8

Planlama: Çok Düzeyli Geri Bildirim Kuyruğu

Bu bölümde, en iyi bilinen zamanlama yaklaşımlarından biri olan Çok Düzeyli Geri Bildirim Kuyruğu (MLFQ)'yu geliştirme problemine göz atacağız. Çok Düzeyli Geri Bildirim Kuyruğu (MLFQ) zamanlayıcısı ilk olarak 1962 yılında Corbato vb. tarafından [C+62] bir sistem olarak tanımlanmış olan Uyumlu Zaman Bölüştürme Sistemi (CTSS) içinde tarif edilmiştir ve bu çalışma, Multics'te yapılan sonraki çalışmalarla birlikte, ACM tarafından Corbato'ya **Turing Ödülü** verilmesine yol açmıştır. Zamanlayıcı, yıllar boyunca modern sistemlerde karşılaşacağınız uygulamalara göre geliştirilmiştir.

MLFQ'nun çözmeye çalıştığı temel problem iki katmandan oluşmaktadır. Öncelikle, döngü süresini optimize etmeyi ister; ancak, işletim sistemi genellikle bir işin ne kadar süre çalışacağını tam olarak bilmez, bu da SJF (veya STCF) gibi algoritmaların gerektiği bilgilerdir. İkincisi, MLFQ etkileşimli kullanıcıların (yani, bir sürecin bitmesini bekleyen ve ekrana bakan kullanıcıların) sistemi duyarlı kılmayı ve böylece yanıt süresini en aza indirmeyi ister; ancak, Round Robin gibi algoritmalar yanıt süresini azaltır ancak döngü süresi için kötüdür. Böylece, bizim problemimiz: genellikle bir sürecin hiçbir şeyini bilmediğimiz halde, nasıl bu hedefleri gerçekleştirebilecek bir zamanlayıcı geliştirebiliriz? Zamanlayıcı, sistem çalışırken, çalıştırdığı işlerin özelliklerini nasıl öğrenebilir ve böylece daha iyi zamanlama kararları alabilir?

ÇÖZÜM NOKTASI:
MÜKEMMEL BİLGİ OLMADAN NASIL PLANLAMAK GEREKİR?
Bir işin süresini önceden bilmeden, hem etkileşimli işler için yanıt süresini en aza indirmeyi hem de döngü süresini en aza indirmeyi nasıl sağlayacak bir zamanlayıcı tasarımı yapabiliriz?

İpucu: Geçmişten Öğrenin
Çoklu geribildirim kuyruğu, geçmişten geleceği tahmin etmek için kullanılan iyi bir sistem örneğidir. Böylesine yaklaşımlar işletim sistemlerinde (ve bilgisayar bilimlerinde birçok diğer yerde, donanım yönlendirme tahmincileri ve önbellekleme algoritmaları dahil) yaygındır. Bu tür yaklaşımlar, işlerin davranış fazlarına sahip olduğu ve bu nedenle tahmin edilebilir olduğunda işe yarar; tabii ki, böylesine tekniklerle dikkatli olunması gerekir, çünkü yanılabilirler ve sistemi hiçbir bilgiye sahip olmaksızın yapabileceğinden daha kötü kararlar almaya yönlendirebilirler..

8.1 MLFQ: Temel Kurallar

"Bir çok kuyruklu geri bildirim sıralayıcısının temel algoritmalarını anlatarak, bu bölümde bir çok kuyruklu geri bildirim sıralayıcısının nasıl inşa edileceğini açıklayacağız. Özellikleri birçok yönde farklı olsa da, birçok MLFQ yaklaşımı benzerdir. Bizim anlatımımızda, MLFQ'nın farklı öncelik seviyelerine sahip birkaç ayrı kuyruğu vardır. Herhangi bir anda, çalışmaya hazır bir iş bir **kuyrukta(queues)** bulunur. MLFQ, hangi işin hangi zamanda çalışacağına karar vermek için **öncelikleri(priority level)** kullanır: daha yüksek bir önceliğe (yani daha yüksek bir kuyruğa) sahip bir iş seçilir ve çalıştırılır.".

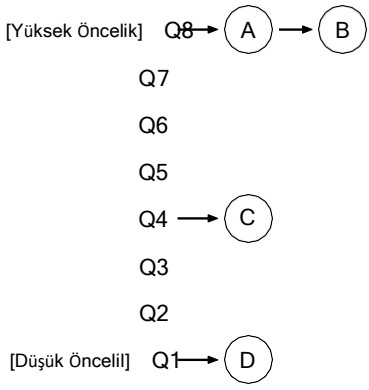
Tabii ki, Bir düğüm birden fazla işleme özelliğine sahip olabilir ve bu nedenle aynı önceliğe sahip olabilir. Bu durumda, bu işler arasında round-robin planlama kullanacağız.

Birinci ve ikinci temel kuralımız MLFQ için şu şekildedir:

- **Kural 1:** Eğer (A) önceliği > (B) önceliği ise A çalışır (B çalışmaz).
- **Kural 2:** Eğer (A) önceliği = (B) önceliği ise A ve B round – robin'de çalışır.

Bu, MLFQ planlayıcısı için anahtar yerlerin nasıl ayarlandığına ilişkin bir açıklamadır. İşlerin sabit bir önceliği vermek yerine, MLFQ, gözlenen davranışlarına göre işin önceliğini değiştirir. Örneğin, bir iş klavyeden girdi beklerken sürekli CPU'yu bırakırsa, MLFQ etkileşimli bir işlem nasıl davranacağını tahmin edecektir ve bu nedenle işin önceliğini yüksek tutar. Bunun yerine, bir iş CPU'yu uzun süre yoğun bir şekilde kullanırsa, MLFQ önceliğini düşürür. Bu şekilde, MLFQ işler hakkında çalışırken öğrenecek ve böylece işin geçmişini gelecekteki davranışını tahmin etmek için kullanacaktır.

"MLFQ, bir işin geçmiş davranışını kullanarak gelecekteki davranışını tahmin etmeyi amaçlayan bir sistemdir. Böylece, bir iş sıklıkla klavyeden girdi beklerken CPU'yu bırakırsa, MLFQ yüksek bir önceliğe sahip olacaktır çünkü bu bir etkileşimli işlemin nasıl davranabileceğini gösterir. Eğer bir iş zamanın çoğunu yoğun bir şekilde CPU kullanırsa, MLFQ önceliğini düşürecektir. Bu şekilde, MLFQ işlerin nasıl çalıştıklarını öğrenmeye çalışacak ve bu sayede işin geçmişini kullanarak gelecekteki davranışını tahmin edecektir."-



Şekil 8.1: **MLFQ Örneği**

İş önceliğinin zaman içinde nasıl değiştiğini anlamaya çalışın. Ve bu, bu kitaptaki bir bölümü ilk kez okuyanlar için sadece bir sürpriz olan, tam olarak bir sonraki adımımız olacaktır.

