-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

-----------------------------------------------1. UNITE----------------------------------------------

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

\*\* İşletim Sistemi = Bilgisayar kullanıcısı ile donanım arasında aracı rolü oynayan programdır.

Hedefleri:

- Kullanıcı programlarını çalıştırmak ve kullanıcı problemlerini çözmeyi kolaylaştırmak

- Bilgisayar sistemini rahat kullanılır hale getirmek

- Bilgisayar donanımını etkin kullanmak

\* İşletim sistemi bir kaynak paylaştırıcıdır; Bütün kaynakları yönetir. Etkin ve adil kaynak

kullanımı arasında dengeyi sağlar, çelişen istekleri karara bağlar.

\* İşletim sistemi bir kontrol programıdır; Programların çalışmasını denetleyerek hataları ve

uygun olmayan kullanımı önler.

\*\* Kernel (çekirdek) = Bilgisayarda sürekli çalışan tek program

\*\* Bootstrap Programı = İlk açılışta veya yeniden başlatma sonucu yüklenir. Genellikle ROM veya

EPROM içinde saklanır ve firmware olarak bilinir. Sistemi her yönüyle başlangıç durumuna getirir.

İşletim sistemi çekirdeğini yükler ve yürütmeye başlar.

----------------------------------------------------------------------------------------------------

\*\* Interrupt (Kesme) = Yazılım sistem çağrılarını kullanması veya kernel ile haberleşmesine olanak

sağlayan bir mekanizmadır. User space uygulamasının kernal space çağrıları ile gerçekleştirilmesi

için kullanılır. Kesme, genellikle dednetimi kesme servis rutinine aktarır. Bu aktarım, kesme vektörü yoluyla yapılır.

\* Interrupt Vector (Kesme Vectörü) = Bütün servis rutinlerinin adreslerini içerir.

Kesme mimarisi kesilen komutun adresini kaydetmelidir. Hata veya kullanıcı isteği sonucu yazılım

tarafından üretilen kesmeye trap (tuzak) veya exception (istisna) denir.

\* Interrupt Driven = İşletim sistemi kesme yönlendirmelidir.

\* Polling = Birşeyi sürekli kontrol edip olmasını bekleme

-----------------------------------------------------------------------------------------------------

\* Ana Bellek = CPU'nun doğrudan erişebildği tek büyük depolama ortamı. Rastgele erişimlidir. Genellikle

uçucudur (Volatile).

\* İkincil Depolama = Ana belleğin uzantısı olarak büyük kapasiteli kalıcı depolama sunar.

Depolama Sistemleri Hiyerarşisi

- Hız

- Maliyet

- Uçuculuk/kalıcılık (Volatile)

\*\* Caching = Bilgiyi daha hızlı olan depolama birimine kopyalamak. Ana bellek ikincil depo için bir

cache olarak görülebilir.

\*\* Device Driver (Cihaz Sürücüsü) = Denetçi ve çekirdek arasında tek tip arayüz sağlar.

-----------------------------------------------------------------------------------------------------

Çok işlemcili Sistemler = Bunlara paralel sistem, sıkı bağlı (tightly-coupled) sistemler de denir.

Avantajları:

- Daha fazla üretilen iş (throughput)

- Ölçek ekonomisi (Sistem büyüdükçe birim maliyetin düşmesi)

- Daha fazla güvenilirlik (Nadiren bozulma veya hata toleransı)

İkiye ayrılır:

- Asimetrik Çokişlemcili Mimari = Her işlemci özel bir işe atanır.

- Simetrik Çokişlemcili Mimari = Her işlemci bütün işleri yapar.

-----------------------------------------------------------------------------------------------------

Kümelenmiş (Clustered) Sistemler = Çokişlemcili sistemlere benzer ama birlikte çalışan birden fazla

sistem vardır. Yüksek erişilebilirlik özelliğine sahip bir servis sağlar, arızaları atlatır.

\* Asimetrik Kümeleme = Bir makine sıcak yedek olarak bekler.

\* Simetrik Kümeleme = Uygulama çalıştıran birden fazla düğüm vardır, bunlar birbirlerini izler.

