run() {  
        int sum = 0;  
        for (int i = 0; i <= upper; i++)  
            sum += i;  
        sumValue.setSum(sum);  
    }  
}  
  
public class Driver  
{  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            if (Integer.parseInt(args[0]) < 0)  
                System.err.println(args[0] + " must be >= 0.");  
            else {  
                Sum sumObject = new Sum();  
                int upper = Integer.parseInt(args[0]);  
                Thread thrd = new Thread(new Summation(upper, sumObject));  
                thrd.start();  
                try {  
                    thrd.join();  
                    System.out.println("The sum of "+upper+" is "+sumObject.getSum());  
                } catch (InterruptedException ie) {}  
            }  
        } else  
            System.err.println("Usage: Summation <integer value>");  
    }  
}
```

Aynı thread için gereklidir.

Yeni thread oluşturuldu.

Yeni thread başlıyor.
run() çağırılır.

fork-join stratejisi
Thread'ler senkron çalışıyor.

34

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- **Dolaylı thread oluşturma**
- Thread çalışma kuralları
- Windows ve Linux thread'leri

35

Dolaylı thread oluşturma

- Multicore işlemcilerdeki gelişmelerle birlikte, uygulamalar yüzlerce hatta binlerce thread içermektedirler.
- Çok sayıda thread ile uygulama geliştirmek oldukça zordur ve hata olma olasılığı vardır.
- Thread oluşturma işinin **uygulama geliştiriciler yerine, compiler tarafından yapılması** günümüzde giderek popüler hale gelmektedir.
- Bu stratejiye **implicit threading** denilmektedir.

36

Dolaylı thread oluşturma

Thread pools

- Multithreaded bir Web sunucusu, gelen isteklerin her birisi için yeni thread oluşturur.
- Multihreaded bir Web sunucusu, eşzamanlı çok sayıda istemciye servis sağlayabilir.
- **Gelen istek sayısı çok artarsa** sistem kaynakları (CPU time, memory, ...) tükenir.
- **Multithreaded sistemlerde belirli sayıda thread oluşturulmasına izin vermek için thread pool oluşturulur.**
- Yeni istek geldiğinde thread pool içerisinde kullanılabilir **thread varsa cevaplanır**, yoksa bir thread'in serbest hale gelmesi beklenir.
- **Varolan thread'in kullanımı yeni thread oluşturmaya göre daha hızlıdır.**

37

Dolaylı thread oluşturma

Thread pools - Windows

- Örnekte `PoolFunction()` fonksiyonu thread olarak çalıştırılmaktadır.

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```

- Thread pool API içerisindeki `QueueUserWorkItem()` fonksiyonu, **pool içerisindeki bir thread ile PoolFunction() fonksiyonunu çalıştırır.**

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

- **Parametreler:**

- LPTHREAD_START_ROUTINE, thread olarak çalışacak fonksiyonun pointer'i
- PVOID, Gönderilecek parametre
- ULONG, Bayrak bitleri (bekleme süresi, I/O gerekliliği, ...)

38

Dolaylı thread oluşturma

OpenMP

- OpenMP, C, C++ ve Fortran için yazılmış bir grup compiler direktifidir.
- Shared memory yaklaşımını kullanılır.
- **OpenMP ile paralel çalışacak kod blokları tanımlanır.**
- OpenMP ile `#pragma omp parallel` direktifi paralel çalışacak bloğun hemen başında kullanılır.
- OpenMP farklı türdeki deyimler için ayrı direktifler kullanır.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

- OpenMP, Linux, Windows ve Mac OS X sistemlerdeki çok sayıda açık kaynak ve ticari compiler'larda kullanılabilir.

39

Dolaylı thread oluşturma

OpenMP - Örnek

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```

printf() deyimi paralel çalışacaktır.

40

Dolaylı thread oluşturma

Grand central dispatch

- Grand central dispatch (GCD), **Apple Mac OS X ve iOS işletim sistemlerinde paralel çalışacak kısımları belirlemek için kullanılır.**
- Paralel çalışacak bloğun belirtilmesi için ^ simbolü kullanılır.

```
^{ printf("I am a block"); }
```
- GCD blokları **dispatch queue** içerisinde yerleştirir.
- Bir blok kuyruktan atılırsa, tekrar thread havuzundaki bir thread'e atanabilir.
- Dispatch queue, **serial** veya **concurrent** şeklinde oluşturulabilir.
- **Serial queue**, FIFO çalışır ve **sadece bir blok kuyruktan alınabilir**.
- **Concurrent queue**, FIFO çalışır ve kuyruktan **birden fazla blok aynı anda alınabilir**.

41

Dolaylı thread oluşturma

Grand central dispatch

- Concurrent queue, önceliklendirilmiş 3 tane dispatch kuyruğa sahiptir: **low**, **default** ve **high**.
- Önceliklendirme ile blokların önem derecesi belirlenmektedir.
- Aşağıdaki örnekte, default önceliğe sahip concurrent kuyruğa bir blok eklenmektedir.

```
dispatch_queue_t queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```

Kuyruğa blok eklendi.

default öncelikli concurrent kuyruk

42

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- **Thread çalışma kuralları**
- Windows ve Linux thread'leri

43

Thread çalışma kuralları

fork() ve exec() sistem çağrıları

- **Bazı Unix sistemlerde, iki tür fork() çağrısı vardır.**
- Birisi ile **tüm thread'ler duplicate yapılır**, diğeri ile **sadece fork() ile başlatılan thread duplicate yapılır**.
- **exec()** sistem çağrısı ile **tüm thread'leri ile birlikte process duplicate yapılır**.

44

Thread çalışma kuralları

Signal handling

- Unix sistemlerde bir **signal** belirli bir olayın gerçekleştiğini gösterir.
- **Oluşan olaya karşılık gelen sinyal bir process'e iletilir.**
- Oluşan sinyal, **senkron** veya **asenkron** alınabilir.
- **Senkron sinyal, sinyalin oluşmasına neden olan olayı gerçekleştiren process'e iletilir.**
- Senkron sinyale illegal hafıza erişimi veya 0'a bölme verilebilir.
- **Asenkron sinyal, sinyali oluşturan process'ten başka bir process'e iletilir.**
- Asenkron sinyale <ctrl><C> tuşlarına birlikte basmak verilebilir.

45

Thread çalışma kuralları

Signal handling

- İşletim sistemlerinde **sinyaller farklı hedeflere gönderebilir**:
 - Process içerisindeki **sadece bir thread'e gönderebilir** (Senkron).
 - Process içerisindeki **tüm thread'lere gönderebilir** <ctrl><C>.
 - Process içerisindeki **bazı thread'lere gönderebilir**.
 - Bir process için **tüm sinyalleri almak üzere bir thread atanabilir**.
- **Senkron sinyaller** sadece **oluşturan thread'e** gönderilir.
- Aşağıdaki UNİX fonksiyonu ile ID değeri verilen process'e iletilir.

```
kill(pid_t pid, int signal)
```
- POSIX Pthreads ile aşağıdaki fonksiyon kullanılır.

```
pthread_kill(pthread_t tid, int signal)
```

45

Thread çalışma kuralları

Thread iptal etme

- Bir thread'in çalışması tamamlanmadan iptal edilebilir.
- İstenen bir sonucun bir thread tarafından bulunması halinde diğerleri iptal edilebilir.
- Bir Web sayfası yüklenirken stop butonuna basıldığında process içerisindeki tüm thread'ler iptal edilir.
- Bir thread başka bir thread'i aniden sonlandırabilir (**Asenkron cancellation**).
- Bir thread başka bir thread'in kendi kendisini sonlandırmasını sağlayabilir (**Deferred cancellation**).

47

Thread çalışma kuralları

Thread iptal etme

- Pthreads aşağıdaki tabloda verilen üç farklı iptal etme modunu destekler.

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
...  
  