8.2 Girişim #1: Öncelik Nasıl Değiştirilir?

Şimdi MLFQ'nun bir işin öncelik düzeyini (ve böylece hangi kuyrukta olduğunu) bir işin yaşam süresi boyunca nasıl değiştireceğini kararlaştırmalıyız. Bunu yapmak için, iş yükümüzü göz önünde bulundurmalıyız: Kısa süren (ve CPU'yu sıklıkla terketme olasılığı olan) etkileşimli işlerin ve yanıt süresine önem verilmeyen ancak çok CPU zamanı gerektiren daha uzun süren "CPU'ya bağlı" işlerin bir karışımı. İşte öncelik ayar algoritmasına ilk denememiz:

- **Kural 3:** Bir iş sisteme girdiğinde, en yüksek öncelikte (en üstteki kuyrukta) verlestirilir.
- **Kural 4a** Bir is bir zaman dilimini tamamen kullanırken çalışıyorsa, önceliği azaltılır (vani, bir kuvruk asağıva tasınır).
- **Kural 4b:** Bir is zaman dilimi bitmeden CPU'yu terk ederse, aynı öncelik düzeyinde kalır.

Örnek 1: Tek Uzun Süreli İş

İlk olarak, sistemde uzun süreli bir işin olup olmadığını inceleyelim. Şekil 8.2, bu işin zaman içinde üç kuyrukludan bir zamanlayıcıda nasıl değiştiğini gösterir.

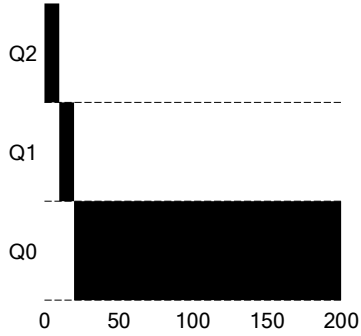


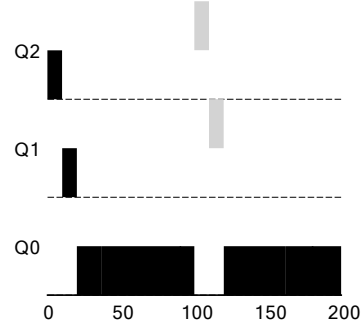
Figure 8.2: **Long-running Job Over Time**

"İş, en yüksek önceliğe (Q2) girdiğinde görülür. Bir tek 10 ms'lik zaman diliminden sonra, planlayıcı işin önceliğini bir azaltır ve böylece iş Q1'de olur. Q1'de bir zaman dilimi içinde çalıştıktan sonra, iş en düşük önceliğe (Q0) düşürülür ve orada kalır. Basit değil mi?"

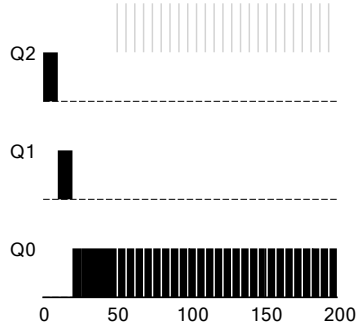
Örnek 2: Kısa Bir İş Gelişti

Bu daha karmaşık bir örnekle, MLFQ'nın SJF'ye nasıl yaklaştığını görelim. Bu örnekte, A ve B olmak üzere iki iş vardır. A, uzun süre çalışan bir CPU yoğun işidir ve B, kısa süre çalışan bir etkileşimli iştir. A'nın zamanı geçtikten sonra B'nin geldiğini varsayalım. Ne olacak? MLFQ B için SJF'yi taklit eder mi?

Şekil 8.3, bu senaryonun sonuçlarını çizmektedir. A (siyah olarak gösterilmiştir), en düşük öncelikli kuyrukta (herhangi bir uzun süreli CPU yoğun işleri gibi) çalışmaktadır; B (gri olarak gösterilmiştir), T = 100 zamanında gelir ve böylece en yüksek kuyruğa eklenir; çalışma süresi kısa olduğundan (yalnızca 20 ms), B, alt kuyruğa varmadan önce tamamlanır, iki zaman diliminde; daha sonra A (düşük öncelikte) tekrar çalışmaya başlar.



Şekil 8.3: **Bir Etkileşimli İş Geldi**



Şekil 8.4: **I/O ve CPU'nun Şiddetli İş Yükünün Karışımı**

En yüksek kuyruğa eklenir; çalışma süresi kısa olduğundan (sadece 20 ms), B en alt kuyruğa ulaşmadan tamamlanır, iki zaman diliminde; sonra A (düşük öncelikte) tekrar çalışmaya devam eder.

Bu örnekten, algoritmanın büyük amaçlarından birini anlayabileceğinizi umuyoruz: bir işin kısa bir iş mi yoksa uzun süren bir iş mi olacağını bilmiyor olmasından dolayı, ilk olarak kısa bir iş olma ihtimalini düşünür ve böylece işe yüksek öncelik verir. Eğer gerçekten kısa bir işse, hızlı bir şekilde çalışıp tamamlanacaktır; eğer kısa bir iş değilse, yavaş yavaş kuyruklarda aşağı doğru hareket edecek ve böylece kendini uzun süreli, daha çok toplu işlem gibi bir süreç olarak kanıtlayacaktır. Bu şekilde, MLFQ SJF'yi yaklaştırır.

Örnek 3: Peki ya I/O

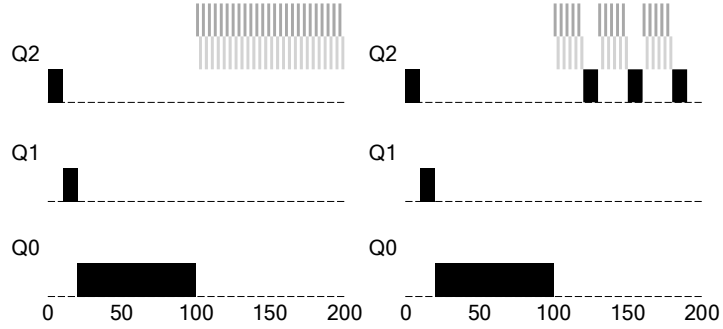
Şimdi biraz I/O ile birlikte bir örnek inceleyelim. Yukarıda 4b kuralında belirtildiği gibi, bir iş zaman dilimini kullanmadan önce işlemciden vazgeçerse, aynı öncelik düzeyinde tutarız. Bu kuralın amacı basittir: örneğin, bir etkileşimli iş çok miktarda I/O yapıyorsa (örneğin, kullanıcı girişi için fare veya klavye'den bekleyen), zaman dilimini tamamlamadan CPU'yu terk edecektir; bu tür bir durumda, işi cezalandırmak istemeyiz ve sadece aynı seviyede tutarız.

Şekil 8.4 Bu nasıl çalıştığını gösteren bir örnek verir, etkileşimli iş B (gri olarak gösterilmiş) sadece 1 ms için CPU'ya ihtiyaç duyar ve CPU'yu bir uzun süren toplu iş A (siyah olarak gösterilmiş) ile yarışır. MLFQ yaklaşımı, B'yi en yüksek öncelikte tutar çünkü B CPU'yu sürekli bırakır; eğer B etkileşimli bir işse, MLFQ daha da etkileşimli işleri hızlı çalıştırma amacına ulaşır.

Mevcut MLFQ'muzla İlgili Sorunlar

Böylece temel bir MLFQ'ya sahibiz. Uzun süren işlerle CPU'yu dengeli bir şekilde paylaştığı ve kısa veya IO yoğun etkileşimli işleri hızlı çalıştırdığı gibi oldukça iyi bir iş çıkarmaya görünüyor. Ne yazık ki, şimdiye kadar geliştirdiğimiz yaklaşım ciddi hatalar içerir. Bu konuda bir fikriniz var mı??

