YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

KİLİTLENME PROBLEMLERİNİN BELİRLENMESİ VE ÇÖZÜLMESİ

Elektrik Elektronik Mühendisi Mustafa Bülent MUTLUOĞLU

F.B.E. Bilgisayar Bilimleri ve Mühendisliği Anabilim Dalında Hazırlanan

YÜKSEK LİSANS TEZİ

Tez Danışmanı : Doç. Dr. Selim AKYOKUŞ
Prof. M. Yahya KARSLIGİL
Prof. Dr. Oruç BİLGİÇ

İSTANBUL, 1999

İÇİNDEKİLER Sayfa

ŞEKİL LİSTESİ iv

ÖNSÖZ v

ÖZET vi

ABSTRACT vii

1. GİRİŞ 1

2. İŞLETİM SİSTEMLERİ VE İŞLEMLER 2

2.1 İşlemler (Processes) 2
2.2 İşlem Durumları (Process States) 3
2.3 İşlemlerarası Haberleşme (Interprocess Communication) 4
2.4 Karşılıklı Dışlama (Mutual Exclusion) 5
2.4.1 Semaforlar 5
2.4.2 Monitörler 6
2.4.3 Mesaj Yollama 7

3. KİLİTLENME 8

3.1 Kilitlenme Tanımı 8
3.2 Kilitlenme'yi Doğuran Şartlar 8
3.3 Kilitlenme Örnekleri 8
3.4 Kilitlenme Çeşitleri 10
3.5 Kilitlenmeyle Başa Çıkma Yöntemleri 10
3.5.1 Önleme (Prevention) 10
3.5.2 Sakınma (Avoidance) 11
3.5.3 Belirleme ve Çözme (Detection and Resolution) 11

4. JAVA DİLİNDE ÇOK İŞLEM PARÇACIKLILIĞI 13

4.1 Java Dili'nde Eşzamanlılık 13
4.2 Çok İşlem Parçacıklılığı (Multithreading) 13
4.3 İşlem Parçacığı Senkronizasyonu 14

5. KİLİTLENME BELİRLEME ALGORİTMALARI 16

5.1 Merkezi Sistemlerde Kilitlenme Belirleme Algoritmaları 16
5.2 Dağıtılmış Sistemlerde Kilitlenme Belirlenmesi 17
5.2.1 Merkezi Belirleme 18
5.2.2 Hiyerarşik Belirleme 18
5.2.3 Dağıtılmış Belirleme 18

6. MERKEZİ BİR SİSTEMDE KİLİTLENME BELİRLEME UYGULAMASI 20

6.1 Başlıca Sınıflar 20
6.2 Başlıca Veri Yapıları 21
6.3 Programın Genel Akışı 23
6.3.1 Kaynak Grafiğinin Oluşturulması 25
6.3.2 Kullanılan Kilitlenme Belirleme Algoritması 26

7. SONUÇ 28

KAYNAKLAR 29

EKLER 30

Ek 1 Program Listesi 30
Ek 2 Sözlük 80

ÖZGEÇMİŞ 81

ŞEKİL LİSTESİ

		Sayfa
Şekil 2.1	İşlemlerin bulunabileceği durumlar	3
Şekil 2.2	Durum diyagramı	4
Şekil 3.1	Kilitlenme örneği	9
Şekil 3.2	Kilitlenme örneği	9
Şekil 3.3	Kilitlenme örneği	9
Şekil 4.1	Java dilinde işlem parçacıklarının bulunabilecekleri durumlar	14
Şekil 5.1	Kaynak grafiği örnekleri	16
Şekil 6.1	Thread ve resource veri yapıları	21
Şekil 6.2	RequestList, acquireList ve releaseList veri yapıları	22
Şekil 6.3	ResourceGraph veri yapısı	23
Şekil 6.4	Programın genel akışı	24
Şekil 6.5	Bağlı olmayan bir grafik	25
Şekil 6.6	Bağlı olmayan grafiğin saklanması	25
Şekil 6.7	Kaynak grafiğinin indirgenmesi	26
Şekil 6.8	Kilitlenme Belirleme Algoritmasının Akışı	27

T E Ş E K K Ü R

Yüksek lisans eğitimim boyunca desteğini benden esirgemeyen değerli hocam Sayın Prof. M. Yahya KARSLIGİL'e ve tezin hazırlanması aşamasında görüşleriyle beni yönlendiren değerli hocam Sayın Doç. Dr. Selim Akyokuş'a teşekkür ederim.

ÖZET

Paralel ya da eşzamanlı çalışan işlemlerden oluşan sistemlerde, işlemlerin sonsuza dek birbirini beklemesi anlamına gelen kilitlenme, önemli bir problemdir. Bu sistemlerde, pek çok işlem aynı anda çalışır, birbirleriyle haberleşir ve sistem kaynaklarını ortaklaşa kullanırlar. İşlemler kaynakları talep eder, kaynak boştaysa elde eder ve kaynakla işi bittikten sonra serbest bırakır. Bir işlem aynı anda pek çok kaynağı elinde bulundurabilir.

Bir kaynağı talep eden bir işlem, eğer kaynak boşta değilse kaynağı elinde tutmakta olan işlemi beklemeye başlar. Bu işlem de, başka bir kaynağı elde etmek için bir başka işlemi bekliyor olabilir. Öyle bir an gelir ki, bir döngü içerisinde, bir küme işlem, sonsuza dek birbirlerini beklemeye başlar. Bu durumada kilitlenme oluşmuş demektir ve tespit edilip çözülmesi gerekir.

Bu çalışmada, merkezi sistemler için geliştirilmiş olan bir kilitlenme belirleme algoritması, simülasyona gerek kalmadan uygulanmıştır. Programlama dili olarak, eşzamanlılığı desteklediği için Java seçilmiştir. Çalışma sonucunda, algoritmanın merkezi sistemlerde kilitlenme belirlemesi için doğru çalıştığı gösterilmiştir.

ABSTRACT

Deadlock, which means that the processes waiting for each other forever, is a serious problem for the systems consisting of processes running in parallel or concurrently. In these systems, lots of processes run concurrently, communicate with each other and share the system resouces. The processes request the resources, acquire it if it is free and release it when they are finished with that resource. A process may hold lots of resources at the same time.

If the resource a process requesting is not free, the process starts to wait for the process holding that resource. That process may be waiting for another process to acquire another resource. And at a certain time, a set of processes start to wait each other in a loop infinitely. Then it means that deadlock has occured and it must be detected and resolved.

In this thesis, a deadlock detection algorithm developed for centralized systems is applied, without need to a simulation. Because it supports concurrency, Java is choosed as the programming language. As a result of the work, it is shown that the algorithm works correctly for the centralized systems.

1. GİRİŞ

Birçok işlemin, sınırlı sayıda kaynak için yarıştığı, çok işlemli ortamlarda kilitlenme önemli bir sorun olarak karşımıza çıkmaktadır. Bir kaynağı talep eden bir işlem, eğer kaynak boşta değil ise bekleme durumuna geçer ve kaynak elde edilebilir duruma gelinceye kadar beklemede kalır. Bazen öyle bir durum oluşur ki, pek çok işlem, diğer bir işlemin elinde olan bir kaynak için beklemeye başlar ve işlemler sonsuza dek birbirlerini beklerler. Bu duruma kilitlenme diyoruz.

Önceden önlem alınmaması durumunda, kilitlenme oluşması durumunda bunun belirlenmesi ve çözülmesi gerekmektedir. Bu çalışmada, merkezi sistemlerde kilitlenme belirlemesi için geliştirilen bir algoritma uygulaması yapılmıştır. Daha sonra algoritma, "Dining Philosophers" problemine uygulanmıştır.

İkinci bölümde, çok işlemli sistemlerde işlemlerin çalışması, birbirleriyle haberleşme ve karşılıklı dışlama yöntemleri anlatılmaktadır.

Üçüncü bölümde, kilitlenmenin ne olduğu örneklerle anlatılmakta ve başa çıkma yöntemleri hakkında bilgi verilmektedir.

Dördüncü bölümde, Java dilinde çok işlem parçacıklılığı anlatılmaktadır.

Beşinci bölümde, merkezi ve dağıtılmış sistemlerde geliştirilmiş belli başlı kilitlenme belirleme algoritmaları hakkında genel bilgi verilmektedir.

Altıncı bölümde de, bu tez çalışmasında uygulana yöntem, programda kullanılan önemli veri yapıları ve algoritma akışları anlatılmaktadır.

Program listesi, tezin sonuna eklenmiştir.

2. İŞLETİM SİSTEMLERİ ve İŞLEMLER

Bir işletim sistemi, uygulama programlarının çalışmasını denetleyen ve bilgisayar donanımı ile kullanıcı arasında bir arayüz görevi gören bir programdır. Üzerindeki yazılımlar olmadan, bir bilgisayar işe yaramaz bir metal yığınıdır. Üzerine, bahsedilen yazılımlar eklendiğinde, bu metal yığını, bilgileri saklayabilen, bu bilgiler üzerinde hesaplamalar yapabilen, bir yazıdaki yazım yanlışlarını denetleyebilen, dünyanın diğer ucundaki başka bir metal yığınından istenilen bir konuda makaleler elde edebilen, çocukların eğitimine katkıda bulunabilen ve artık pek çoğumuz için vazgeçilmez olan çok yararlı bir alet durumuna gelmektedir. Bilgisayar yazılımları iki ana gruba ayrılabilir:

- Bilgisayarın kendisinin çalışmasını yöneten "sistem programları" ve,
- Kullanıcıların problemlerini halleden "uygulama programları".

Sistem programlarının en önemli ve temel olanı "**işletim sistemi**"dir. Zira bu sistem, sistem kaynaklarını kontrol ederek, uygulama programlarının yazılabilmesi için gerekli temeli sağlar. İşletim sisteminin üç temel amacı olduğu söylenebilir:

1) Rahatlık: Program yazmayı daha rahat ve kolay hale getirmek.
2) Verimlilik: Bilgisayar kaynaklarının daha verimli kullanılmasını sağlamak.
3) Gelişmeye Yatkın Olma: Yeni sistem fonksiyonlarının geliştirilmesine açık olmak.

Yıllar içinde işletim sistemlerinde meydana gelen gelişmelerin en önemlilerinden ikisi "çok programlılık" (multiprogramming) ve "zaman paylaşımı" (timesharing) özelliklerine sahip işletim sistemlerinin geliştirilmesidir.

2.1 İşlemler (Processes)

Bütün işletim sistemlerindeki anahtar kavramlardan biri "**işlem**"dir. Bir işlem, temel olarak, çalışmakta olan bir programdır. İçeriği, çalışabilir bir programın, programın verileri ve yığın (stack), program sayıcısı (program counter), yığın işaretçisi (stack pointer), diğer yazıcılar (register) ve programın çalışması için gereken diğer bilgilerden oluşmaktadır.

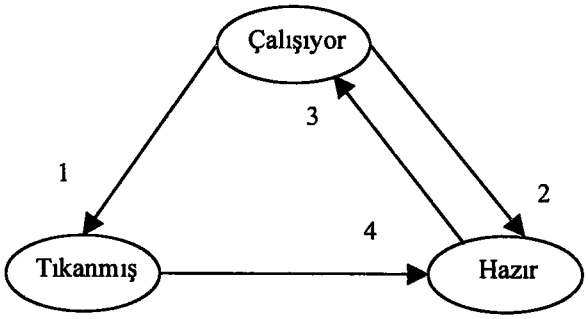
Zaman paylaşımlı bir sistemde, işlemler, belirli aralıklarla çalışırlar. Süresi dolması, başka bir olayı beklemesi gibi nedenlerle bir işlem, işletim sistemi tarafından durdurulmakta ve başka bir işlemin çalışması sağlanmaktadır. Bu şekilde durdurulan bir işlemin, daha sonra tekrar kaldığı yerden devam ettirilebilmesi için gerekli bilgilerin (program sayıcısı, yığın işaretçisi gibi) hafızada saklanması gerekir.

2.2 İşlem Durumları (Process States)

Zaman paylaşımlı, çok programlı sistemlerde, bir işlem değişik sebeplerden dolayı durdurulabilir ve başka bir işlem başlatılabilir. Eğer bir işlem, kullabileceği maksimum işlemci zamanını kullandığı için durdurulmuşsa, **hazır** (ready) durumunda demektir. Yani işlemci sırası kendisine geldiğinde, çalışması için hiçbir engel yoktur. Eğer bir işlem, işlemci sırası kendisinde olduğu halde, harici bir nedenden dolayı çalışamaz duruma gelirse, bu durumda bu işlem **tıkanmış** (blocked) durumdadır. Genel olarak bir işlem aşağıdaki üç durumda bulunabilir:

1. **Çalışıyor**: O sırada işlemciyi kullanıyor demektir.
2. **Hazır**: Çalışabilir ama başka bir işlemin çalışması için geçici olarak durdurulmuş.
3. **Tıkanmış**: Harici bir olay meydana gelene kadar çalışamaz.

Şekil 2.1'de bir işlemin bulunabileceği durumlar bir durum diyagramında gösterilmektedir.

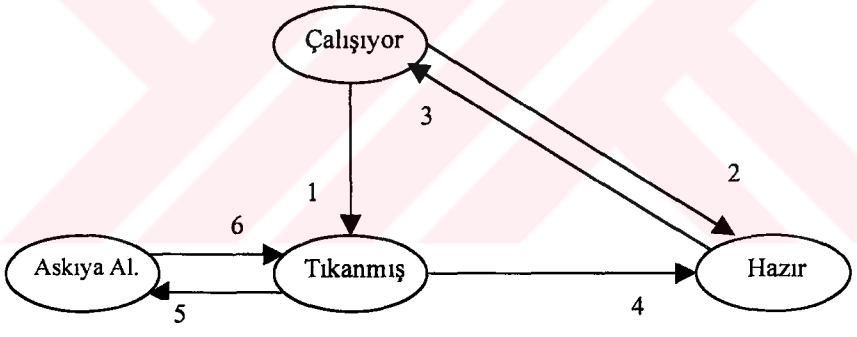


Şekil 2.1 İşlemlerin bulunabileceği durumlar

Şekilde de görüleceği gibi, durumlar arasında dört değişik geçiş mümkündür:

1. İşlem, harici bir olayı beklediğinden, devam edemeyeceğini anlar.
2. İşlem, kendisine verilen işlemci zamanını doldurmuş olduğundan, işletim sistemi tarafından durdurulur.
3. İşlemci sırası kendisine gelen hazır bir işlem, çalışmaya başlatılır.
4. Beklenen harici olay gerçekleşmiş olduğundan, tıkanmış işlem hazır duruma geçer.

İşlemlerin içinde bulunabileceği üç temel duruma, dördüncü bir durum eklenebilir: **Askıya alınmış** (suspended). Buna ihtiyaç duyulmasının nedeni sınırlı hafıza kapasitesidir. Tıkanmış ya da hazır durumda bulunan tüm işlemlere ait bilgiler hhafızada tutulduğundan, bir süre sonra hafıza yetersiz hale gelebilir. Bu durumda işletim sistemi, tıkanmış durumdaki bir işleme ait bilgileri diske kaydedip hafızada yer açabilir. Bu işleme maruz kalan işlem askıya alınmış durumuna geçmektedir. Bu durumda, durum diyagramımız Şekil 2.2'deki gibi olur.



Şekil 2.2 Durum diyagramı

2.3 İşlemlerarası Haberleşme (Interprocess Communication)

Pek çok durumda, işlemler birbirleriyle haberleşme ihtiyacı duyarlar. Bu, en yaygın olarak, genel bir değişken (global variable) aracılığıyla yapılır. İşlemler, bunun gibi daha başka **ortak veriyi** (shared data) kullanıyor olabilirler, diskteki bir dosya gibi. Her işlem bu ortak veriye erişebildiği, okuyabildiği ve değiştirebildiği için, önemli sorunlar meydana gelebilmektedir.

Örnek olarak, bir sistemde, belli bir işi sürekli olarak yapmakta olan iki işlem olduğunu ve

"işlem sayısı" isminde ortak bir değişkene sahip olduklarını düşünelim. Her işlem, tamamladığı her işten sonra işlem sayısı değişkenini bir artırmaktadır. Belli bir anda, ortak değişkenin değerinin 3 olduğunu kabul edelim. Çalışmakta olan bir işlem elindeki işi bitirir ve işlem sayısı değişkenin değerini bir artırmaya karar verir. Önce değişkenin değerini okur: 3. Tam bu sırada işletim sistemi bu işlemi durdurur ve sırayı diğer işleme verir. Bu işlem de elindeki işi tamamlayıp ortak geğişkenin değerini değiştirmeye karar verir. Değerini okur: 3, ve bir artırıp değerini 4 yapar. İşlem sırası tekrar diğer işleme geldiğinde, kaldığı yerden devam eder, elinde ortak değişkenin 3 olduğuna dair bilgi vardır (oysa 3 değil 4'tür) ve bir artırır ve 4 yapar, oysaki 5 yapması gerekirdi.

