Piri Reis Üniversitesi, Denizcilik Meslek Yüksekokulu, Bilgisayar Programcılığı Programı

İŞLETİM SİSTEMLERİNDE KİLİTLENME (DEADLOCK) YÖNETİMİ: PYTHON TABANLI DİNAMİK SİMÜLASYON VE ALGORİTMA PERFORMANS ANALİZİ

Yazar: 20240108044 - Emir Ecrin MALKOÇ

Dersin Adı/Kodu : İŞLETİM SİSTEMLERİ / BIP 2027

Akademik Yıl / Dönem : 2025-2026 / Güz

Dersin Öğretim Görevlisi: Refik Tanju SİRMEN

Tarih: 07.01.2026

İÇİNDEKİLER

1. ÖZET

Modern işletim sistemlerinde, birden fazla işlemin (process) aynı anda sınırlı kaynaklara erişmeye çalışması, "Ölümcül Kilitlenme" (Deadlock) adı verilen sistemsel tıkanıklıklara yol açabilmektedir. Bu çalışmada, Deadlock problemini analiz etmek ve farklı çözüm stratejilerini karşılaştırmak amacıyla Python tabanlı dinamik bir simülasyon ortamı geliştirilmiştir. Geliştirilen simülatör üzerinde; Önleyici (Prevention), Kaçınan (Avoidance), Tespit Edip Sonlandıran (Detection-Termination) ve Alternatif (Rollback) yaklaşımlar test edilmiştir. Elde edilen bulgular, işlemci maliyeti ile sistem kararlılığı arasındaki ters orantıyı ortaya koymakta ve senaryoya bağlı olarak optimum algoritmanın değişkenlik gösterdiğini kanıtlamaktadır.

2. GİRİŞ

Çok görevli (multitasking) işletim sistemlerinin en temel problemlerinden biri, sınırlı sayıdaki sistem kaynağının (CPU, RAM, G/Ç cihazları vb.) süreçler arasında nasıl paylaştırılacağıdır. Bir süreç, ihtiyaç duyduğu kaynağı alabilmek için başka bir sürecin o kaynağı serbest bırakmasını beklediğinde ve o süreç de ilk süreci beklediğinde, sistem "Deadlock" (Kilitlenme) durumuna girer **[1]**.

Literatürde bir Deadlock durumunun oluşması için Coffman tarafından tanımlanan dört temel şartın (Karşılıklı Dışlama, Tut ve Bekle, Bölünemezlik, Döngüsel Bekleme) aynı anda gerçekleşmesi gerekmektedir **[2]**. Bu şartlardan biri kırıldığında kilitlenme çözülür. Ancak bu çözümün sisteme maliyeti (CPU kullanımı) ve veri bütünlüğüne etkisi (işlem sonlandırma) farklılık göstermektedir.

Bu çalışmanın temel amacı; teorik olarak anlatılan Deadlock algoritmalarının, geliştirilen kapsamlı bir Python simülasyonu üzerinde koşturularak somut performans verilerinin elde edilmesidir. Çalışma kapsamında sadece kilitlenmenin tespiti değil; "En Az Maliyetli Kurban

Seçimi", "Zamanı Geri Sarma (Rollback)" ve "Kaynak Hiyerarşisi" gibi farklı stratejilerin, farklı senaryolar (Kaynak Kıtlığı, Kaotik Yük vb.) altındaki davranışları incelenmiştir.

3. MATERYAL VE YÖNTEM

3.1. Simülasyon Mimarisi

Çalışma kapsamında Python programlama dili kullanılarak modüler bir "İşletim Sistemi Çekirdeği (Kernel)" simüle edilmiştir. Sistem, gerçek bir işletim sistemi gibi döngüsel (tick-based) bir zaman dilimi üzerinde çalışmaktadır. Simülatörün temel işleyişi şu şekildedir:

1. **Kaynak Yöneticisi:** Toplam kaynak kapasitesini, o an boşta olan kaynakları (Available Resources) ve gerekli istatistiksel değerleri takip eder.
2. **Süreç Yönetimi:** Her süreç (process); "Hazır", "Çalışıyor", "Bekliyor" veya "Tamamlandı" durumlarından birinde bulunur.
3. **Talep İşleme:** Bir süreç kaynak talep ettiğinde, sistemde aktif olan algoritma modülü devreye girer ve talebin onaylanıp onaylanmayacağına karar verir.