(Bu, düşüncelerinizi olabildiğince kurnazca bir şekilde düşünmeniz gereken yerdir.)



Şekil 8.5: **(Sol) Olmadan (Sağ) Olmuşken Öncelik Arttırmak**

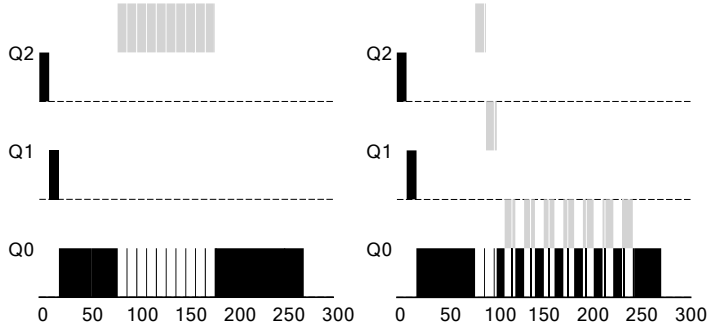
İlk olarak, **aclık(starvation)** problemi vardır: sistemde "cok fazla" etkilesimli is varsa, bunlar tüm CPU zamanını tüketecek ve böylece uzun süreli isler hicbir CPU zamanı almayacaktır (açlık çekerler). Bu senaryoda bu işler üzerinde bazı ilerlemeler yapmak isteriz.

İkinci olarak, akıllı bir kullanıcı programını planlayıcıyı aldatmak için yeniden yazabilir. Planlayıcıyı aldatma genellikle, adil payınızdan daha fazla kaynak elde etmek için bir şey yapmanın kurnaz bir yoludur. Açıkladığımız algoritma aşağıdaki saldırıya açıktır: zaman diliminin bitmesinden önce bir I/O işlemini gerçekleştirin (umursamadığınız bir dosyaya) ve böylece CPU'yu bırakın; böylece aynı kuyrukta kalabilir ve böylece daha yüksek bir CPU zamanı payı elde edebilirsiniz. Doğru bir şekilde yapıldığında (örneğin, bir zaman diliminin % 99'unu kullanarak CPU'yu bırakarak), bir iş CPU'yu neredeyse monopolize edebilir.

Son olarak, bir program zaman içinde davranışını değiştirebilir; CPU ile bağlantılı olan bir şey, etkileşim dönemine geçebilir. Mevcut yaklaşımımızla, böyle bir iş talihsiz olur ve sistemdeki diğer etkileşimli işler gibi değerlendirilmez.

İpucu: ZAMANLAMA, SALDIRIDAN GÜVENLİ OLMALIDIR.

Bir zamanlama politikasının bir **güvenlik(security)** kaygısı olup olmadığını düşünebilirsiniz, içerisinde bulunduğu işletim sisteminde (bu makalede tartışıldığı gibi) veya daha geniş bir bağlamda (örneğin, bir dağıtık depolama sisteminin I / O istek işleme [Y + 18] işleme), ancak giderek daha fazla durumda tam olarak budur. Modern bir veri merkezini düşünün, bu veri merkezinde dünya çapındaki kullanıcılar CPU'ları, bellekleri, ağları ve depolama sistemlerini paylaşır; politika tasarımı ve uygulamalarına dikkat edilmezse, tek bir kullanıcı diğerlerini olumsuz etkileyebilir ve kendisi için bir avantaj sağlayabilir. Bu nedenle, zamanlama politikası bir sistemin güvenliğinin önemli bir parçasıdır ve dikkatle yapılmalıdır.



Şekil 8.6: **(Sol) Olmadan (Sağ) Olan Oyun Toleransı**

8.3 Grişim #2: Önceliği Arttırmak

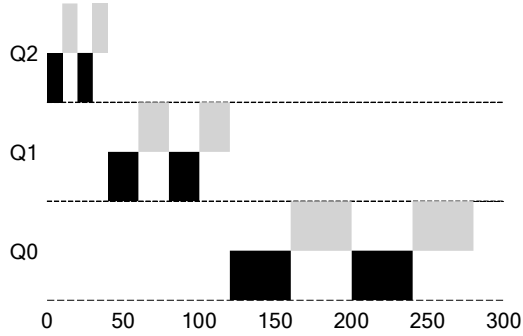
Yoksunluk sorununu önlemek için kuralları değiştirmeyi ve CPU bağlı işlerin bazı ilerlemelerde bulunup bulunamayacağını görmek için deneyelim. Bu amaçla ne yapabiliriz?
Bu basit fikir, sistemdeki tüm işlerin periyodik olarak önceliklerini **artırmaktır(boost)**. Bunu gerçekleştirmek için çok fazla yol vardır, ancak basit bir şey yapalım: hepsini en üst kuyruğa atın; bu nedenle, yeni bir kural:

- **Kural 5** Süre biriminden sonra, sistemdeki tüm işleri en üst kuyruğa taşı.

Yeni kuralımız aynı anda iki sorunu çözer. Öncelikle, işlerin yoksunluk yaşamaması garanti edilir: en üst kuyrukta oturarak, bir iş round-robin şeklinde diğer yüksek öncelikli işlerle CPU'yu paylaşacak ve böylece sonunda hizmet alacaktır. İkinci olarak, eğer CPU-bağlı bir iş etkileşimli hale geldiyse, zamanlayıcı öncelik artışını aldıktan sonra onu doğru bir şekilde işler.

Öreneğin, Bu senaryoda, uzun süren bir işin iki kısa süren etkileşimli işle rekabet ederken davranışını gösteriyoruz. Şekil 8.5 (sayfa 6)'da iki grafik gösterilir. Solda, bir öncelik artışı yoktur ve bu nedenle uzun süren iş, iki kısa iş gelir gelmez yoksun kalır; sağda, her 50 ms'te bir öncelik artışı vardır (bu muhtemelen çok küçük bir değerdir, ancak burada örnek için kullanılır) ve böylece en azından uzun süren işin bazı ilerlemelerde bulunacağını garanti ederiz, her 50 ms'de en yüksek önceliğe yükseltilerek düzenli olarak çalışmaya devam eder.

Tabii ki, zaman periyodu S'nin eklenmesi açık bir soruya yol açar: S ne olmalıdır? John Ousterhout, iyi bilinen bir sistem araştırmacısı [O11], sistemlerdeki bu değerleri **voo-doo sabitleri(voo-doo constants)** olarak adlandırırdı, çünkü doğru olarak ayarlamak için bir tür kara büyü gerekiyormuş gibi görünüyordu. Ne yazık ki, S de bu tadı taşıyor. Çok yüksek ayarlanırsa, uzun süren işler yoksun kalabilir; çok düşükse, etkileşimli işler CPU'ya doğru bir pay almayabilir.



Şekil 8.7: **Düşük Öncelik, Uzun Kuantumlar.**

8.4 Girişim #3: Daha İyi Hesap

Artık çözmemiz gereken bir problem daha var: planlayıcımızı nasıl önlenebilir? Gerçek suçlu burada, tahmin ettiğiniz gibi, 4a ve 4b kurallarıdır, bu kurallar bir işin zaman dilimi dolmadan önce CPU'yu bırakmasını sağlar. Ne yapmalıyız?