Birden çok işlemin ortak bir veriye erişebilir durumda olduğu ve nihayi sonucun, tam olarak hangi işlemin ne zaman çalıştığına göre değiştiği durumlara "yarışma koşulları" (race conditions) denilmektedir.

2.4 Karşılıklı Dışlama (Mutual Exclusion)

Yarışma koşullarını engellemenin yolu **karşılıklı dışlama**dır. Bunun anlamı, aynı andan birden çok işlemin ortak veriye erişmesini engellemektir. Bir işlemin, programının ortak veriye eriştiği komutları içeren alanına **kritik bölge** (critical section) denilmektedir. Böylece karşılıklı dışlama, aynı anda birden çok işlemin kritik bölgelerine girmesinin engellenmesi olarak da tanımlanabilir.

Karşılıklı dışlama ve **eşzamanlılığı** (concurrency) sağlama yöntemleri üçe ayrılmaktadır:

1) **Semaforlar** (semaphores)
2) **Monitörler** (monitors)
3) **Mesaj yollama** (message passing)

2.4.1 Semaforlar

Eşzamanlı çalışan işlemlerden kaynaklanan problemlerin çözümüyle ilgili olarak ilk ilerlemeyi, 1965 yılında Dijkstra sağlamıştır. Bu çözümde temel prensip şudur: İki veya daha fazla işlem, basit sinyaller sayesinde birlikte çalışabilirler. Örneğin bir işlem, başka

bir işlemden bir sinyal gelene kadar beklemek zorunda bırakılabilir. Uygun sinyal yapılarıyla, herhangi karmaşık bir işbirliği sağlanabilir. Sinyal gönderme işlemi için, **semafor** diye adlandırılan özel değişkenler kullanılmaktadır.

S semaforu aracılığı ile bir sinyal göndermek isteyen bir işlem *signal(s)* komutunu çalıştırır. Sinyal almak için de *wait(s)* komutunu çalıştırır; beklenen sinyal gönderilmediyse, gönderme işlemi gerçekleşene kadar işlem beklemeye alınır. İstenen etkinin sağlanabilmesi için semafor değişkenini şu üç koşula uyması gerekmektedir:

1. Bir semaforun ilk değeri negatif olamaz.
2. *Wait* komutu, semaforun değerini bir azaltır. Semafor negatif bir değere düşerse, *wait* komutunu çalıştıran işlem beklemeye alınır.
3. *Signal* komutu, semaforun değerini bir artırır. Semaforun değeri pozitif değilse, *wait* komutuyla beklemeye alınan bir işlem hazır duruma getirilir.

Bu işlemler dışında hiçbir şekilde bir semaforun değeri değiştirilemez. Ayrıca *wait* ve *signal* işlemleri atomiktir, yani kesilemezler (uninterrupted). Bir semafor için beklemekte olan işlemler için bir kuyruk oluşturulur ve gerektiğinde bu kuyruktan bir işlem hazır duruma getirilir.

2.4.2 Monitörler

Karşılıklı dışlama için semaforlar çok güçlü ve esnek bir imkan sağlamaktadır ama bunları kullanarak yazılan programlarda hata yapma olasılığı çok yüksektir. Semaforları etkileyen komutlar programın her yerine dağılmış olabilir ve bunların etkiledikleri semafor üzerindeki toplu etkilerini görmek çok zordur. Bu zorluğu aşmak için, Hoare (1974) ve Brinch Hansen (1975), daha yüksek seviyeli bir senkronizasyon yapısı önerdiler: **monitör**. Bir monitör, özel bir paket içinde gruplanmış yordamlar (procedures), değişkenler ve veri yapılarından meydana gelmektedir. İşlemler, bir monitördeki yordamları istedikleri zaman çağırabilirler ama bunun dışında, monitördeki veri yapılarına ulaşamazlar. Monitörlerin karşılıklı dışlamayı sağlamak için bir özelliği vardır: Aynı anda sadece bir işlem monitörün içinde aktif olabilir.

Bir işlem, monitördeki bir yordamı çağırdığında, işletim sistemi önce monitörün içinde aktif olan başka bir işlem olup olmadığını kontrol eder. Eğer varsa, yordamı çağıran işlem, monitörün içinde aktif olan işlem monitörü terk edene kadar bekler.

Monitör kullanmanın en önemli dezavantajı, bunları destekleyen programlama dillerinin pek olmamasıdır. Ama, gelecekte çok daha fazla yaygınlaşacağı düşünülen **Java** dili sayesinde, monitörler de artık programcıların kullanabileceği yapılar arasına girmiş bulunmaktadır.

2.4.3 Mesaj Yollama

Semaforlar ve monitörler sayesinde ortak hafızalı bir sistemdeki problemleri halledebiliriz ama ya dağıtılmış sistemlerdeki (distributed systems) problemler? Bu sistemler için **mesaj yollama** yöntemi geliştirilmiştir. İki komut kullanılmaktadır: *send* ve *receive*. Bu yöntem, semafor yöntemine benzemektedir. En önemli fark, sinyallerin farklı makineler arasında dolaşmasıdır. Bu yüzden pek çok yeni problemle karşılaşılmaktadır: Mesajlar, ağ üzerinde kaybolabilmektedir; mesaj doğru alındığı halde, mesajın alındığına dair gönderilen "alındı" mesajı kaybolabilmektedir...

3. KİLİTLENME

3.1 Kilitlenme Tanımı

Kilitlenme en genel şekliyle şöyle tanımlanmaktadır: Bir işlem kümesindeki her işlem, ancak kümedeki başka bir işlemin meydana getirebileceği bir olayı bekliyorsa, bu işlem kümesine "**kilitlenmiş**" denmektedir.

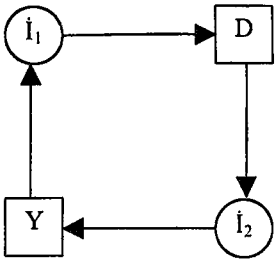
3.2 Kilitlenme'yi Doğuran Şartlar

1. Karşılıklı dışlama (Mutual exclusion) şartı: Her kaynak, ya tam olarak bir işleme verilmiştir ya da boştadır.
2. Sakla ve bekle şartı: Elinde kaynak olan bir işlem başka kaynakları talep edebilir.
3. Zorla alamama (No preemption) şartı: Kaynaklar, daha önce verildikleri işlemden zorla geri alınamaz. Ancak işlemin kendisi tarafından bırakabilir.
4. Dairesel bekleme şartı: Her biri zincirin bir sonraki elemanının elinde tuttuğu kaynağı beklemekte olan, iki ya da daha çok işlemden oluşan bir zincirin oluşmuş olması gerekir

3.3 Kilitlenme Örnekleri

Kilitlenmeler birbirinden farklı olabilirler. Kaynak ve işlem sayısına göre çok basit bir kilitlenme olabileceği gibi, çok karmaşık bir kilitlenme de olabilir.

Ilk örnek olarak, aynı anda hem diskteki bir dosyaya (D) hem de yazıcıya (Y) sahip olmak için birbirleriyle yarışan iki işlemin ($İ_1$, $İ_2$) bulunduğu bir durumu ele alalım. İki işlemin de kaynaklardan birine sahip olması ve diğer kaynağı talep etmesi durumunda kilitlenme meydana gelir. Bu durum Şekil 3.1'de gösterilmiştir. $İ_1$ işlemi yazıcıyı almış dosyayı beklemekte, $İ_2$ işlemi de dosyayı almış yazıcıyı beklemektedir. Bir şey yapılmadıkça, birbirlerini sonsuza kadar beklerler.



Şekil 3.1 Kilitlenme örneği

Başka bir örnek olarak, toplam hafızanın 20K olduğu bir sistemde, Şekil 3.2'deki gibi hafıza taleplerinde bulunan iki işlemin olduğu bir durumu ele alalım.

$İ_1$	$İ_2$
8K talep eder	7K talep eder
6K talep eder	8K talep eder

Şekil 3.2 Kilitlenme örneği

Eğer işlemlerden herhangi birinin iki talebi de, diğer işlemin talebi karşılanmadan önce karşılanırsa kilitlenme olmaz. Sırayla ikisi de talep ettikleri hafızayı elde ederler. Ama, eğer ikisi de ilk taleplerini elde ederlerse, ikisinin de ikinci talebi için yeterli hafıza alanı kalmayacağından, sistem kilitlenir.

Başka bir örnek de Şekil 3.3'te görülmektedir.

$İ_1$	$İ_2$
Al ($İ_2$, M)	Al ($İ_1$, M)
Gönder ($İ_2$, M')	Gönder ($İ_1$, M')

Şekil 3.3 Kilitlenme örneği

Bu örnekte, iki işlem de diğerine mesaj göndermek için, ondan mesaj beklediğinden, iki işlem, sonsuza kadar birbirlerinden gelecek mesajları bekler.

3.4 Kilitlenme Çeşitleri

Kilitlenmiş işlemlerin, meydana gelmesini bekledikleri olaya göre kilitlenme ikiye ayrılmaktadır. Kilitlenme, beklenen olay, bir kaynağın serbest bırakılması ise "**kaynak kilitlenmesi**", bir mesajın alınması ise "**haberleşme kilitlenmesi**" diye adlandırılmaktadır. Kaynak kilitlenmesi konusunda pek çok araştırma yapılmış olmasına karşın, haberleşme kilitlenmesi konusunda yapılmış araştırmalar sınırlı kalmıştır.

Tasarım aşamasında kilitlenme problemi ihmal edilirse, daha sonra belirlenmesi ve bazı işlemlerin durdurulması yoluyla çözülmesi gerekir.

3.5 Kilitlenmeyle Başa Çıkma Yöntemleri

Kilitlenme problemiyle başa çıkma yöntemleri üç çeşittir: "**Önleme**" (**prevention**), "**sakınma**" (**avoidance**) ve "**belirleme ve çözme**" (detection and resolution) dir.

3.5.1 Önleme (Prevention)

Kilitlenme olması için gerekli olan dört şarttan birinin oluşması imkansız hale getirilerek çalışır. İşlemlerin kaynakları talep etmeleri konusunda getirilen sınırlamalara göre, önleme yöntemleri değişir. Daha çok aşağıdaki üç yöntem kullanılır:

1) İşlemin, işleme başlamadan önce, ihtiyaç duyacağı bütün kaynakları elde etmesi gerekir. Bu şart her zaman sağlanamayabilir çünkü, sadece dağıtılmış sistemlerde değil, merkezi sistemlerde de, telep edilecek kaynaklar veriye bağımlı olabilir ve işlemin en başında hangi kaynakların talep edileceği bilinemeyebilir. Ayrıca, her işlemin, işleme başlamadan önce bütün kaynakları elde etmesi şartı, sistemin eşzamanlılığını (concurrency) da olumsuz olarak etkilediği için, bu yöntem elverişli değildir.
2) Yeni bir kaynak talep etmeden önce, her işlemin, elindeki bütün kaynakları bırakması gerekir. Yani, eşzamanlı gereken kaynakların bir grup halinde talep edilmesi şartı vardır.
3) Kaynaklar sıralıdır ve her işlem, gerekli kaynakları bu sıraya uyarak talep edebilir.

Önleme yönteminin iki dezavantajı vardır. Birincisi, gerekmediği halde kaynakların önceden tahsis edilmesi, sistemin eşzamanlılığını düşürür. İkincisi, yapılan taleplerin güvenilirliğinin ölçülmesi ağır bir yük getirmektedir. Önleme yönteminin en önemli avantajı, kilitlenmeden dolayı meydana gelen işlemleri baştan başlatma yükünden kurtulunmuş olmasıdır.

Özetle, yapısı önceden belirlenebilecek sistemler dışında, önleme yönteminin elverişsiz olduğu düşünülmektedir.

3.5.2 Sakınma (Avoidance)

Bu yöntemde, işlemler ve ilişkili kaynaklarla ilgili bilgi, bir "işlem-kaynak grafiği" aracılığı ile tutulmaya çalışılır. Bu grafik, halen hangi kaynakların hangi işlemlere tahsis edilmiş olduğunu ve olası talepleri gösterir. Bu grafik aracılığı ile, bütün işlemlerin bitene kadar çalışmasını sağlayacak bir faaliyet sırası olup olmadığına bakılıp, kilitlenmeden sakınmaya yarayacak bir kaynak tahsisi stratejisi bulunmaya çalışılır. Talep edilen bir kaynak, talep eden işleme tahsis edilmeden önce, bu tahsisin güvenli olup olmadığına bakılır ve ancak bundan sonra tahsis gerçekleşir. Bu nedenle, işlemlerin hangi kaynakları talep edecekleri konusunda bir ön bilgiye ihtiyaç duyulduğu için bu yöntem de elverişli değidir.

3.5.3 Belirleme ve Çözme (Detection and Resolution)

Bu yöntemde kilitlenme olmasını engellemek için hiçbir şey yapılmaz. İşlemler, hiçbir sınırlama olmaksızın, istedikleri kaynak için beklemeye geçebilirler. Kilitlenmiş bir grup işlem farkedildiğinde, bunlardan bazıları sonlandırılarak problem çözülür. Bu işlem için bir işlem-kaynak grafiğine ihtiyaç vardır.

Grafiğin düğümleri işlemleri ya da kaynakları temsil eder. Kenarlar ise, kaynakların hangi işlemlere tahsis edilmiş olduğunu ve hangi işlemlerin hangi kaynaklar için beklediğini belirtir. Bu grafikte bir halka oluşursa, kilitlenme meydana gelmiş demektir. Bu yüzden, belli sürelerde, grafikte halka oluşup oluşmadığının belirlenmesi gerekmektedir.

Belirleme ve Çözme yöntemlerinin iki önemli dezavantajı vardır. Birincisi, kaynak grafiğinde halka oluşup oluşmadığının belirlenmesi sisteme çok ağır bir yük getirmesidir. İkincisi de, kilitlenmenin çözülmesinin getirdiği fiyattır, yani işlemlerin durdurulması ya da baştan başlatılması.

4. JAVA DİLİNDE ÇOK İŞLEM PARÇACIKLILIĞI

4.1 Java Dili'nde Eşzamanlılık

Pek çok programlama dili, programcılara eşzamanlı programlar yazma imkanını vermemektedir. Popüler genel amaçlı programlama dilleri arasında, sadece Java dili eşzamanlılığı desteklemektedir. Programcılar, uygulamalarını, belli **işlem parçacıklarının** (**thread**) eşzamanlı çalışması şeklinde programlayabilmektedirler.

4.2 Çok İşlem Parçacıklılığı (Multithreading)

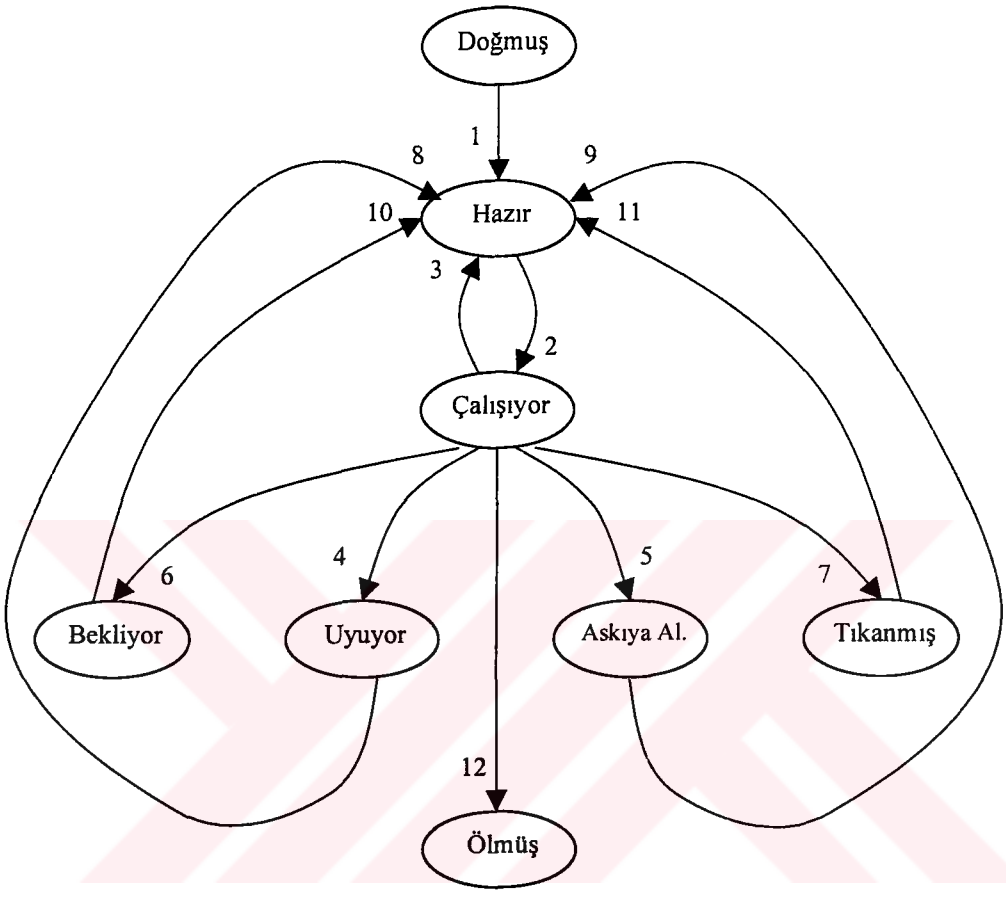
Java dilinin eşzamanlılığı desteklediğini söyledik. Bu, işlem parçacıkları ile sağlanmaktadır. Bir işlem parçacığını, bir işlemden ayıran en önemli özelliği, kendine ait kaynakları olmamasıdır. Bütün işlem parçacıkları, kendilerini yaratan işlemin kaynaklarını ortaklaşa kullanırlar.