\* Storage-Area Network (SAN): Depo Paylaşımı

\* Distributed Lock Manager (DLM): Dağıtık Kilit Yöneticisi. Çatışan işlemleri önler.

\* High-Performance Computing (HPC): Yüksek performanslı hesaplama.

-----------------------------------------------------------------------------------------------------

Multiprogramming (Çoklu programlama) = Verimlilik için gereklidir. Tek kullanıcı CPU ve I/O cihazlarını

sürekli meşgul tutamaz. Çoklu programlama, işleri organize ederk CPU'nun her zaman yürütecek bir işi

olmasını sağlar. Sistemdeki işlerin bir alt kümesi bellekte tutulur. Bir iş seçili ve çalıştırılır.

(job scheduling). İşin beklemesi gerekirse, OS başka işe geçiş yapar.

Multitasking (Çoklu Görev) (Timesharing - Zaman Paylaşımı) = CPU'nun çok sıkça iş geçişi yaparak

kullanıcıya çalışan her işle etkileşme imkanı verdiği durumdur.

\* CPU Scheduling = Aynı anda çalışmaya hazır birden çok iş varsa.

\* Swapping (Değiş-Tokuş) = Süreçler belleğe sığmazsa swapping ile girme-çıkma yönetilir.

\* Sanal Bellek = Tümüyle bellekte olmayan süreçlerin yürütülmesini sağlar.

\* Dual-Mode Operation = OS'in kendisini ve diğer sistem bileşenlerini korumasını sağlar.

\* Mode Bit = Sistemin kullanıcı kodu mu çekirdek kodu mu çalıştırdığını anlamayı sağlar(1=User,0=Kernel)

System call, modu çekirdek moduna değiştirir. Çağrıdan dönünce modu kullanıcı moduna geri alır.

\* Timer = Sonsuz döngüyü, kaynakları tek başına tüketen süreçleri engellemek için kullanılır.

-----------------------------------------------------------------------------------------------------

Süreç = Yürütülen bir programdır. Sistemdeki iş birimidir. Program pasif, süreç ise aktif varlıktır.

Süreç görevini tamamlamak için kaynaklara ihtiyaç duyar. Süreç sonlandırmak için, yeniden kullanılabilir

tüm kaynakların geri alınması gerekir.

- Tek parçalı süreçte; bir program counter (program sayacı) vardır ve yürütülecek bir sonraki komutun

konumunu belirtir. Süreç, komutları birer birer, sırayla ve tamamlanana kadar yürütür.

- Çok parçalı süreçte; her iş parçası (thread) başına bir program sayacı bulunur.

\*\* Multiplexing (Çoklama) = CPU'ları süreçler arasında düzenleyerek eşzamanlılığı sağlar.

-----------------------------------------------------------------------------------------------------

Bellek yönetimi etkinlikleri

\* Belleğin hangi parçalarının kim tarafından kullanılmakta olduğunu takip etmek

\* Hangi süreçlerin ve verilerin belleğe girip bellekten çıkacağına karar vermek

\* Bellek alanı tahsis etmek ve geri almak

-----------------------------------------------------------------------------------------------------

HARD DISK -> MEMORY -> CACHE -> REGISTER

\*\* Coherency (Bellek tutarlılığı) = CPU'lar cache'lerinde en güncel değere sahip olmalı

-----------------------------------------------------------------------------------------------------

OS'in bir amacı donanım cihazlarının kendilerine has özelliklerini kullanıcıdan gizlemektir.

I/O Subsystem görevleri

- Buffering = Arabelleğe kaydetme (Veri aktarılırken geçici olarak saklama)

- Caching = Verinin parçalarını depoda saklama

- Spooling = Bir işin çıktısının başka bir işin girdisiyle örtüşmesi

- Genel cihaz sürücü arayüzü

- Özel donanım cihazları için sürücüler

-----------------------------------------------------------------------------------------------------

Protection = OS'in süreç ve kullanıcıların kaynaklara erişimini denetlemek için tanımladığı mekanizma

Security = Sistemin iç ve dış saldırılara karşı savunulması

\* User ID = Erişim denetimi için o kullanıcının bütün dosya ve süreçleriyle ilişkilendirilir.