Buradaki çözüm, MLFQ'nun her seviyesinde CPU zamanını **daha iyi hesaplamaktır(better accounting).** Bir işlemin bir seviyede kullandığı zaman dilimini unutmayıp, planlayıcı izlemeli; bir işlemin ayrıldığı yerde, bir sonraki öncelik kuyruğuna düşürülür. Kullandığı zaman dilimini uzun bir patlama veya birçok küçük patlama olarak kullanıp kullanmadığı önemli değildir. Böylece 4a ve 4b kurallarını aşağıdaki tek kurala yeniden yazıyoruz:

- **Kural 4:** Bir iş, bir seviyedeki zamanını kullandıktan sonra (CPU'yu ne kadar verdiği önemli değil), önceliği azaltılır (yani bir kuyruk düşer).

  Bir örnek bakalım. Şekil 8.6 (sayfa 7), eski 4a ve 4b Kuralları (solda) ve yeni anti-oyun Kuralları 4 ile bir yük dengesinin planlayıcıyı aldatmaya çalıştığını gösterir. Oyun koruma olmadan, bir işlem, bir zaman diliminin sonuna çok yakın bir I / O verilebilir ve böylece CPU zamanını domine edebilir. Böyle koruma mekanizmaları yerleştirildiğinde, işlemin I / O davranışı ne olursa olsun, yavaş yavaş kuyruklara iner ve böylece adil bir CPU payı elde edemez.

8.5 MLFQ Ayarı Ve Diğer Sorunlar

MLFQ zamanlama ile bazı diğer sorunlar ortaya çıkar. Bir büyük soru, bu tür bir planlayıcının nasıl **parametreleştirileceğidir**. Örneğin, kaç kuyruk olmalıdır? Her kuyruk için zaman diliminin ne kadar büyük olması gerekir? Açlık önlemek ve davranış değişikliklerini hesaba kattırmak için öncelik sıklıkla artırılmalıdır? Bu sorulara kolay cevaplar yoktur ve böylece yük dengesi ile biraz deneyim ve daha sonra planlayıcının ayarı, memnuniyet verici bir dengenin sağlanmasına yol açacaktır.

İpucu : VOO-DOO SABİTLERİNDEN KAÇININ (OUSTERHOUT KANUNU)
Mümkün olduğunca voo-doo sabitlerinden kaçınmak için iyi bir fikirdir. Ne yazık ki, yukarıdaki örnekte olduğu gibi, genellikle zordur. Birisi iyi bir değer öğrenmeye çalışabilir, ancak bu da basit değildir. Sık sonuç: varsayılan parametre değerleriyle dolu bir yapılandırma dosyası, bir şey tam olarak doğru çalışmadığında bir deneyimli yönetici tarafından ayarlanabilir. Sanırım bu genellikle değiştirilmez ve bu nedenle, varsayılan değerlerin sahada iyi çalıştığını umuyoruz. Bu ipucunu eski OS profesörümüz John Ousterhout'a borçluyuz ve bu nedenle **Ousterhout Kanunu(Law)**olarak adlandırıyoruz..

Örneğin, çoğu MLFQ varyantı farklı kuyruklar arasında değişen zaman dilimi uzunluğuna izin verir. Yüksek öncelikli kuyruklar genellikle kısa zaman dilimlerine sahip olur; nihayetinde etkileşimli işlerden oluşurlar ve bu nedenle aralarında hızlıca değiştirmek mantıklıdır (örn., 10 veya daha az milisaniye). Bu karşın, düşük öncelikli kuyruklar, CPU bağlı uzun süren işleri içerir; bu nedenle, daha uzun zaman dilimleri iyi çalışır (örn., 100 ms). Şekil 8.7 (sayfa 8) 'da, iki işin en yüksek kuyrukta 20 ms (10 ms zaman dilimiyle), orta kuyrukta 40 ms (20 ms zaman dilimiyle) ve en düşük kuyrukta 40 ms zaman dilimiyle çalıştığı bir örnek gösterilir.

Solaris MLFQ uygulaması – Zaman paylaşımı, zamanlama sınıfı veya TS - özellikle yapılandırmaya kolaydır; bir işin önceliğini süresince nasıl değiştireceğini, her zaman diliminin ne kadar süreceğini ve bir işin önceliğinin ne sıklıkla artırılacağını belirleyen bir dizi tablo sağlar [AD00]; bir yönetici bu tablo ile oynayarak planlayıcının farklı şekillerde davranmasını sağlayabilir. Tablo için varsayılan değerler 60 kuyruktur, yavaş yavaş artan zaman dilim uzunlukları 20 milisaniyeden (en yüksek öncelik) birkaç yüz milisaniyeye (en düşük) kadar ve önceliklerin yaklaşık 1 saniyede bir artırıldığı yerlerdir.

Diğer MLFQ planlayıcılar bu bölümde açıklanan kurallara dayanan bir tablo veya tam olarak kullanmazlar; bunun yerine matematiksel formüller kullanarak öncelikleri ayarlar. Örneğin, FreeBSD planlayıcısı (versiyon 4.3) bir işin mevcut öncelik seviyesini hesaplamak için bir formül kullanır ve bu işlem tarafından kullanılan CPU'ya dayanır [LM + 89]; ayrıca, kullanım zamanla azalır, bu kitapta açıklanan farklı bir yöntemle istenen öncelik artışı sağlar. Bu tip **bozulan-kullanım(decay-usage)** algoritmaları ve özellikleri hakkında harika bir genel bakış için Epema'nın makalesini inceleyin [E95].

Son olarak, birçok planlayıcının diğer özellikleri de olabilir. Örneğin, bazı planlayıcılar en yüksek öncelik seviyelerini işletim sistemi işi için ayırır; bu nedenle tipik kullanıcı işleri sistemdeki en yüksek öncelik seviyelerine asla ulaşamaz. Bazı sistemler ayrıca bazı kullanıcı **tavsiyelerine(advice)** yardımcı olmak için de izin verir; örneğin, komut satırı yardımcısı nice kullanarak bir işin önceliğini (biraz) artırabilir veya azaltabilir ve böylece herhangi bir zamanda çalışma şansını artırabilir veya azaltabilir. Daha fazla bilgi için man sayfasına bakın.

İpucu: MÜMKÜNSE ÖNERİ KULLANIN

İşletim sistemi, genellikle sistemin her bir işlemini en iyi şekilde nasıl yöneteceğini bilmez, bu nedenle kullanıcıların veya yöneticilerin işletim sistemine bazı ipuçları vermesine olanak sağlayan arabirimlerin sağlanması çoğu zaman yararlıdır. Böyle **ipuçlarını(hints)** öneriler olarak adlandırırız, çünkü işletim sistemi mutlaka ona dikkat etmek zorunda değildir, ancak daha iyi bir karar vermek için önerivi dikkate alabilir. Bu ipuçları, işletim sisteminde çok sayıda yerde yararlıdır, bu yerler arasında planlayıcı (örneğin, nice ile), bellek yöneticisi (örneğin, madvise ile) ve dosya sistemi (örneğin, bilgiye dayalı önceden alma ve önbellekleme [P + 95] ile) dahil olmak üzere.

8.6 MLFQ: Özet

Çok Seviyeli Geri Bildirim Kuyruğu (MLFQ) olarak bilinen bir zamanlama yaklaşımı açıkladık. Umarım neden böyle olduğunu anlayabilirsiniz: birden fazla kuyruk seviyesi var ve bir işin önceliğini belirlemek için geri bildirim kullanır. Tarih onun rehberidir: işlerin zaman içinde nasıl davrandığına dikkat edin ve ona göre davranın.