/* cancel the thread */  
pthread_cancel(tid);
```

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

48

Konular

- Thread'ler
- Thread'lerin sağladığı faydalar
- Multicore programlama
 - Multicore programlamanın zorlukları
 - Paralel çalışma türleri
- Multithreading modelleri
 - Many-to-one
 - One-to-one
 - Many-to-many
- Thread kütüphaneleri
- Dolaylı thread oluşturma
- Thread çalışma kuralları
- **Windows ve Linux thread'leri**

49

Windows ve Linux thread'leri

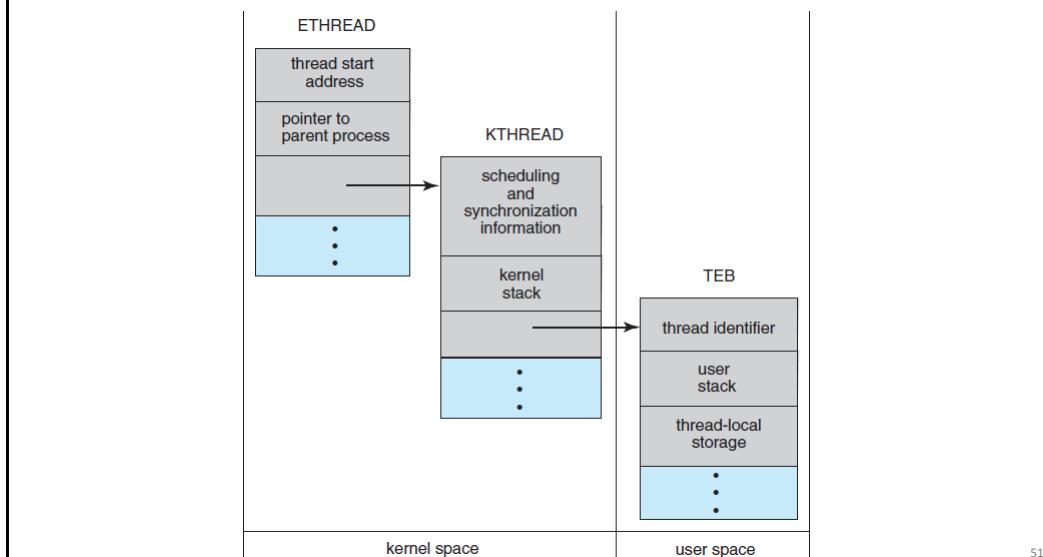
Windows thread'leri

- Microsoft işletim sistemleri için temel API olarak **Windows API** kullanılır.
- Bir Windows uygulaması ayrı bir process olarak çalışır ve **her process bir veya daha fazla thread içerebilir.**
- **Windows, kullanıcı thread'leri ile kernel thread'leri arasında one-to-one eşleştirme yapar.**
- Bir thread için ayrılan **register kümesi, stack, depolama alanı, context** olarak adlandırılır.
- Windows bir thread için aşağıdaki veri yapılarını kullanır:
 - **ETHREAD:** Yürüttü thread blok
 - **KTHREAD:** Kernel thread blok
 - **TEB:** Thread environment (ortam) blok

50

Windows ve Linux thread'leri

Windows thread'leri



51

Windows ve Linux thread'leri

Linux thread'leri

- Linux, `fork()` sistem çağrısının yanı sıra `clone()` sistem çağrıları ile thread oluşturabilir.
- Linux, `clone()` ile yeni bir görev başlattığında, parent task ile child task arasında paylaşım miktarını da gönderir.
- **Dosya sistemi, hafıza aralığı, sinyaller veya açık olan dosyalar paylaşılabilir.**
- Görevler, Linux kernel içerisinde bir veri yapısına sahiptir (`struct task_struct`) ve açık dosyalar, virtual memory ve sinyal bilgilerini gösterir.
- Linux, `fork()` ile **yeni bir görev başlattığında, parent task veri yapısı kopyalanır.**

52

Windows ve Linux thread'leri

Linux thread'leri

- Linux, clone() ile yeni bir görev başlattığında bayrak bitlerine göre veri yapısı oluşturulur.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- Çoklu process veya scheduling
- Gerçek zamanlı CPU scheduling

Temel kavramlar

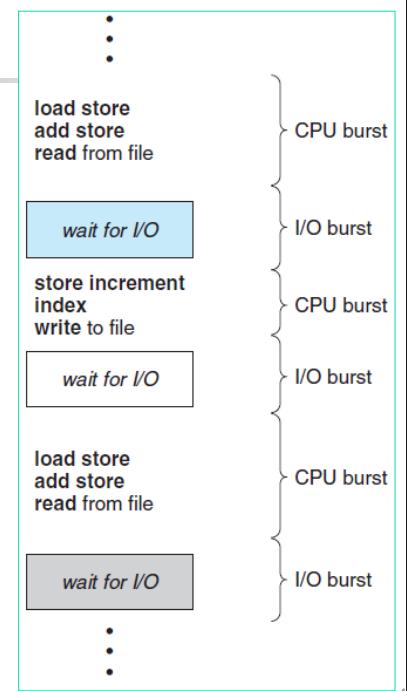
- CPU scheduling (planlama), multiprogramming çalışan işletim sistemlerinin temelini oluşturur.
- CPU, process'ler arasında geçiş yaparak bilgisayarı daha verimli hale getirir.
- **Her zaman aralığında bir process'in çalıştırılması amaçlanır.**
- Tek işlemcili sistemlerde, bir anda sadece bir process çalıştırılabilir.
- CPU, process'lerde ortaya çıkacak bekleme durumlarında başka process'leri çalıştırır.
- Hafızada çok sayıda process bulundurulur.
- Bir process herhangi bir şekilde beklemeye geçtiğinde CPU başka bir process'e geçiş yapar.
- Bilgisayardaki **tüm kaynaklar** kullanılmadan önce zamana göre **planlanır**.

3

Temel kavramlar

CPU Burst Cycle ve I/O Burst Cycle

- Process çalışma, **CPU execution** ve **I/O wait** döngüsünü içermektedir.
- Process'ler bu iki durum arasında geçiş yaparlar.
- Process'ler çalışmaya **CPU burst** ile başlarlar ve **I/O burst** ile devam ederler.

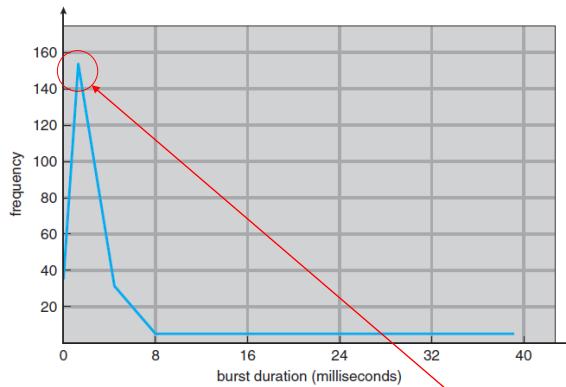


4

Temel kavramlar

CPU Burst Cycle ve I/O Burst Cycle

- CPU burst süresi, process'ten process'e ve bilgisayardan bilgisayara çok farklı olabilmektedir.



- Process'ler için CPU burst süresi sıklıkla kısa olmaktadır.
- Process'ler kısa aralıklarla durumunu değiştirmektedir.

5

Temel kavramlar

CPU Schedulers

- CPU bekleme durumuna **geçtiğinde**, işletim sistemi hazır kuyruğundan (**ready queue**) bir process'i çalıştırılmak üzere seçmek zorundadır.
- Bu seçme işlemi kısa dönem planlayıcı (**short-term scheduler** veya **CPU scheduler**) tarafından gerçekleştirilir.
- Hazır kuyruğu, ilk gelen ilk çıkar (**first-in-first-out, FIFO**) olmak zorunda değildir.
- Hazır kuyruğu, **FIFO**, **priority queue**, **ağaç**, **sırasız bağlı liste** şeklinde oluşturulabilir.
- Hazır kuyruğunda bekleyen tüm process'lerin CPU tarafından çalıştırılmak üzere seçilme olasılıkları vardır.
- Kuyruk içindeki kaytlarda, **process control block (PCB)** tutulur.

6

Temel kavramlar

Preemptive Scheduling

- CPU-scheduling kararı 4 durum altında gerçekleştirilir:
 1. Bir process **çalışma** durumundan **bekleme** durumuna geçtiğinde (I/O isteği),
 2. Bir process **çalışma** durumundan **hazır** durumuna geçtiğinde (interrupt),
 3. Bir process **bekleme** durumundan **hazır** durumuna geçtiğinde (I/O tamamlanması),
 4. Bir process'in **sonlandırıldıgendı.**
- Eğer scheduling işlemi **1. ve 4. durumlarda gerçekleşmişse**, buna **nonpreemptive** veya **cooperative** scheduling denir.
- **2. ve 3. durumlarda gerçekleşmişse preemptive** scheduling denir.

7

Temel kavramlar

Preemptive Scheduling

- **Nonpreemptive scheduling'te**, CPU bir process'e tahsis edilmişse, bu process **sonlandırılıncaya kadar**, **CPU'yu serbest bırakıncaya kadar** veya **bekler durumuna geçinceye kadar** tutar.
- Windows 3.1, nonpreemptive scheduling kullanmıştır.
- Diğer tüm Windows versiyonları preemptive scheduling kullanmıştır.
- Mac OS X işletim sistemi de preemptive scheduling kullanmaktadır.
- **Preemptive scheduling veri paylaşımı yaptığında race condition** gerçekleşir.
- **Bir process kernel verisi üzerinde değişiklik yaparken** yanında kesilerek **başka bir process'e geçilmesi** ve aynı veriye erişim yapılması **halinde çakışma** meydana gelir.

8

Temel kavramlar

Dispatcher

- CPU scheduling işlevini gerçekleştiren bileşen **dispatcher** olarak adlandırılır.
- **Dispatcher, short-term scheduler tarafından CPU'ya atanacak process'i seçer.**
- Dispatcher aşağıdaki işlevleri içermektedir:
 - Context geçisi
 - Kullanıcı moduna geçiş
 - Programı yeniden başlatmak için kullanıcı programında uygun konuma atlama
- Dispatcher'ın çok hızlı bir şekilde geçiş yapması zorunludur.
- Process'ler arasında **geçiş süresine dispatch latency** denilmektedir.

9

Konular

- Temel kavramlar
- **Scheduling kriterleri**
- Scheduling algoritmaları
- Çoklu process veya scheduling
- Gerçek zamanlı CPU scheduling

10

Scheduling kriterleri

- CPU scheduling algoritmaları çok sayıda farklı kritere göre karşılaştırılır:
 - **CPU utilization:** CPU'nun olabildiği kadar kullanımında olması istenir. CPU kullanım oranı %0 - %100 arasındadır. Gerçek sistemlerde bu oran %40 ile %90 arasındadır.
 - **Throughput:** Her zaman aralığında tamamlanan process sayısıdır.
 - **Turnaround time:** Bir process'in hafızaya alınmak için bekleme süresi, hazır kuyruğunda bekleme süresi, CPU'da çalıştırılması ve I/O işlemi yapması için geçen sürelerin toplamıdır.
 - **Waiting time:** Bir process'in hazır kuyruğunda beklediği süredir.
 - **Response time:** Bir process'e gönderilen isteğe cevap dönünçeye kadar geçen süredir.
- **CPU utilization'ı ve throughput'u maksimum, turnaround time, waiting time ve response time'ı minimum yapmak** amaçlanır.
- Genellikle ortalama değerler optimize edilmeye çalışılır.

11

Konular

- Temel kavramlar
- Scheduling kriterleri
- **Scheduling algoritmaları**
- Çoklu process veya scheduling
- Gerçek zamanlı CPU scheduling

12

Scheduling algoritmaları

- CPU scheduling algoritmaları, **hazır kuyruğunda bekleyen process'lerden hangisinin CPU'ya atanacağını belirlerler.**
 - First-Come, First-Served Scheduling
 - Shortest-Job-First Scheduling
 - Priority Scheduling
 - Round-Robin Scheduling
 - Multilevel Queue Scheduling
 - Multilevel Feedback Queue Scheduling

13

Scheduling algoritmaları

First-Come, First-Served Scheduling

- En basit CPU scheduling algoritmasıdır ve **first-come first served (FCFS)** şeklinde çalışır.
- CPU'ya ilk istek yapan process, CPU'ya ilk atanmış process'tir.
- **FIFO kuyruk yapısıyla yönetilebilir.**
- FCFS algoritmasıyla **ortalama bekleme süresi genellikle yüksektir.**
- **Bekleme süreleri process'lerin kuyruğa geliş sırasına göre çok değişmektedir.**

14

Scheduling algoritmaları

First-Come, First-Served Scheduling

- Aşağıdaki 3 process için CPU'da çalışma süreleri ms olarak verilmiştir.

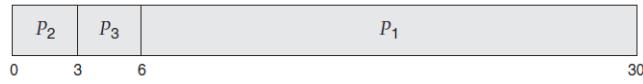
Process	Burst Time
P_1	24
P_2	3
P_3	3

- Process'ler P_1 , P_2 , P_3 sırasıyla gelirse Gantt şeması aşağıdaki gibidir.



- Ortalama bekleme süresi $(0+ 24 + 27) / 3 = 17 \text{ ms}$ olur.

- P_2 , P_1 , P_3 sırasıyla gelirse Gantt şeması aşağıdaki gibidir.



- Ortalama bekleme süresi $(0+ 3 + 6) / 3 = 3 \text{ ms}$ olur.

15

Scheduling algoritmaları

First-Come, First-Served Scheduling

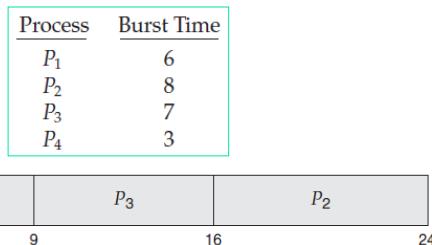
- FCFS algoritmasında, process'lerin çalışma süreleri çok farklıysa ortalama bekleme süreleri çok değişken olur.
- Bir CPU-bound process ile çok sayıda I/O bound process varsa, CPU-bound process CPU'da çalışırken tüm I/O bound process'ler hazır kuyruğunda bekler, I/O cihazları boş kalır.
- Çok sayıda küçük process'in büyük bir process'in CPU'yu terketmesini beklemesine **convoy effect** denilmektedir.
- Bir process'e CPU tahsis edildiğinde sonlanana veya I/O isteği yapana kadar CPU'yu elinde tutar.