Java'da işlem parçacıklarının bulunabilecekleri durumlar Şekil 4.1'de gösterilmiştir.

Durumlar arasında olabilecek geçişler şekilde numaralanmıştır. Bu geçişler, şu durumlarda gerçekleşebilir:

1. *start* yordamı çalıştırılınca,
2. İşlemci sırası işlem parçacığına geldiği zaman,
3. İşlem parçacığı kullanabileceği maksimum işlemci süresini doldurduğu, kesildiği (interrupt) ya da başka bir işlem parçacığının çalışmasına imkan vermek için *yield* yordamı çalıştırılınca,
4. *sleep* yordamı çalıştırılınca,
5. *suspend* yordamı çalıştırılınca,
6. *wait* yordamı çalıştırılınca,
7. Giriş/çıkış (I/O) isteği yapılınca
8. Uyuma süresi sona erince,
9. *resume* yordamı çalıştırılınca,
10. *notify* yordamı çalıştırılınca,

11. Giriş/çıkış işlemi tamamlanınca
12. İşlem parçacığı işini tamamlayınca ya da *stop* yordamı çalıştırılınca.



Şekil 4.1 Java dilinde işlem parçacıklarının bulunabilecekleri durumlar

4.3 İşlem Parçacığı Senkronizasyonu

Java, karşılıklı dışlamayı ve senkronizasyonunu sağlamak için, monitörleri kullanmaktadır. İçinde, *synchronized* sıfatlı bir yordam bulunan her nesne, bir monitördür. Monitör, aynı anda sadece bir işlem parçacığının senkronize bir yordamı çalıştırmasına izin verir. Bu, senkronize yordam çağırılınca, nesnenin *kilitlenmesiyle* sağlanır; buna *kilidin elde edilmesi* de denilmektedir.

Eğer bir monitörde, yani bir nesnede birden çok senkronize yordam varsa, aynı anda

bunlardan sadece bir tanesi çalışıyor durumda olabilir. Başka herhangi bir senkronize yordamı çağıran her işlem parçacığı, beklemek zorundadır. Çalışmakta olan senkronize yordam tamamlandığında, nesne üzerindeki kilit serbest bırakılır ve monitör, hazır olan işlem parçacıkları arasında en yüksek önceliğe sahip olan işlem parçacığının devam etmesine izin verir.

Senkronize bir yordamı çalıştırmakta olan bir işlem parçacığı, artık çalışamayacağına karar verebilir. Bu durumda bu işlem parçacığı, *wait* yordamını çağırır. Hem işlemci, hem de nesneye ait kilit işlem parçacığından alınır ve bu işlem parçacığı, nesneye ait kilidi bekleyen işlem parçacıklarından oluşan kuyruğa katılır. Senkronize bir yordamı çalıştıran bir işlem parçacığı, bu yordamla işi bittiğinde, *notify* yordamını çağırarak beklemekte olan bir işlem parçacığını hazır duruma sokabilir.

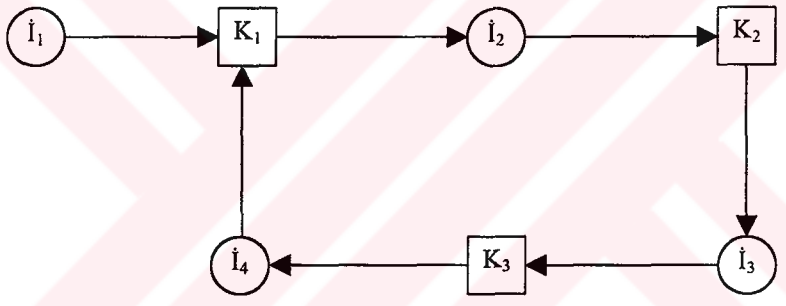
Monitör, senkronize yordamlarından birini çağırıp, meşgul olduğu için içeri giremeyen tüm işlem parçacıklarından oluşan bir kuyruk meydana getirir. Ayrıca, manitörün içindeyken *wait* yordamını çağıran işlem parçacıkları da bu kuyruğa alınır. Yalnız, kuyruktaki bu iki sınıf işlem parçacıkları arasında önemli bir fark vardır. Monitör meşgul olduğu için beklemeye başlayan işlem parçacıkları, monitör serbest olduğu anda içeriye girebilirler ama, *wait* yordamını çağırmış olan bir işlem parçacığı, ancak başka bir işlem parçacığı tarafından *notify* yordamının çağırılmasıyla uyandırıldıktan sonra monitöre girebilir.

5. KİLİTLENME BELİRLEME ALGORİTMALARI

5.1 Merkezi Sistemlerde Kilitlenme Belirleme Algoritmaları

Bu sistemlerde, belirleme algoritmalarının temelinde, bir kaynak grafiği oluşturulması yatar. Bu grafikte, bir veya daha fazla halka varsa, kilitlenme oluşmuş demektir. Halka yoksa, sistem kilitlenmemiş demektir.

(a)



(b)

Şekil 5.1 Kaynak grafiği örnekleri

Örnek olarak, Şekil 5.1'de, (a) şıkkındaki kaynak grafiğinde bir halka olmadığından sistem kilitlenmemiştir. (b) şıkkındaki kaynak grafiğinde ise, bir halka meydana geldiğinden, sistem kilitlenmiştir.

Yukardaki şekildeki gibi basit kaynak grafiklerinde, grafikte halka olup olmadığı gözle dahi belirlenebildiği halde, gerşek sistemlerde, karmaşık grafiklerdeki halkaları belirlemek için formal bir algoritmaya ihtiyacımız vardır. Bu algoritmaların bir örneği aşağıda verilmiştir. Düğüm listelerinden oluşan bir L veri yapısı kullanılmaktadır:

1. Grafikteki her N düğümü için, N başlangıç düğümü olacak şekilde, aşağıdaki beş adımı uygulayın.
2. L'yi boş listeye eşitleyin ve grafikteki her kolu işaretlenmemiş olarak belirleyin.
3. Şu anki düğümü L'nin sonuna ekleyip, bu düğümün L'de iki kez bulunup bulunmadığına bakın. Eğer durum buysa, grafikte, L'nin içinde listelenmiş, bir halka vardır ve algoritma sona erer.
4. Şu anki düğümden çıkan, işaretlenmemiş kol olup olmadığına bakın. Varsa 5. Adıma; yoksa 6. Adıma gidin.
5. İşaretlenmemiş kollardan birini seçip işaretleyin. Kolun işaret ettiği düğüme gidip 3. Adıma gidin.
6. Ölü bir sona ulaşmış bulunuyoruz. Bir önceki düğüme gidin ve 3. Adıma gidin. Eğer bu düğüm, grafikteki ilk düğüm ise, grafikte halka yok demektir ve algoritma sona erer.

5.2 Dağıtılmış Sistemlerde Kilitlenme Belirlenmesi

Genel olarak, dağıtılmış bir sistem, her biri merkezi bir sistemden oluşan birden çok sitelerden oluşur. Bu, sistemde, tekrarlanmış veri, değişik sitelerde paralel olarak çalışan bir işlem gibi yeni problemlerin doğmasına neden olmaktadır. Tasavvur edilebileceği gibi, dağıtılmış bir sistemde kilitlenmenin belirlenmesi, merkezi bir sisteme göre daha zordur. Bunun sebebi, her sitenin, genel sistem hakkında sadece yerel bilgiye sahip olmasıdır.

Hem kaynak hem de haberleşme kilitlenmeleri dağıtılmış olabilir. Dağıtılmış sistemlerde, yerel olmayan veriye erişmek isteyen işlemler, diğer sitelere göç edip orada bir alt işlem yaratırlar. Alt işlemler eşzamanlı olarak çalışabilirler.

Dağıtılmış sistemlerde, kilitlenme belirleme algoritmaları üç ana grupta toplanabilir:

1) Merkezi Belirleme (Centralized Deadlock Detection)
2) Hiyerarşik Belirleme (Hierarchical Deadlock Detection)
3) Dağıtılmış Belirleme (Distributed Deadlock Detection)

5.2.1 Merkezi Belirleme

Merkezi belirleme algoritmalarında, bir site, merkezi belirleyici, genel kilitlenmelerin belirlenmesinden sorumludur. İki temel yaklaşım vardır: *Peryodik* belirleme yaklaşımında, peryodik olarak pek çok site, kilitlenmenin oluşup oluşmadığını belirlemek için kontrol edilir. *Sürekli* yaklaşımda ise, her yerel site, kendi kaynak grafiğinde bir değişiklik olduğunda, merkezi belirleyiciyi uyarır.

Merkezi belirlemenin sakıncalarından biri, kilitlenme olmadığı halde kilitlenmenin varolduğunun sanılmasıdır. Bir diğer sakınca da, siteler arasındaki yoğun haberleşme trafiğinin getirdiği yüktür. Önemli sakıncalarından biri de, merkezi belirleyici siteye bağımlı olmasıdır. Bu sitenin çalışmaz hale gelmesi durumunda, bütün algoritma çöker.

5.2.2 Hiyerarşik Belirleme

Merkezi belirlemede, bütün sistemin kaynak grafiğine ait bilgiler bir sitede toplanıyordu. Hiyerarşik belirleme, merkezi belirleme ile dağıtılmış belirleme arasında bir yerdedir.

Merkezi belirlemede olduğu gibi, her site, kendi yerel kaynak grafiğini oluşturur. Merkezi belirlemeden farklı olarak, genel kaynak grafiği, belli sayıda belirleyici siteler arasında dağıtılmıştır. Bu belirleyiciler, bir ağaç şeklinde organize edilmiştir. Bu ağacın her dalı, yerel bir kaynak grafiğine aittir. Her site, kaynak grafiğindeki değişiklikleri, ağaç hiyerarşisine göre, ebeveyn sitesine bildirir. Her ebeveyn belirleyici, kendisi ve altındaki sitelerde oluşabilecek kilitlenmelerin belirlenmesinden sorumludur.

Hiyerarşik belirleme, genel kaynak grafiğini oluşturmanın getirdiği yüksek maliyet sorununu kısmen de olsa çözer ama, hala merkezi belirleyicilere bağımlıdır. Bunlardan bir veya birkaçının çalışamaz duruma gelmesi halinde, algoritma çalışamaz hale gelir.

5.2.3 Dağıtılmış Belirleme

Bu yöntemde, yeterli bilgiye sahip herhangi bir site kilitlenmeyi belirleyebilir. Çoğunlukla, genel kaynak grafiğinde oluşabilecek halkalar, iki ya da üç uzunluğa sahip olduğundan, dağıtılmış belirleme, diğer yöntemlere göre daha iyi sonuçlar verebilmektedir. Çoğunlukla,

sadece iki site, kilitlenmenin içinde yer alacaktır. Bu durumda, kilitlenmenin belirlenebilmesi için, tüm sisteme ait genel kaynak grafiğine gerek yoktur. Kilitlenme çok daha hızlı ve gereksiz haberleşme trafiğine neden olmadan belirlenebilir.

6. MERKEZİ BİR SİSTEMDE KİLİTLENME BELİRLEME UYGULAMASI

Java programlama dili, çok işlem parçacıklılığı desteklediğinden, bir simülasyona gerek kalmamış, algoritma aynen uygulanabilmiştir. Detaylara girmeden önce, algoritmanın uygulanması sırasında yapılan kabullenmeleri (assumption) listeleyelim:

1. Her işlem aynı anda en çok bir kaynağı talep edebilir. Talep ettiği kaynağı almadan, başka bir kaynak talebinde bulunamaz.
2. Sistemdeki her işlemin düzgün programlandığı kabul edilmiştir; yani hiçbir işlem, sonsuz döngü ve benzeri hataları içermez.
3. Sistemde interaktif işlemler yoktur.

Sistemde, belli sayıda işlem ve belli sayıda kaynak olduğu kabul edilmiş ve bu sayıların aldığı değişik değerler için algoritmanın çalıştığı gösterilmiştir.

6.1 Başlıca Sınıflar

Pogramda aşağıdaki sınıflar (class) mevcuttur:

- ✓ Deadlock
- ✓ DrawCanvas
- ✓ ButtunHandler
- ✓ BackForward
- ✓ Detector
- ✓ Process
- ✓ Scheduler
- ✓ Resource
- ✓ ListNode
- ✓ Node
- ✓ StartNode
- ✓ DrawNode
- ✓ DrawStartNode

Deadlock sınıfı, programın asıl sınıfını oluşturmaktadır. Applet üzerindeki gerekli elemanların yaratılması, genel işlem parçacıklarıyla kilitlenme belirleyicisi olan "Detector" işlem parçacığının yaratılıp başlatılması ve kaynakların başlatılmasını sağlayan kodu içermektedir.

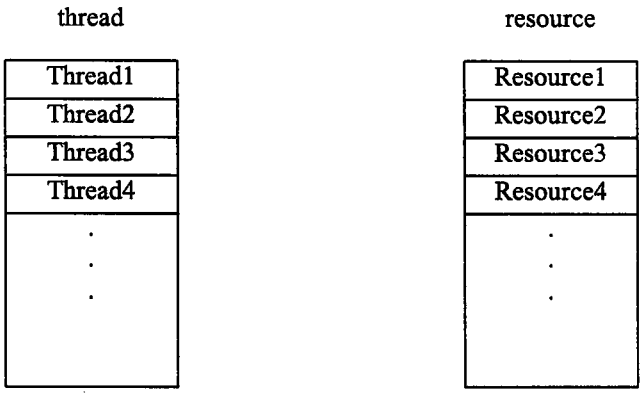
Process sınıfı, genel işlem parçacıklarının çalışmasını sağlayan kodu içermektedir. Resource sınıfı ise, işlem parçacıklarının talep ettikleri kaynakları ve bunları talep ederken kullanacakları yordamları içermektedir.

Detector sınıfı, kilitlenme olup olmadığını belirlemeye yarayacak yordamlarla, kaynak grafiğinin oluşturulup applet üzerinde çizilmesini sağlamak için oluşturulacak grafiğin yaratılması için gereken yordamları içermektedir.

ListNode, Node, StartNode, DrawNode ve DrawStartNode sınıfları kaynak grafiği ve applet üzerinde çizilecek grafiğin düğümlerini oluşturan sınıflardır.

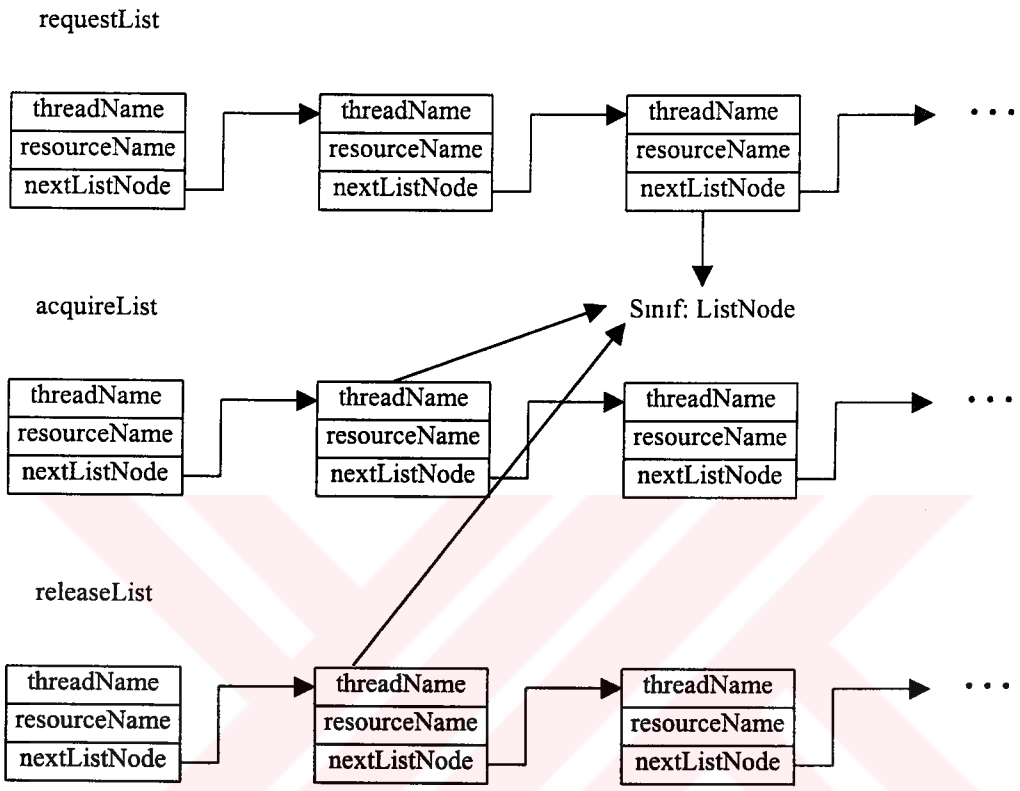
6.2 Başlıca Veri Yapıları

Programdaki başlıca veri yapıları şunlardır: thread (bütün genel işlem parçacıklarından oluşan dizi), resource (kaynaklardan oluşan dizi). Bu iki veri yapısı Şekil 6.1'de gösterilmiştir.