Bölüm boyunca dağıtılan MLFQ kurallarının iyileştirilmiş kümesi, keyfiniz için burada tekrarlandı:

- **Kural 1:** Eğer (A)'nın önceliği > (B)'nin önceliği ise, A çalışır (B çalışmaz).
- **Kural 2:** (A)'nın önceliği = (B)'nin önceliği ise, A & B çalışır in round-robin verilen kuyruğun zaman dilimini (kantum uzunluğunu) kullanarak.
- **Kural 3:** Bir iş sisteme girdiğinde, en yüksek öncelikte (en üstteki kuyrukta) yerleştirilir.
- **Kural 4:** "Bir iş, verilen seviyedeki zaman kotasını kullandıktan sonra (CPU'dan kaç kere vazgeçtiğine bakılmaksızın), önceliği azaltılır (yani, bir sıraya düşer)."
- **Kural 5:** "Belirli bir zaman dilimi S sonrasında, sistemdeki tüm işleri en üstteki sıraya taşıyın."

MLFQ, aşağıdaki nedenlerden dolayı ilginçtir: bir işin doğası hakkında a priori bilgi talep etmek yerine, bir işin çalışmasını gözlemler ve buna göre öncelik verir. Bu şekilde, en iyi iki dünyayı da elde eder: kısa süren etkileşimli işler için harika bir genel performans (SJF / STCF benzeri) sağlar ve uzun süren CPU yoğun yükler için adil ve ilerleme sağlar. Bu nedenle, BSD UNIX türevleri [LM + 89, B86], Solaris [M06] ve Windows NT ve sonraki Windows işletim sistemleri [CS97] gibi birçok sistem, temel planlayıcı olarak bir MLFQ formunu kullanır.

Referanslar

[AD00] "Multilevel Feedback Queue Scheduling in Solaris" by Andrea Arpaci-Dusseau. Available: http://www.ostep.org/Citations/notes-solaris.pdf. *A great short set of notes by one of the authors on the details of the Solaris scheduler. OK, we are probably biased in this description, but the notes are pretty darn good.*

[B86] "The Design of the UNIX Operating System" by M.J. Bach. Prentice-Hall, 1986. *One of the classic old books on how a real UNIX operating system is built; a definite must-read for kernel hackers.*

[C+62] "An Experimental Time-Sharing System" by F. J. Corbato, M. M. Daggett, R. C. Daley. IFIPS 1962. *A bit hard to read, but the source of many of the first ideas in multi-level feedback scheduling. Much of this later went into Multics, which one could argue was the most influential operating system of all time.*

[CS97] "Inside Windows NT" by Helen Custer and David A. Solomon. Microsoft Press, 1997. *The NT book, if you want to learn about something other than UNIX. Of course, why would you? OK, we're kidding; you might actually work for Microsoft some day you know.*

[E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors" by D.H.J. Epema. SIGMETRICS '95. *A nice paper on the state of the art of scheduling back in the mid 1990s, including a good overview of the basic approach behind decay-usage schedulers.*

[LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System" by S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman. Addison-Wesley, 1989. *Another OS classic, written by four of the main people behind BSD. The later versions of this book, while more up to date, don't quite match the beauty of this one.*

[M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture" by Richard McDougall. Prentice-Hall, 2006. *A good book about Solaris and how it works.*

[O11] "John Ousterhout's Home Page" by John Ousterhout. `www.stanford.edu/~ouster/`. *The home page of the famous Professor Ousterhout. The two co-authors of this book had the pleasure of taking graduate operating systems from Ousterhout while in graduate school; indeed, this is where the two co-authors got to know each other, eventually leading to marriage, kids, and even this book. Thus, you really can blame Ousterhout for this entire mess you're in.*

[P+95] "Informed Prefetching and Caching" by R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka. SOSP '95, Copper Mountain, Colorado, October 1995. *A fun paper about some very cool ideas in file systems, including how applications can give the OS advice about what files it is accessing and how it plans to access them.*

[Y+18] "Principled Schedulability Analysis for Distributed Storage Systems using Thread Architecture Models" by Suli Yang, Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '18, San Diego, California. *A recent work of our group that demonstrates the difficulty of scheduling I/O requests within modern distributed storage systems such as Hive/HDFS, Cassandra, MongoDB, and Riak. Without care, a single user might be able to monopolize system resources.*

Ödev (Simulasyon)

"Bu program, mlfq.py, bu bölümde sunulan MLFQ planlayıcısının nasıl davranacağını gösterir. Ayrıntılar için README'ye bakın.".

Sorular

1. Sadece iki iş ve iki kuyrukla birkaç rastgele oluşturulmuş problemi çalıştırın; her bir için MLFQ çalışma izini hesaplayın. Her işin uzunluğunu sınırlandırarak ve I / O'ları kapatarak yaşamınızı kolaylaştırın.

Basit bir rastgele üretim örneği

Rastgele çekirdek: -s 1
Kuyruk sayısı: -n 2
İş sayısı: -j 2
Maksimum iş uzunluğu: -m 10
IO yürütmesi yok: -M 0

```
emre@EMRE:~/Desktop/ostep/ostep-homework/cpu-sched-mlfq$ ./mlfq.py -s 1 -n 2 -j 2 -m 10 -M 0 -c
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime   2 - ioFreq   0
  Job  1: startTime   0 - runTime   7 - ioFreq   0


Execution Trace:

[ time 0 ] JOB BEGINS by JOB 0
[ time 0 ] JOB BEGINS by JOB 1
[ time 0 ] Run JOB 0 at PRIORITY 1 [ TICKS 9 ALLOT 1 TIME 1 (of 2) ]
[ time 1 ] Run JOB 0 at PRIORITY 1 [ TICKS 8 ALLOT 1 TIME 0 (of 2) ]
[ time 2 ] FINISHED JOB 0
[ time 2 ] Run JOB 1 at PRIORITY 1 [ TICKS 9 ALLOT 1 TIME 6 (of 7) ]
[ time 3 ] Run JOB 1 at PRIORITY 1 [ TICKS 8 ALLOT 1 TIME 5 (of 7) ]
[ time 4 ] Run JOB 1 at PRIORITY 1 [ TICKS 7 ALLOT 1 TIME 4 (of 7) ]
[ time 5 ] Run JOB 1 at PRIORITY 1 [ TICKS 6 ALLOT 1 TIME 3 (of 7) ]
[ time 6 ] Run JOB 1 at PRIORITY 1 [ TICKS 5 ALLOT 1 TIME 2 (of 7) ]
[ time 7 ] Run JOB 1 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 1 (of 7) ]
[ time 8 ] Run JOB 1 at PRIORITY 1 [ TICKS 3 ALLOT 1 TIME 0 (of 7) ]
[ time 9 ] FINISHED JOB 1

Final statistics:
  Job  0: startTime   0 - response   0 - turnaround   2
  Job  1: startTime   0 - response   2 - turnaround   9

  Avg  1: startTime n/a - response 1.00 - turnaround 5.50
```

2 öncelik kuyruğu vardır, İş 0'ın uzunluğu 2'dir ve 1. kuyrukta 2 ms boyunca yürütülür ve zaman dilimi 2 ms tüketir. Aynı kuyrukta İş 1'e geçin ve İş-1 7 ms boyunca yürütülür ve zaman dilimi 7 ms tüketir.