\* Group ID = Kullanıcı kümelerinin tanımlanmasına, süreç ve dosyalarla ilişkilendirilerek denetimin

yönetilmesine imkan verir.

\*\* Privilege Escalation (Yetki Yükseltme) = Kullanıcının daha fazla hakka sahip ID'ye geçmesini sağlar.

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

-------------------------------------------------2. UNITE--------------------------------------------

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

İşletim Sistemi Kullanıcı Servisleri

- Kullanıcı Arayüzü = Neredeyse tüm işletim sistemleri bir kullanıcı arayüzüne sahiptir. CLI, GUI..

- Program Çalıştırma = Sistem, bir programı hafızaya yükleyebilmeli çalıştırabilmeli ve sonlandırabilmeli

- I/O İşlemleri = Çalışan bir program I/O işlemi gerektirebilir.

\* İletişim = İşlemler bilgi alışverişinde bulunabilirler. İletişim ortak hafıza (shared memory) ile veya

mesaj gönderimi (message passing) ile sağlanabilir. Paketler, işletim sistemi tarafından taşınır.

\* Hata Tespiti = İşletim sistemi sürekli olarak olası hataları takip eder.

\* Kaynak Paylaştırma = Birden fazla kullanıcı ya da birden fazla iş aynı anda çalıştırıldığında kaynaklar her birine adil şekilde paylaştırılmalıdır.

- Özel Pay Alma Kodu Kullananlar = CPU zamanı, ana hafıza, dosya kayıt birimi

- İsteme ve İade Etme Kodu Kullananlar = I/O Cihazları

\* Hesap Tutma = Hangi kullanıcının hangi tip sistem kaynağının ne kadarını kullandığını takip etmek

\* Koruma = Sistem kaynaklarına her tür erişimin kontrol altında tutulmasını gerektirir.

\* Güvenlik = Sisteme dışarıdan erişmek isteyenlerin kimlik doğrulamasını yapmaktan I/O cihazlarının

geçersiz erişim isteklerine engel olunmasına kadar çeşitlilik gösterir.

\*\* System Çağrıları (System Calls) = İşletim sistemi tarafından sunulan servisler için programlama

arayüzüdür. Genelde direkt çağrılmazlar, API aracılığıyla kullanılırlar.

\*\*\* Sistem Çağrıları yerine neden API'ler kullanılır ? Basitlik ve Taşınabilirlik

İşletim Sistemine parametre göndermeyi sağlayan üç genel yöntem:

1 - Parametreleri yazmaçlar (registers) içinde göndermek

2 - Parametreler hafızada bir blokta veya tabloda tutulur ve blokun adresi yazmaç ile gönderilir.

3 - Parametreler program tarafından yığına (stack) atılır ve işletim sistemi tarafından yığından çekilir.

İşletim sistemi tasarımında kullanıcı ve sistem hedefleri

- Kullanıcı Hedefleri = İşletim sistemi rahat kullanılmalı, kolayca öğrenilmeli, tutarlı, güvenli, hızlı ol

- Sistem Hedefleri = İşletim sistemi kolay tasarlanmlaı ve gerçekleştirilmeli, bakımı kolay yapılmalı. Esnek olmalı, tutarlı, hatasız ve verimli çalışmalı

\* İlke (Policy) = Ne yapılmalı ?

\* Mekanizma (Mechanism) = Nasıl yapılmalı ?

Katmanlı Yaklaşım = İşletim sistemi belli sayıda katmana (layer) ayrılır. Her katman alt seviyedeki

diğer katmanların üzerine yerleşir. En alt katman (layer 0 ) donanım katmanıdır. En üst katman (layer N)

kullanıcı arayüzüdür. Sistemin modüler olması için, üst katmanlar sadece altındaki katmanların

fonksiyonlarını ve servislerini kullanmalıdır.