- FCFS algoritması belirli zaman aralıklarıyla CPU'yu paylaşan time-sharing sistemler için uygun değildir.

16

Scheduling algoritmaları

Shortest-Job-First Scheduling

- Shortest-Job-First Scheduling (SJF) algoritmasında, CPU'ya bir sonraki işlem süresi en kısa olan (shortest-next-CPU-burst) process atanır.



- Ortalama bekleme süresi, $(0 + 3 + 9 + 16) / 4 = 7$ ms'dir. FCFS kullanılsaydı 10,25 ms olurdu $((0 + 6 + 14 + 21) / 4)$.
- SJF algoritması minimum ortalama bekleme süresini elde eder.

17

Scheduling algoritmaları

Shortest-Job-First Scheduling

- SJF algoritmasındaki en büyük zorluk, sonraki çalışma süresini tahmin etmektir.
- Long-term (job) scheduling için kullanıcının belirlediği süre alınabilir.
- SJF algoritması genellikle long-term scheduling için kullanılır.
- SJF algoritması short-term scheduling seviyesinde kullanılamaz.
- Short-term scheduling'te CPU'da sonraki çalışma süresini bilmek mümkün değildir.
- Short-term scheduling'te sonraki çalışma süresi tahmin edilmeye çalışılır.
- Sonraki çalışma süresinin önceki çalışma süresine benzer olacağı beklenir.

18

Scheduling algoritmaları

Shortest-Job-First Scheduling

- Sonraki CPU burst süresi, **önceki** CPU burst sürelerinin **üstel ortalama (exponential average)** değeri olarak tahmin edilir.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

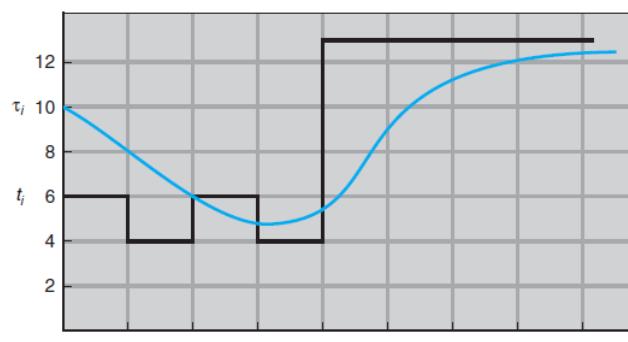
- Burada, $0 < \alpha < 1$ (genellikle $1/2$), τ_{n+1} sonraki tahmin edilen süreyi, t_n ise n . CPU burst süresini gösterir.
- SJF algoritması preemptive veya nonpreemptive olabilir.
- Çalışmakta olan process'ten daha kısa süreye sahip yeni bir process geldiğinde, preemptive SJF çalışmakta olanı keser, nonpreemptive SJF çalışmakta olanın sonlanmasına izin verir.**
- Preemptive SJF, **shortest-remaining-time-first scheduling** olarak adlandırılır.

19

Scheduling algoritmaları

Shortest-Job-First Scheduling

- $\alpha = 1/2$ ve $\tau_0 = 10$ için exponential average görülmektedir.



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

20

Scheduling algoritmaları

Priority Scheduling

- Shortest-job-first (SJF) algoritması, priority scheduling algoritmalarının özel bir durumudur.
- CPU en yüksek önceliğe sahip process'e atanır.
- Eşit önceliğe sahip olanlar ise FCFS sırasıyla atanır.
- SJF algoritması tahmin edilen CPU-burst süresine göre önceliklendirme yapar.
- SJF algoritmasında, CPU burst süresi azaldıkça öncelik artar, CPU burst süresi arttıkça öncelik azalır.

21

Scheduling algoritmaları

Priority Scheduling

- Aşağıda 5 process için öncelik değerine göre gantt şeması verilmiştir.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



- Ortalama bekleme süresi $(1+ 6 + 16 + 18) / 4 = 8,2 \text{ ms}$ olur.

22

Scheduling algoritmaları

Priority Scheduling

- Önceliklendirme kriterleri aşağıdakilerden bir veya birkaç tanesi olabilir:
 - Zaman sınırı
 - Hafıza gereksinimi
 - Açılan dosya sayısı
 - I/O burst ve CPU burst oranı
 - Process'in önemi
- Priority scheduling preemptive veya nonpreemptive olabilir.
- **Preemptive yönteminde**, bir process hazır kuyruğuna geldiğinde, çalışmakta olan process'ten daha öncelikli ise, **çalışmakta olan kesilir**.
- **Nonpreemptive yönteminde**, bir process hazır kuyruğuna geldiğinde, çalışmakta olan process'ten daha öncelikli bile olsa, **çalışmakta olan durum değiştirene kadar devam eder**.

23

Scheduling algoritmaları

Priority Scheduling

- Priority scheduling algoritmasında, **CPU sürekli yüksek öncelikli process'leri çalıştırabilir** ve bazı processler sürekli hazır kuyruğunda bekleyebilir (**indefinite blocking, starvation**).
- **Sınırsız beklemeyi engellemek için düşük öncelikli process'ler kuyrukta beklerken öncelik seviyesi artırılır** (Örn. her 15 dakikada 1 artırılır).
- Öncelik değeri artırılarak **en düşük önceliğe sahip process'in** bile belirli bir süre sonunda çalışması sağlanır.

24

Scheduling algoritmaları

Round-Robin Scheduling

- Round-robin (RR) scheduling, genellikle time-sharing sistemlerde kullanılır.
- Hazır kuyruğundaki process'ler belirli bir zaman aralığında (time slice) CPU'ya sıralı atanır.
- Zaman aralığı genellikle 10 ms ile 100 ms aralığında seçilir.
- Time slice aralığından daha kısa sürede sonlanan process CPU'yu serbest bırakır.
- Round-robin scheduling ile ortalama bekleme süresi genellikle uzundur.

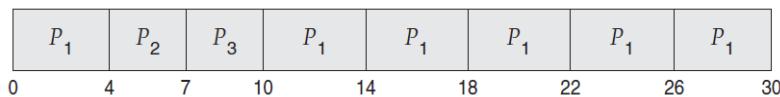
25

Scheduling algoritmaları

Round-Robin Scheduling

- Aşağıda 3 process için CPU-burst time ve gantt şeması verilmiştir.
- Örnekte **time slice = 4 ms** olarak alınmıştır.

Process	Burst Time
P_1	24
P_2	3
P_3	3



- P_1 için $= 10 - 4 = 6$, P_2 için 4, P_3 için 7 ms bekleme süresi vardır.
- Ortalama bekleme süresi ise $17 / 3 = 5,66$ ms'dir.
- q time slice süresiyle n process çalışan sistemde, bir process için en fazla bekleme süresi $(n - 1) * q$ olur.

26

Scheduling algoritmaları

Round-Robin Scheduling

- Time slice süresi **çok büyük olursa** çalışma FCFS yöntemine benzer.
- Time slice süresi **çok küçük olursa context switch işlemi çok fazla yapılır.**
- Context switch süresi overhead olur ve çok fazla context switch yapılması istenmez.
- **Time slice süresi, context switch süresinin genellikle 10 katı alınır.**
- CPU'nun %10 süresi context switch için harcanır.

27

Scheduling algoritmaları

Multilevel Queue Scheduling

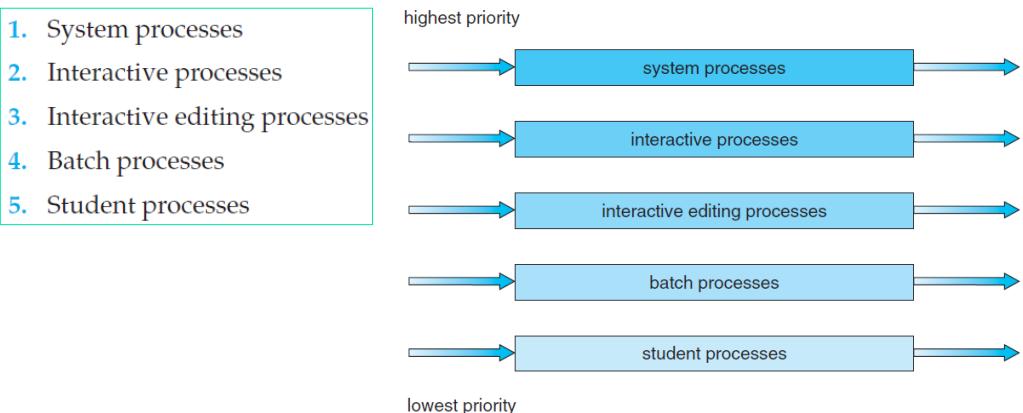
- **Multilevel Queue Scheduling (MQS) algoritmasında, process'ler farklı gruplar halinde sınıflandırılır.**
- Örneğin process'ler **foreground** (interaktif) ve **background** (batch) olarak 2 gruba ayrılabilir.
- Foreground process'lerde response-time kısa olması gereklidir ve background process'lere göre önceliklidir.
- **Multilevel queue scheduling** algoritması **hazır kuyruğunu** **parçalara böler** ve kendi aralarında önceliklendirir.
- Process'ler bazı özelliklerine göre (hafıza boyutu, öncelik, process türü, ...) bir kuyruğa atanır.
- Her kuyruk kendi scheduling algoritmasına sahiptir.

28

Scheduling algoritmaları

Multilevel Queue Scheduling

- Her kuyruğa öncelik derecesine göre time slice atanabilir.
- Her kuyruğa diğerlerine göre **mutlak öncelik tanımlanabilir ve kendisinde process varken düşük öncelikli kuyruğa geçilmez.**



Scheduling algoritmaları

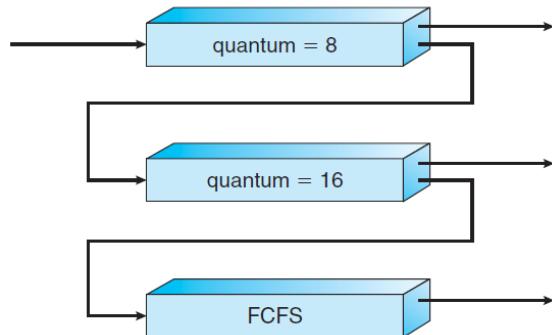
Multilevel Feedback Queue Scheduling

- **Multilevel Queue Scheduling (MQS) algoritmasında, process'ler farklı kuyruklar arasında geçiş yapabilirler.**
- Bu yöntemde, I/O bound ve interaktif process'ler yüksek öncelikli kuyruğa atanır.
- **Düşük öncelikli kuyrukta çok uzun süre bekleyen process'ler yüksek öncelikli kuyruğa aktarılır** (indefinite lock engellenir).
- Hazır kuyruğuna gelen process öncelikle en yüksek öncelikli kuyruğa alınır.
- En yüksek öncelikli kuyruk tamamen boşalırsa ikinci öncelikli kuyruğa geçilir.

Scheduling algoritmaları

Multilevel Feedback Queue Scheduling

- Şekilde üstteki kuyruk en yüksek önceliğe, en alttaki kuyruk en düşük önceliğe sahiptir.
- Kuyruklarda time slice (quantum) süreleri farklı olabilir.



31

Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- **Multi-processor scheduling**
- Gerçek zamanlı CPU scheduling

32

Multi-processor scheduling

- Çok işlemci kullanılan sistemlerde yük paylaşımı (**load sharing**) yapılabilir, ancak scheduling çok daha karmaşık hale gelir.
- **Asymmetric multiprocessing** yaklaşımında, CPU'lardan birisi (**master**) scheduling algoritmaları, I/O işlemleri ve diğer sistem aktivitelerini yönetir. Diğer CPU'lar kullanıcı kodlarını çalıştırır.
- **Symmetric multiprocessing** yaklaşımında, her CPU kendi scheduling algoritmasına sahiptir ve **master CPU yoktur**.
- Tüm CPU'lar ortak hazır kuyruğuna sahip olabilir veya ayrı ayrı hazır kuyruğu olabilir.
- **Birden fazla CPU'nun paylaşılan veri yapısına erişimi engellenmelidir**.
- Birden fazla CPU'nun **aynı process'i çalıştırması engellenmelidir**.
- Windows, Linux ve Mac OS X işletim sistemleri SMP desteğini sağlarlar.

33

Multi-processor scheduling

İşlemci ile atanan process ilişkisi

- Bir process başka bir işlemciye aktarıldığında, eski işlemcideki cache bellek bilgileri aktarılmaz.
- Yeni aktarılan işlemcinin **cache bellek bilgileri oluşana kadar hit rate oranı çok düşük kalır**.
- Bir process çalışmaktan olduğu **işlemci ile ilişkilendirilir (processor affinity)** ve sonraki çalışacağı işlemci de aynı olur.
- **Bazı sistemlerde**, process bir işlemciye atanır, ancak **aynı işlemcide çalışmayı garanti etmez (soft affinity)**.
- **Bazı sistemlerde**, process bir işlemciye atanır **ve her zaman aynı işlemcide çalışmayı garanti eder (hard affinity)**.
- Linux işletim sistemi soft affinity ve hard affinity desteğine sahiptir.

34

Multi-processor scheduling

Yük dengeleme

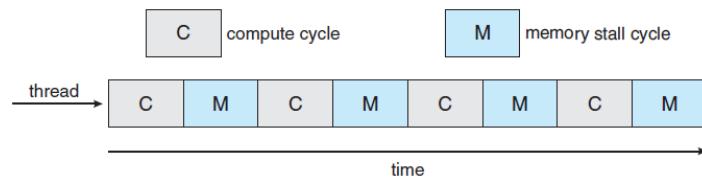
- **Yük dengeleme (load balancing)**, SMP sistemlerde tüm işlemciler üzerinde iş yükünü dağıtarak verimi artırmayı amaçlar.