2. Bölümdeki örnekleri tekrarlayabilmek için planlayıcıyı nasıl çalıştırırsınız?

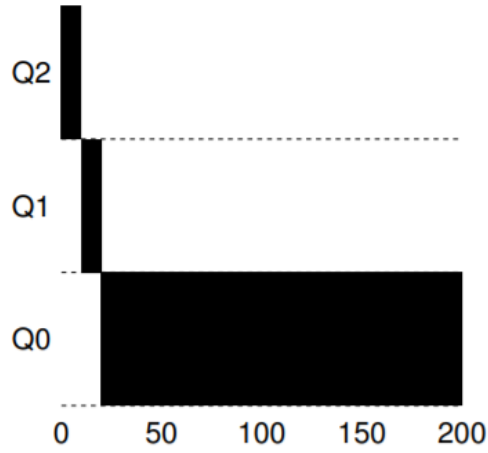


Figure 8.2: **Long-running Job Over Time**

Süreçlerin her kuyrukta 1 zaman dilimi (kota), yani 10 ms boyunca çalıştığı üç seviyeli öncelik kuyrukları

Yalnızca bir süreç vardır, bu nedenle kuyruğun son seviyesi dönüşümlü olarak çalışır, yani süreç her zaman çalışır

Çalışma sonuçları:

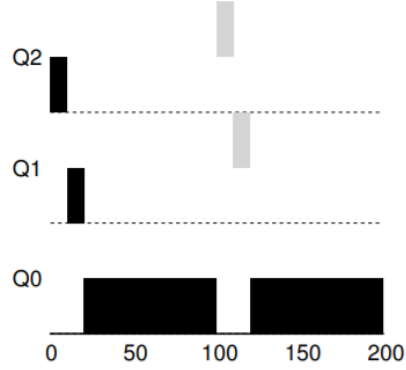


Figure 8.3: **Along Came An Interactive Job**

IO işlemleri içermez

İlk iş 0 ms'de gelir ve GÇ olmadan 180 ms boyunca çalışır: 0,180,0
İkinci iş 100 ms'de gelir ve IO olmadan 20 ms boyunca çalışır: 100,20,0
Zaman dilimi uzunluğu 10 ms: -q 10
3 öncelik kuyruğu
Kuyruk başına 1 zaman dilimi uzunluğu
Varsayılan olarak hiçbir öncelik yükseltilmemiştir
Çalışma sonuçları:

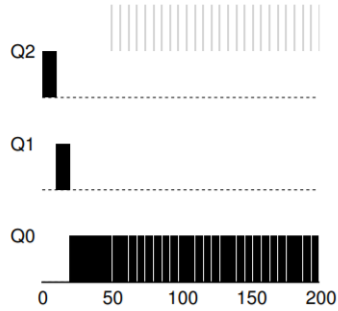


Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

Önceden JOB 0 çalışır ve son kuyruğa taşınana kadar bir sonraki öncelik kuyruğuna geçer
İŞ 1 her 1 ms'de bir IO yürütür
İŞ 0 her 5 ms'de bir çalışır JOB 1
Çalışma sonuçları:

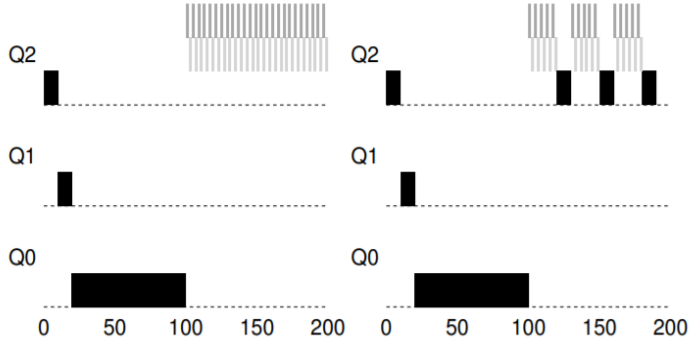


Figure 8.5: **Without (Left) and With (Right) Priority Boost**

İlk süreç 0 ms'de gelir ve en düşük önceliğe indirilene kadar çalışır
İkinci ve üçüncü süreçler 100 ms'de gelir ve her 5 ms'de bir IO işlemleri gerçekleştirir, her IO 5 ms sürer
GÇ yürütüldükten sonra öncelik korunur
İlk süreç, sırayla IO ve CPU'yu işgal ettiği için açlık çekiyor

Çalışma sonuçları:

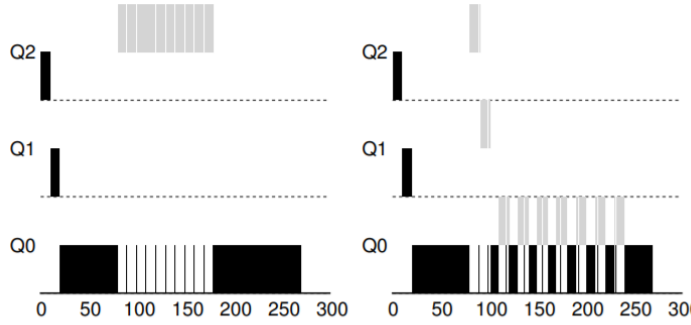


Figure 8.6: **Without (Left) and With (Right) Gaming Tolerance**

Burada sürecin çalışma süresi kısaltılır
Çalışma sonuçları:

Şekil 8.6 (sağda)

IO işlemleri gerçekleştirmeyen süreçlerin çalışma zamanı, CPU'nun boşta kalmasını önlemek için uzatılmıştır
Çalışma sonuçları:

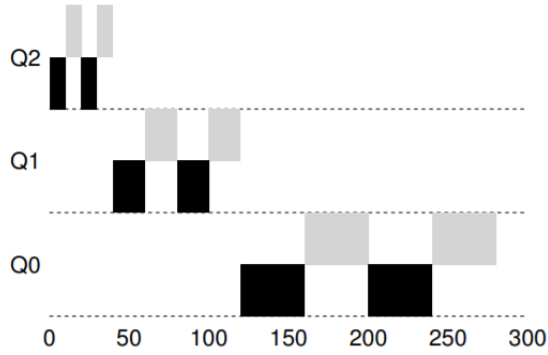


Figure 8.7: **Lower Priority, Longer Quanta**

Öncelik ne kadar düşükse, zaman dilimi o kadar uzun olur.