Mikroçekirdek Sistem Yapısı = Çekirdekte gerçekleşeyen şeylerden taşınabilir olan herşeyi user moduna taşır. Modüller arasindaki iletişimi mesaj gönderme yoluyla gerçekleşir.

Faydaları:

- Mikroçekirdeğin özelliklerini arttırmak daha kolay

- İşletim sistemini yeni mimarilere geçirmek daha kolay

- Daha tutarlı ve güvenli

Zararları:

- Çekirdek modu ve kullanıcı modu arasında iletişimin getirdiği aşırı yükleme, performans sorunu yaratır

Modüller = Birçok işletim sistemi çekirdek modullerini kullanır. Nesneye yönelik yaklaşımı kullanır. Tüm

temel bileşenler ayrıdır. Birbirleriyle belirli arayüzler üzerinden iletişim kurarlar. Her biri gerektirğinde çekirdeğe yüklenebilir durumdadır. Katmanlara benzer fakat daha çok esneklik sağlarlar.

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

-------------------------------------------------3. UNITE--------------------------------------------

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

İşlem = Çalışmakta olan bir programdır. Aşağıdakileri içermelidir:

- Program sayacı (Program Counter)

- Yığın (Stack)

- Veri Bölümü (Data section)

İşlem Durumları

- yeni (new): İşlem oluşturuldu

- çalışıyor (running): İşlem komutları çalıştırılıyor

- bekliyor (waiting): İşlem bir olayın gerçekleşmesini bekliyor

- hazır (ready): İşlem bir işlemciye atanmayı bekliyor

- sonlandırılmış (terminated): İşlem çalışmayı bitirmiş

\* PCB = İşlem Kontrol Bloğu, herhangi bir işlem ile ilgili bilgileri tutar.

İşlem Zamanlama Kuyrukları

- İş kuyruğu (job queue) = Sistemdeki tüm işlemlerin kümesi

- Hazır kuyruğu (Ready queue) = Ana hafızadaki, tüm işlemlerin kümesi

- Cihaz kuyrukları (Device queues) = Bir I/O cihazını kullanmayı bekleyen işlemler kümesi

Zamanlayıcılar

- Uzun Vadeli Zamanlayıcılar (Long-term Scheduler) = İş zamanlayıcısı. Hangi işlemlerin hazır kuyruğuna

alınması gerektiğine karar verir. Çok sık çalıştırılır (milisaniye).

- Kısa Vadeli Zamanlayıcılar (Short-term Scheduler) = CPU zamanlayıcısı. Sıradaki hangi işlemin CPU

tarafından çalıştırılacağına karar verir. Çok sık çalıştırılmaz (saniye, dakika).

Context Switch (Bağlamsal Anahtarlama) = CPU başka bir işleme geçerken, sistem eski işlemin durumunu

kaydetmeli ve yeni işlemin kaydedilmiş durumunu yüklemelidir. Bir işlemin bağlamı (context) PCB'de tutulur.

Bağlamsal Anahtarlama için harcanan zaman ek yüktür. Değişim sırasında işlemci kullanıcın işine direkt

yarayan bir iş yapmamaktadır. Harcanan zaman donanım desteğine bağlıdır.

İşlem Oluşturma

Ana işlem (parent process) çocuk işlemleri (children processes) oluşturur. Çocuk işlemlerde yeni işlem oluşturabileceğinden, sistemde işlemlerin bir ağacı oluşur. Genellikle işlemler işlem belirteci (process identifier - pid) kullanılarak birbirlerinden ayrılırlar ve yönetilirler.

Kaynak paylaşımı 3 olasılık:

- Ana işlem ve çocuklar tüm kaynakları paylaşırlar

- Çocuklar ana işlemin kaynaklarının belli bir alt kümesini paylaşır

- Ana işlem ve çocuklar kaynak paylaşmazlar

Çalıştırma

- Ana işlem ve çocuklar aynı anda çalışır.

- Ana işlem, çocuk işlemler sonlanana kadar bekler

Hafıza alanı (address space)

- Çocuk ana işlemin kopyasına sahip

- Çocuk adres alanına yeni bir program yükler

\* Fork = Fork sistem çağrısı yeni bir işlem oluşturur

\* Exec = Exec sistem çağrısı, fork sistem çağrısı ile yeni işlem oluşturulduktan sonra, işlemin hafıza alanına yeni bir program yüklemek için kullanılır.