Şekil 6.1 Thread ve resource veri yapıları

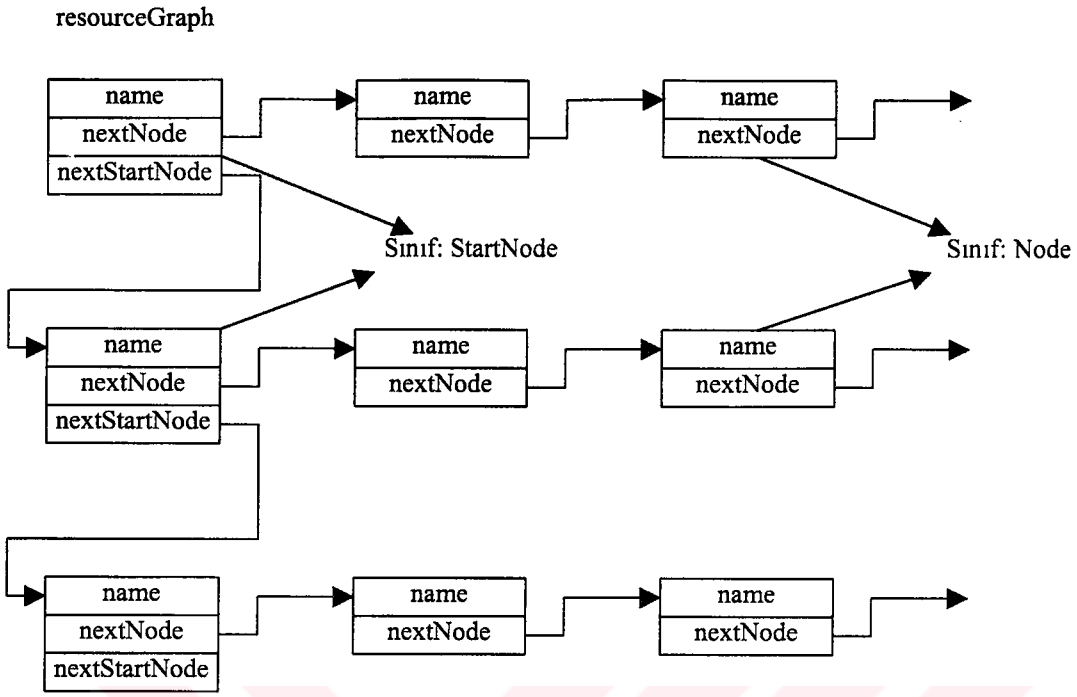
İşlem parçacıklarının yaptığı kaynak talepleri requestList, acquireList ve releaseList linkli listelerinde saklanmaktadır. Bunlar Şekil 6.2'de gösterilmiştir.



Şekil 6.2 RequestList, acquireList ve releaseList veri yapıları

Kaynak grafiğinin saklandığı, resourceGraph linkli listesi, programın en önemli veri yapısıdır. Bu veri yapısı Şekil 6.3'te gösterilmiştir.

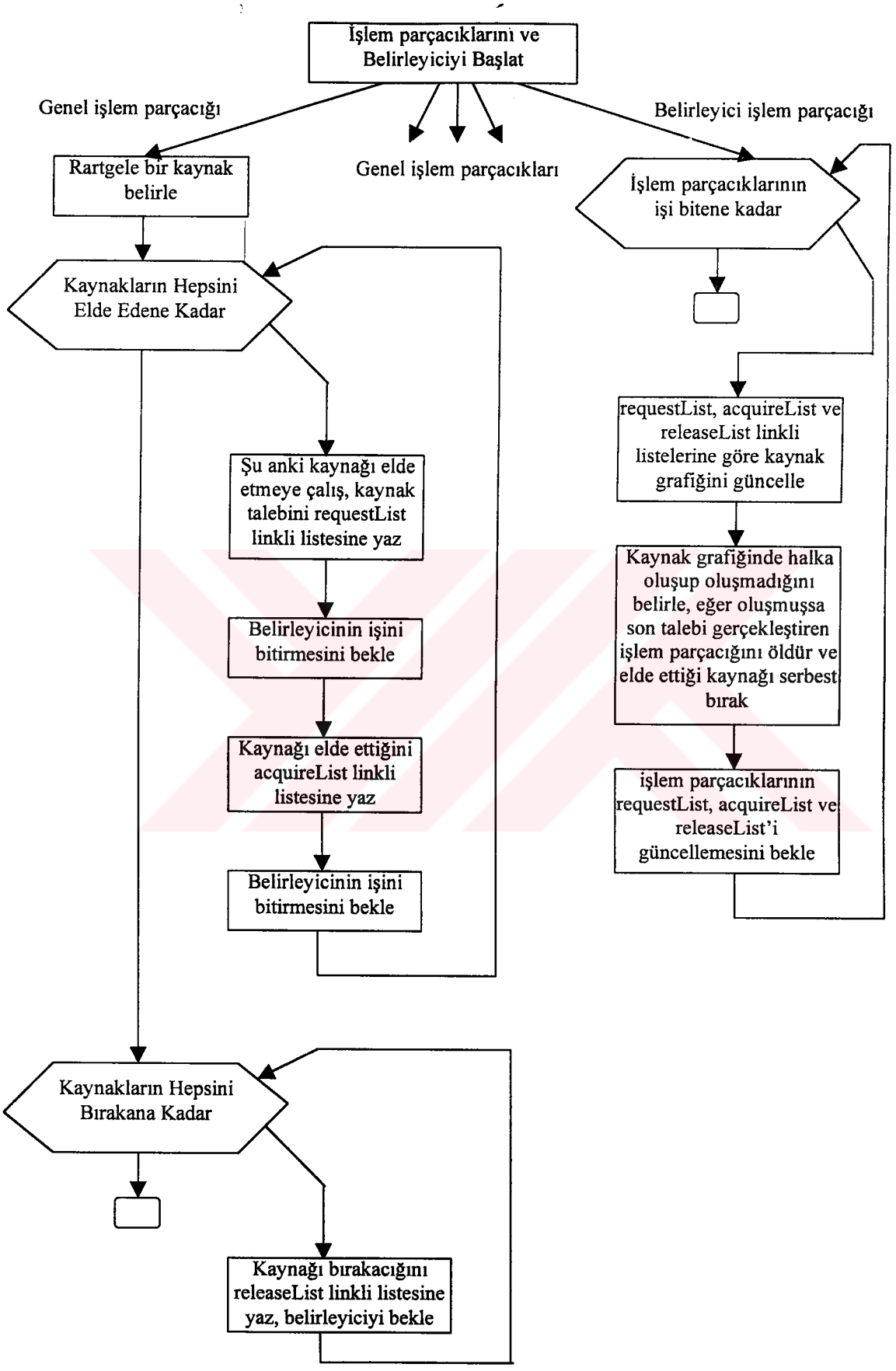
Kaynak grafiği (resourceGraph) kullanılarak elde edilen ve applet üzerinde kaynak grafiğinin çizilmesinde kullanılan veri yapısı ise graphToDraw'dur. Bu linkli listenin elemanları, DrawNode ve DrawStartNode sınıflarının örnekleridir (instance).



Şekil 6.3 ResourceGraph veri yapısı

6.3 Programın Genel Akışı

Programın akışı kısaca Şekil 6.4'teki gibi şöyle özetlenebilir. Programın başında applet elemanları yaratılıp genel işlem parçacıkları ve belirleyici işlem parçacığı başlatılmaktadır. Genel işlem parçacıkları rastgele bir kaynak numarası belirleyip bu kaynaktan başlayarak her kaynağı elde etmeye çalışmaktadır. Kaynak grafiğinde değişiklik yaratabilecek her değişiklik, genel işlem parçacıkları tarafından uygun linkli listelere (requestList, acquireList ve releaseList) yazılmaktadır ve değişikliği gerçekleştiren işlem parçacığı, belirleyicinin grafikte halka oluşup oluşmadığını belirlemesi için beklemeye başlamaktadır. Bu şekilde bütün kaynakları elde ettikten sonra, sırayla kaynakları bırakmaktadır.

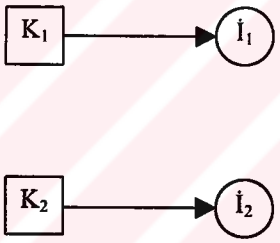


Şekil 6.4 Programın genel akışı

Belirleyici işlem parçacığı, requestList, acquireList ve releaseList linkli listelerinden herhangi birinde bir değişiklik meydana geldiğinde çalışmaya başlamakta, bu yeni değişikliğe göre kaynak grafiğini güncellemekte ve kilitlenmenin oluşup oluşmadığını belirlemek için grafikte halka oluşup oluşmadığına bakmaktadır. Kilitlenme durumunda, son talebi gerçekleştiren genel işlem parçacığını öldürüp yeniden başlatmakta, bu genel işlem parçacığının elde etmiş olduğu kaynakları da serbest bırakmaktadır.

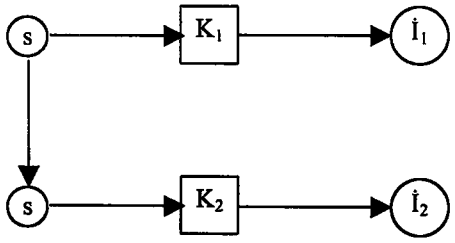
6.3.1 Kaynak Grafiğinin Oluşturulması

Kaynak grafiği bir linkli listeden oluşmaktadır ve düğümleri Node ve StartNode sınıflarının bir örneğidir. Grafiğin düğümlerinin tek bir sınıfın örnekleri olmaması, grafiğin her durumda, bağlı olmamasından (connected) kaynaklanmaktadır. Örneğin Şekil 6.5'teki kaynak grafiği bağlı bir grafik değildir.



Şekil 6.5 Bağlı olmayan bir grafik

Bu yüzden bu grafik Şekil 6.6'daki gibi saklanmaktadır.

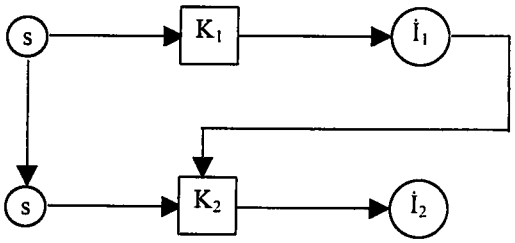


Şekil 6.6 Bağlı olmayan grafiğin saklanması

Böylece, sadece bir başlangıç düğümünün hafızadaki adreste saklanmasıyla, grafiğin tüm düğümlerine ait adreslere ulaşılabilmektedir. Örnekteki grafikte, her başlangıç düğümüne

bağlı (daire içinde "s" ile gösterilmiştir) olan normal düğümlerin hepsine alt grafik (subgraph) adı verilmiştir.

Yukarıdaki kaynak grafiğine sahip olan bir sistemde, $İ_1$ işleminin K_2 kaynağını talep etmesi durumunda kaynak grafiği Şekil 6.7-a'daki gibi olacaktır.



(a)

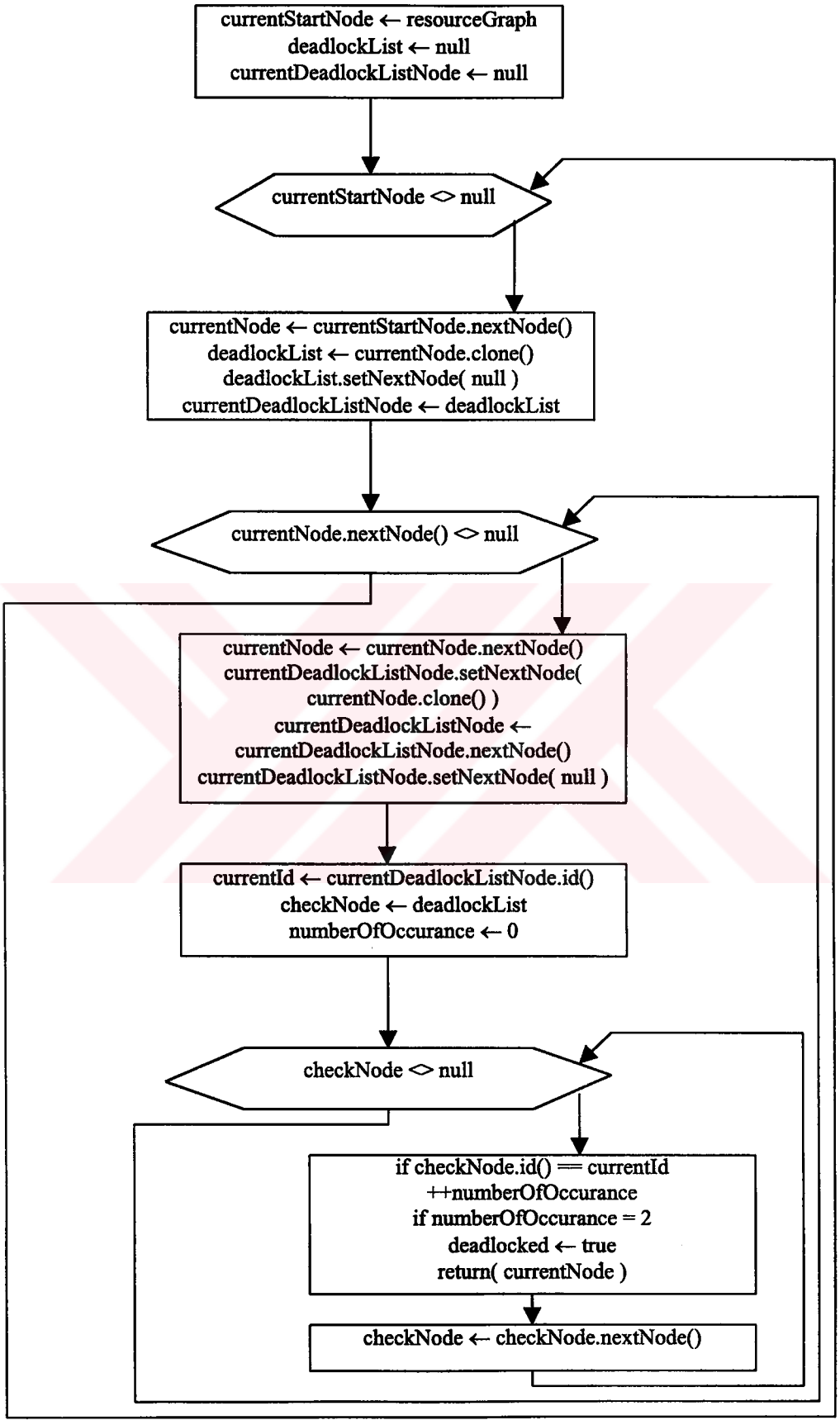
(b)

Şekil 6.7 Kaynak grafiğinin indirgenmesi

Ama bu durumda grafiğinin, sadece bir başlangıç düğümünün adresiyle saklanabileceğinden, grafiğin Şekil 6.7-b'deki duruma getirilmesi, yani indirgenmesi gerekmektedir (reduceGraph). Bu durumda, iki alt grafik birleştirilip bir alt grafik oluşturulmuştur. Alt grafiklerin belirlenebilmesi için, her alt grafiğe ait her düğümün bir alt grafik numarası (subgraphNumber) belirlenmiştir.

6.3.2 Kullanılan Kilitlenme Belirleme Algoritması

Kilitlenmenin belirlenmesinde 5.1 bölümünde anlatılan algoritma kullanılmıştır. Bu algoritmanın akışı Şekil 6.8'de verilmiştir.