- Her işlemcinin kendi kuyruğuna sahip olduğu sistemlerde, yük dengeleme iyi yapılmazsa bazı işlemciler boş beklerken diğer işlemciler yoğun çalışabilir.
- Ortak kuyruk kullanan sistemlerde yük dengellemeye ihtiyaç olmaz.
- **Yük dağılımı için iki yöntem kullanılır: push migration ve pull migration.**
- **Push migration** yönteminde, **bir görev** işlemcilerin iş yükünü kontrol eder ve **boş olanlara dolu olan diğer işlemcilerdeki process'leri aktarır**.
- **Pull migration** yönteminde, boş kalan işlemci dolu olan diğer işlemcilerde **bekleyen bir process'i kendi üzerine alır**.

35

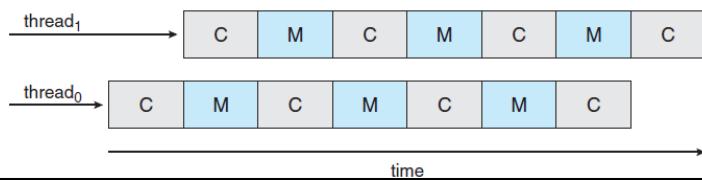
Multi-processor scheduling

Multicore sistemler

- **İşletim sistemi için her core ayrı bir işlemci olarak görülür.**
- İşlemcinin hafıza erişimi uzun süre alır (**memory stall**).



- Şekildeki işlemci, %50 süreyi hafızayı beklerken geçirmektedir.
- Bir core'a birden fazla thread atanarak aralarında geçiş yapılır.



36

Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- Multi-processor scheduling
- **Gerçek zamanlı CPU scheduling**

37

Gerçek zamanlı CPU scheduling

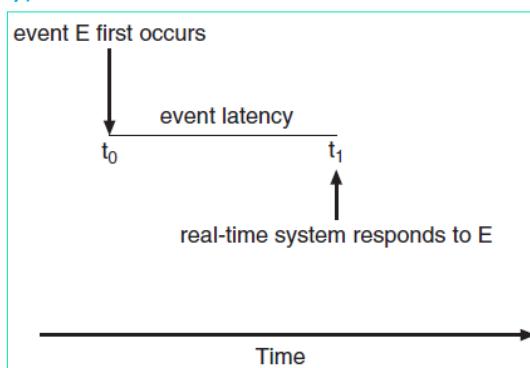
- **Gerçek zamanlı sistemler iki gruba ayrılır:**
 - **Soft real-time sistemler**
 - **Hard real-time sistemler**
- **Soft real-time sistemler**, zaman kritik process'lere diğerlerine göre öncelik verir, ancak **çalışma süresine garanti vermez**.
- **Hard real-time sistemler**, zaman kritik process'leri **deadline süresinde çalıştırmayı garanti eder**.

38

Gerçek zamanlı CPU scheduling

Minimizing latency

- Sistemde bir olay gerçekleştiğinde, olabildiği kadar kısa sürede gerekli işlemin yapılması zorunludur.
- Bir olay oluştuktan sonra işlemin gerçekleşmesine kadar bir süre geçer (**event latency**).

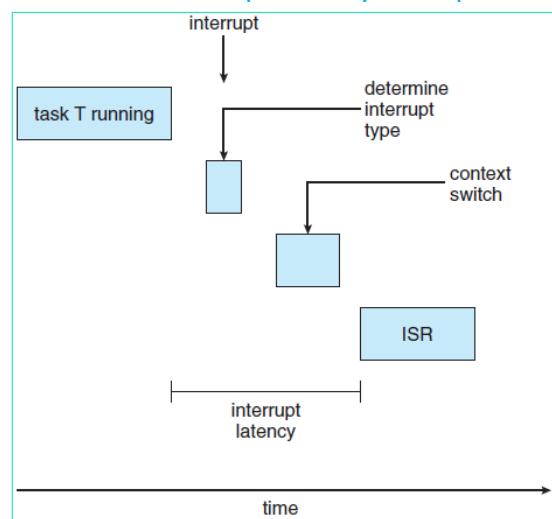


39

Gerçek zamanlı CPU scheduling

Minimizing latency

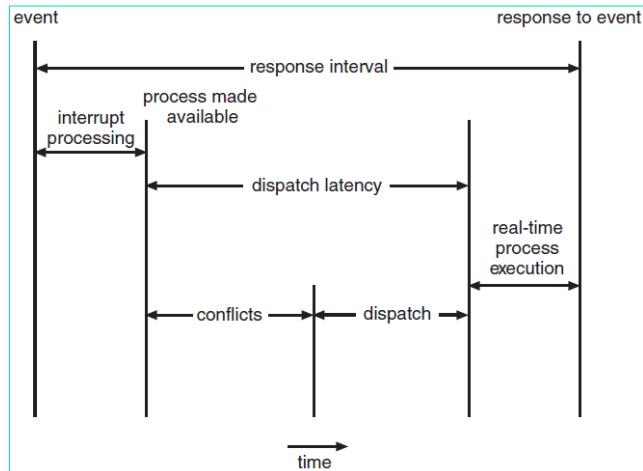
- Gerçek zamanlı sistemlerin performansını **interrupt latency** ve **dispatch latency** etkiler.
- Interrupt latency**, CPU'ya interrupt gelmesi ile CPU'nun istenen işleme başlaması için geçen süredir.



Gerçek zamanlı CPU scheduling

Minimizing latency

- Bir process'in durdurularak diğer process'in başlatılması için geçen süreye **dispatch latency** denir.
- **Conflict** aşamasında yeni process için kaynak aktarımı veya bir process'in durdurulması gerçekleştirilir.



Gerçek zamanlı CPU scheduling

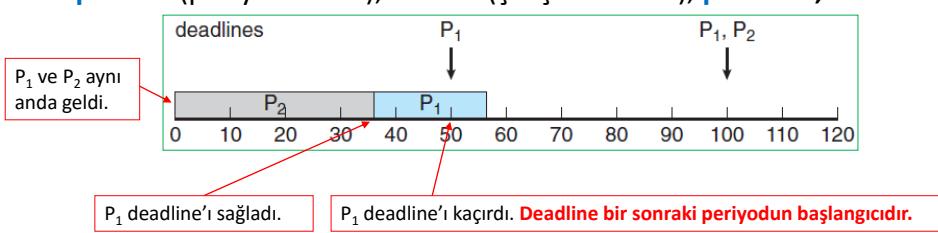
Priority-Based Scheduling

- Gerçek zamanlı işletim sistemleri **öncelik tabanlı algoritma** kullanmak zorundadır.
- Bir process CPU'da çalışırken yüksek öncelikli process geldiğinde kesilir ve gelen process çalıştırılır.
- Windows 32 seviyeli önceliklendirme yapar. **16-31 arasındaki öncelik değerlerini gerçek zamanlı process'lere ayırmıştır.**
- Bu şekildeki çalışma ile (preemptive, priority-based scheduler) soft real-time garanti edilir.
- Bir process için deadline'dan önce çalışma garantisı yoktur.
- Aynı öncelik seviyesinde bekleyen process'ler varsa preemptive yapılmaz.

Gerçek zamanlı CPU scheduling

Rate-Monotonic Scheduling

- **Rate-Monotonic Scheduling** algoritmasında, her process sisteme geldiğinde periyot süresiyle ters orantılı (sisteme gelme sıklığı ile doğru orantılı) öncelik seviyesi atanır.
- Periyot süresi kısalıkça öncelik seviyesi artar, **arttıkça** öncelik seviyesi azalır. CPU'yu kullanma sıklığı artan process'lere öncelik verilir.
- **P₂ daha önceliklidir** (CPU'ya gelme sıklığı göz önüne alınmamıştır.).
- **p₁ = 50** (periyot süresi), **t₁ = 20** (çalışma süresi), **p₂ = 100**, **t₂ = 35**.

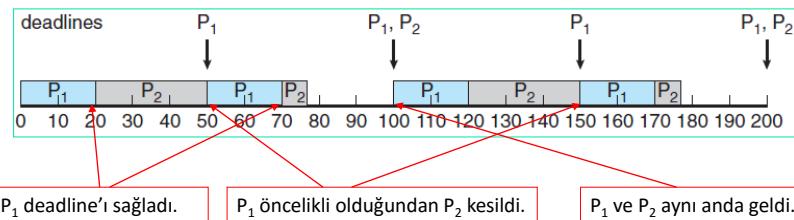


43

Gerçek zamanlı CPU scheduling

Rate-Monotonic Scheduling

- Şekilde **P₁ daha önceliklidir** (CPU'ya gelme sıklığı fazladır.).
- **p₁ = 50** (periyot süresi), **t₁ = 20** (çalışma süresi), **p₂ = 100**, **t₂ = 35**.

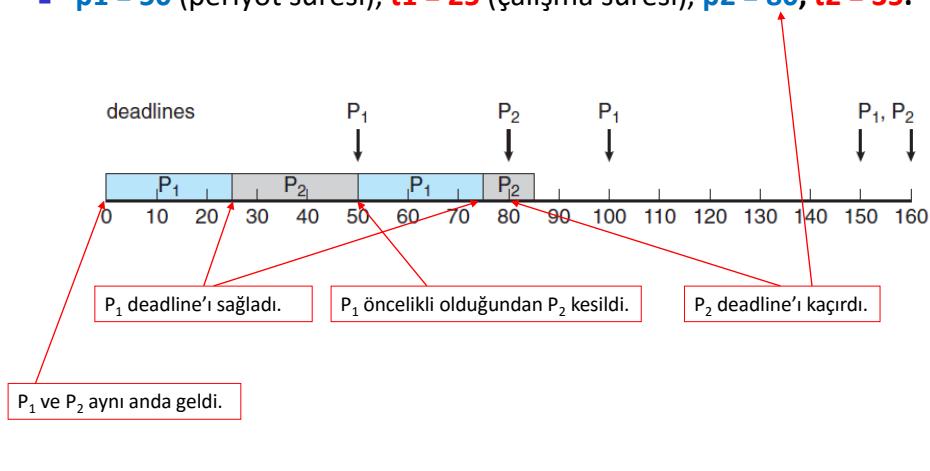


44

Gerçek zamanlı CPU scheduling

Rate-Monotonic Scheduling

- Şekilde P_1 daha önceliklidir (CPU'ya gelme sıklığı fazladır.).
- $p1 = 50$ (periyot süresi), $t1 = 25$ (çalışma süresi), $p2 = 80$, $t2 = 35$.

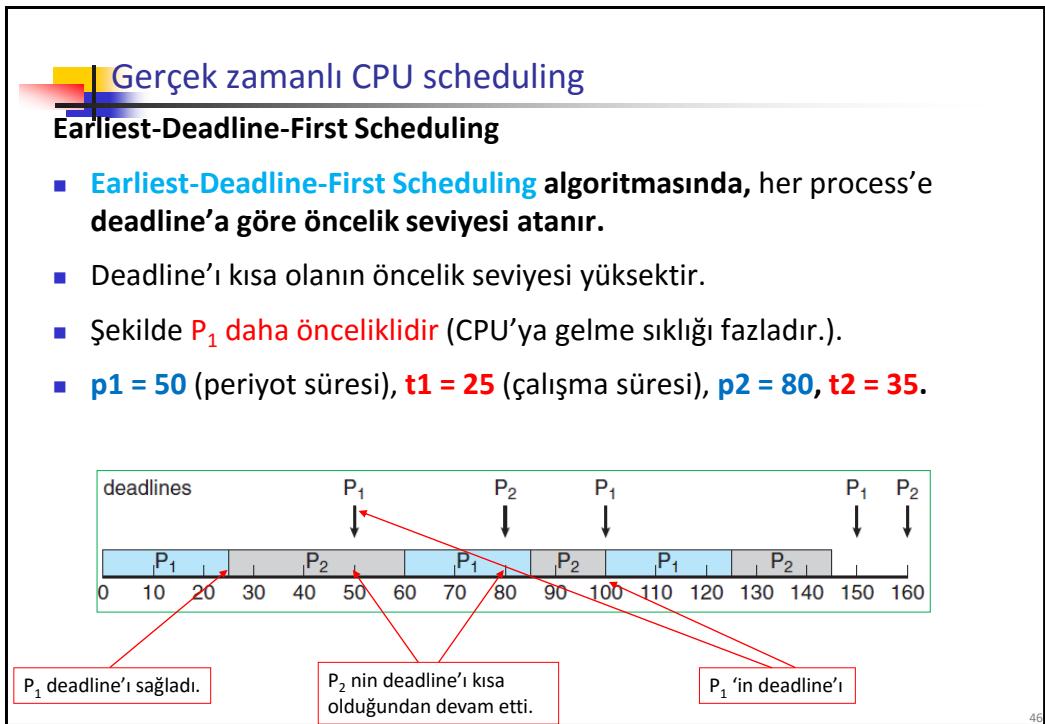


45

Gerçek zamanlı CPU scheduling

Earliest-Deadline-First Scheduling

- Earliest-Deadline-First Scheduling algoritmasında, her process'e deadline'a göre öncelik seviyesi atanır.
- Deadline'ı kısa olanın öncelik seviyesi yüksektir.
- Şekilde P_1 daha önceliklidir (CPU'ya gelme sıklığı fazladır.).
- $p1 = 50$ (periyot süresi), $t1 = 25$ (çalışma süresi), $p2 = 80$, $t2 = 35$.



45

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- **Sistem modeli**
 - Deadlock tanımı ve özellikler
 - Deadlock yönetimi için metodlar
 - Deadlock önleme
 - Deadlock'tan kaçınma
 - Deadlock algılama
 - Deadlock'tan kurtulma

Sistem modeli

- Multiprogramming ortamlarda çok sayıda process sınırlı kaynağı kullanmak için yarışır.
- Bir process bir kaynağa istek yaptığında, kaynak dolu ise bekleme durumuna geçer.
- **Bekleme durumundaki process, bazı durumlarda hiçbir zaman durumunu değiştiremez.** Buna **deadlock** (kilitlenme) denir.
- İşletim sistemleri genellikle deadlock önleme mekanizmalarını sağlamazlar.
- **Program geliştiricinin deadlock oluşmayacak şekilde program geliştirmesi gereklidir.**

3

Sistem modeli

- Bir sistemdeki kaynaklar, CPU cycle, dosyalar, I/O cihazları (yazıcı, DVD sürücü) gibi farklı türdedir.
- Bir kaynaktan birden fazla varsa (2 CPU, farklı noktalardaki 2 yazıcı), bu kaynakların da birbirinden ayrılması gereklidir.
- Bir process, bir kaynağı kullanmadan önce istek yapar (**request**), kullandıktan sonra da serbest bırakır (**release**).
- Bir process bir kaynağı aşağıdaki sırayla kullanır:
 - **Request:** Process kaynağı istek yapar. **Kaynak kullanılabilir değilse bekler.**
 - **Use:** Process kaynak üzerindeki **işlemi**ni gerçekleştirir.
 - **Release:** Process kaynağı serbest bırakır.
- **Cihaz** için **request()** ve **release()**, **dosya** için **open()** ve **close()**, **hafıza** için **allocate()** ve **free()** şeklindedir.

4

Sistem modeli

- Request ve release işlemleri, **semaforlar** için **wait()** ve **signal()**, **mutex kilitlemesi** için **acquire()** ve **release()** şeklinde tanımlanabilir.
- İşletim sistemleri **kaynakların boş veya atanmış olduğunu bir tablo ile tutarlar.**
- Kaynaklar, **fiziksel** (yazıcı, hafıza alanı, ...) veya **mantıksal** (semafor, mutex lock) olabilir.
- Bir sistemde, 3 tane CD WR sürücüsü olsun. 3 process bu kaynakları kullanırken, **her process diğer CD WR sürücülerinden birisine istek yaparsa deadlock oluşur.**
- Bir sistemde, 1 yazıcı ve 1 DVD sürücü olsun. **P0 process'i yazıcıyı, P1 process'i DVD sürücüyü tutarken, P1 yazıcıya ve P0 DVD sürücüye istek yaparsa deadlock oluşur.**
- Multithread uygulama geliştiriciler deadlock olasılığına dikkat etmelidir.

5

Konular

- Sistem modeli
 - **Deadlock tanımı ve Özellikleri**
 - Deadlock yönetimi için metodlar
 - Deadlock önleme
 - Deadlock'tan kaçınma
 - Deadlock algılama
 - Deadlock'tan kurtulma

6

Deadlock tanımı ve özellikler

Gerekli şartlar

- Bir deadlock durumunda, process'ler hiçbir zaman sonlanamaz, sistem kaynakları atanmış durumdadır ve başka işler başlatılamaz.
- Aşağıdaki 4 şart aynı anda oluştuğunda deadlock ortaya çıkabilir:
 - **Mutual exclusion:** Paylaşımzsız kaynak bir process tarafından tutulurken başka bir process bu kaynağa istek yaparsa, ikinci process kaynak boşalıncaya kadar bekler.
 - **Hold and wait:** Bir process bir kaynağı tutarken, başka bir kaynağı da bekler durumundadır. Bu kaynak ise başka bir process tarafından kullanılır durumdadır.
 - **No preemption:** Kaynaklar önceden boşaltılamaz. Bir kaynağın boşaltılması için kullanan process'in serbest bırakması gereklidir.
 - **Circular wait:** $\{P_0, P_1, \dots, P_n\}$ processleri birbirini beklemektedir. P_0 process'i P_1 'i, P_1 process'i P_2 'yi, ..., P_n process'i de P_0 'yı beklemektedir.
- Her durum birbirinden tamamen bağımsız değildir. **Circular wait** oluştuğunda, **hold and wait** durumu vardır.

7

Deadlock tanımı ve özellikler

Resource-allocation graph

- Deadlock'lar **bir yönlü graf (directed graph)** kullanılarak (**system resource-allocation graph**) tanımlanabilir.
- Graf üzerinde **düğümler V (vertices)** ile **kenarlar E (edge)** ile gösterilir.
- **V kümesi iki kısma** ayrırlar:
 - $P = \{P_1, P_2, \dots, P_n\}$ aktif process'ler, gösterir.
 - $R = \{R_1, R_2, \dots, R_m\}$ sistemdeki tüm kaynakları gösterir.
- Bir P_i process'inden R_j kaynağına çizilen kenar $P_i \rightarrow R_j$ şeklinde gösterilir.
- $P_i \rightarrow R_j$ kenarı ile P_i process'ının R_j kaynağına istek yaptığı ve beklediği ifade edilir.
- $R_j \rightarrow P_i$ kenarı ile de R_j kaynağının P_i process'ine atandığı ifade edilir.
- $P_i \rightarrow R_j$ **request edge**, $R_j \rightarrow P_i$ **assignment edge** olarak adlandırılır.