Simülasyon, her algoritma için senaryoyu sıfırdan başlatarak, algoritmalar arasında adil bir karşılaştırma (benchmark) ortamı sağlar.

3.2. Kullanılan Algoritmalar ve Yaklaşımlar

Projede dört farklı kategoride toplam yedi algoritma/strateji uygulanmıştır:

A. Önleyici Yaklaşım (Prevention)

- **Resource Ordering (Kaynak Sıralaması):** Havender tarafından önerilen bu yöntemde, tüm kaynaklara hiyerarşik bir numara verilir **[3]**. Bir süreç, elindekinden daha düşük numaralı bir kaynak isteyemez. Eğer isterse, elindeki yüksek numaralı kaynakları zorla bırakması (preemption) sağlanır. Bu yöntem "Döngüsel Bekleme" şartını matematiksel olarak imkansız kılar.

B. Tespit Etme ve Son Verme (Detection-Termination)

Bu gruptaki algoritmalar, sistemde bir döngü (cycle) oluşup oluşmadığını DFS (Depth First Search) yöntemiyle sürekli tarar **[4]**. Döngü tespit edilirse bir "Kurban" (Victim) seçilir ve sonlandırılır.

- **Random Victim:** Döngüdeki süreçlerden rastgele biri sonlandırılır.
- **Minimum Victim:** İşleminin henüz çok başında olan süreç kurban edilir. Amaç, harcanan işlemci emeğini (CPU Time) korumaktır.
- **Maximum Victim:** Elinde en fazla kaynak tutan süreç kurban edilir. Amaç, sisteme anında büyük miktarda kaynak kazandırıp tıkanıklığı açmaktır.
- **Valued Victim:** Öncelik değeri (Priority/Value) en düşük olan süreç kurban edilir. Kritik işlemlerin korunması hedeflenir.

C. Kaçınma Yaklaşımı (Avoidance)

- **Banker's Algorithm:** Dijkstra'nın geliştirdiği bu algoritma, bir kaynak verilmeden önce "Bu kaynağı verirsem sistem gelecekte güvenli (safe) bir durumda kalır mı?" simülasyonunu yapar **[5]**. Eğer güvensiz (unsafe) bir durum oluşma ihtimali varsa, kaynak verilmez ve süreç bekletilir.

D. Alternatif Yaklaşım

- **Rollback (Geri Sarma):** Kilitlenme tespit edildiğinde süreç öldürülmez. Bunun yerine, sürecin yaptığı işlemler hafızada geri alınır, elindeki kaynaklar bıraktırılır ve süreç "reset"lenerek en başa döndürülür **[6]**. Veri kaybını önler ancak zaman kaybı yaratır.

Not: Simülasyon çıktılarında "Overhead Cost" (Maliyet Puanı) metriği, algoritmanın karar verirken harcadığı işlem gücünü temsil etmek için kullanılmıştır.