Şekil 6.8 Kilitlenme Belirleme Algoritmasının Akışı

7. SONUÇ

Bu tez çalışmasında, çok işlemli merkezi sistemler için geliştirilmiş olan bir kilitlenme belirleme algoritması başarıyla uygulanmıştır. Algoritmanın uygulanmasında simülasyona gerek kalmamıştır çünkü programlama dili olarak, çok işlem parçacıklılığını destekleyen bir dil olan Java kullanılmıştır. Algoritmanın, kilitlenme oluşan her durumda, kilitlenmeyi doğru olarak tespit ettiği ve çözebildiği gözlemlenmiştir. Ayrıca algoritma, kilitlenme oluşmadığı halde kilitlenmenin var olduğu gibi yanlış sonuçlar da vermemektedir.

Program değişik işlem parçacığı ve kaynak sayıları için çalıştırılmış ve bu değişik durumlarda doğru çalışıp çalışmadığına bakılmıştır. İşlem parçacığı ve kaynak sayısının dört-beş civarında ve altında olduğu durumlarda, kaynak grafiği applet üzerinde çizdirilmiş ve kilitlenmenin meydana geldiği durum gözle gözlenebilmiştir. İşlem parçacığı ve kaynak sayısının büyük olduğu durumlarda ise, kaynak grafiği çizilmemiş, sadece kaç adımda kilitlenme oluştuğu, nasıl çözüldüğü vb. bilgiler applet üzerinde gösterilmiştir.

Daha sonra, algoritmanın bir uygulaması olarak, "Dining Philosophers" problemine uygulaması yapılmıştır. Bu problem kısaca şöyledir:

Yuvarlak bir masa etrafında, belli sayıda filozof oturmaktadır. Her iki filozof arasında bir çatal, masanın ortasında bir tabak içerisinde makarna bulunmaktadır. Her filozof, sürekli olarak, bir süre düşünüp biraz makarna yemekte ve bu sürekli olarak böylece devam etmektedir. Bir filozofun makarna yiyebilmesi için önce sağındaki sonra da solundaki çatalı alması gerekmektedir. Makarna yedikten sonra da çatalları masaya geri bırakmaktadır. Problem şu ki, her filozofun sağ tarafında bulunan çatalı alması durumunda, hepsi birbirlerini sonsuza dek beklemeye başlar ve kilitlenme meydana gelmiş olur.

Beş adet filozofa kadar değişik filozof sayıları için algoritmanın uygulaması yapılmış ve kilitlenmeyi doğru çözdüğü görülmüştür.

KAYNAKLAR

Akgün, S. F., (1989), "Deadlock Detection Problem in Computing Systems", Yüksek Lisans Tezi, Boğaziçi Üniversitesi.

December, J., Morrison, M., (1996), Java Unleashed, Sams.net publishing, USA.

Deitel, H. M., (1984), An Introduction To Operating Systems, Addison Wesley.

Deitel, H. M. ve Deitel, P. J., (1997), Java How To Program, Prentice-Hall.

Goscinski, A., (1991), Distributed Operating Systems, Addison Wesley.

Lea, D., (1997), Concurrent Programming in Java, Addison Wesley

Raynal, M., (1988), Distributed Algorithms and Protocols, Wiley Editions.

Stallings, W., (1998), Operating Systems, Internals and Design Principles, Prentice Hall.

Tanenbaum, A. S., (1992), Modern Operating Systems, Prentice-Hall.