8

Deadlock tanımı ve özellikler

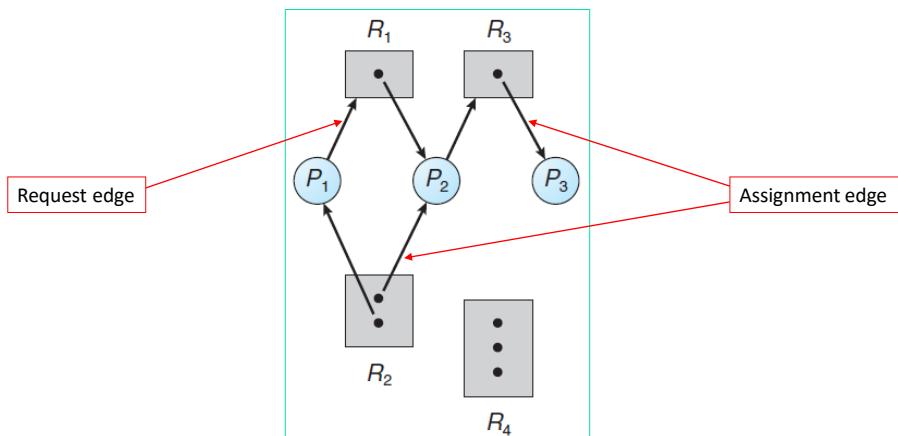
Resource-allocation graph

- Graf üzerinde her bir process daire ile, her bir kaynak dikdörtgenle gösterilir.
- Bir kaynaktan birden fazla örnek varsa (birden fazla CD WR) dikdörtgen içerisinde her örnek ayrı nokta ile gösterilir.
- Bir process, bir kaynaktan bir örneğe istek yaparsa **request edge** çizilir.
- Bir process'in yaptığı istek karşılanmasırsa **assignment edge**'e dönüştürülür.
- Kaynak serbest bırakıldığında ise assignment edge silinerek kaynak serbest bırakılır.

Deadlock tanımı ve özellikler

Resource-allocation graph

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

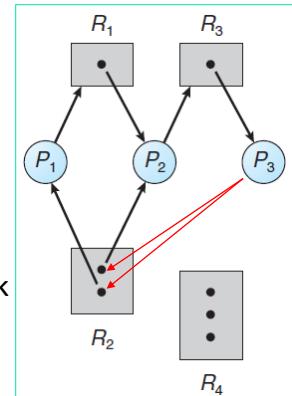


10

Deadlock tanımı ve özellikleri

Resource-allocation graph

- Eğer graf üzerinde **döngü yoksa deadlock yoktur**,
döngü varsa deadlock olabilir.
- Her kaynaktan sadece 1 örnek varken graf**
üzerinde **döngü varsa, deadlock oluşur.**
- Döngü içerisinde yer alan tüm process'ler deadlock
durumundadır.
- Kaynaklardan birden fazla olması durumunda,**
deadlock için döngü oluşması gereklidir ancak
yeterli değildir.
- Döngü dışındaki bir process'e atanmış kaynak seçilebilir.

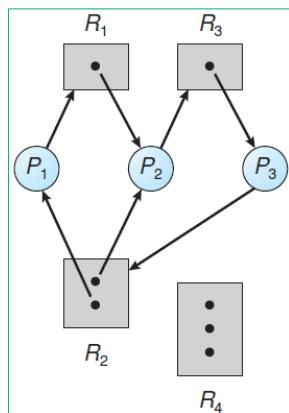


11

Deadlock tanımı ve özellikleri

Resource-allocation graph

- Şekilde P_1, P_2 ve P_3 deadlock durumundadır.



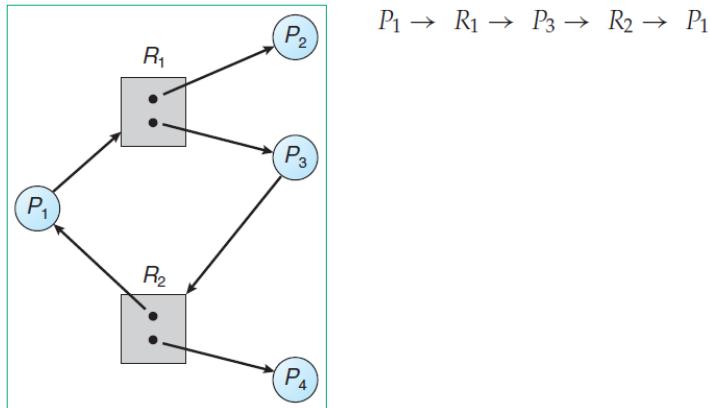
$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
 $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

12

Deadlock tanımı ve özellikler

Resource-allocation graph

- Şekilde döngü vardır, ancak deadlock yoktur.
- P4, R2'nin kopyasını serbest bırakırsa P3 kullanabilir.



13

Konular

- Sistem modeli
- Deadlock tanımı ve özellikler
- **Deadlock yönetimi için metodlar**
- Deadlock önleme
- Deadlock'tan kaçınma
- Deadlock algılama
- Deadlock'tan kurtulma

14

Deadlock yönetimi için metotlar

- Deadlock problemi için 3 farklı yol izlenebilir:
 - **Deadlock'lardan kaçınmak için protokol kullanılabilir.** Sistem hiçbir zaman deadlock durumuna düşmez.
 - Sistemin deadlock durumuna düşmesine izin verilir, **deadlock algılanır ve çözülür.**
 - **Deadlock problemi tamamen gözardı edilir** ve sistemde deadlock hiçbir zaman olmayacağı gibi davranışır.
- Üçüncü durum işletim sistemleri tarafından yaygın kullanılır (Linux, Windows).
- **Linux ve Windows işletim sistemleri, deadlock yönetimini uygulama geliştiricilere bırakır.**

15

Deadlock yönetimi için metotlar

- Bir sistemde hiçbir zaman **deadlock olmamasını garanti etmek için, deadlock-prevention veya deadlock-avoidance yöntemleri kullanılabilir.**
- **Deadlock prevention, kaynak isteklerini sınırlı olarak** deadlock olmasını önler.
- **Deadlock avoidance, bir kaynağa istek yapan process'in yanı sıra, hangi zaman aralığında kullanacağını da bilmek ister.**
- Deadlock avoidance, kaynak isteği yapan process'in beklemesine veya kaynağın atanmasına karar verebilir.
- Bir sistem, **deadlock prevention veya deadlock avoidance yöntemlerini kullanmazsa deadlock oluşabilir.**
- Bu sistemler, **deadlock olup olmadığını kontrol eden bir algoritma ve deadlock oluştuğunda çözümünü sağlayan bir algoritma** sağlamalıdır.

16

Deadlock yönetimi için metotlar

- Bir sistemde **deadlock algılama ve çözme algoritması yoksa**, aynı kaynağa istek yapan ve kilitlenen process sayısı artmaya başlar, bir süre sonra sistem çalışmaz hale gelir ve **en sonunda manuel olarak sistemin restart yapılması gereklidir.**
- Deadlock algılama ve çözümleme yöntemleri çoğu işletim sisteminde kullanılmaz.
- Bazı sistemler, başka durumlar için kullandığı yöntemleri deadlock yönetiminde de kullanırlar.
- Bazı sistemlerde, deadlock durumu yoktur. Bunun yerine **process'in donmuş durumu (frozen state) vardır.**
- Bu durumda **process işletim sistemine geri dönemez.**

17

Konular

- Sistem modeli
- Deadlock tanımı ve özellikler
- Deadlock yönetimi için metotlar
- **Deadlock önleme**
- Deadlock'tan kaçınma
- Deadlock algılama
- Deadlock'tan kurtulma

18

Deadlock önleme

Mutual exclusion

- Deadlock oluşması için 4 durumun da (**mutual exclusion, hold and wait, no preemption, circular wait**) gerçekleşmesi gereklidir.
- Bu şartların birisi engellenirse deadlock önlenmiş olur.
- Mutual exclusion'da en azından bir kaynak paylaşılamaz (eş zamanlı erişilemez) durumdadır.
- Paylaşılabilebilir kaynaklar deadlock oluşturmaz.
- **Read-only dosyalar** paylaşılabilen kaynaklardır ve **deadlock oluşturmaz**.
- **Paylaşılamaz kaynaklara birden fazla process tarafından eş zamanlı erişim yapılamaz** (Semafor veya mutex lock ile engellenir).

19

Deadlock önleme

Hold and wait

- Bir sistemde **hold and wait** durumunun oluşmaması için, bir process bir kaynağa istek yaptığında **başka bir kaynağı tutmaması** gereklidir.
- Bir protokol kullanılarak, **bir process çalışmaya başlamadan önce ihtiyaç duyacağı tüm kaynaklar kendisine atanabilir**.
- Başka bir protokolde ise, **eger bir process kaynak kullanmıyorsa yeni kaynak için istek yapabilir**.
- Bir process, DVD sürücüdeki dosyayı diske kaydetsin, dosyayı sıralasın, ardından yazıcıdan çıktı alsin.
- Birinci protokolde, tüm çalışma süresince DVD sürücü, dosya ve yazıcı process'e atanır (**verimsiz kullanım**).
- İkinci protokolde, kaynaklar ihtiyaç duyduğunda sırayla process'e atanır (**Bir process sık istek yapılan bir kaynağı süresiz bekleyebilir**).

20

Deadlock önleme

No preemption

- Bir kaynak bir process tarafından tutuluyorsa, **kaynak process tarafından serbest bırakılmadan önce boşaltılamaz**.
- Bir protokol kullanılarak, bir process bir kaynak isteğiinde bulunursa ve **bekleme durumuna geçerse, tutmakta olduğu tüm kaynakları serbest bırakır**.
- Başka bir protokol kullanılarak,
 - **Bir process bir kaynağa istek yaptığından kullanılabilir olduğu kontrol edilir.** Uygunsa tahsis edilir.
 - **İstek yapılan kaynak başka process tarafından tutuluyorsa**, tutan process'in başka bir kaynağı bekleyip beklememiği kontrol edilir. **Başka bir kaynağı bekliyorsa**, istek yapılan kaynak boşaltılıp (preempt) yeni istek yapan process'e atanır.
 - **İstek yapılan kaynak uygun değilse ve kullanan process başka bir kaynağı beklemiyorsa**, istek yapan process bekletilir.

21

Deadlock önleme

Circular wait

- Circular wait durumunun önlenmesi için **tüm kaynaklar sıralanır ve bir process kaynak isteğini artan sırada yapabilir**.

$$\begin{aligned}F(\text{tape drive}) &= 1 \\F(\text{disk drive}) &= 5 \\F(\text{printer}) &= 12\end{aligned}$$

- Bir process, **başlangıçta bir kaynağı isteyebilir**. Ardından **yapacağı istekler artan sırada olmak zorundadır**.
- Bir process, R_i kaynağından sonra R_j kaynağını isteyebilir ($F(R_i) < F(R_j)$).
- Eğer bir process **tape drive ile yazıcıyı aynı anda kullanmak isterse**, **önce tape drive atanır ardından yazıcı atanır**.

22

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metodlar
- Deadlock önleme
- **Deadlock'tan kaçınma**
- Deadlock algılama
- Deadlock'tan kurtulma

23

Deadlock'tan kaçınma

- **Deadlock önleme algoritmaları ile kaynak erişimi sınırlanır.**
- **Deadlock oluşmasını sağlayan durumlardan en az bir tanesi engellenir.**
- Bu durumda, **verimsiz kullanım ve düşük throughput** ortaya çıkar.
- **Deadlock'tan kaçınma yöntemlerinde, deadlock oluşumunu engellemek için kaynakların nasıl istendiğinin bilinmesi gereklidir:**
 - P process'i önce tape sürücü ardından yazıcı istemektedir ve ikisini birlikte serbest bırakmaktadır.
 - Q process'i ise önce yazıcı ardından tape sürücü istemektedir.
- **Sistem, ileride deadlock oluşmayacak şekilde kaynakları planlayarak tahsis eder.**
- Basit bir yöntemde, **tüm process'ler ihtiyaç duydukları kaynakları başlangıçta bildirirler, sistem circular wait oluşmayacak şekilde kaynakları tahsis eder.**

24

Deadlock'tan kaçınma

Safe state

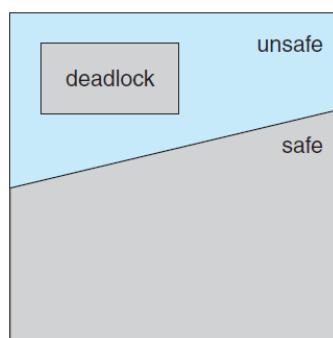
- Eğer bir **sistem**, kaynakları process'lere belirli bir sırada (**safe sequence**) **maksimum ihtiyaçları kadar** atayabiliyorsa ve **deadlock oluşmuyorsa** bu durum **safe state** olarak adlandırılır.
- Eğer bir sistemde **safe sequence** varsa sistem safe state durumundadır.
- **<P1, P2, ..., Pn>** sırası **mevcut atama durumu için safe suquence'dır**, eğer aşağıdaki durumlar sağlanırsa:
 - Pj process'inin tüm kaynak istekleri mevcut **boş kaynaklarla** ve tüm Pi'ler ($i < j$) tarafından tutulan **dolu kaynaklarla** karşılanır (**tüm Pi'lerin sonlanması gereklidir**).
 - Eğer Pj'nin istediği kaynaklar kullanılabilir değilse, **tüm Pi'ler sonlanıp kaynakları boşaltıncaya kadar bekler**.
 - Pj kaynak kullanımını **sonlandırdığında**, Pj+1 process'i kaynağı alıp kullanabilir.

25

Deadlock'tan kaçınma

Safe state

- Bir **safe sequence yoksa** sistem **unsafe durumundadır (deadlock oluşabilir)**.
- Safe, unsafe ve deadlock state şekilde görülmektedir.



26

Deadlock'tan kaçınma