3. Planlayıcı parametrelerini tamamen round – robin planlayıcısı gibi davranacak şekilde nasıl yapılandırırsınız?

-Sadece bir kuyruk kullanın

```
emre@EMRE:~/Desktop/ostep/ostep-homework/cpu-sched-mlfq$ ./mlfq.py -n 1 -q 5 -l 0,10,0:0,10,0 -c
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 1
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is   5
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  10 - ioFreq   0
  Job  1: startTime   0 - runTime  10 - ioFreq   0


Execution Trace:

[ time 0 ] JOB BEGINS by JOB 0
[ time 0 ] JOB BEGINS by JOB 1
[ time 0 ] Run JOB 0 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 9 (of 10) ]
[ time 1 ] Run JOB 0 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 8 (of 10) ]
[ time 2 ] Run JOB 0 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 7 (of 10) ]
[ time 3 ] Run JOB 0 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 6 (of 10) ]
[ time 4 ] Run JOB 0 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 5 (of 10) ]
[ time 5 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 9 (of 10) ]
[ time 6 ] Run JOB 1 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 8 (of 10) ]
[ time 7 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 7 (of 10) ]
[ time 8 ] Run JOB 1 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 6 (of 10) ]
[ time 9 ] Run JOB 1 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 5 (of 10) ]
[ time 10 ] Run JOB 0 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 4 (of 10) ]
[ time 11 ] Run JOB 0 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 3 (of 10) ]
[ time 12 ] Run JOB 0 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 2 (of 10) ]
[ time 13 ] Run JOB 0 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 1 (of 10) ]
[ time 14 ] Run JOB 0 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 0 (of 10) ]
[ time 15 ] FINISHED JOB 0
[ time 15 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 4 (of 10) ]
[ time 16 ] Run JOB 1 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 3 (of 10) ]
[ time 17 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 2 (of 10) ]
[ time 18 ] Run JOB 1 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 1 (of 10) ]
[ time 19 ] Run JOB 1 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 0 (of 10) ]
[ time 20 ] FINISHED JOB 1

Final statistics:
  Job  0: startTime   0 - response   0 - turnaround  15
  Job  1: startTime   0 - response   5 - turnaround  20

  Avg  1: startTime n/a - response 2.50 - turnaround 17.50
```

Her iki süreç de 0 ms'de gelir, her ikisi de 10 ms boyunca çalışır, ilk sürecin IO'su vardır, ikinci sürecin IO'su yoktur

Bir kuyruk kullanın
IO bittiğinde kuyruğun başına taşınır.

4. İki iş ve planlayıcı parametreleriyle bir iş yükü oluşturun ve bir iş, planlayıcıyı yönetmek ve belirli bir zaman aralığında CPU'nun %99'unu elde etmek için eski 4a ve 4b kurallarını (-S bayrağı ile açılmış) kullanır.

İki iş için yük ve zamanlayıcı parametrelerini tasarlayın, böylece bir iş zamanlayıcıyı "kandırmak" ve belirli bir zaman aralığında CPU'nun %99'unu almak için önceki 4a ve 4b (-S) kurallarını kullanır

Kural 4a: bir iş tüm zaman dilimini kullandıktan sonra önceliğini düşürür (bir sonraki kuyruğa geçer)
Kural 4b: Bir iş kendi zaman dilimi içinde CPU'yu aktif olarak serbest bırakırsa, öncelik aynı kalır

Zaman dilimi uzunluğu 10 ve IO süresi 1 olan 2 kuyruk kullanıldığında, bir IO talep edildikten sonra öncelik değişmez (CPU'dan vazgeçilir); ilk işlem (JOB 0) 0 ms'de gelir, 50 ms boyunca çalışır ve her 9 ms'de bir IO işlemi talep eder; ikinci işlem (JOB 1) 0 ms'de gelir, 50 ms boyunca çalışır ve bir IO gerçekleştirmez.

19 - 37 döneminde, ikinci süreç (JOB 1) sadece 1 ms CPU alırken, ilk süreç (JOB 0) 18 ms alır, bu da %94,7'ye denk gelir ...

```
emre@EMRE:~/Desktop/ostep/ostep-homework/cpu-sched-mlfq$ ./mlfq.py -n 2 -q 10 -i 1 -l 0,50,9:0,50,0 -S -c
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 1
OPTIONS stayAfterIO True
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  50 - ioFreq   9
  Job  1: startTime   0 - runTime  50 - ioFreq   0


Execution Trace:

[ time 0 ] JOB BEGINS by JOB 0
[ time 0 ] JOB BEGINS by JOB 1
[ time 0 ] Run JOB 0 at PRIORITY 1 [ TICKS 9 ALLOT 1 TIME 49 (of 50) ]
[ time 1 ] Run JOB 0 at PRIORITY 1 [ TICKS 8 ALLOT 1 TIME 48 (of 50) ]
[ time 2 ] Run JOB 0 at PRIORITY 1 [ TICKS 7 ALLOT 1 TIME 47 (of 50) ]
[ time 3 ] Run JOB 0 at PRIORITY 1 [ TICKS 6 ALLOT 1 TIME 46 (of 50) ]
[ time 4 ] Run JOB 0 at PRIORITY 1 [ TICKS 5 ALLOT 1 TIME 45 (of 50) ]
[ time 5 ] Run JOB 0 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 44 (of 50) ]
[ time 6 ] Run JOB 0 at PRIORITY 1 [ TICKS 3 ALLOT 1 TIME 43 (of 50) ]
[ time 7 ] Run JOB 0 at PRIORITY 1 [ TICKS 2 ALLOT 1 TIME 42 (of 50) ]
[ time 8 ] Run JOB 0 at PRIORITY 1 [ TICKS 1 ALLOT 1 TIME 41 (of 50) ]
[ time 9 ] IO_START by JOB 0
IO DONE
[ time 9 ] Run JOB 1 at PRIORITY 1 [ TICKS 9 ALLOT 1 TIME 49 (of 50) ]
[ time 10 ] IO_DONE by JOB 0
```

```
[ time 58 ] Run JOB 1 at PRIORITY 0 [ TICKS 6 ALLOT 1 TIME 36 (of 50) ]
[ time 59 ] IO_DONE by JOB 0
[ time 59 ] Run JOB 0 at PRIORITY 1 [ TICKS 9 ALLOT 1 TIME 4 (of 50) ]
[ time 60 ] Run JOB 0 at PRIORITY 1 [ TICKS 8 ALLOT 1 TIME 3 (of 50) ]
[ time 61 ] Run JOB 0 at PRIORITY 1 [ TICKS 7 ALLOT 1 TIME 2 (of 50) ]
[ time 62 ] Run JOB 0 at PRIORITY 1 [ TICKS 6 ALLOT 1 TIME 1 (of 50) ]
[ time 63 ] Run JOB 0 at PRIORITY 1 [ TICKS 5 ALLOT 1 TIME 0 (of 50) ]
[ time 64 ] FINISHED JOB 0
[ time 64 ] Run JOB 1 at PRIORITY 0 [ TICKS 5 ALLOT 1 TIME 35 (of 50) ]
[ time 65 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 34 (of 50) ]
[ time 66 ] Run JOB 1 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 33 (of 50) ]
[ time 67 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 32 (of 50) ]
[ time 68 ] Run JOB 1 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 31 (of 50) ]
[ time 69 ] Run JOB 1 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 30 (of 50) ]
[ time 70 ] Run JOB 1 at PRIORITY 0 [ TICKS 9 ALLOT 1 TIME 29 (of 50) ]
[ time 71 ] Run JOB 1 at PRIORITY 0 [ TICKS 8 ALLOT 1 TIME 28 (of 50) ]
[ time 72 ] Run JOB 1 at PRIORITY 0 [ TICKS 7 ALLOT 1 TIME 27 (of 50) ]
[ time 73 ] Run JOB 1 at PRIORITY 0 [ TICKS 6 ALLOT 1 TIME 26 (of 50) ]
[ time 74 ] Run JOB 1 at PRIORITY 0 [ TICKS 5 ALLOT 1 TIME 25 (of 50) ]
[ time 75 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 24 (of 50) ]
[ time 76 ] Run JOB 1 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 23 (of 50) ]
[ time 77 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 22 (of 50) ]
[ time 78 ] Run JOB 1 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 21 (of 50) ]
[ time 79 ] Run JOB 1 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 20 (of 50) ]
[ time 80 ] Run JOB 1 at PRIORITY 0 [ TICKS 9 ALLOT 1 TIME 19 (of 50) ]
[ time 81 ] Run JOB 1 at PRIORITY 0 [ TICKS 8 ALLOT 1 TIME 18 (of 50) ]
[ time 82 ] Run JOB 1 at PRIORITY 0 [ TICKS 7 ALLOT 1 TIME 17 (of 50) ]
[ time 83 ] Run JOB 1 at PRIORITY 0 [ TICKS 6 ALLOT 1 TIME 16 (of 50) ]
[ time 84 ] Run JOB 1 at PRIORITY 0 [ TICKS 5 ALLOT 1 TIME 15 (of 50) ]
[ time 85 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 14 (of 50) ]
[ time 86 ] Run JOB 1 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 13 (of 50) ]
[ time 87 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 12 (of 50) ]
[ time 88 ] Run JOB 1 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 11 (of 50) ]
[ time 89 ] Run JOB 1 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 10 (of 50) ]
[ time 90 ] Run JOB 1 at PRIORITY 0 [ TICKS 9 ALLOT 1 TIME 9 (of 50) ]
[ time 91 ] Run JOB 1 at PRIORITY 0 [ TICKS 8 ALLOT 1 TIME 8 (of 50) ]
[ time 92 ] Run JOB 1 at PRIORITY 0 [ TICKS 7 ALLOT 1 TIME 7 (of 50) ]
[ time 93 ] Run JOB 1 at PRIORITY 0 [ TICKS 6 ALLOT 1 TIME 6 (of 50) ]
[ time 94 ] Run JOB 1 at PRIORITY 0 [ TICKS 5 ALLOT 1 TIME 5 (of 50) ]
[ time 95 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 4 (of 50) ]
[ time 96 ] Run JOB 1 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 3 (of 50) ]
[ time 97 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 2 (of 50) ]
[ time 98 ] Run JOB 1 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 1 (of 50) ]
[ time 99 ] Run JOB 1 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 0 (of 50) ]
[ time 100 ] FINISHED JOB 1

Final statistics:
  Job  0: startTime   0 - response   0 - turnaround  64
  Job  1: startTime   0 - response   9 - turnaround 100

  Avg  1: startTime n/a - response 4.50 - turnaround 82.00
```