EKLER

EK 1 Program Listesi

```
// Deadlock.java
// A program demonstrating a deadlock

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Deadlock extends Applet
{
    public static final int THRNUM = 2;
    public static final int RESOURCE_NUMBER = 2;
    public Detector detector;
    public static Process thread[];
    public static ThreadGroup threads;
    public static Resource resource[];

    public static DrawCanvas drawCanvas;
    public Panel bottomPanel;
    public Panel buttonPanel;
    public Panel textPanel[];

    public Button oneStep;
    public Button fiveSteps;
    public Button _25Steps;
    public Button _125Steps;
    public Button pauseContinue;
    public Button back;
    public Button forward;

    public TextArea allActions;
    public TextArea currentAction;
    public TextArea deadlockArea;

    public void init()
    {
        // create and run the detector thread
        detector = new Detector( "Detector", this );
        detector.setPriority( Thread.MAX_PRIORITY );
        detector.start();

        // create the resources
        resource = new Resource[ RESOURCE_NUMBER ];
        for ( int i = 0; i < RESOURCE_NUMBER; i++ )
        {
            resource[ i ] = new Resource( Integer.toString( i+1 ),
                detector );
        }

        // create and run the threads
        threads = new ThreadGroup( "Threads" );
        threads.setMaxPriority( Thread.MIN_PRIORITY );
        thread = new Process[ THRNUM ];
        for ( int i = 0; i < THRNUM; i++ )
        {
```

```
        thread[ i ] = new Process( resource, "Thread" + (i+1),
              threads );
        thread[ i ].setPriority( Thread.MIN_PRIORITY );
        thread[ i ].start();
}

setLayout( new BorderLayout() );

// create the canvas for the drawing area
drawCanvas = new DrawCanvas( this );

// create the panel for the button panel and the
// text areas
bottomPanel = new Panel();
bottomPanel.setLayout( new GridLayout( 1, 4, 0, 0 ) );

// add
add( drawCanvas, BorderLayout.CENTER );
add( bottomPanel, BorderLayout.SOUTH );

// create a panel for the buttons
buttonPanel = new Panel();
buttonPanel.setLayout( new GridLayout( 8, 1 ) );
// add
bottomPanel.add( buttonPanel );

// create the panels for the text areas
textPanel = new Panel[ 3 ];
for ( int i=0; i<3; i++ )
{
      textPanel[ i ] = new Panel();
      bottomPanel.add( textPanel[ i ] );
}

// create the buttons
oneStep = new Button( "1 step" );
fiveSteps = new Button( "5 steps" );
_25Steps = new Button( "25 steps" );
_125Steps = new Button( "125 steps" );
pauseContinue = new Button( "Pause / Continue" );
//_continue = new Button( "Continue" );
back = new Button( "Back" );
forward = new Button( "Forward" );
buttonPanel.add( oneStep );
buttonPanel.add( fiveSteps );
buttonPanel.add( _25Steps );
buttonPanel.add( _125Steps );
buttonPanel.add( pauseContinue );
//buttonPanel.add( _continue );
buttonPanel.add( back );
buttonPanel.add( forward );
ButtonHandler buttonHandler = new ButtonHandler( this );
oneStep.addActionListener( buttonHandler );
fiveSteps.addActionListener( buttonHandler );
_25Steps.addActionListener( buttonHandler );
_125Steps.addActionListener( buttonHandler );
pauseContinue.addActionListener( buttonHandler );
back.addActionListener( buttonHandler );
```

```
        forward.addActionListener( buttonHandler );

        // create the text areas
        allActions = new TextArea( 10, 25 );
        currentAction = new TextArea( 10, 25 );
        deadlockArea = new TextArea( 10, 25 );
        textPanel[ 0 ].add( allActions );
        textPanel[ 1 ].add( currentAction );
        textPanel[ 2 ].add( deadlockArea );
        allActions.setEditable( false );
        currentAction.setEditable( false );
        deadlockArea.setEditable( false );
    }


    public void paint( Graphics g )
    {
        drawCanvas.paint( g );
    }


    public void stop()
    {
        detector.stop();
        for ( int i = 0; i < THRNUM; i++ )
        {
            thread[ i ].stop();
        }
    }

}
```

```
// ButtonHandler.java
// The handler of the buttons

import java.awt.*;
import java.awt.event.*;

public class ButtonHandler implements ActionListener
{
    private static boolean pause = true;
    private Deadlock deadlock;


    // constructor
    public ButtonHandler( Deadlock deadlock )
    {
        this.deadlock = deadlock;
    }


    // action performed
    public void actionPerformed( ActionEvent e )
    {
        if ( e.getSource() == deadlock.oneStep )
        {
            oneStep();
        }
        else if ( e.getSource() == deadlock.fiveSteps )
        {
            fiveSteps();
        }
        else if ( e.getSource() == deadlock._25Steps )
        {
            _25Steps();
        }
        else if ( e.getSource() == deadlock._125Steps )
        {
            _125Steps();
        }
        else if ( e.getSource() == deadlock.pauseContinue )
        {
            pauseContinue();
        }
        else if ( e.getSource() == deadlock.back )
        {
            back();
        }
        else if ( e.getSource() == deadlock.forward )
        {
            forward();
        }
    }

    public void oneStep()
    {
        deadlock.detector.resume();
        Detector.runNumber = 1;
    }
```

```
    public void fiveSteps()
    {
        deadlock.detector.resume();
        Detector.runNumber = 5;
    }

    public void _25Steps()
    {
        deadlock.detector.resume();
        Detector.runNumber = 25;
    }

    public void _125Steps()
    {
        deadlock.detector.resume();
        Detector.runNumber = 125;
    }

    public void pauseContinue()
    {
        if ( pause )
        {
            deadlock.detector.suspend();
            pause = false;
        }
        else
        {
            deadlock.detector.resume();
            pause = true;
        }
    }

    public void back()
    {
        BackForward.backForward = BackForward.BACK;
        new Thread( new BackForward( deadlock ), "Back" ).run();
    }

    public void forward()
    {
        BackForward.backForward = BackForward.FORWARD;
        new Thread( new BackForward( deadlock ), "Forward" ).
            run();
    }

}
```

```
// BackForward.java
// The class for the back and forward threads

public class BackForward implements Runnable
{
    private Deadlock deadlock;
    public static final int NONE = 0;
    public static final int BACK = 1;
    public static final int FORWARD = 2;
    public static int backForward = NONE;

    // constructor
    public BackForward( Deadlock deadlock )
    {
        this.deadlock = deadlock;
    }

    public void run()
    {
        if ( backForward == BACK )
        {
            deadlock.drawCanvas.draw( deadlock.detector.
                previousGraphToDraw );
        }
        else
        {
            deadlock.drawCanvas.draw( deadlock.detector.
                graphToDraw );
        }
    }

}
```

```
// DrawCanvas.java
// The area on the applet where the resource graph is drawn

import java.awt.*;
import java.lang.Math;

public class DrawCanvas extends Canvas
{
    private final int DRAW_UNITS = 6;
    private final int FONT_SIZE = ( int ) ( 1.2 * Detector.
        DRAW_DISTANCE / DRAW_UNITS );
    private final int ARC_LENGTH = ( Detector.DRAW_DISTANCE /
        DRAW_UNITS ) / 1 ;
    private final double ARC_ANGLE = Math.PI / 8;

    private Deadlock deadlock;
    private DrawStartNode graphToDraw;

    public DrawCanvas( Deadlock deadlock )
    {
        this.deadlock = deadlock;
    }

    public void draw( DrawStartNode graphToDraw )
    {
        this.graphToDraw = graphToDraw;
        repaint();
    }

    public void paint( Graphics g )
    {

        DrawNode drawNodeInDeadlock = null;
        int drawNodeInDeadlockPassed = 0;
        if ( Detector.deadlocked )
        {
            drawNodeInDeadlock = deadlock.detector.findDrawNode(
                deadlock.detector.nodeInDeadlock() );
        }

        DrawStartNode currentDrawStartNode = graphToDraw;
        DrawNode currentDrawNode = currentDrawStartNode;

        while ( currentDrawStartNode != null )
        {

            boolean endOfSubgraph = false; // the subgraph ends
            // either if the next node is null or the id of the
            // next node is not equal to the current id

            do
            {
                currentDrawNode = currentDrawNode.nextDrawNode();

                if ( ( Detector.deadlocked ) && ( currentDrawNode ==
                    drawNodeInDeadlock ) )
                {
                    drawNodeInDeadlockPassed += 1;
```

```
        if ( drawNodeInDeadlockPassed == 2 )
        {
            break;
        }
    }

    if ( BackForward.backForward == BackForward.BACK )
    {

        deadlock.deadlockArea.append( "\nBack" );

        if ( currentDrawNode.id().equals( Detector.
            previousModifiedNodeName ) )
        {
            g.setColor( Color.red );
        }
    }
    else if ( BackForward.backForward == BackForward.
        FORWARD )
    {

        deadlock.deadlockArea.append( "\nForward" );

        if ( currentDrawNode.id().equals( Detector.
            modifiedNodeName ) )
        {
            g.setColor( Color.red );
        }
    }
    else if ( BackForward.backForward == BackForward.NONE )
    {

        deadlock.deadlockArea.append( "\nNonee" );

        if ( currentDrawNode.id().equals( Detector.
            modifiedNodeName ) )
        {
            g.setColor( Color.red );
        }
    }


    // draw the node and its edge, if exists
    drawTheNode( g, currentDrawNode );

    g.setColor( Color.black );

    // look if the end of the subgraph is reached
    if ( currentDrawNode.nextDrawNode() == null )
    {
        endOfSubgraph = true;
    }
    else if ( currentDrawNode.nextDrawNode().
        graphNumber()!= currentDrawNode.graphNumber() )
    {
        endOfSubgraph = true;
    }
```

```
            } while ( !endOfSubgraph );

            currentDrawStartNode = currentDrawStartNode.
                nextDrawStartNode();
            currentDrawNode = currentDrawStartNode;
        }

        BackForward.backForward = BackForward.NONE;
    }

    // draws the node, its name and if exists, the edge
    // originating from it
    public void drawTheNode( Graphics g, DrawNode nodeToDraw )
    {
        int drawUnit = Detector.DRAW_DISTANCE / DRAW_UNITS;

        String fontName = g.getFont().getName();
        int fontStyle = g.getFont().getStyle();
        int fontSize = g.getFont().getSize();
        g.setFont( new Font( fontName, fontStyle, FONT_SIZE ) );

        // get the draw coordinates of the node
        int drawX = nodeToDraw.drawX();
        int drawY = nodeToDraw.drawY();

        // move the cursor to the upper left corner of the node
        int cursorX = drawX;
        int cursorY = drawY - drawUnit;

        // draw the node
        if ( nodeToDraw.id().startsWith( "T" ) )
        {
            g.drawOval( cursorX, cursorY, 2*drawUnit, 2*drawUnit );

            // move the cursor to the bottom left corner of the
            // string to be drawn and draw the string
            cursorX = drawX + drawUnit - ( FONT_SIZE / 2 );
            cursorY = drawY + FONT_SIZE / 2 ;
            g.drawString( nodeToDraw.id(), cursorX, cursorY );
        }
        else
        {
            g.drawRect( cursorX, cursorY, 2*drawUnit,
                2*drawUnit );

            // move the cursor to the bottom left corner of the
            // string to be drawn and draw the string
            cursorX = drawX + drawUnit / 3 ;
            cursorY = drawY + FONT_SIZE / 2 ;
            g.drawString( nodeToDraw.id(), cursorX, cursorY );
        }

        // look whether the next node is null or not
        // if so, we are ok, return
        if ( nodeToDraw.nextDrawNode() == null )
        {
            return;
        }
```

```
// the next node is not null, so we have to draw the edge
// get the draw coordinates of the next node
int nextDrawX = nodeToDraw.nextDrawNode().drawX();
int nextDrawY = nodeToDraw.nextDrawNode().drawY();

// determine the origin and destination coordinates
// of the edge to be drawn
int originX;
int originY;
int destinationX;
int destinationY;

if ( nextDrawY == drawY )
{
      // The two nodes are in the same subgraph.
      // The next node cannot be at the left of the first one,
      // which would mean that a deadlock was occured
      // so nextDrawX > drawX
      originX = drawX + 2*drawUnit;
      originY = drawY;
      destinationX = nextDrawX;
      destinationY = nextDrawY;
}
else if ( nextDrawY > drawY )
{
      // downward
      originX = drawX + drawUnit;
      originY = drawY + drawUnit;
      destinationX = nextDrawX + drawUnit;
      destinationY = nextDrawY - drawUnit;
}
else
{
      // upward
      originX = drawX + drawUnit;
      originY = drawY - drawUnit;
      destinationX = nextDrawX + drawUnit;
      destinationY = nextDrawY + drawUnit;
}

// now we have got the origin and destination coordinates
// of the edge to be drawn
if ( ( destinationY == originY ) && ( destinationX <
      originX ) )
{
      int add = ( int ) ( Detector.DRAW_DISTANCE / 3 );
      g.drawLine( originX, originY, originX - add , originY -
            add );
      g.drawLine( originX - add , originY - add, destinationX +
            add, destinationY - add );
      g.drawLine( destinationX +  add, destinationY - add,
            destinationX, destinationY );
      originX = destinationX + add;
      originY = destinationY - add;
}
else
{
```

```
            g.drawLine( originX, originY, destinationX, destinationY );
        }

        // draw the arrow
        drawArrow( g, originX, originY, destinationX, destinationY );

    }


    public void drawArrow( Graphics g, int originX, int originY,
        int destinationX, int destinationY )
    {
        double pi = Math.PI;
        double alpha;
        double beta;
        double gama;
        int leftX;
        int leftY;
        int rightX;
        int rightY;

        if ( destinationY > originY )
        {
            if ( destinationX > originX )
            {
                alpha = Math.atan( ( ( double ) destinationY - originY )
                    / ( destinationX - originX ) );
                if ( alpha >= ARC_ANGLE )
                {
                    beta = ( pi - ( 3 * ARC_ANGLE ) ) - alpha;
                    gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                    leftX = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                    leftY = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                    rightX = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                    rightY = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                    g.drawLine( destinationX, destinationY,
                        destinationX +leftX, destinationY - leftY);
                    g.drawLine( destinationX, destinationY,
                        destinationX - rightX, destinationY - rightY);
                }
                else
                {
                    beta = ( ( pi / 2 ) - ARC_ANGLE ) - alpha;
                    gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                    leftX = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                    leftY = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                    rightX = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                    rightY = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                    g.drawLine( destinationX, destinationY,
                        destinationX - leftX, destinationY - leftY);
                    g.drawLine( destinationX, destinationY,
                        destinationX - rightX, destinationY + rightY);
                }
            }
            else if ( destinationX == originX )
            {
                beta = ( pi / 2 ) - ARC_ANGLE;
                gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
```

```
            leftX = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
            leftY = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
            rightX = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
            rightY = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
            g.drawLine( destinationX, destinationY,
                  destinationX +leftX, destinationY - leftY);
            g.drawLine( destinationX, destinationY,
                  destinationX - rightX, destinationY - rightY);
      }
      else // destinationX < originX
      {
            alpha = Math.atan( ( ( double ) destinationY - originY )
                  / ( originX - destinationX ) );
            if ( alpha >= ARC_ANGLE  )
            {
                  beta = alpha - ARC_ANGLE;
                  gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                  leftX = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                  leftY = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                  rightX = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                  rightY = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                  g.drawLine( destinationX, destinationY,
                        destinationX +leftX, destinationY - leftY);
                  g.drawLine( destinationX, destinationY,
                        destinationX - rightX, destinationY - rightY);
            }
            else
            {
                  beta = ( ( pi / 2 ) - ARC_ANGLE ) + alpha;
                  gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                  leftX = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                  leftY = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                  rightX = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                  rightY = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                  g.drawLine( destinationX, destinationY,
                        destinationX +leftX, destinationY + leftY);
                  g.drawLine( destinationX, destinationY,
                        destinationX +rightX, destinationY - rightY);
            }
      }
}
else if ( destinationY == originY )
{
      if ( destinationX > originX )
      {
            beta = ( pi / 2 )- ARC_ANGLE;
            gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
            leftX = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
            leftY = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
            rightX = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
            rightY = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
            g.drawLine( destinationX, destinationY,
                  destinationX - leftX, destinationY - leftY);
            g.drawLine( destinationX, destinationY,
                  destinationX - rightX, destinationY + rightY);
      }
      else // destinationX < originX
      {
```

```
            beta = ( pi / 2 ) - ARC_ANGLE;
            gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
            leftX = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
            leftY = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
            rightX = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
            rightY = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
            g.drawLine( destinationX, destinationY,
                destinationX +leftX, destinationY + leftY);
            g.drawLine( destinationX, destinationY,
                destinationX +rightX, destinationY - rightY);
        }
    }
    else if ( destinationY < originY )
    {
        if ( destinationX > originX )
        {
            alpha = Math.atan( ( ( double ) originY - destinationY )
                / ( destinationX - originX ) );
            if ( alpha >= ARC_ANGLE )
            {
                beta = alpha - ARC_ANGLE;
                gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                leftX = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                leftY = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                rightX = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                rightY = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                g.drawLine( destinationX, destinationY,
                    destinationX - leftX, destinationY + leftY);
                g.drawLine( destinationX, destinationY,
                    destinationX +rightX, destinationY + rightY);
            }
            else
            {
                beta = ( ( pi / 2 ) - ARC_ANGLE ) + alpha;
                gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                leftX = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                leftY = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                rightX = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                rightY = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                g.drawLine( destinationX, destinationY,
                    destinationX - leftX, destinationY - leftY);
                g.drawLine( destinationX, destinationY,
                    destinationX - rightX, destinationY + rightY);
            }
        }
        else if ( destinationX == originX )
        {
            beta = ( pi / 2 ) - ARC_ANGLE;
            gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
            leftX = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
            leftY = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
            rightX = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
            rightY = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
            g.drawLine( destinationX, destinationY,
                destinationX - leftX, destinationY + leftY);
            g.drawLine( destinationX, destinationY,
                destinationX +rightX, destinationY + rightY);
        }
```

```
                else // destinationX < originX
                {
                    alpha = Math.atan( ( ( double ) originY - destinationY )
                        / ( originX - destinationX ) );
                    if ( alpha >= ARC_ANGLE  )
                    {
                        beta = ( pi - ARC_ANGLE ) - alpha;
                        gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                        leftX = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                        leftY = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                        rightX = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                        rightY = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                        g.drawLine( destinationX, destinationY,
                            destinationX - leftX, destinationY + leftY);
                        g.drawLine( destinationX, destinationY,
                            destinationX +rightX, destinationY + rightY);
                    }
                    else
                    {
                        beta = ( ( pi / 2 ) - ARC_ANGLE ) - alpha;
                        gama = ( pi - ( 2 * ARC_ANGLE ) ) - beta;
                        leftX = ( int ) ( Math.sin( beta ) * ARC_LENGTH );
                        leftY = ( int ) ( Math.cos( beta ) * ARC_LENGTH );
                        rightX = ( int ) ( Math.sin( gama ) * ARC_LENGTH );
                        rightY = ( int ) ( Math.cos( gama ) * ARC_LENGTH );
                        g.drawLine( destinationX, destinationY,
                            destinationX +leftX, destinationY + leftY);
                        g.drawLine( destinationX, destinationY,
                            destinationX +rightX, destinationY - rightY);
                    }
                }
            }
        }

}
```

```
// Process.java
// Part of a program demonstrating a simple deadlock

public class Process extends Thread
{
    public static final int MAX_SLEEP_TIME = 1;
    private final int LOOP_NUMBER = 555;
    private int sleepTime;
    private Resource resource[];

    // constructor
    public Process( Resource resource[], String name,
        ThreadGroup threadGroup )
    {
        super( threadGroup, name );
        this.resource = resource;
        sleepTime = ( int ) ( java.lang.Math.random() *
            MAX_SLEEP_TIME );
    }

    // all the threads use this method to sleep for a random duration
    public void sleepRandom()
    {
        try
        {
            Thread.currentThread().sleep( sleepTime );
        }
        catch ( InterruptedException e )
        {
            System.err.println( "Exception: " + e.toString() );
        }
    }

    public void startFromTheBeginning()
    {
        this.stop();
        this.run();
    }

    public void run()
    {
        for ( int loop = 1; loop <= LOOP_NUMBER; loop++)
        {
            // randomly select a first resource to try to get
            // and try to get the remaining ones in the increasing
            // order of the resource numbers
            int firstResource = ( int ) ( Deadlock.
                RESOURCE_NUMBER * java.lang.Math.random() );

            for( int i=firstResource; i < resource.length; i++ )
            {
                resource[ i ].get();
            }

            for( int i=0; i < firstResource; i++ )
            {
                resource[ i ].get();
```

```
                    // sleep for a while before releasing the Resource
                    sleepRandom();

                }

                for( int i=0; i<resource.length; i++ )
                {
                    resource[ i ].release();
                }
            }
        }
}
```

```
// Resource.java
// Class defining the resourses

public class Resource
{
    private Detector detector;