İşlem Sonlandırma (Process Termination) = İşlemler son komutlarını çalıştırdıktan sonra işletim sistemine kendilerini silmelerini isterler (exit). İşlemin dönüş değeri çocuktan ana işleme döndürülür (wait ile). İşlemin kaynakları işletim sistemi tarafından geri alınır. Eğer;

- Çocuk kendine ayrılan kaynakları aşmışsa

- Çocuk işleme atanan göreve artık ihtiyaç yoksa

- Eğer ana işlem sonlandırılıyorsa, Ana işlem çocuk işlemlerin çalışmasını sonlandırabilir (abort). Bazı

işletim sistemlerinde ana işlem sonlandırıldıktan sonra çocukların çalışmaya devam etmesine izin vermez.

Peşpeşe sonlandırma (cascading termination) ile tüm çocuk işlemler sonlandırılır.

İşlemler Arası İletişim (Interprocess Communication) = Bir sistemdeki işlemler bağımsız (independent ) ya da işbirliği yapıyor (cooperating) olabilir. İşbirliği yapan işlemler birbirlerini etkileyebilirler veya veri paylaşabilirler. İşlemler bilgi paylaşımı, işlem hızını artırmak, modülerlik sağlamak için işbirliği

yaparlar. İşbirliği yapan işlemler işlemler arası iletişime (interprocess communication - IPC) ihtiyaç duyar. İşlemlerin iletişim kurması ve eylemlerini senkronize etmelerini sağlayan mekanizma. IPC’nin iki modeli:

- Paylaşımlı bellek (shared memory)

- Mesaj gönderme (message passing)

İletişim bağlantısının uygulanması:

Fiziksel

- Shared memory

- Hardware bus

- Network

Mantıksal

- Direct or indirect

- Synchronous or asynchronous

- Automatic or explicit buffering

Mesaj sistemi, işlemler birbirleriyle paylaşımlı değişkenlere bağlı kalmaksızın haberleşir. IPC mekanizması iki işlemi sağlar:

- send (mesaj gönderme) = Mesaj boyutu sabit yada değişken olabilir.

- receive (mesaj alma)

Üretici-Tüketici Problemi (Producer-Consumer Problem)

İşbirliği yapan işlemler için problem örneği: üretici işlem bilgi üretirken, tüketici işlem üretilen bilgiyi tüketir.

- Sınırsız tampon bellek (unbounded-buffer): tampon bellek için herhangi bir pratik sınır getirmez.

- Sınırlı tampon bellek (bounded-buffer): Sınırlı boyutta bir tampon bellek kullanıldığını varsayar.

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

-------------------------------------------------4. UNITE--------------------------------------------

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

Tek ve Çok İş Parçacıklı İşlemlerin Faydaları:

- Cevap Verebilirlik (Responsiveness)

- Kaynak Paylaşımı (Resource Sharing)

- Ekonomi (Economy)

- Ölçeklenebilirlik (Scalability)

Çok Çekirdekli Programlama: Çok çekirdekli sistemler programcıları çok iş parçacıklı uygulamalar yazmaya

zorluyor. Bu konudaki zorluklar:

- Aktiviteleri bölmek (dividing activities)

- Denge (balance)

- Bilgileri Ayırmak (Data Splitting)

- Veri Bağımlılığı (Data dependency)

- Test ve Hata ayıklama (testing and debugging)

\* Kullanıcı İş Parçacıkları: İş parçacığı yönetimi kullanıcı seviyesinde tanımlı iş parçacığı kütüphaneleri ile sağlanır.

\* Çekirdek İş Parçacıkları: Çekirdek tarafından desteklenirler.

ÇOKLU İŞ PARÇACIĞI MODELLERİ

- Çoktan-Teke (Many-to-One)

- Teke-Tek (One-to-One)

- Çoktan-Çoka (Many-to-Many)

\* Çoktan-Teke = Pek çok kullanıcı seviyesindeki iş parçacığı tek bir çekirdek iş parçacığı ile eşleşir.