4. DENEYSEL KURGU (SENARYOLAR)

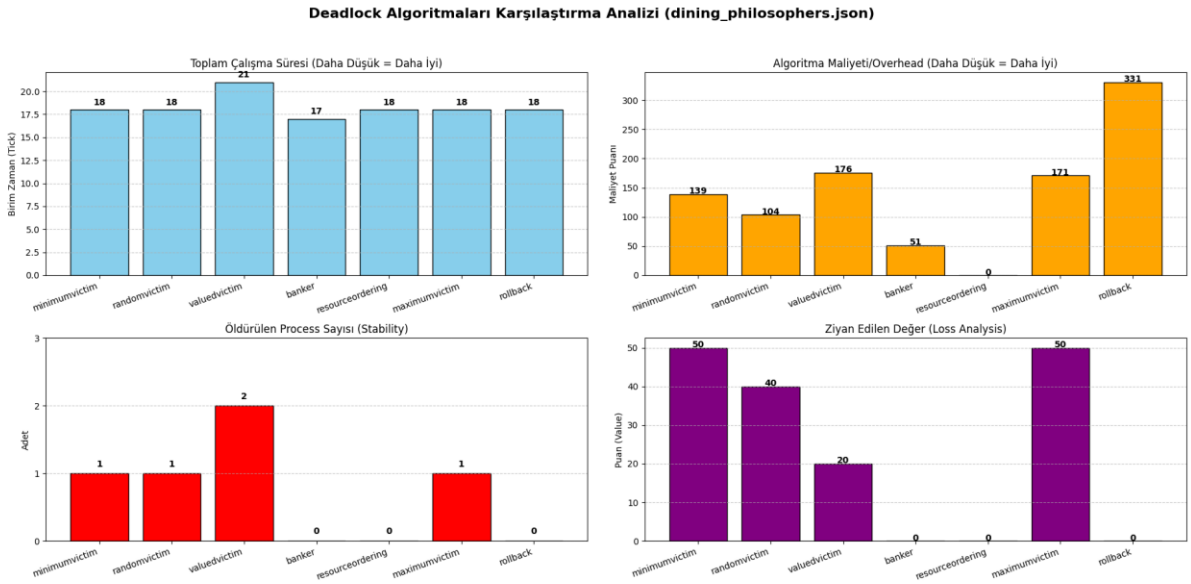
Algoritmaların teknik yeterliliklerini ve performansını daha elde tutulur veriler ile ölçebilmek amacıyla tasarlanan karakteristik senaryolar arasından aşağıdaki seçilen üç senaryo uygulanmıştır:

1. **Kaynak Kıtlığı (Starvation) Senaryosu:** Sistemin sınırlı kaynaklarla çalıştığı ve süreçlerin kaynaklar için yoğun rekabete girdiği bir ortamdır. Bu senaryonun temel amacı; algoritmaların "Kurban Seçimi" veya "Değer Koruma" başarısını değil, doğrudan **"Overhead Cost" (Maliyet Puanı)** üzerindeki etkisini ölçmektir. Algoritmaların karar verirken işlemciyi ne kadar meşgul ettiği ve hesaplama maliyetleri burada karşılaştırılır.
2. **Kaotik Yük (Chaos) Senaryosu:** Farklı nitelikteki çok sayıda sürecin sisteme rastgele zamanlarda giriş yaptığı bir **Benchmark (Kıyaslama)** testidir. Sistemin "Heavy Load" (Ağır Yük) altındaki tepkisini, toplam çalışma süresinin nasıl değiştiğini ve algoritmaların sistem kararlılığını (Stability) nasıl etkilediğini ölçmeyi hedefler.
3. **Yemek Yiyen Filozoflar (Dining Philosophers):** Klasik bir kilitlenme problemidir. Beş sürecin dairesel bir şekilde birbirini kilitlediği bu senaryoda, tüm algoritmaların bu kısır döngüye nasıl tepki verdiği, döngüyü kırmak için nasıl bir yol izlediği ve çözüm maliyetleri analiz edilir. **[7]**.

5. BULGULAR VE TARTIŞMA

Simülasyon sonucunda elde edilen veriler; **Toplam Çalışma Süresi (Total Execution Time)**, **Sistem Yükü (Overhead Cost)**, **Öldürülen Süreç Sayısı (Kill Count)** ve **Ziyan Edilen Değer (Wasted Value)** metrikleri üzerinden analiz edilmiştir.