    private String name; // the name of the resource
    private boolean free; // true indicates that the resource
    // is available

    public Resource( String name, Detector detector )
    {
        this.name = "R" + name;
        this.detector = detector;
        free = true;
    }

    public String name()
    {
        return( name );
    }

    public synchronized void free()
    {
        free = true;
        notifyAll();

        System.out.println( "Resource " + name + " freed." );

    }

    // method to get the resource
    public synchronized void get()
    {

        while ( Detector.scheduler.threadPermission() == false )
        {
            Detector.scheduler.schedulerWait();
        }

        // about to request a Resource,
        // so modify the request list of the Detector
        // before entering the wait mode
        String threadName = Thread.currentThread().getName();
        String resourceName = name;
        detector.addRequestListNode( threadName, resourceName );

        while ( !free )
        {
            try
            {
                wait();
            }
            catch ( InterruptedException e )
            {
                System.err.println( "Exception: " + e.toString() );
            }
        }
```

```
        free = false;

        while ( Detector.scheduler.threadPermission() == false )
        {
            Detector.scheduler.schedulerWait();
        }

        // just acquired a Resource,
        // so modify the acquire list of the Detector
        detector.addAcquireListNode( threadName, resourceName );
    }

    public synchronized void release()
    {
        free = true;

        while ( Detector.scheduler.threadPermission() == false )
        {
            Detector.scheduler.schedulerWait();
        }

        // just released a Resource,
        // so modify the release list of the Detector
        // before notifying
        String threadName = Thread.currentThread().getName();
        String resourceName = name;
        detector.addReleaseListNode( threadName, resourceName );
        notify();
    }
}
```

```
// Scheduler.java
// To control when to run the regular threads

public class Scheduler
{

    private boolean threadPermission;

    public Scheduler()
    {
        threadPermission = false;
    }

    public synchronized boolean threadPermission()
    {
        if ( threadPermission )
        {
            setThreadPermission( false );
            return( true );
        }
        else
        {
            return( false );
        }
    }

    public synchronized void setThreadPermission( boolean
        threadPermission )
    {
        this.threadPermission = threadPermission;
    }

    public synchronized void schedulerWait()
    {
        try
        {
            wait();
        }
        catch ( InterruptedException e ) {}
    }

    public synchronized void schedulerNotifyAll()
    {
        notifyAll();
    }
}
```

```
// Detector.java
// Thread to detect the deadlocks

import java.applet.Applet;

public class Detector extends Thread
{
    private final long LOOP_NUMBER = 1; // 8 0's
    private final long DELAY_FOR_PRINT = 1; // 8 0's
    public static final int DRAW_DISTANCE = 60;

    public static int runNumber = 1;
    private int totalRunNumber = 0;
    private int totalDeadlockNumber = 0;

    public static boolean deadlocked = false;

    public static Scheduler scheduler = new Scheduler(); // to
    // schedule the regular threads

    private ListNode requestList; // used for maintaining the
    private ListNode acquireList; // request, acquire and
    private ListNode releaseList; // release lists

    private StartNode resourceGraph; // the resource graph
    public DrawStartNode graphToDraw; // the graph to draw
    public DrawStartNode previousGraphToDraw; // the graph to draw

    public static String modifiedNodeName;
    public static String previousModifiedNodeName;

    private Deadlock deadlock;

    // constructor
    public Detector( String name, Deadlock deadlock )
    {
        super( name );
        this.deadlock = deadlock;
        requestList = null;
        acquireList = null;
        releaseList = null;
        resourceGraph = new StartNode( "S" );
    }


    // Adds a node to the end of the request list
    public synchronized void addRequestListNode( String threadName,
        String resourceName )
    {
        ListNode currentListNode; // used to navigate till
        // the end of the list

        // print the contents of the current list
        ListNode printNode = requestList;

        while ( printNode != null )
        {
            printNode = printNode.nextListNode();
```

```
        }

        while ( deadlock.currentAction == null ) {}
        deadlock.currentAction.setText( threadName +
            " requests " + "Resource" + resourceName.
            substring( 1 ) + "\n" );

        while ( deadlock.allActions == null ) {}
        totalRunNumber++;
        deadlock.allActions.append( Integer.toString(
            totalRunNumber ) );
        deadlock.allActions.append( threadName + " requests "
            + "Resource" + resourceName.substring( 1 ) + "\n" );

        // navigate till the end of the list
        currentListNode = requestList;

        if ( currentListNode == null )
        {
            // add the new ListNode to the end of the list
            requestList = new ListNode( threadName, resourceName );
        }
        else
        {
            while ( currentListNode.nextListNode() != null )
            {
                currentListNode = currentListNode.nextListNode();
            }

            // add the new ListNode to the end of the list
            currentListNode.setNextListNode( new ListNode(
                threadName, resourceName ) );
        }

        // notify the detector which is waiting for an update
        // of the lists
        notify();

    }


    // Adds a node to the end of the request list
    public synchronized void addAcquireListNode( String threadName,
        String resourceName )
    {
        ListNode currentListNode; // used to navigate till
        // the end of the list

        if ( deadlock.allActions != null )
        {
            totalRunNumber++;
            deadlock.allActions.append( Integer.toString(
                totalRunNumber ) );
            deadlock.allActions.append( threadName + " acquires "
                + "Resource" + resourceName.substring( 1 ) + "\n" );
        }
        if ( deadlock.currentAction != null )
        {
```

```
            deadlock.currentAction.setText( threadName +
                " acquires " + "Resource" + resourceName.
                substring( 1 ) + "\n" );
        }

        // navigate till the end of the list
        currentListNode = acquireList;

        if ( currentListNode == null )
        {
            // add the new ListNode to the end of the list
            acquireList= new ListNode( threadName, resourceName );
        }
        else
        {
            while ( currentListNode.nextListNode() != null )
            {
                currentListNode = currentListNode.nextListNode();
            }

            // add the new ListNode to the end of the list
            currentListNode.setNextListNode( new ListNode(
                threadName, resourceName ) );
        }

        // notify the detector which is waiting for an update
        // of the lists
        notify();

}


// Adds a node to the end of the request list
public synchronized void addReleaseListNode( String threadName,
        String resourceName )
{
        ListNode currentListNode; // used to navigate till
        // the end of the list

        if ( deadlock.allActions != null )
        {
            totalRunNumber++;
            deadlock.allActions.append( Integer.toString(
                totalRunNumber ) );
            deadlock.allActions.append( threadName + " releases "
                + "Resource" + resourceName.substring( 1 ) + "\n" );
        }
        if ( deadlock.currentAction != null )
        {
            deadlock.currentAction.setText( threadName +
                " releases " + "Resource" + resourceName.
                substring( 1 ) + "\n" );
        }


        // navigate till the end of the list
        currentListNode = releaseList;
```

```
        if ( currentListNode == null )
        {
            // add the new ListNode to the end of the list
            releaseList= new ListNode( threadName, resourceName );
        }
        else
        {
            while ( currentListNode.nextListNode() != null )
            {
                currentListNode = currentListNode.nextListNode();
            }

            // add the new ListNode to the end of the list
            currentListNode.setNextListNode( new ListNode(
                threadName, resourceName ) );
        }

        // notify the detector which is waiting for an update
        // of the lists
        notify();

    }


    // modify the graph with respect to the lists
    public synchronized void modifyGraph()
    {
        // The regular threads should wait till the components
        // of the applet are created.
        while ( deadlock.deadlockArea == null )
        {
        }

        scheduler.setThreadPermission( true );
        scheduler.schedulerNotifyAll();

        // wait until at least one of the lists is updated
        try
        {
            wait();
        }
        catch ( InterruptedException e ) {}

        // suspend untill resumed by one of the buttons
        runNumber -= 1;
        if ( runNumber == 0 )
        {
            suspend();
        }

        previousGraphToDraw = graphToDraw;

        // navigate till the end of the request list
        // and add the new nodes to the graph
        while ( requestList != null )
        {
            addRequestNodes( requestList.threadName(),
                requestList.resourceName() );
```

```
            modifiedThreadNodeName( requestList.threadName() );
            requestList = requestList.nextListNode();
        }

        // navigate till the end of the acquire list
        // and modify the acquire nodes
        while ( acquireList != null )
        {
            modifyAcquireNodes( acquireList.threadName(),
                acquireList.resourceName() );
            modifiedThreadNodeName( acquireList.threadName() );
            acquireList = acquireList.nextListNode();
        }

        // navigate till the end of the request list
        // and modify the release nodes
        while ( releaseList != null )
        {
            modifyReleaseNodes( releaseList.threadName(),
                releaseList.resourceName() );
            modifiedThreadNodeName( releaseList.threadName() );
            releaseList = releaseList.nextListNode();
        }

}


private synchronized void modifiedThreadNodeName( String
        modifiedThreadNodeName )
{
        if ( modifiedNodeName != null )
        {
            previousModifiedNodeName = modifiedNodeName.substring( 0 );
        }
        modifiedNodeName = "T" + modifiedThreadNodeName.
            substring( 6 );
}



// add the request nodes to the graph
public synchronized void addRequestNodes( String threadName,
        String resourceName)
{
        // look whether the nodes already exist or not
        Node threadNode = findNode( threadName );
        Node resourceNode = findNode( resourceName );

        if ( threadNode != null )
        {
            if ( resourceNode != null )
            {
                // both of the nodes already exist
                // so create an edge from the thread node
                // to the resource node
                threadNode.setNextNode( resourceNode );

                // find the previous node of the resource
```

```
            // node, because the graph may be reducable
            Node previousNode = previousNodeOf( resourceNode );

            if ( ( previousNode instanceof StartNode ) &&
                 ( threadNode.graphNumber() != resourceNode.
                 graphNumber() ) )
            {
                 // the destination of the new edge is the first
                 // element of a subgraph, so the subgraph should
                 // be attached to the graph (reduce)
                 reduceGraph( threadNode, ( StartNode )
                      previousNode );
            }
       }
       else // resourceNode == null
       {
            // the thread node does, but the resource node does
            // not exist in the graph so it should be created
            resourceNode = new Node( resourceName, threadNode.
                 graphNumber() );
            threadNode.setNextNode( resourceNode );
       }
  }
  else // threadNode == null
  {
       if ( resourceNode != null )
       {
            // the thread node does not exist, the resource node
            // exists so create a thread node, a start node, an
            // edge from it to the new node and an edge from the
            // new node to the resource node
            StartNode newStartNode = newStartNode();
            threadNode = new Node( threadName,  newStartNode.
                 hashCode() );
            newStartNode.setNextNode( threadNode );
            threadNode.setNextNode( resourceNode );

            // find the previous node of the resource
            // node, because the graph may be reducable
            Node previousNode = previousNodeOf( resourceNode );

            if ( previousNode instanceof StartNode )
            {
                 // the destination of the new edge is the first
                 // element of a subgraph, so the subgraph should
                 // be attached to the graph (reduce)
                 reduceGraph( threadNode, ( StartNode )
                      previousNode );
            }
       }
       else
       {
            // both the thread node and the resource node does
            // not exist, so create the nodes and a start node
            // and the necessary edges
            StartNode newStartNode = newStartNode();
            threadNode = new Node( threadName,  newStartNode.
                 hashCode() );
```

```
                resourceNode = new Node( resourceName, threadNode.
                    graphNumber() );
                newStartNode.setNextNode( threadNode );
                threadNode.setNextNode( resourceNode );
            }
        }

}

// modifies the acquire nodes
public synchronized void modifyAcquireNodes( String threadName,
    String resourceName)
{
    // find the nodes
    Node threadNode = findNode( threadName );
    Node resourceNode = findNode( resourceName );

    // both of the nodes already exist because an
    // acquisition of a resource cannot occur without
    // a request made before
    // so reverse the edge between the thread node
    // and the resource node

    Node previousNode = previousNodeOf( threadNode );

    if ( previousNode instanceof StartNode )
    {
        // if the thread node is the first node of a subgraph
        // the nodes may be interchanged in the resource graph
        previousNode.setNextNode( resourceNode );
        resourceNode.setNextNode( threadNode );
        threadNode.setNextNode( null );// here, the next
        // node of the resource node is not lost because there
        // cannot be a next node, which would mean that the
        // resource was already acquired by another thread

        Node previousNodeInDifferentSubgraph =
            previousNodeInDifferentSubgraph( resourceNode );
        if ( previousNodeInDifferentSubgraph != null )
        {
            reduceGraph( previousNodeInDifferentSubgraph,
                startNodeOf( resourceNode ) );
        }
    }
    else
    {
        // otherwise a new subgraph should be created for the
        // resource node
        StartNode newStartNode = newStartNode();
        newStartNode.setNextNode( resourceNode );
        resourceNode.setGraphNumber( newStartNode.hashCode() );
        resourceNode.setNextNode( threadNode );
        threadNode.setNextNode( null );// here, the next
        // node of the resource node is not lost because there
        // cannot be a next node, which would mean that the
        // resource was already acquired by another thread
    }
```

```
        if ( ( previousNode instanceof StartNode ) &&
             ( threadNode.graphNumber() !=
             resourceNode.graphNumber() ) )
        {
             // the destination of the new edge is the first
             // element of a subgraph, so the subgraph should
             // be attached to the graph (reduce)
             reduceGraph( threadNode, ( StartNode ) previousNode );
        }
    }


    // modifies the release nodes
    public synchronized void modifyReleaseNodes( String threadName,
        String resourceName)
    {
        // find the nodes
        Node resourceNode = findNode( resourceName );
        Node threadNode = findNode( threadName );

        // both of the nodes already exist because a
        // release of a resource cannot occur without
        // a request and acquire made before
        // so delete the edge between the resource node
        // and the thread node

        // clear the edge between them
        resourceNode.setNextNode( null );

        Node previousNode = previousNodeOf( resourceNode );
        if ( previousNode instanceof StartNode )
        {
             // the source of the just deleted edge is the first
             // element of a subgraph, so the subgraph should
             // be deleted (only the start node is deleted here,
             // because the source node will be garbage collected
             deleteStartNode( ( StartNode ) previousNode );
        }

        //deadlock.deadlockArea.append( "\npre res ok" );

        previousNode = previousNodeOf( threadNode ); // the
        // previous node cannot be a start node

        //deadlock.deadlockArea.append( "\nsuspended" );
        //suspend();

        if ( previousNode != null )
        // if it is null, nothing necessary to do
        // because the thread node will be garbage collected
        {
             // the thread node still has a previous node but
             // was it in the same subgraph with its previous node
             // before the deletion of the edge or not
             if ( previousNode.graphNumber() != threadNode.
                  graphNumber() )
             // if they are equal nothing necessary to do
             {
```

```
                // they were not in the same subgraph
                // so attach the remaining nodes to the subgraph
                // containing previous node of the thread node
                attachSubgraph( previousNode, threadNode );
            }
        }

}


// finds the node with its id equal to a given id
public synchronized Node findNode( String nodeId )
{
    if ( resourceGraph == null )
    {
        return( null );
    }

    // if the graph is empty the node cannot exist
    if ( resourceGraph.nextNode() == null )
    {
        return( null );
    }

    StartNode currentStartNode = resourceGraph;
    Node currentNode = currentStartNode;
    boolean found = false;

    while ( ( !found ) && ( currentStartNode != null ) )
    {
        boolean endOfSubgraph = false; // the subgraph ends
        // either if the next node is null or the id of the
        // next node is not equal to the current id

        do
        {
            currentNode = currentNode.nextNode();

            // look if the end of the subgraph is reached
            if ( currentNode.nextNode() == null )
            {
                endOfSubgraph = true;
            }
            else if ( currentNode.nextNode().graphNumber()!=
                currentNode.graphNumber() )
            {
                endOfSubgraph = true;
            }

            if ( currentNode.id().equals( nodeId ) )
            {
                found = true;
            }
        } while ( ( !found ) && ( !endOfSubgraph ) );

        if ( !found )
        {
            currentStartNode = currentStartNode.nextStartNode();
```

```
                currentNode = currentStartNode;
            }
        }

        if ( found )
        {
            return( currentNode );
        }
        else
        {
            return( null );
        }
}


// finds the previous node of a given node
// if it has a previous node in the same subgraph
// that node is returned
// otherwise, if exists, any other previous node is
// or null is returned
public synchronized Node previousNodeOf( Node node )
{
    if ( resourceGraph == null )
    {
        return( null );
    }

    // look if the node can be found given its id
    // if not, this means that because of a recent
    // modification of the graph, it does not have a
    // previous node in the same subgraph with itself
    if ( findNode( node.id() ) != null )
    {
        return( previousNodeInSameSubgraph( node ) );
    }
    else
    {
        return( previousNodeInDifferentSubgraph( node ) );
    }
}


// finds the previous node of a given node
// given that it has a previous node in its subgraph
public synchronized Node previousNodeInSameSubgraph(
    Node node )
{
    StartNode currentStartNode = resourceGraph;
    Node currentNode = currentStartNode;
    Node previousNode = currentStartNode;
    boolean found = false;

    while ( ( !found ) && ( currentStartNode != null ) )
    {
        boolean endOfSubgraph = false; // the subgraph ends
        // either if the next node is null or the id of the
        // next node is not equal to the current id
```

```
            do
            {
                previousNode = currentNode;
                currentNode = currentNode.nextNode();

                // look if the end of the subgraph is reached
                if ( currentNode.nextNode() == null )
                {
                    endOfSubgraph = true;
                }
                else if ( currentNode.nextNode().graphNumber()!=
                    currentNode.graphNumber() )
                {
                    endOfSubgraph = true;
                }

                if ( currentNode == node )
                {
                    found = true;
                }
            } while ( ( !found ) && ( !endOfSubgraph ) );

            if ( !found )
            {
                currentStartNode = currentStartNode.nextStartNode();
                currentNode = currentStartNode;
            }
        }
        return( previousNode );

}


// finds the previous node of a given node
// given that it does not have a previous node in its subgraph
// if not any, it returns null
public synchronized Node previousNodeInDifferentSubgraph(
    Node node )
{
    if ( resourceGraph == null )
    {
        return( null );
    }
    if ( resourceGraph.nextNode() == null )
    {
        return( null );
    }

    StartNode currentStartNode = resourceGraph;
    Node currentNode = currentStartNode;
    boolean found = false;

    while ( ( !found ) && ( currentStartNode != null ) )
    {
        boolean endOfSubgraphGraphNumber = false; // indicates
        // that we are at the end of a subgraph because the
        // graph number of the next node is different from
        // the current one
```

```
            boolean endOfSubgraphNull = false; // indicates
            // that we are at the end of a subgraph because the
            // next node is null

            do
            {
                //previousNode = currentNode;
                currentNode = currentNode.nextNode();

                // look if the end of the subgraph is reached
                if ( currentNode.nextNode() == null )
                {
                    endOfSubgraphNull = true;
                }
                else if ( currentNode.nextNode().graphNumber()!=
                    currentNode.graphNumber() )
                {
                    endOfSubgraphGraphNumber = true;
                }

                if ( endOfSubgraphGraphNumber && ( currentNode.
                    nextNode() == node ) )
                {
                    found = true;
                }
            } while ( ( !found ) && ( !endOfSubgraphNull ) &&
                ( !endOfSubgraphGraphNumber ) );

            if ( !found )
            {
                currentStartNode = currentStartNode.nextStartNode();
                currentNode = currentStartNode;
            }
        }

        if ( found )
        {
            return( currentNode );
        }
        else
        {
            return( null );
        }

    }


    // finds the start node of a given node
    public synchronized StartNode startNodeOf( Node node )
    {
        StartNode currentStartNode = resourceGraph;
        Node currentNode = currentStartNode;
        boolean found = false;

        while ( ( !found ) && ( currentStartNode != null ) )
        {
            boolean endOfSubgraph = false; // the subgraph ends
            // either if the next node is null or the id of the
```

```
            // next node is not equal to the current id

            do
            {
                currentNode = currentNode.nextNode();

                // look if the end of the subgraph is reached
                if ( currentNode.nextNode() == null )
                {
                    endOfSubgraph = true;
                }
                else if ( currentNode.nextNode().graphNumber()!=
                    currentNode.graphNumber() )
                {
                    endOfSubgraph = true;
                }

                if ( currentNode == node )
                {
                    found = true;
                }
            } while ( ( !found ) && ( !endOfSubgraph ) );

            if ( !found )
            {
                currentStartNode = currentStartNode.nextStartNode();