\* Teke-Tek = Her bir kullanıcı seviyesi iş parçacığı tek bir çekirdek iş parçacığı ile eşleşir.

\* Çoktan-Çoka = Pek çok kullanıcı seviyesi iş parçacığı pek çok çekirdek iş parçacığı ile eşleşir. İşletim

sisteminin yeterince çekirdek iş parçacığı oluşturmasını sağlar.

\* İki-Seviye Model = Çoktan-çoka benzer. Farklı olarak kullanıcı iş parçacığıının çekridek iş parçasına bağlanmasına izin verir (bound to kernel thread).

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

-------------------------------------------------6. UNITE--------------------------------------------

-----------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------

\* CPU-I/O İşlem Döngüsü = Bir işlemin çalıştırılması birbirlerini takip eden CPU kullanımı ve I/O beklemsi döngüsünden oluşur.

\* CPU Burst = İşlemcinin bir processi işlemek için i/o wait gelene kadar ihtiyacı olan zaman aralığı. yani bir başka deyişle cpu'nun tek process için tek bi seferde harcadığı vakit.

\* I/O Burst = CPU'nun daha fazla yürütme için I/O'yi beklediği zamandır. I/O burstten sonra süreç, bir sonraki CPU burst için hazır kuyruğa girer.

CPU Scheduling

Çalışmaya hazır şekilde hafızada bekleyen işlemlerden birini seçerek işlemciyi ona ayırır. İşlemci zamanlaması şu durumlarda gerçekleşebilir:

1. Çalışma durumundan (running state) bekleme durumuna (waiting state) geçişte

2. Çalışma durumundan hazır durumuna (ready state) geçişte

3. Bekleme durumundan hazır durumuna geçişte

4. İşlem sonlandığında

1 ve 4 durumlarında zamanlama: kesmeyen (nonpreemptive)

Diğer durumlarda: kesen (preemptive)

package Demo1;

class Runner extends Thread{

public void run() {

for(int i = 0 ; i<10 ; i++) {

System.out.println("Hello"+i);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class App {

public static void main(String[] args) {

Runner runner1 = new Runner();

runner1.start();

Runner runner2 = new Runner();

runner2.start();

}

}

-------------------------------------------------> Extend ile thread olusturma.

package Demo2;

class Runner implements Runnable{

public void run() {

for(int i = 0 ; i<10 ; i++) {

System.out.println("Hello"+i);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class App {

public static void main(String[] args) {

Thread t1 = new Thread(new Runner());

Thread t2 = new Thread(new Runner());

t1.start();

t2.start();

}

}

-----------------------------------------------------> runnable ile thread olusturma

package Demo3;

public class App {

public static void main(String[] args) {

Thread t1 = new Thread(new Runnable() {

@Override

public void run() {

for(int i = 0 ; i<10 ; i++) {

System.out.println("Hello"+i);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

});

Thread t2 = new Thread(new Runnable() {

@Override

public void run() {

for(int i = 0 ; i<10 ; i++) {

System.out.println("Hello"+i);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

});

t1.start();

t2.start();

}

}

-------------------------------------------------------------------------------> Farkli bir yazim sekli ile thread.

Bu threadler su sekilde calisir;

Hello0

Hello0

Hello1

Hello1

Hello2

Hello2

Hello3

Hello3

Hello4

Hello4

Hello5

Hello5

Hello6

Hello6

Hello7

Hello7

Hello8

Hello8

Hello9

Hello9

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

SADI HOCA SENKRON EDILMIS THREADLER.

package SadiHoca;

public class App {

public static void main(String args[]) {

Data d = new Data();

d.deger = 100;

d.bayrak = false;

Producer p = new Producer(d);

Consumer c = new Consumer(d);

Thread t = new Thread(c);

p.start();

t.start();

}

}

class Producer extends Thread {

Data d;

public Producer(Data d) {

this.d = d;

}

public void run(){

for(int i = 0; i<10;i++){

synchronized(d){

d.deger++;