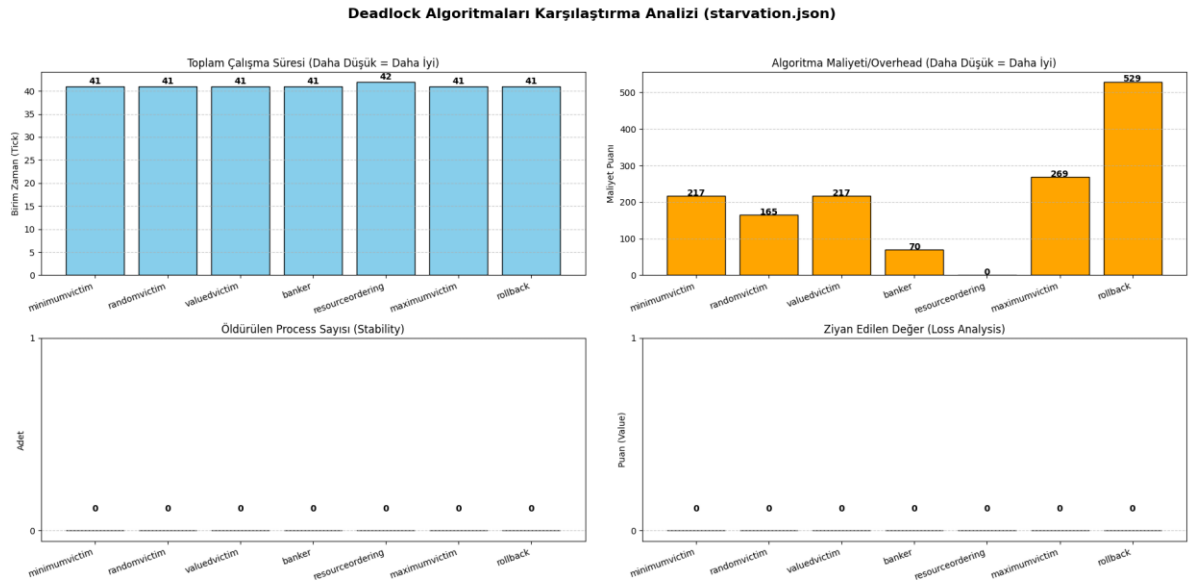
5.1. Filozoflar Problemi (Dining Philosophers) Analizi



Dairesel beklemenin (Circular Wait) zorunlu olduğu bu klasik problemde algoritmaların çözüm yetenekleri test edilmiştir.

- **Resource Ordering (Prevention):** 0 Maliyet ve 0 Kill Count ile problemi çözen tek algoritmadır. Filozoflar problemindeki dairesel kilitlenmenin tek yapısal çözümü, kaynaklara hiyerarşi getirmektir. Algoritma, son çatalı alacak filozofun kural ihlali yapacağını baştan engellediği için kilitlenme hiç oluşmamıştır **[10]**.
- **Banker Algoritması (Avoidance):** 51 Maliyet Puanı ile düşük bir yükle çalışmış ve Kill Count 0'dır. Banker algoritması, son çatalın alınması durumunda sistemin "Unsafe State"e düşeceğini hesaplayarak izni vermemiş, böylece süreçleri güvenli beklemeye almıştır **[1]**.
- **Rollback Algoritması (Alternatif):** 331 Maliyet Puanı ile yine en yüksek maliyeti oluşturmuştur. Kilitlenme oluştukça zamanı geri sarmış, bu da problemi çözse de çalışma süresini ve işlemci maliyetini gereksiz yere artırmıştır.
- **Victim (Detection) Algoritmaları:** Her üç victim algoritması da kilitlenmeyi çözmek için en az 1 veya 2 filozofu öldürmek zorunda kalmıştır. Deadlock oluştuktan sonra müdahale eden bu algoritmalar, bu tip dairesel bağımlılıklarda kurban vermeden çözüm üretememektedir. Detection yaklaşımları bu tür sıkı döngülerde sürekli bir "Abort-Restart" döngüsüne girme riski taşır **[7]**. Özellikle Valued Victim, 2 süreç öldürerek en başarısız sonucu vermiştir; çünkü tüm filozofların değeri birbirine yakın olduğunda seçim yapmakta zorlanmıştır.