                currentNode = currentStartNode;
            }
        }

        return( currentStartNode );

}


// creates a new start node for a new subgraph and
// returns this new start node if there is at least two
// nodes in the graph, otherwise it returns the origin
// of the graph
public synchronized StartNode newStartNode()
{
        if ( resourceGraph == null )
        {
            resourceGraph = new StartNode( "S" );
            return( resourceGraph );
        }
        if ( resourceGraph.nextNode() == null )
        {
            return( resourceGraph );
        }
        else
        {
            StartNode currentStartNode = resourceGraph;
            while ( currentStartNode.nextStartNode() != null )
            {
                currentStartNode = currentStartNode.nextStartNode();
            }
```

```
            StartNode newStartNode = new StartNode( "S" );
            currentStartNode.setNextStartNode( newStartNode );
            return( newStartNode );
        }
    }

    // deletes a start node
    // only the start node is deleted because its subgraph
    // will be garbage collected, if not null of course
    public synchronized void deleteStartNode( StartNode startNode )
    {
        // if the first start node is to be deleted
        // first start node should be modified
        if ( startNode == resourceGraph )
        {
            resourceGraph = resourceGraph.nextStartNode();
        }
        else
        {
            // find the previous start node of the given one
            StartNode previousStartNode = resourceGraph;
            while ( previousStartNode.nextStartNode() !=
                startNode )
            {
                previousStartNode = previousStartNode.nextStartNode();
            }

            previousStartNode.setNextStartNode( startNode.
                nextStartNode() );
        }
    }


    // adds the subgraph after the second node
    // to the subgraph containing the first node
    // the only thing done is to make the graph numbers
    // of the nodes after the second node same as the
    // graph number of the first node
    public synchronized void attachSubgraph( Node firstNode,
        Node secondNode )
    {
        Node currentNode = secondNode;
        int newGraphNumber = firstNode.graphNumber(); // will be
        // assigned to the graph numbers of the nodes after the
        // second node (till the end of the subgraph)

        // look whether the end of the subgraph is reached
        boolean endOfSubgraph = false;
        if ( currentNode.nextNode() == null )
        {
            endOfSubgraph = true;
        }
        else if ( currentNode.nextNode().graphNumber() !=
            currentNode.graphNumber() )
        {
            endOfSubgraph = true;
        }
```

```
        currentNode.setGraphNumber( newGraphNumber );

        while ( !endOfSubgraph )
        {
            currentNode = currentNode.nextNode();

            // look whether the end of the subgraph is reached
            if ( currentNode.nextNode() == null )
            {
                endOfSubgraph = true;
            }
            else if ( currentNode.nextNode().graphNumber() !=
                currentNode.graphNumber() )
            {
                endOfSubgraph = true;
            }

            currentNode.setGraphNumber( newGraphNumber );
        }

    }


    // cleares the reducable start node and adds its subgraph
    // to the subgraph containing the node graphNumberNode
    public synchronized void reduceGraph( Node graphNumberNode,
        StartNode reducableStartNode )
    {
        // If deadlock, do nothing
        if ( deadlocked )
        {
            return;
        }
        deleteStartNode( reducableStartNode );

        // since the graph number node might be a member of the
        // subgraph of the reducable start node,
        // new graph number is the graph number of the node
        // prior to graph number node, not of itself
        graphNumberNode.setGraphNumber( previousNodeOf(
            graphNumberNode ) instanceof StartNode ? previousNodeOf(
            graphNumberNode ).hashCode() : previousNodeOf(
            graphNumberNode ).graphNumber() );

        if ( graphNumberNode.nextNode() != null )
        // nothing necessary to do if null
        {
            attachSubgraph( graphNumberNode, graphNumberNode.
                nextNode() );
        }
    }


    // searches for a deadlock
    public synchronized Node nodeInDeadlock()
    {
        // nothing to do if no nodes
        if ( resourceGraph == null )
```

```
        {
            return( null );
        }
        StartNode currentStartNode = resourceGraph;
        Node deadlockList = null;
        int numberOfOccurance = 0;

        do
        {
            Node currentNode = currentStartNode.nextNode();
            deadlockList = ( Node ) currentNode.clone(); // the
            // remaining members of the list will be garbage
            // collected
            deadlockList.setNextNode( null );
            Node currentDeadlockListNode = deadlockList;

            do
            {
                currentNode = currentNode.nextNode(); // there are at
                // least two nodes following a start node
                currentDeadlockListNode.setNextNode( ( Node )
                    currentNode.clone() );
                currentDeadlockListNode.nextNode().setNextNode( null );
                currentDeadlockListNode = currentDeadlockListNode.
                    nextNode();

                String currentId = currentDeadlockListNode.id();
                Node currentCheckDeadlockListNode = deadlockList;
                numberOfOccurance = 0;
                do
                {
                    if ( currentCheckDeadlockListNode.id().equals(
                        currentId ) )
                    {
                        ++numberOfOccurance;
                    }
                    if ( numberOfOccurance == 2 )
                    {
                        deadlocked = true;
                        return( currentNode );
                    }
                    currentCheckDeadlockListNode =
                        currentCheckDeadlockListNode.nextNode();
                } while ( currentCheckDeadlockListNode != null );

            } while ( currentNode.nextNode() != null );

            currentStartNode = currentStartNode.nextStartNode();
        } while ( currentStartNode != null );

        return( null );

    }


    // Creates the resource graph to be drawn.
    // The resulting graph is the exact copy of the actual
    // resource graph with the draw coordinates of the
```

```
// nodes added
public synchronized void graphToDraw()
{
    if ( resourceGraph == null )
    {
        graphToDraw = null;
        return;
    }

    // if no nodes, no draw graph to be created
    if ( resourceGraph.nextNode() == null )
    {
        graphToDraw = null; // the draw graph
        return;
    }

    StartNode currentStartNode = resourceGraph;
    Node currentNode = currentStartNode;

    graphToDraw = new DrawStartNode( "S", currentStartNode.
        hashCode(), currentStartNode ); // the draw graph

    DrawStartNode currentDrawStartNode = graphToDraw;
    DrawNode currentDrawNode = currentDrawStartNode;

    int drawX = 0;
    int drawY = 0;

    while ( currentStartNode != null )
    {
        drawX = 0;
        drawY += DRAW_DISTANCE;

        boolean endOfSubgraph = false; // the subgraph ends
        // either if the next node is null or the id of the
        // next node is not equal to the current id

        do
        {
            drawX += DRAW_DISTANCE;
            currentNode = currentNode.nextNode();

            currentDrawNode.setNextDrawNode( new DrawNode(
                currentNode.id(), currentNode.graphNumber(),
                drawX, drawY, currentNode ) );
            currentDrawNode = currentDrawNode.nextDrawNode();

            // look if the end of the subgraph is reached
            if ( currentNode.nextNode() == null )
            {
                endOfSubgraph = true;
            }
            else if ( currentNode.nextNode().graphNumber()!=
                currentNode.graphNumber() )
            {
                endOfSubgraph = true;
            }
```

```
            } while ( !endOfSubgraph );

            currentStartNode = currentStartNode.nextStartNode();
            currentNode = currentStartNode;

            if ( currentStartNode != null )
            {
                currentDrawStartNode.setNextDrawStartNode( new
                    DrawStartNode( "S", currentStartNode.
                    hashCode(), currentStartNode ) );
                currentDrawStartNode = currentDrawStartNode.
                    nextDrawStartNode();
                currentDrawNode = currentDrawStartNode;
            }
        }

        // Now we have got the exact copy of the resource graph,
        // except the edges originating from the end of the
        // subgraphs. Those are to be copied now.
        currentDrawStartNode = graphToDraw;

        do
        {
            // navigate till the end of the subgraph
            currentDrawNode = currentDrawStartNode;
            while ( currentDrawNode.nextDrawNode() != null )
            {
                currentDrawNode = currentDrawNode.nextDrawNode();
            }

            // look if there is an edge at the end of the actual
            // subgraph
            if ( currentDrawNode.actualNode().nextNode() != null )
            {
                currentDrawNode.setNextDrawNode( findDrawNode(
                    currentDrawNode.actualNode().nextNode() ) );
            }

            // go to the next subgraph
            currentDrawStartNode = currentDrawStartNode.
                nextDrawStartNode();

        } while ( currentDrawStartNode != null );

}


// Creates the resource graph to be drawn during a deadlock.
// The resulting graph is the exact copy of the actual
// resource graph with the draw coordinates of the
// nodes added
public synchronized void graphToDrawDuringDeadlock()
{
    StartNode currentStartNode = resourceGraph;
    Node currentNode = currentStartNode;

    graphToDraw = new DrawStartNode( "S", currentStartNode.
        hashCode(), currentStartNode ); // the draw graph
```

```
DrawStartNode currentDrawStartNode = graphToDraw;
DrawNode currentDrawNode = currentDrawStartNode;

int drawX = 0;
int drawY = 0;

while ( currentStartNode != null )
{
    drawX = 0;
    drawY += DRAW_DISTANCE;

    boolean endOfSubgraph = false; // the subgraph ends
    // either if the next node is null or the id of the
    // next node is not equal to the current id

    do
    {
        drawX += DRAW_DISTANCE;

        currentNode = currentNode.nextNode();

        DrawNode drawNodeInDeadlock = findDrawNode(
            currentNode );
        if ( drawNodeInDeadlock != null )
        {
            currentDrawNode.setNextDrawNode(
                drawNodeInDeadlock );
            break;
        }

        currentDrawNode.setNextDrawNode( new DrawNode(
            currentNode.id(), currentNode.graphNumber(),
            drawX, drawY, currentNode ) );
        currentDrawNode = currentDrawNode.nextDrawNode();

        // look if the end of the subgraph is reached
        if ( currentNode.nextNode() == null )
        {
            endOfSubgraph = true;
        }
        else if ( currentNode.nextNode().graphNumber()!=
            currentNode.graphNumber() )
        {
            endOfSubgraph = true;
        }

    } while ( !endOfSubgraph );

    currentStartNode = currentStartNode.nextStartNode();
    currentNode = currentStartNode;

    if ( currentStartNode != null )
    {
        currentDrawStartNode.setNextDrawStartNode( new
            DrawStartNode( "S", currentStartNode.
            hashCode(), currentStartNode ) );
        currentDrawStartNode = currentDrawStartNode.
```

```
                    nextDrawStartNode();
                currentDrawNode = currentDrawStartNode;
            }
        }

        // Now we have got the exact copy of the resource graph,
        // except the edges originating from the end of the
        // subgraphs. Those are to be copied now.
        currentDrawStartNode = graphToDraw;

        DrawNode drawNodeInDeadlock = findDrawNode(
            nodeInDeadlock() );

        do
        {
            // navigate till the end of the subgraph
            currentDrawNode = currentDrawStartNode;
            while ( currentDrawNode.nextDrawNode() != null )
            {
                currentDrawNode = currentDrawNode.nextDrawNode();

                if ( currentDrawNode == drawNodeInDeadlock )
                {
                    break;
                }
            }

            // look if there is an edge at the end of the actual
            // subgraph
            if ( ( currentDrawNode != drawNodeInDeadlock ) &&
                ( currentDrawNode.actualNode().nextNode() != null ) )
            {
                currentDrawNode.setNextDrawNode( findDrawNode(
                    currentDrawNode.actualNode().nextNode() ) );
            }

            // go to the next subgraph
            currentDrawStartNode = currentDrawStartNode.
                nextDrawStartNode();

        } while ( currentDrawStartNode != null );

    }


    // Finds the draw node in a given graph with its actual
    // node equal to a given node
    public synchronized DrawNode findDrawNode( Node nodeToFind )
    {
        if ( graphToDraw == null )
        {
            return( null );
        }
        if ( graphToDraw.nextDrawNode() == null )
        {
            return( null );
        }
```

```
Node nodeInDeadlock = null;
int nodeInDeadlockPassed = 0;
if ( deadlocked )
{
    nodeInDeadlock = nodeInDeadlock();
}

DrawStartNode currentDrawStartNode = graphToDraw;
DrawNode currentDrawNode = currentDrawStartNode;
boolean found = false;

while ( ( !found ) && ( currentDrawStartNode != null ) )
{
    if ( currentDrawStartNode.nextDrawNode() == null )
    {
        return( null );
    }

    boolean endOfSubgraph = false; // the subgraph ends
    // either if the next node is null or the id of the
    // next node is not equal to the current id

    do
    {
        currentDrawNode = currentDrawNode.nextDrawNode();

        if ( deadlocked )
        {
            if ( currentDrawNode.actualNode() == nodeInDeadlock )
            {
                nodeInDeadlockPassed += 1;
                if ( nodeInDeadlockPassed == 2 )
                {
                    break;
                }
            }
        }

        // look if the end of the subgraph is reached
        if ( currentDrawNode.nextDrawNode() == null )
        {
            endOfSubgraph = true;
        }
        else if ( currentDrawNode.nextDrawNode().
            graphNumber()!= currentDrawNode.graphNumber() )
        {
            endOfSubgraph = true;
        }

        if ( currentDrawNode.actualNode() == nodeToFind )
        {
            found = true;
        }
    } while ( ( !found ) && ( !endOfSubgraph ) );

    if ( !found )
    {
        currentDrawStartNode = currentDrawStartNode.
```

```
                    nextDrawStartNode();
                currentDrawNode = currentDrawStartNode;
            }
        }

        if ( found )
        {
            return( currentDrawNode );
        }
        else
        {
            return( null );
        }
    }


    public void killThread( String nameOfThreadToKill )
    {
        int threadIndex = Integer.parseInt( nameOfThreadToKill.
            substring( 1 ) ) - 1;
        deadlock.thread[ threadIndex ].stop();
        deadlock.thread[ threadIndex ] = new Process(
            deadlock.resource, "Thread" + ( threadIndex + 1 ),
            deadlock.threads );
        deadlock.thread[ threadIndex ].setPriority( Thread.
            MIN_PRIORITY );
        deadlock.thread[ threadIndex ].start();

        deadlocked = false;
        deadlock.deadlockArea.append( "Thread" + ( threadIndex +
            1 ) + " destroyed.\n" );

        for ( int i = 0; i < deadlock.resource.length; i++ )
        {
            Node resourceNode = findNode( deadlock.resource[ i ].
                name() );
            if ( resourceNode == null )
            {
                continue;
            }
            if ( resourceNode.nextNode() == null )
            {
                continue;
            }

            if ( resourceNode.nextNode().id().equals( "Thread" +
                nameOfThreadToKill.substring( 1 ) ) )
            {
                deadlock.resource[ i ].free();
                deadlock.deadlockArea.append( "Resource" +
                    deadlock.resource[ i ].name().substring( 1 ) +
                    " freed.\n");
                modifyReleaseNodes( ( "Thread" + nameOfThreadToKill.
                    substring( 1 ) ), resourceNode.id() );
                graphToDraw();
                deadlock.drawCanvas.draw( graphToDraw );
            }
        }
```

```
    }

    public void run()
    {
        while ( true )
        {
            modifyGraph();
            Node nodeInDeadlock = nodeInDeadlock();
            if ( nodeInDeadlock != null )
            {
                ++totalDeadlockNumber;
                Node modifiedNode = findNode( "Thread" +
                    modifiedNodeName.substring( 1 ) );
                graphToDrawDuringDeadlock();
                deadlock.drawCanvas.draw( graphToDraw );
                deadlock.deadlockArea.append( "!!!DEADLOCK!!!\n" );
                deadlock.deadlockArea.append( "total deadlocks:\n" );
                deadlock.deadlockArea.append( Integer.toString(
                    totalDeadlockNumber ) + "\n" );
                killThread( modifiedNodeName );
            }
            else
            {
                graphToDraw();
            }

            deadlock.drawCanvas.draw( graphToDraw );

        }
    }
}
```

```
// ListNode.java
// Nodes of the lists of the Detector

public class ListNode
{
    private String threadName; // name of the thread
    private String resourceName; // name of the resource
    private ListNode nextListNode; // next list node

    // constructor
    public ListNode( String threadName, String resourceName )
    {
        this.threadName = threadName;
        this.resourceName = resourceName;
        nextListNode = null;
    }

    // get the Process name of the ListNode
    public String threadName()
    {
        return( threadName );
    }

    // get the Resource name of the ListNode
    public String resourceName()
    {
        return( resourceName );
    }

    // set the next ListNode
    public void setNextListNode( ListNode listNode )
    {
        nextListNode = listNode;
    }

    // get the next ListNode
    public ListNode nextListNode()
    {
        return( nextListNode );
    }

}
```

```
// Node.java
// Nodes of the directed graph

public class Node implements Cloneable
{
    private String name; // name of the Node
    private Node nextNode; // next Node in the Graph

    // Because we may have more than one subgraphs, every node
    // should know to which subgraph it belongs
    private int graphNumber;

    // no-argument constructor
    public Node()
    {
        nextNode = null;
    }

    // constructor
    public Node( String name )
    {
        this.name = name;
        nextNode = null;
    }

    // constructor
    public Node( String name, int graphNumber )
    {
        this.name = name;
        this.graphNumber = graphNumber;
        nextNode = null;
    }

    // get the name of the Node
    public String id()
    {
        return( name );
    }

    // set the next Node
    public void setNextNode( Node node )
    {
        nextNode = node;
    }

    // get the next Node
    public Node nextNode()
    {
        return( nextNode );
    }

    // set the graph number
    public void setGraphNumber( int graphNumber )
    {
        this.graphNumber = graphNumber;
    }

    // get the graph number
```

```
    public int graphNumber()
    {
        return( graphNumber );
    }

    // clone
    protected Object clone()
    {
        Object clonedNode = null;
        try
        {
            clonedNode = super.clone();
        }
        catch ( CloneNotSupportedException cnse ) {}

        return( clonedNode );
    }

}
```

```
// StartNode.java
// Starting Nodes of the parts of the directed Graph

public class StartNode extends Node
{
    private StartNode nextStartNode; // next starting node

    // no-argument constructor
    public StartNode()
    {
        // implicit call to superclass constructor
        nextStartNode = null;
    }

    // constructor
    public StartNode( String name )
    {
        super( name );
        nextStartNode = null;
    }

    // set the next starting Node
    public void setNextStartNode( StartNode startNode )
    {
        nextStartNode = startNode;
    }

    // get the next starting Node
    public StartNode nextStartNode()
    {
        return( nextStartNode );
    }

}
```

```
// DrawNode.java
// Nodes of the graph to be drawn

public class DrawNode
{
    private String name; // name of the node

    private int graphNumber; // Because we may have more than one subgraph,
    // every node should know to which subgraph it belongs

    private int drawX; // where to draw
    private int drawY; // the node
    private DrawNode nextDrawNode; // next node in the graph

    private Node actualNode; // corresponding node in the
    // actual graph

    // no-argument constructor
    public DrawNode()
    {
        nextDrawNode = null;
        actualNode = null;
    }

    // constructor
    public DrawNode( String name, int graphNumber, Node actualNode )
    {

        System.out.println( "Draw node created: " + name );

        this.name = formName( name );
        this.graphNumber = graphNumber;
        nextDrawNode = null;
        setActualNode( actualNode );
    }

    // constructor
    public DrawNode( String name, int graphNumber, int drawX,
        int drawY, Node actualNode )
    {
        this.name = formName( name );
        this.graphNumber = graphNumber;
        this.drawX = drawX;
        this.drawY = drawY;
        nextDrawNode = null;
        setActualNode( actualNode );
    }

    // Creates a name for the draw node.
    // If it is a thread node returns "T" followed by the
    // thread number, else it returns "R" followed by the
    // hash code of the resource.
    // For a start node in the graph, the return string
    // of this function is not important because, the
    // start nodes will not be drawn
    public String formName( String name )
    {
        if ( name.startsWith( "T" ) )
```

```
        {
            return( "T" + name.substring( 6 ) );
        }
        else
        {
            return( name );
        }
    }

    // get the name of the draw node
    public String id()
    {
        return( name );
    }

    // set the graph number
    public void setGraphNumber( int graphNumber )
    {
        this.graphNumber = graphNumber;
    }

    // get the graph number
    public int graphNumber()
    {
        return( graphNumber );
    }

    // get the x coordinate
    public int drawX()
    {
        return( drawX );
    }

    // set the x coordinate
    public void setDrawX( int drawX )
    {
        this.drawX = drawX;
    }

    // get the y coordinate
    public int drawY()
    {
        return( drawY );
    }

    // set the y coordinate
    public void setDrawY( int drawY )
    {
        this.drawY = drawY;
    }

    // get the next draw node
    public DrawNode nextDrawNode()
    {
        return( nextDrawNode );
    }

    // set the next draw node
```

```
    public void setNextDrawNode( DrawNode drawNode )
    {
        nextDrawNode = drawNode;
    }

    // get the actual node
    public Node actualNode()
    {
        return( actualNode );
    }

    // set the actual node
    public void setActualNode( Node actualNode )
    {
        this.actualNode = actualNode;
    }

}
```

```
// DrawStartNode.java
// Nodes of the graph to be drawn

public class DrawStartNode extends DrawNode
{

    private DrawStartNode nextDrawStartNode; // next start node
    // in the graph

    // no-argument constructor
    public DrawStartNode()
    {
        super();
        nextDrawStartNode = null;
    }

    // constructor
    public DrawStartNode( String name, int graphNumber,
        Node actualNode )
    {
        super( name, graphNumber, actualNode );
        nextDrawStartNode = null;
    }


    // constructor
    public DrawStartNode( String name, int graphNumber,
        int drawX, int drawY, Node actualNode )
    {
        super( name, graphNumber, drawX, drawY, actualNode );
        nextDrawStartNode = null;
    }

    // get the next draw start node
    public DrawStartNode nextDrawStartNode()
    {
        return( nextDrawStartNode );
    }

    // set the next draw start node
    public void setNextDrawStartNode( DrawStartNode
        drawStartNode )
    {
        nextDrawStartNode = drawStartNode;
    }

}
```

EK 2 SÖZLÜK

Allocation	Tahsis etme
Assumption	Kabullenme
Avoidance	Sakınma
Critical Section	Kritik Bölge
Concurrent	Eşzamanlı
Distributed	Dağıtılmış
Instance	Örnek
Multiprocess	Çok İşlemli
Multiprogramming	Çok Programlı
Mutual Exclusion	Karşılıklı Dışlama
Prevention	Önleme
Procedure	Yordam
Process	İşlem
Register	Yazıcı
Semaphore	Semafor
Stack	Yığın
Thread	İşlem Parçacığı

ÖZGEÇMİŞ

Doğum Tarihi	17.10.1973	
Doğum Yeri	Trabzon	
Lise	1983-1990	Trabzon Anadolu Lisesi
Lisans	1990-1994	Bilkent Üniversitesi Mühendislik Fak. Elektrik Elektronik Mühendisliği Bölümü
Yüksek lisans	1994-1999	Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Bilimleri ve Müh. Bölümü
Çalıştığı kurum	1996-Devam ediyor	YTÜ Bilgisayar Bilimleri ve Müh. Böl. Araştırma Görevlisi