Safe state - örnek

- Bir sistemde **12 tape sürücü** ve **3 process** (P_0 , P_1 ve P_2) olsun.
- P_0 process'i maksimum **10 tane**, P_1 process'i **4 tane** ve P_2 process'i **9 tane** tape sürücüye ihtiyaç duymaktadır.
- **t0 anında, P_0 process'i 5 tane, P_1 process'i 2 tane ve P_2 process'i 2 tane tape sürücü kullanıyor.**
- **t0 anında, 3 tane tape sürücü boştadır.**

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- **t0 anında $\langle P_1, P_0, P_2 \rangle$ safe sequence'inde sistem safe durumundadır.**

27

Deadlock'tan kaçınma

Safe state - örnek

- **$\langle P_1, P_0, P_2 \rangle$ safe sequence için, t1 anında P_2 process'i 1 tane kaynak daha alsın (Safe sequence bozuldu!!!). Boş kaynak sayısı 2'ye düşer.**

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

3 olur.

- Bu durumda sadece P_1 process'i tüm kaynaklarını alabilir (2 tane daha).
- **P_1 tüm kaynaklarını boşalrsa toplam 4 kaynak boşta olur.**
- **P_0 ve P_2 aynı anda tüm kaynakları kullanmak isterse deadlock oluşur.**
- P_0 process'i 5 kaynak ister, P_2 process'i ise 6 kaynak ister (**4 kaynak boş**).
- **P_2 'ye yeni kaynak ataması yapılmadan önce, kendisinden önceki processlerin tüm kaynaklarını serbest bırakması beklenmelidir.**

28

Deadlock'tan kaçınma

Resource-allocation-graph algorithm

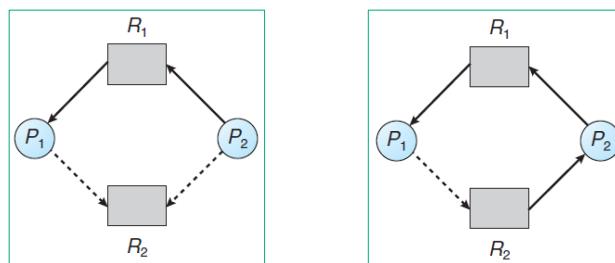
- Her kaynaktan bir örnek olan sistemlerde resource-allocation graf değiştirilerek kullanılabilir.
- Resource-allocation grafına yeni bir kenar eklenir (**claim edge**).
- $P_i \rightarrow R_j$ claim edge, P_i process'inin R_j kaynağını **ileriki zamanda isteyebileceğini gösterir**.
- Claim edge graf üzerinde **noktalı çizgiyle gösterilir**.
- P_i process'i R_j kaynağını **istediğinde** ise **request edge** yapılır.
- R_j kaynağı P_i process'i tarafından **serbest bırakıldığında**, assignment edge **claim edge'e dönüştürülür**.
- P_i process'i çalışmaya başlamadan önce tüm claim edge'leri graf üzerinde görüntülenir.

29

Deadlock'tan kaçınma

Resource-allocation-graph algorithm

- P_i process'i R_j kaynağını **istediğinde** claim edge request edge'e dönüşür.
- Assignment edge yapılabilmesi için **resource-allocation graf** üzerinde **döngü oluşmaması gereklidir**.
- Şekilde **P2** process'i, **R2'yi** **istediğinde boş olmasına rağmen atanmaz** (Döngü oluştugundan **sistem unsafe durumuna geçer**).
- Daha sonra **P1** process'i **R2'yi** **istediğinde deadlock oluşur**.



30

Deadlock'tan kaçınma

Banker algoritması

- Resource-allocation algoritması **aynı kaynaktan birden fazla olan sistemlerde uygulanamaz.**
- Aynı kaynaktan birden fazla olan sistemlerde **banker algoritması (banker's algorithm) kullanılabilir.**
- Banka sisteminde, bir banka parasını tüm müşterilerinin ihtiyacını karşılayamayacak seviyeye hiçbir zaman düşürmez.
- **Bir process sisteme geldiğinde** tüm kaynaklar için **maksimum ihtiyaçlarını bildirir.**
- **Bu kaynak miktarı sistemin tüm kaynaklarından fazla olamaz !!!**
- **Bir process bir grup kaynak istediğinde**, sistem bu atamanın yapılması halinde **safe state'in korunup korunmadığına bakar.**

31

Deadlock'tan kaçınma

Banker algoritması

- Banker algoritması aşağıdaki veri yapılarını kullanır:
 - **Available:** Bir vektördür. Sistemde boştaki tüm kaynakların sayısını tutar.
Available[j] = k ise, sistemdeki **R_j** kaynağından **k adet boşadır.**
 - **Max:** n * m matristir. Her process'in maksimum kaynak talebini tutar.
Max[i][j] = k ise **P_i process'i R_j kaynağından en fazla k tane isteyebilir.**
 - **Allocation:** n * m matristir. Her process'in kullanmakta olduğu kaynak miktarını tutar. **Allocation[i][j] = k** ise, **P_i process'i R_j kaynağından k adet kullanmaktadır.**
 - **Need:** n * m matristir. Her process'in kalan ihtiyaç miktarını tutar.
Need[i][j] = k ise **P_i process'i R_j kaynağından k adet daha kullanabilir.**
Need[i][j] = Max[i][j] – Allocation[i][j] olur.
- Veri yapısındaki veri boyutu (satır) ve değerleri dinamik olarak değişimdir (**çalışan process sayısı değişebilir**).

32

Deadlock'tan kaçınma

Banker algoritması - örnek

- Sistemde 5 process vardır. A kaynağından 10, B kaynağından 5 ve C kaynağından 7 tane var (10, 5, 7). T0 anında sistem görüntüsü aşağıdadır.

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- Yukarıdaki durumda $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ safe sequence'dir.
- P_1 ihtiyacının tamamını alırsa, boşta (2, 1, 0) kalır. Tümünü bırakırsa (5, 3, 2) boşadır. Ardından P_3 tümünü alabilir!!!

33

Deadlock'tan kaçınma

Banker algoritması - örnek

- $P_1 = (1, 0, 2)$ isteğinde bulunursa, öncelikle boşta olanların sağlayıp sağlamadığına bakılır ($(1, 0, 2) \leq (3, 3, 2)$). Boşta kalan (2, 3, 0).
- Kaynak ataması yapıldıktan sonraki durum aşağıdadır.

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Yeni durum için $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ safe sequence'dir.
- Eğer, $P_4 = (3, 3, 0)$ isterse yeterli kaynak olmadığından atanamaz.
- $P_0 = (0, 2, 0)$ atanırsa (P_0 need (7, 2, 3) kalır), boş (2, 1, 0) ve sistem unsafe durumuna geçer.

34

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metodlar
- Deadlock önleme
- Deadlock'tan kaçınma
- **Deadlock algılama**
- Deadlock'tan kurtulma

35

Deadlock algılama

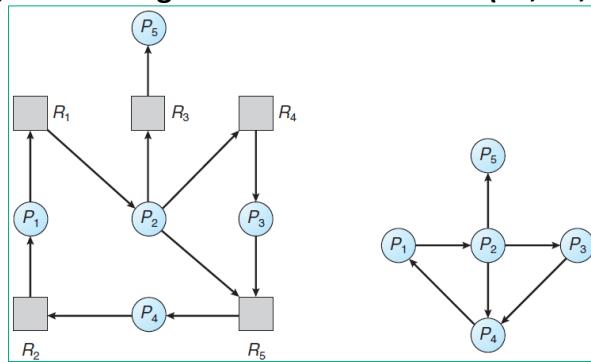
- Bir sistem **deadlock önleme veya deadlock'tan kaçınma algoritmasına sahip değilse**, deadlock oluşabilir.
- Bu tür bir sistem, deadlock probleminin çözümü için aşağıdakileri sağlayabilir:
 - Sistemin deadlock durumunda olup olmadığına karar verecek bir algoritma
 - Deadlock durumunun çözülmesi için bir algoritma
- **Deadlock algılama algoritmaları**, her kaynaktan bir tane olması veya her kaynaktan birden fazla olması durumuna göre **farklılık gösterir**.

36

Deadlock algılama

Her kaynaktan bir örnek olması

- Resource-allocation grafi kullanılarak wait-for grafi oluşturulabilir.
- Wait-for grafında, kaynak düğümleri kaldırılarak sadece process düğümleri bırakılır.
- Wait-for grafında döngü varsa deadlock vardır (P_1, P_2, P_3, P_4).



37

Deadlock algılama

Her kaynaktan birden fazla örnek olması

- Bu tür sistemlerde birden fazla veri yapısı kullanılır:
 - Available: Her kaynak türü için boştaki kaynak sayısını tutar.
 - Allocation: $n * m$ matristir. Her process için mevcut atanmış kaynak sayısını tutar.
 - Request: $n * m$ matristir. Her process'in mevcut istek sayısını tutar.
Request[i][j] = k ise P_i process'i R_j kaynağından k tane daha istemektedir.

38

Deadlock algılama

Her kaynaktan birden fazla örnek olması - örnek

- Sistemde 5 process, A = 7, B = 2 ve C = 6 kaynak vardır.
- T0 anındaki kaynak atama durumu aşağıdadır.

	<i>Allocation</i>	<i>Request</i>	<i>Available</i>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ safe sequence olsun.
- $P_2 = (0, 0, 1)$ isteği gelirse istek matrisi yandaki gibi olur ve sistem deadlock olabilir. **P0 tümünü bıraksa bile!!!**
- **P0 tümünü bıraksa bile**, kalan kaynak miktarı, **P1, P2, P3 ve P4 için yeterli değildir** ve deadlock oluşur.

	<i>Request</i>
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

Konular

- Sistem modeli
 - Deadlock tanımı ve özellikler
 - Deadlock yönetimi için metodlar
 - Deadlock önleme
 - Deadlock'tan kaçınma
 - Deadlock algılama
 - Deadlock'tan kurtulma

Deadlock'tan kurtulma

Process termination

- Sistemde deadlock olduğunda **operatör tarafından manuel giderilebilir.**
- Sistem otomatik olarak **deadlock recover yapabilir.**
- Process termination yönteminde 2 tür çözüm olabilir:
 - **Tüm deadlock durumundaki process'ler sonlandırılır.** Process'lerin sonuçları kaybolabilir.
 - **Deadlock döngüsü ortadan kalkıncaya kadar her adımda bir deadlock process sonlandırılır.** Her process sonlandığında deadlock cycle kontrolü yapılması gereklidir.
- Bir process'in **abort edilmesi sonucunda tutarsızlıklar ortaya çıkabilir.**
 - Dosya yazma işlemi yarıda kalabilir.
 - Yazıcıda devam eden yazdırma işlemi yarıda kalabilir.

41

Deadlock'tan kurtulma

Resource preemption

- **Deadlock cycle ortadan kalkıncaya kadar** process'lere atanmış kaynaklar alınarak başka process'lere atanır.
 - **Selecting a victim:** Kaynak veya process seçiminde maliyet hesaplanabilir. Deadlock olma sıklığı, çalışma süresi, vb.
 - **Rollback:** Bir process'ten bir kaynağı alınca, normal çalışmasına devam edip etmeyeceğine karar verilir. Buna karar vermek zor olduğundan process abort edilir ve yeniden başlatılır.
 - **Starvation:** Eğer bir process sürekli kesiliyorsa görevini hiçbir zaman tamamlayamayabilir (**starvation**). Bir process'in victim olarak seçilmesinde maksimum limit belirlenebilir.

42

İşletim Sistemleri

Hazırlayan: M. Ali Akcayol

Gazi Üniversitesi

Bilgisayar Mühendisliği Bölümü

Bu dersin sunumları, "Abraham Silberschatz, Greg Gagne, Peter B. Galvin, Operating System Concepts 9/e, Wiley, 2013." kitabı kullanılarak hazırlanmıştır.