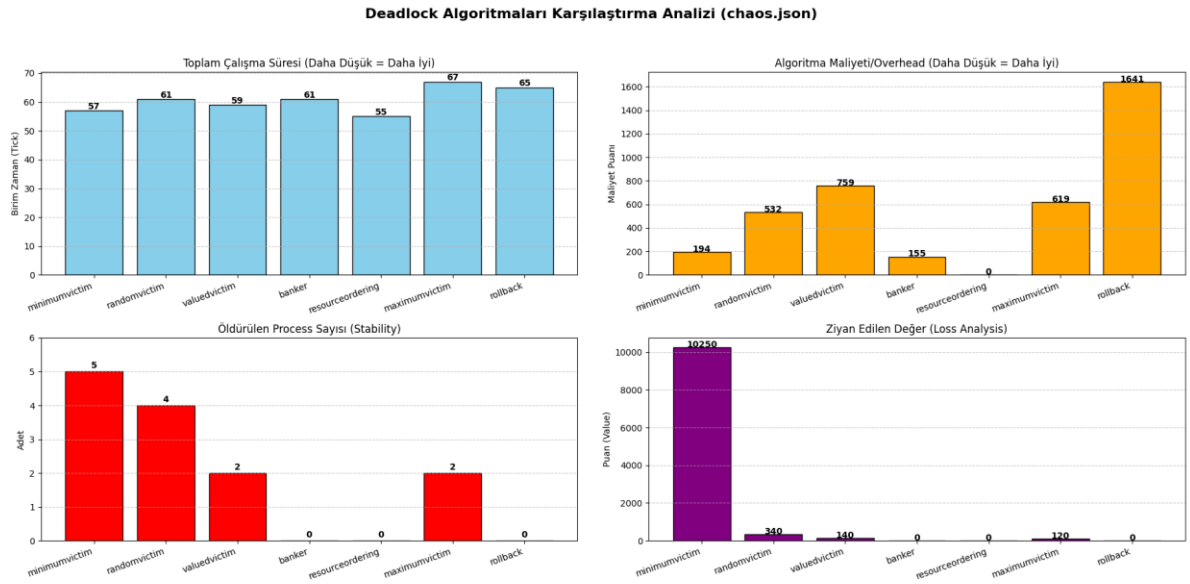
5.2. Kaynak Kıtlığı (Starvation) Analizi



Bu senaryoda sistem kilitlenmeye (deadlock) düşmemiş, ancak algoritmaların "kontrol mekanizmalarının" işlemciye getirdiği yük (Overhead) test edilmiştir.

- **Resource Ordering (Prevention):** Grafikte görüldüğü üzere 0 maliyet puanı ile çalışmıştır. Havender prensibine dayanan bu yaklaşım, kaynaklara statik numaralar atadığı ve çalışma zamanında (runtime) herhangi bir karmaşık hesaplama yapmadığı için sisteme ek yük bindirmemiştir **[3]**. Kaynak sıralaması kuralı, algoritma maliyetini minimize etmiştir.
- **Banker Algoritması (Avoidance):** 70 Maliyet Puanı ile Rollback ve Victim algoritmalarından daha düşük, ancak Resource Ordering'den daha yüksek bir maliyet sergilemiştir. Sistemde deadlock riski oluşmasa bile, Banker algoritması her kaynak talebinde $O(m \times n^2)$ karmaşıklığında olan güvenlilik (safety) matrisini hesaplamak zorundadır **[1]**. Bu durum, sistem güvenli olsa dahi kaçınılmaz bir baz maliyet oluşturmuştur.
- **Rollback Algoritması (Alternatif):** 529 Maliyet Puanı ile bu senaryonun en maliyetli algoritması olmuştur. Algoritma herhangi bir işlem sonlandırmamış olsa da, olası bir deadlock durumuna karşı sürekli olarak sistemin durumunu izlemesi (monitoring), işlemci üzerinde "Detect-and-Recover" stratejilerinin en büyük dezavantajı olan yüksek overhead'e neden olmuştur **[10]**.
- **Detection (Victim) Algoritmaları:** Minimum, Maximum, Random ve Valued algoritmaları 165-269 bandında orta seviye bir maliyet üretmiştir. Bu algoritmalar sürekli olarak *Wait-For* grafiği üzerinde döngü taraması (Cycle Detection) yaptıkları için Banker'den daha fazla, ancak Rollback'in ağır işlem yükünden daha az kaynak tüketmiştir **[9]**.

5.3. Kaotik Yük (Chaos) Analizi



Sistemin ağır yük altında olduğu ve çok sayıda deadlock riskinin oluştuğu bu senaryoda, algoritmaların kriz anındaki kararları ve kayıpları (Wasted Value) incelenmiştir.