System.out.println("uretici"+d.deger);

}

}

}

}

class Consumer implements Runnable {

Data d;

public Consumer(Data d) {

this.d = d;

}

public void run() {

for(int i = 0; i<10;i++){

synchronized(d){

d.deger--;

System.out.println("tuketici"+d.deger);

}

}

}

}

class Data {

int deger;

boolean bayrak;

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

SENKRON EDILMIS

package Synchronized;

public class App {

private int count = 0;

public synchronized void increment() {

count++;

}

public static void main(String[] args) {

App app = new App();

app.doWork();

}

public void doWork() {

Thread t1 = new Thread(new Runnable() {

public void run() {

for(int i =0; i<10000; i++) {

increment();

}

}

});

Thread t2 = new Thread(new Runnable() {

public void run() {

for(int i =0; i<10000; i++) {

increment();

}

}

});

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("Count is: " + count);

}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

VOLATILE

package Volatile;

import java.util.Scanner;

class Processor extends Thread{

private volatile boolean running = true;

public void run() {

while(running) {

System.out.println("Heloo");

try {

Thread.sleep(100);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

public void shutdown() {

running = false;

}

}

public class App {

public static void main(String[] args) {

Processor proc1 = new Processor();

proc1.start();

System.out.println("Press return to stop ...");

Scanner scanner = new Scanner(System.in);

scanner.nextLine();

proc1.shutdown();

}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

iki farkli sirketten birisini ceviriyor

import java.util.HashMap;

import java.util.Map;

public class ThreadExample {

static Map<Integer, String> shippingFirmIdNameMap = new HashMap<>();

static Integer SHIPPING\_FIRM\_ID = 101;

public static void main(String[] args) throws InterruptedException {

Thread thread1 = new Thread(() -> {

if(!shippingFirmIdNameMap.containsKey(SHIPPING\_FIRM\_ID)){

shippingFirmIdNameMap.put(SHIPPING\_FIRM\_ID, "UPS");

}

});

Thread thread2 = new Thread(() -> {

if(!shippingFirmIdNameMap.containsKey(SHIPPING\_FIRM\_ID)) {

shippingFirmIdNameMap.put(SHIPPING\_FIRM\_ID, "Ocean Network Express");

}

});

thread1.start();

thread2.start();

thread1.join();

thread2.join();

System.out.println(shippingFirmIdNameMap.get(SHIPPING\_FIRM\_ID));

}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

ThreadPool

package ThreadPool;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

class Processor implements Runnable{

private int id;

public Processor(int id) {

this.id = id;

}

public void run() {

System.out.println("Starting: " + id);

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

}

System.out.println("Completed: " + id);

}

}

public class App {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(2);

for(int i=0; i<5; i++) {

executor.submit(new Processor(i));

}

executor.shutdown();

System.out.println("All tasks submitted.");

try {

executor.awaitTermination(10,TimeUnit.DAYS);

} catch (InterruptedException e) {

}

System.out.println("All tasks completed.");

}

}

2 thread calisiyor 0. ve 1. calisiyor ve bitiyor sonra 2. basliyor ve 3. basliyor gibi gibi devam ediyor.

All tasks submitted.

Starting: 0

Starting: 1

Completed: 1

Completed: 0

Starting: 2

Starting: 3

Completed: 3

Completed: 2

Starting: 4

Completed: 4

All tasks completed.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

SharedMemory

package App;

public class ThreadExample {

public static void main(String[] args) {

SharedData sharedData = new SharedData();

// Okuma işlemi yapan thread

Thread readerThread = new Thread(() -> {

while (true) {

int data = sharedData.readData();

System.out.println("Okunan Veri: " + data);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// Yazma işlemi yapan thread

Thread writerThread = new Thread(() -> {

int counter = 0;

while (true) {

sharedData.writeData(counter++);

try {

Thread.sleep(1500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// Mod hesaplama işlemi yapan thread

Thread modCalculatorThread = new Thread(() -> {

while (true) {

int modResult = sharedData.calculateMod();

System.out.println("Mod: " + modResult);

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

});