Konular

- Giriş
- Swapping
- Bitişik hafıza atama
- Segmentation
- Paging

Giriş

- Modern bilgisayar sistemlerinin çalışmasında **hafıza merkezi role sahiptir.**
- **Hafıza**, her birisi kendi adresine sahip olan **çok sayıda byte alanına sahiptir.**
- **CPU**, **program counter (PC)** değerine göre **hafızadan bir komutu fetch eder.**
- Hafızadan alınan komutlar, **bir veya birden fazla parametre için hafıza erişimi (operand) gerektirebilirler.**
- Komutun çalıştırılmasından sonra **elde edilen sonuç hafızaya tekrar yazılabilir.**

3

Giriş

Temel donanım yapısı

- **Main memory** ve **general purpose register'lar**, CPU tarafından adreslenen genel amaçlı kayıt alanlarıdır.
- Makine komutlarında **hafıza adresini parametre (operand) olarak alan komutlar vardır**, ancak disk adresini alan komutlar yoktur.
- CPU'nun ihtiyaç duyduğu veri veya komut hafızada değilse, öncelikle **hafızaya alınmalıdır**.
- CPU içerisindeki **register'lara genellikle bir cycle ile erişilebilmektedir.**
- **Hafızaya erişim** bus üzerinden yapılır ve **register'a göre oldukça uzun süre gerektirir.**
- Hafıza ile CPU arası çok daha **hızlı ve CPU'ya yakın** bir saklama alanı oluşturulur (**cache**).

4

Giriş

Temel donanım yapısı

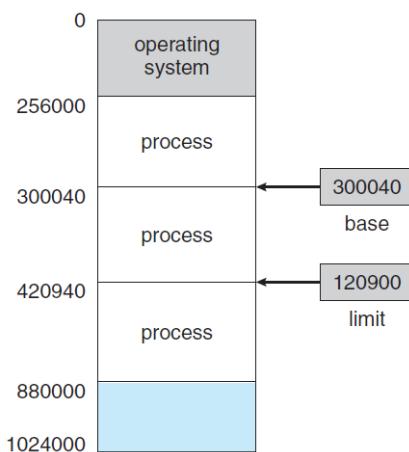
- Diğer kullanıcı process'lerin bir process'e ayrılan alana erişiminin engellenmesi gereklidir.
- Çok kullanıcılı sistemlerde bir kullanıcı process'ine başka kullanıcının erişiminin de engellenmesi gereklidir.
- Bu tür koruma işleri donanımsal düzeyde yapılır. İşletim sistemi düzeyinde yapıldığında performans düşer.
- Her process kendisine ait ayrı bir hafıza alanına sahiptir.
- Böylelikle, process'ler birbirinden ayrılmış olur ve birden fazla process eşzamanlı çalıştırılabilir.

5

Giriş

Temel donanım yapısı

- Bir process için ayrılan alanın **başlangıç adresi (base register)** ve **boyutu (limit register)** belirlenmelidir.

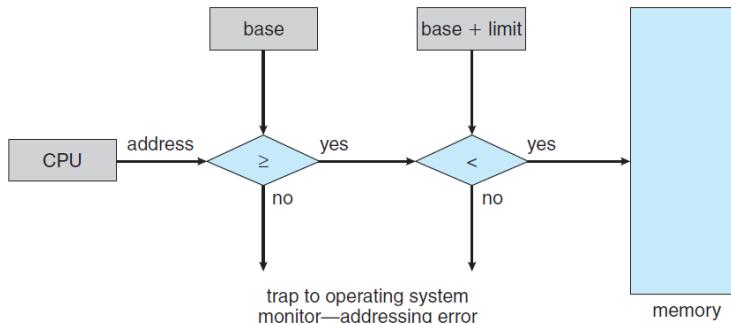


6

Giriş

Temel donanım yapısı

- Hafıza alanının korunması donanımla gerçekleştirilebilir.
- Kullanıcı modunda istek yapılan her hafıza adresinin **base** ile **base+limit** aralığında olduğu kontrol edilir.



- İstenen adres **process için ayrılan alanın dışında ise hata üretilir.**

Giriş

Adres binding

- Bir program disk üzerinde binary dosya olarak bulunur.
- **Bir programın çalıştırılabilmesi için hafızaya alınması gereklidir.**
- Bir process **disk üzerinden hafızaya alınmak için kuyruğa alınır (input queue)**.
- Bir process hafızaya yerleşikten sonra komutları çalıştırır veya hafızadaki veri üzerinde işlem yapar.
- **Process'in çalışması tamamlandığında kullandığı hafıza alanı boşaltılır.**
- Kullanıcı programı çalışmadan önce ve çalışması süresince farklı aşamalardan ve/veya durumlardan geçer.

Giriş

Adres binding

- Kaynak programda adres genellikle semboliktir (**count**).
- Compiler bu adresleri yeniden yerleştirilebilir (**relocatable**) adreslere dönüştürür (Örn.: program başlangıcından itibaren **14.byte**).
- Linkage editör veya loader, bu adresleri mutlak (**absolute**) adreslere dönüştürür (Örn.: **74014**).

9

Giriş

Adres binding

- Komutların veya verilerin hafıza adreslerine bağlanması (binding) farklı sekillerde olabilir:
 - **Compile time:** Compile aşamasında kodun hafızada yerleşeceği yer bilinirse mutlak code (**absolute code**) oluşturulabilir. Yerleşeceği hafıza alanı değiştirse yeniden compile edilmesi gereklidir. (MS-DOS işletim sistemi **.com** programlarını bu şekilde çalıştırır.)
 - **Load time:** Compile aşamasında programın yerleşeceği yer bilinmiyorsa, derleyici yeniden yerleştirilebilir (**relocatable code**) kod oluşturur.
 - **Execution time:** Eğer program çalışması sırasında bir segment'ten başka bir segment'e geçerse, adres binding **run-time**'da yapılır.

10

Giriş

Mantıksal ve fiziksel adres alanı

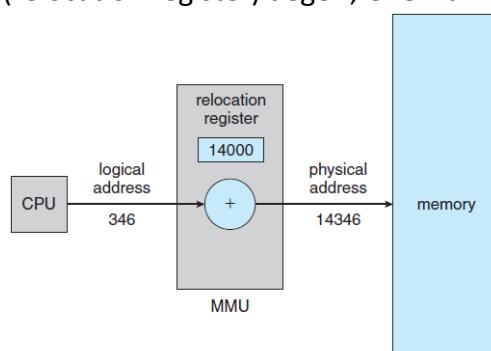
- CPU tarafından oluşturulan adres **mantıksal adres (logical address)** olarak adlandırılır.
- Hafıza biriminin gördüğü adres **fiziksel adres (physical address)** olarak adlandırılır.
- **Compile-time** veya **load-time** adres binding işlemleri **mantıksal** veya **fiziksel adres** üretir.
- **Execution-time** adres binding ise **sanal adres** (virtual address) (**page number + offset**).
- Run-time'da sanal adresin fiziksel adrese dönüştürülmesi **donanım bileşeni (memory-management-unit, MMU)** tarafından yapılır.
- **Base-register** fiziksel adrese dönüştürme için kullanılan **donanım bileşenidir**.

11

Giriş

Mantıksal ve fiziksel adres alanı

- **Base-register** (relocation register) değeri, **CPU'nun hesapladığı adrese eklendir.**



- Kullanıcı programı hiçbir zaman fiziksel adresi bilmez.
- Mantıksal adres **[0, 0+max]** aralığında, fiziksel adres **[R, R+max]** aralığındadır.

12

Giriş

Dynamic loading

- Bir programın tamamı hafızaya yüklenmez gerektikçe modül (blok) halinde yüklenir (**dynamic loading**).
- **Önce main program hafızaya yüklenir ve çalıştırılır.**
- Bir program parçası (**routine**) çalışırken başka routin'i çağrıduğunda, **hafızada yüklü değilse loader tarafından yüklenir.**
- **Çok büyük boyuttaki programların çalıştırılması için hafıza yönetimi açısından fayda sağlar.**
- Sık kullanılmayan rutin'lerin (hata yordamları) hafızada sürekli bulunmasını engeller.

13

Giriş

Dynamic linking ve paylaşılan kütüphaneler

- **Dinamik bağlanan kütüphaneler** sistem kütüphaneleridir (dil kütüphanesi) ve **kullanıcı programına çalışırken bağlanır.**
- Bazı işletim sistemleri statik bağlamayı destekler ve binary programa loader tarafından bağlanır.
- Her kütüphane için küçük bir **kod parçası (stub)** yükleneceği **uygun hafıza alanını** gösterir.
- Tüm programlar aynı kütüphaneyi kullanır.
- Kütüphanelerde yapılacak güncellemeler tüm kullanıcı programlarına kolaylıkla yansıtılır.

14

Konular

- Giriş
- **Swapping**
- Bitişik hafıza atama
- Segmentation
- Paging

15

Swapping

- Bir process çalışmak için hafızada olmak zorundadır.
- Bir process geçici olarak **diske (backing store)** aktarılabilir ve tekrar **hafızaya alınabilir (swapping)**.
- **Ready queue** (hazır kuyruğu), CPU'da çalıştırılmak üzere bekleyen process'leri tutar.
- **CPU scheduler** bir process'i çalıştırımıya karar verdiğiinde **dispatcher'ı** çağrıır.
- Dispatcher, çalışacak process'in hazır kuyruğunda olup olmadığını kontrol eder ve kuyrukta ise çalıştırır.
- Kuyrukta değilse ve **hafızada yeterli yer yoksa başka bir process'i hafızadan atar (swap out)** ve istenen process'i yükler (**swap in**).

16

Swapping

- İki process'in yer değiştirmesi context-switch işlemini gerektirir ve uzun süre alır.
- 100 MB'lık bir process'in 50MB/sn hızındaki bir diske kaydedilmesi için 2sn gerekir. İki process'in yer değiştirmesi 4sn süre alır.
- Process'lerin **dinamik hafıza gereksinimleri** için `request_memory()` ve `release_memory()` sistem çağrıları kullanılır.
- Bir process'in **swap out** yapılabilmesi için **tüm işlemlerini bitirmesi zorunludur**.
- Bir process I/O kuyruğunda bekliyorsa veya başka bir işlem sonucunu bekliyorsa swap out yapılamaz.
- Modern işletim sistemleri **hafıza eşik değerinin altına düşmeden swapping yapmaz**.

17

Swapping

Mobil sistemlerde swapping

- Mobil sistemler swapping işlemini desteklemez.
- Mobil cihazlar kalıcı saklama birimi olarak **hard disk yerine flash bellek** kullanır.
- Flash belleklerde yazma sayısı limiti vardır.
- Mobil cihazlarda, **main memory ile flash bellek arasındaki throughput değeri düşüktür**.
- Apple **iOS** işletim sistemi, **uygulamalardan hafızayı boşaltmasını ister**.
- Read-only veri sistemden atılır ve sonra flash bellekten tekrar yüklenir.
- Değişebilen veriler (stack) hafızadan atılmaz.
- **Android** işletim sistemi, **yeterli hafıza alanı yoksa bir process'i sonlandırır ve durum bilgisini flash belleğe kaydeder**.

18

Konular

- Giriş
- Swapping
- Bitişik hafıza atama
- Segmentation
- Paging

19

Bitişik hafıza atama

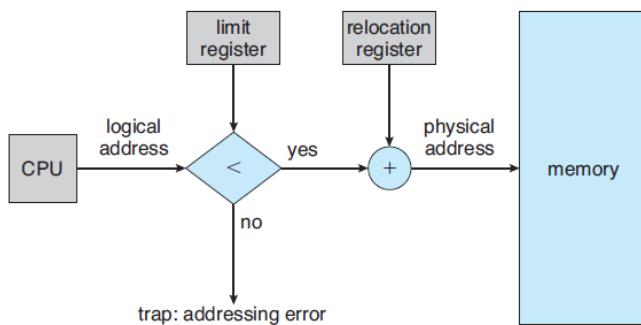
- **Main memory hem işletim sistemini hem de kullanıcı programlarını yerleştirmek zorundadır.**
- Bitişik hafıza atama yönteminde **hafıza iki parçaaya ayrılır:**
 - İşletim sisteminin yerleştiği kısım
 - Kullanıcı process'lerinin yerleştiği kısım
- İşletim sistemi **hafızanın başlangıcına veya sonuna yerleşebilir.**
- Interrupt vector table düşük adrese veya yüksek adrese yerleştirilebilir.
- İşletim sistemi ile interrupt vector tablosu genellikle aynı tarafa yerleştirilir.
- **Bitişik hafıza atama yönteminde bir process için ayrılan alan tek bölümden oluşur** ve sonraki process için ayrılan yere kadar devam edebilir.

20

Bitişik hafıza atama

Hafıza alanı koruma

- Bir process'in sahip olmadığı hafıza alanına erişimini engellemek gerekir.
- Sistemde **relocation register** ve **limit register** ile her process'e kendisine ait hafıza alanı ayrılabilir.