- **Resource Ordering (Prevention):** 0 Öldürülen Süreç ve 0 Ziyan Edilen Değer ile tamamlanmıştır. Yapısal kısıtlama sayesinde deadlock oluşumu matematiksel olarak engellendiği için, sistem hiç duraksamadan ve kurban vermeden en yüksek verimlilikle çalışmıştır **[10]**.
- **Banker Algoritması (Avoidance):** 155 Maliyet Puanı ile çalışmış, hiçbir süreci öldürmemiştir. Kaotik ortamda dahi güvenli durumlara izin vererek deadlock'u engellemiştir. Şaşırtıcı bir şekilde, yoğun yük altında dahi maliyeti (155), Rollback (1641) ve Victim algoritmalarına (Ort. 500+) göre çok daha düşük kalmıştır. Dijkstra'nın belirttiği gibi, doğru bir kaçınma stratejisi, kriz anında kurtarma maliyetinden daha verimli olabilir **[5]**.
- **Rollback Algoritması (Alternatif):** 1641 Maliyet Puanı ile grafikteki en yüksek işlem yüküne ulaşmıştır. Deadlock oluştuğunda süreçleri öldürmek yerine başa sarması (restart), işlemciye binen yükü katlamıştır. Singhal ve Shivaratri'nin belirttiği gibi, checkpoint/rollback mekanizmaları veri bütünlüğünü korusa da sistem performansını (throughput) ciddi oranda düşürmektedir **[6]**.
- **Victim (Detection) Algoritmaları:** Grafikte dikkat çeken en önemli veri, Minimum Victim algoritmasının 10.250 Puanlık devasa bir değer kaybına (Wasted Value) yol açmasıdır. Minimum Victim algoritması, kurban seçerken sadece "işlemin ne kadar ilerlediğine" bakar. Bu senaryoda, çok yüksek değere sahip olan bir işlem, henüz işin başındayken deadlock döngüsüne girdiği için algoritma tarafından defalarca "ucuz kurban" sanılarak öldürülmüştür. Bu durum, Stallings tarafından belirtilen "yanlış kurban seçiminin maliyet artışı" teorisini doğrulamaktadır **[7]**.

6. SONUÇ

Bu çalışmada geliştirilen simülasyon ortamı, işletim sistemlerindeki Deadlock algoritmalarının "en iyisi yoktur, duruma göre en uygunu vardır" prensibini doğrulamıştır.

Elde edilen sonuçlara göre:

1. Banka ve finans sistemleri gibi **veri kaybının tolere edilemediği** sistemlerde, yüksek işlem maliyetine rağmen **Banker Algoritması** veya **Rollback** tercih edilmelidir.
2. Oyun sunucuları veya video işleme gibi **hızın önemli olduğu** sistemlerde, **Minimum Victim** veya **Random Victim** algoritmaları daha performanslı sonuç vermektedir.
3. Sistemin belirli bir hiyerarşiye oturtulabildiği gömülü sistemlerde ise **Resource Ordering** en verimli çözümdür.

7. YAZILIM VE KOD ERİŞİLEBİLİRLİĞİ

Bu çalışmada kullanılan simülasyon algoritmaları, senaryo dosyaları ve performans analizi araçları şeffaflık ilkesi gereği açık kaynak olarak aşağıdaki bağlantıdan ya da alternatif olarak bu makalenin ekinde dijital formatta sunulmuştur.

GitHub Deposu: https://github.com/odidexwastaken/Deadlock-Simulation-and-Performance-Analysis

8. KAYNAKÇA

[1] A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts*, 10. baskı. Hoboken, NJ: Wiley, 2018.

[2] E. G. Coffman, M. Elphick ve A. Shoshani, "System deadlocks," *Computing Surveys*, c. 3, no. 2, ss. 67-78, 1971.

[3] J. W. Havender, "Avoiding deadlock in multitasking systems," *IBM Systems Journal*, c. 7, no. 2, ss. 74-84, 1968.

[4] A. S. Tanenbaum ve H. Bos, *Modern Operating Systems*, 4. baskı. Boston, MA: Pearson, 2014.

[5] E. W. Dijkstra, "Cooperating sequential processes," Technological University, Eindhoven, Hollanda, 1965.

[6] M. Singhal ve N. G. Shivaratri, *Advanced Concepts in Operating Systems*. New York, NY: McGraw-Hill, 1994.

[7] W. Stallings, *Operating Systems: Internals and Design Principles*, 9. baskı. Boston, MA: Pearson, 2017.

[8] R. C. Holt, "Some deadlocks properties of computer systems," *ACM Computing Surveys (CSUR)*, c. 4, no. 3, ss. 179-196, 1972.

[9] E. Knapp, "Deadlock detection in distributed databases," *ACM Computing Surveys*, c. 19, no. 4, ss. 303-328, 1987.

[10] S. S. Isloor ve T. A. Marsland, "The deadlock problem: An overview," *Computer*, c. 13, no. 9, ss. 58-78, 1980.