// Thread'leri başlat

readerThread.start();

writerThread.start();

modCalculatorThread.start();

}

}

class SharedData {

private int data;

// Okuma işlemi

public synchronized int readData() {

return data;

}

// Yazma işlemi

public synchronized void writeData(int newData) {

System.out.println("Yazılan Veri: " + newData);

data = newData;

}

// Mod hesaplama işlemi

public synchronized int calculateMod() {

return data % 5;

}

}

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class DosyaIslemleri {

public static void main(String[] args) {

String okunacakDosyaAdi = "notlar.txt";

String yazilacakDosyaAdi = "notlar\_yeni.txt";

try (BufferedReader bufferedReader = new BufferedReader(new FileReader(okunacakDosyaAdi))) {

// Dosyadan okuma işlemi

String satir;

double toplamNot = 0;

int ogrenciSayisi = 0;

while ((satir = bufferedReader.readLine()) != null) {

// Her satırdaki notu toplam not'a ekle

double not = Double.parseDouble(satir);

toplamNot += not;

ogrenciSayisi++;

}

// Ortalama notu hesapla

double ortalamaNot = toplamNot / ogrenciSayisi;

System.out.println("Notların Ortalaması: " + ortalamaNot);

// Yeni bir dosyaya yazma işlemi

try (BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(yazilacakDosyaAdi))) {

bufferedWriter.write("Notların Ortalaması: " + ortalamaNot);

} catch (IOException e) {

System.err.println("Yazma hatası: " + e.getMessage());

}

} catch (IOException e) {

System.err.println("Okuma hatası: " + e.getMessage());

}

}

}

Heap ve stack, bilgisayar programları tarafından kullanılan bellek yönetimi konseptleridir. İşletim sistemi dersinde bu iki kavramın farkını anlamak önemlidir. İşte heap ve stack'in temel farkları:

1. **Yapıları ve Kullanımları:**
   * **Heap:** Genellikle dinamik bellek tahsisi için kullanılır. Program çalışma zamanında heap üzerinde bellek alanı talep eder ve bu alanı serbest bırakabilir. Yani, heap üzerinde oluşturulan bellek alanları programın ihtiyaçlarına bağlı olarak dinamik olarak değişebilir.
   * **Stack:** Genellikle yerel değişkenlerin, fonksiyon çağrılarının ve fonksiyon dönüş adreslerinin saklandığı bir bellek alanıdır. Bu bellek alanı, genellikle LIFO (Last In, First Out) mantığına göre çalışır.
2. **Bellek Yönetimi:**
   * **Heap:** Programcının kontrolünde olan bir bellek alanıdır. Bellek tahsisini ve serbest bırakmayı programcının yönetmesi gerekebilir. Bellek sızıntıları ve bellek bozulmaları gibi sorunlar, uygun yönetilmediğinde ortaya çıkabilir.
   * **Stack:** İşletim sistemi ve derleyici tarafından yönetilir. Otomatik olarak bellek tahsisi ve serbest bırakma işlemleri gerçekleştirilir.
3. **Bellek Yapısı:**
   * **Heap:** Dinamik bir bellek alanıdır ve bellek blokları rastgele yerleştirilebilir. Bu nedenle, bellek blokları arasında boşluklar olabilir.
   * **Stack:** Yığın bellek yapısı, tipik olarak sabit bir büyüklüğe sahiptir ve sıralı olarak kullanılır. Yeni bir çağrı yapıldığında, stack pointer (yığın işaretçisi) artar ve yerel değişkenler bu artan pointer ile birlikte bellekte yer bulur.
4. **Bellek Yönetiminde Sorumluluk:**
   * **Heap:** Programcının sorumluluğundadır. Bellek tahsisi ve serbest bırakma işlemleri doğru bir şekilde yönetilmelidir.
   * **Stack:** İşletim sistemi ve derleyici tarafından otomatik olarak yönetilir. Programcının doğrudan müdahalesine gerek yoktur.

İşletim sistemi dersinde bu kavramları anlamak, bellek yönetimi ve program performansı üzerinde önemli bir etkiye sahip olabilir.