- CPU tarafından istenen her adresin **[base, base + limit]** arasında olup olmadığı kontrol edilir.



21

Bitişik hafıza atama

Hafıza alanı atama

- Bir process'e **hafıza alanı atama** işletim sisteme göre farklı şekillerde yapılabilir.
- Hafıza çok sayıda **sabit boyutta** küçük **parçaya ayrılabilir** (**fixed-sized partitions**) ve **her parça bir process'i içerebilir** (**multiple partition**).
- Multiprogramming sistemlerde **eşzamanlı çalışan program sayısı partition sayısına bağlıdır**.
- Bir **partition boşaldığında**, hazır kuyruğunda **bekleyen bir process** seçilerek **partition atanır**.
- IBM OS/360 işletim sistemi kullanmıştır günümüzde kullanılmamaktadır.
- Değişken parçalı (**variable-partition**) yönteminde **işletim sistemi hafızanın boş ve dolu olan parçalarını bir tabloda tutar**.
- Bu yöntemde, **her process'e farklı boyutta parça ayrılabilir**.

22

Bitişik hafıza atama

Hafıza alanı atama

- Bir process sisteme girdiğinde **ihtiyaç duyacağı kadar hafıza alanı ayrılabılırse hafızaya yüklenir** ve CPU'yu beklemeye başlar.
- Bir process sonlandığında ise ayrılan **hafıza alanı serbest bırakılır**.
- Herhangi bir anda, işletim sisteminde **kullanılabilir hafıza blokları listesi** ile **process'lerin giriş kuyruğu kümeleri** vardır.
- Kuyruğun başındaki process için **kullanılabilir yeterli alan yoksa beklenir** veya **kuyruktaki process'ler taranarak boş alana uygun olan varsa seçilir**.
- **Hafızaki boş alanların birleştirilmesi, process'e uygun alanın oluşturulması, serbest bırakılan alanların birleştirilmesi** işlemleri **dynamic storage allocation problem** olarak adlandırılır.

23

Bitişik hafıza atama

Hafıza alanı atama

- Dynamic storage allocation problemi için **3 farklı çözüm kullanılabilir**:
 - **First fit**: Yeterli boyuttaki **ilk boş alan atanır** ve listede kalan kısım aranmaz.
 - **Best fit**: Yeterli boyutta alanların **en küçüğü seçilir**. **Tüm liste aranır**.
 - **Worst fit**: Yeterli boyuttaki alanların **en büyüğü seçilir**. **Tüm liste aranır**.
- Simülasyonlarda, **alan atama süresinin first fit ile, hafıza alanı kullanma verimliliğinin best fit ile daha iyi olduğu** görülmüştür.
- **First fit** yöntemi best fit ve worst fit'e göre **daha kısa sürede atama gerçekleştirilmektedir**.

24



Bitişik hafıza atama

Fragmentation

- Process'ler hafızaya yüklenirken ve atılırken **hafıza alanları sürekli parçalanır (fragmentation)**.
- Bir process için yeterli alan olabilir, ancak bunlar **küçük parçalar halinde dağılmış durumda olabilir**.
- En kötü durumda her iki process arasında boş kısım olabilir.
- First fit ile yapılan istatistiksel analize göre, **N tane kullanılmış blok için $N/2$ tane boş blok oluşur**.
- Bu durumda hafızanın 1/3 kısmı kullanılamaz. **Buna %50 kuralı (50-percent rule)** denir.
- Fragmentation çözümünde küçük bloklar yer değiştirilerek büyük blok elde edilir (**fazla süre gerektirir**).
- **Segmentation** ve **paging** yaklaşımları fragmentation çözümünde etkindir.



Konular

- Giriş
- Swapping
- Bitişik hafıza atama
- **Segmentation**
- Paging

Segmentation

- **Segmentation** yaklaşımında, her segment bir isme ve uzunluğa sahiptir.
- Bir **mantıksal adres**, **segment adı** ile **offset** (segment içerisindeki konumu) **değerini belirler**.

<segment number (ad), offset>

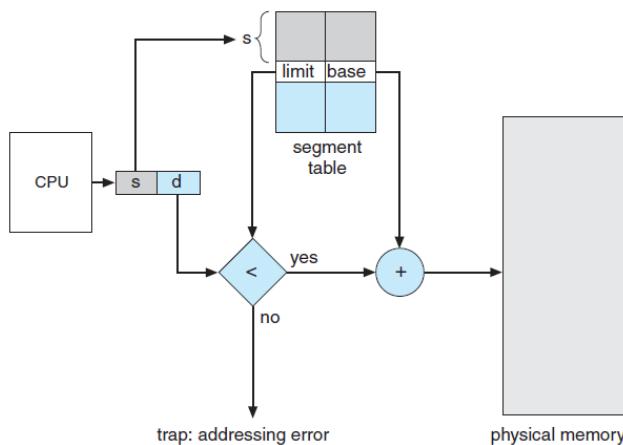
- Bir C derleyicisi aşağıdakiler için ayrı ayrı segment oluşturabilir:
 - Program kodu
 - Global değişkenler
 - Heap (nesneler yerleştirilir)
 - Stack (thread'ler kullanır, lokal değişkenler, call/return)
 - Standart C kütüphanesi
- Derleme sırasında **derleyici segment atamalarını gerçekleştirir**.

27

Segmentation

Segment adresleme donanımı

- Segment ve offset adresiyle **iki boyutlu adresleme** yapılır.
- **Hafıza adresleri tek boyutludur** ve dönüştürme işlemi gereklidir.

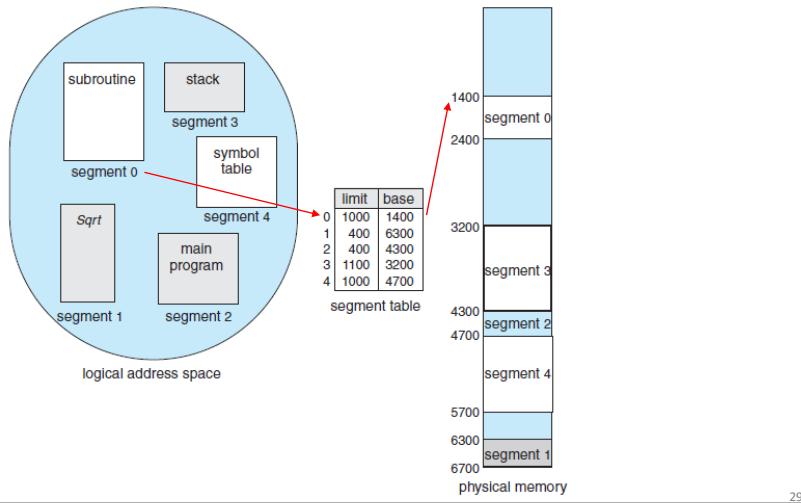


28

Segmentation

Segment adresleme donanımı

- Şekilde 5 segment vardır ve aşağıdaki gibi yerleştirilmiştir.



29

Konular

- Giriş
- Swapping
- Bitişik hafıza atama
- Segmentation
- Paging

30

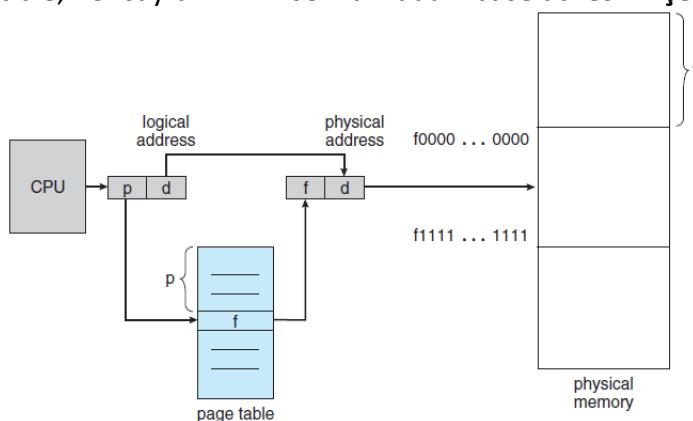
Paging

- Segmentation ile bir process'e atanın fiziksel adres alanının bitişik olmamasına izin verilir.
- **Paging** ile segmentation'da olduğu gibi process'lere bitişik olmayan hafıza adresleri atanabilir.
- Paging yönteminde,
 - Fiziksel hafıza **frame** adı verilen küçük bloklara bölünür.
 - Mantıksal hafıza ise aynı boyutta **page** adı verilen bloklara bölünür.
- Bir process çalıştırılacağı zaman **kaynak kodu diskten alınarak hafızadaki frame'lere yerleştirilir.**
- Mantıksal adres alanı ile fiziksel adres alanı birbirinden ayrılmış durumdadır.

31

Paging

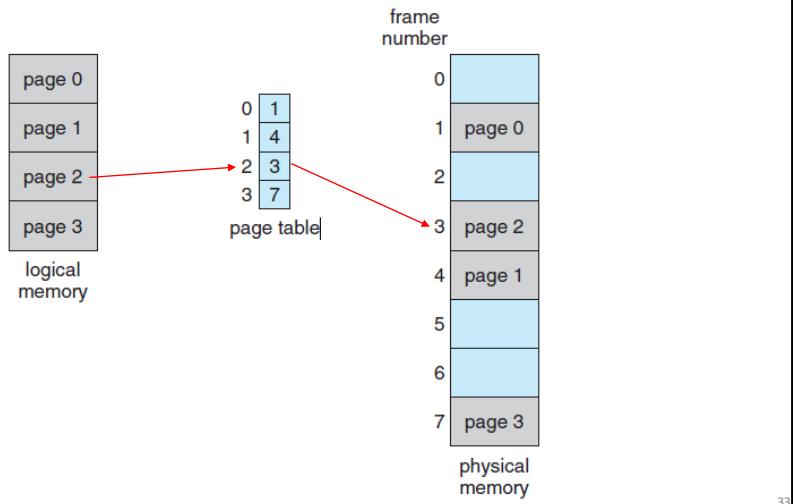
- CPU tarafından oluşturulan **her adres iki parçaya ayrılır: page number(p) ve page offset(d).**
- **Page number, page table** içerisindeki indeks değeri için kullanılır.
- **Page table**, her sayfanın fiziksel hafızadaki **base adresini içerir.**



32

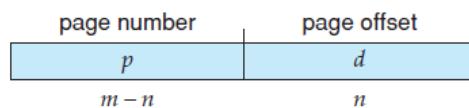
Paging

- Page size donanım tarafından mikroişlemci mimarisinde belirlenir.
- Page size 512 byte ile 1 GB arasında olabilir.



Paging

- Mantıksal adres boyutu 2^m , fiziksel adres (offset) boyutu 2^n byte ise, **sayfa numarası** için soldaki **m-n bit**, **offset değeri** için sağdaki **n bit** alınır.
- Aşağıda örnek mantıksal adres görülmektedir.



- **p** page table içindeki **indeks**, **d** ise sayfadaki **displacement** değeridir.

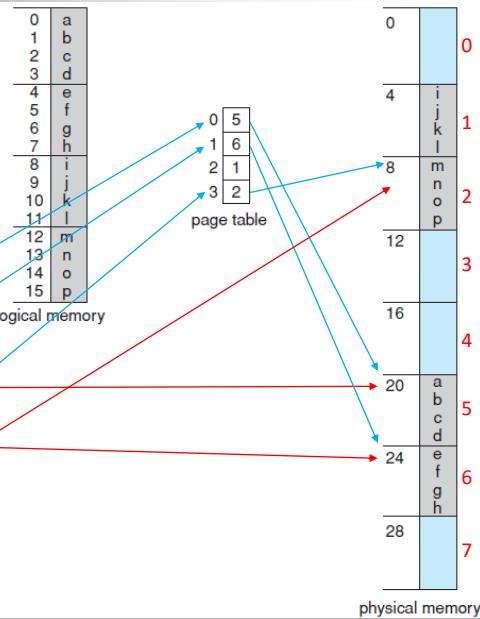
33

34

Paging

- Fiziksel hafızada **8 sayfa** var ve toplam 32 byte boyutundadır.
- **Page size = 4 byte**
- **Mantıksal adres boyutu m = 4 bit**
- **Offset adresi n = 2 bit**
- Mantıksal adres = 0 için **page = 0** ve **offset adres = 0** olur.
- Mantıksal adres = **0000**
- Mantıksal adres = **0100**

Mantıksal adres = 13 (**1101**)
Fiziksel adres = $(2 \times 4) + 1 = 9$



Paging

- **Paging ile fragmentation oluşabilir.**

Page size = 2048 byte

Process size = 72766 byte

Gerekli alan = 35 sayfa + **1086 byte**

1086 byte 36. sayfaya yerleştirilir.

Kullanılmayan alan $2048 - 1086 = 962$ byte

- **36. sayfada 962 byte boş alan kalır.**
- En kötü durumda 1 byte kalır ve ayrı sayfaya yerleştirilir.
- Boş alan $2048 - 1 = 2047$ byte olur.

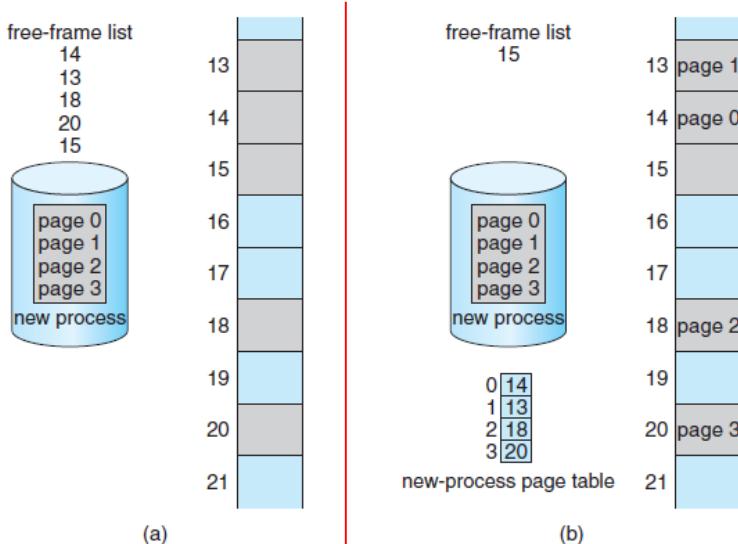
Paging

- 32-bit CPU'da genellikle **32-bit ile page table adresi verilir.**
- 2^{32} adet fiziksel page frame bulunur.
- **Bir frame boyutu 4 KB (2^{12}) ise,**
Toplam adreslenebilir fiziksel hafıza = $2^{32} * 2^{12} = 2^{44}$ olur (16 TB).
- Bir process sisteme çalışmak için geldiğinde, gerekli sayfa sayısı belirlenir.
- **Process'in her sayfası bir frame'e ihtiyaç duyar.**
- Toplam n sayfa varsa, en az n tane frame boş olmalıdır.
- Her sayfa bir frame'e yerleştirilir ve **frame numarası page table'a kaydedilir.**
- **Programcı process'in adresini tek ve bitişik olarak görür.** Frame eşlestirmesini işletim sistemi yapar.

37

Paging

- Şekilde bir process'e ait sayfaların hafızaya yerleştirilmesi görülmektedir.



38

Paging

Translation look-aside buffer (TLB)

- Her işletim sistemi page table saklamak için kendine özgü yöntem kullanır.
- Bazı işletim sistemleri **her process için ayrı page table kullanır**.
- **Her page table için pointer** ayrı bir register'da tutulur.
- Her page table için **bir grup register** oluşturulur.
- **Modern bilgisayarlarda page table çok büyütür** (Örn.: 1 milyon giriş).
- Bu durumda page table için register oluşturulması mantıklı değildir.
- Page table'ın hafızada tutulması halinde, **her adres değişikliğinde hafıza erişimi gerekli olur** (performans düşer).
- Mikroişlemcilerde, **küçük boyutta ve hızlı donanımsal önbellek (translation look-aside buffer)** ile bu problem çözülür.

39

Paging

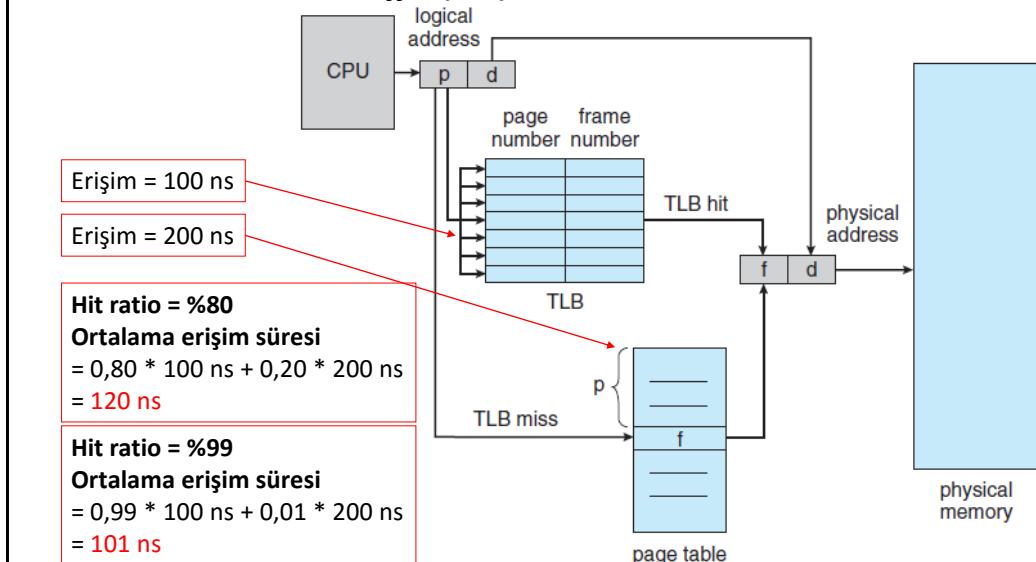
Translation look-aside buffer (TLB)

- TLB içerisindeki **her giriş** satırı **page number** ile **frame number** değerlerini tutar.
- Bir mantıksal adres geldiğinde, **page number** tüm TLB içerisinde aranır (**full associative**).
- Page number değeri bulunursa, **ilgili satırdaki frame number** değeri **alınarak** hafızada **ilgili sayfaya gidilir**.
- TLB içerisinde bulunamayan page number için page table'a gidilir.
- Page table'dan alınan frame number ile page number değeri TLB'ye kaydedilir.
- **TLB dolu ise replacement algoritması** (**least recently used, round robin, random**) ile seçilen satır silinerek yerine yazılır.

40

Paging

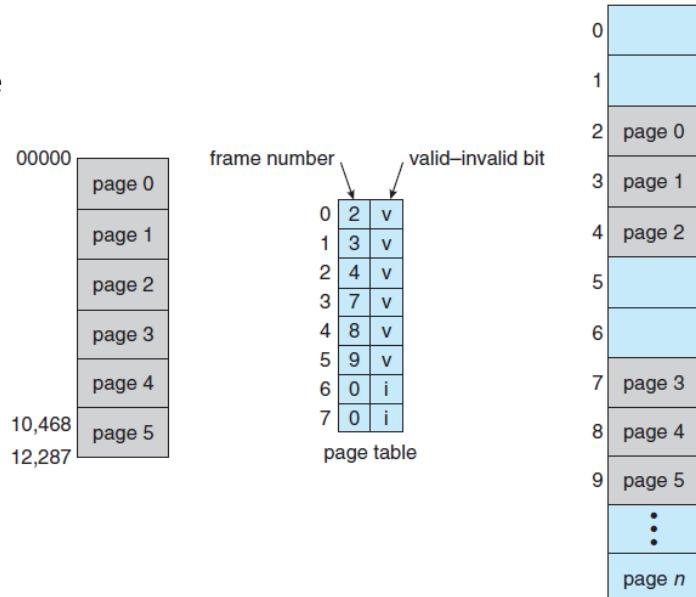
Translation look-aside buffer (TLB)



Paging

Sayfa koruma

- Page table içerisinde her frame için **1 bit protection biti (valid-invalid)** eklenebilir.



Paging

Sayfa paylaşımı

- Birden fazla kullanıcı aynı sayfayı paylaşıp kullanabilir.
- Şekilde 3 process aynı text editörünü farklı verilerle kullanmaktadır.
- Kodun değişmez olması (reentrant / pure code) gereklidir.