Sonuç çok uzun olduğu için kodun sadece başını ve sonunu ekledim.

IO tamamlandığında işlemi kuyruğun başına koyun, böylece ilk 100 ms boyunca CPU'nun %99'unu alır

```
emre@EMRE:~/Desktop/ostep/ostep-homework/cpu-sched-mlfq$ ./mlfq.py -n 2 -q 100 -i 1 -l 0,100,99:0,2,0 -S -I -c
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is 100
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is 100
OPTIONS boost 0
OPTIONS ioTime 1
OPTIONS stayAfterIO True
OPTIONS iobump True


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime 100 - ioFreq  99
  Job  1: startTime   0 - runTime   2 - ioFreq   0


Execution Trace:

[ time 0 ] JOB BEGINS by JOB 0
[ time 0 ] JOB BEGINS by JOB 1
[ time 0 ] Run JOB 0 at PRIORITY 1 [ TICKS 99 ALLOT 1 TIME 99 (of 100) ]
[ time 1 ] Run JOB 0 at PRIORITY 1 [ TICKS 98 ALLOT 1 TIME 98 (of 100) ]
[ time 2 ] Run JOB 0 at PRIORITY 1 [ TICKS 97 ALLOT 1 TIME 97 (of 100) ]
[ time 3 ] Run JOB 0 at PRIORITY 1 [ TICKS 96 ALLOT 1 TIME 96 (of 100) ]
[ time 4 ] Run JOB 0 at PRIORITY 1 [ TICKS 95 ALLOT 1 TIME 95 (of 100) ]
[ time 5 ] Run JOB 0 at PRIORITY 1 [ TICKS 94 ALLOT 1 TIME 94 (of 100) ]
[ time 6 ] Run JOB 0 at PRIORITY 1 [ TICKS 93 ALLOT 1 TIME 93 (of 100) ]
[ time 7 ] Run JOB 0 at PRIORITY 1 [ TICKS 92 ALLOT 1 TIME 92 (of 100) ]
[ time 8 ] Run JOB 0 at PRIORITY 1 [ TICKS 91 ALLOT 1 TIME 91 (of 100) ]
[ time 9 ] Run JOB 0 at PRIORITY 1 [ TICKS 90 ALLOT 1 TIME 90 (of 100) ]
[ time 10 ] Run JOB 0 at PRIORITY 1 [ TICKS 89 ALLOT 1 TIME 89 (of 100) ]
```

```
[ time 64 ] Run JOB 0 at PRIORITY 1 [ TICKS 35 ALLOT 1 TIME 35 (of 100) ]
[ time 65 ] Run JOB 0 at PRIORITY 1 [ TICKS 34 ALLOT 1 TIME 34 (of 100) ]
[ time 66 ] Run JOB 0 at PRIORITY 1 [ TICKS 33 ALLOT 1 TIME 33 (of 100) ]
[ time 67 ] Run JOB 0 at PRIORITY 1 [ TICKS 32 ALLOT 1 TIME 32 (of 100) ]
[ time 68 ] Run JOB 0 at PRIORITY 1 [ TICKS 31 ALLOT 1 TIME 31 (of 100) ]
[ time 69 ] Run JOB 0 at PRIORITY 1 [ TICKS 30 ALLOT 1 TIME 30 (of 100) ]
[ time 70 ] Run JOB 0 at PRIORITY 1 [ TICKS 29 ALLOT 1 TIME 29 (of 100) ]
[ time 71 ] Run JOB 0 at PRIORITY 1 [ TICKS 28 ALLOT 1 TIME 28 (of 100) ]
[ time 72 ] Run JOB 0 at PRIORITY 1 [ TICKS 27 ALLOT 1 TIME 27 (of 100) ]
[ time 73 ] Run JOB 0 at PRIORITY 1 [ TICKS 26 ALLOT 1 TIME 26 (of 100) ]
[ time 74 ] Run JOB 0 at PRIORITY 1 [ TICKS 25 ALLOT 1 TIME 25 (of 100) ]
[ time 75 ] Run JOB 0 at PRIORITY 1 [ TICKS 24 ALLOT 1 TIME 24 (of 100) ]
[ time 76 ] Run JOB 0 at PRIORITY 1 [ TICKS 23 ALLOT 1 TIME 23 (of 100) ]
[ time 77 ] Run JOB 0 at PRIORITY 1 [ TICKS 22 ALLOT 1 TIME 22 (of 100) ]
[ time 78 ] Run JOB 0 at PRIORITY 1 [ TICKS 21 ALLOT 1 TIME 21 (of 100) ]
[ time 79 ] Run JOB 0 at PRIORITY 1 [ TICKS 20 ALLOT 1 TIME 20 (of 100) ]
[ time 80 ] Run JOB 0 at PRIORITY 1 [ TICKS 19 ALLOT 1 TIME 19 (of 100) ]
[ time 81 ] Run JOB 0 at PRIORITY 1 [ TICKS 18 ALLOT 1 TIME 18 (of 100) ]
[ time 82 ] Run JOB 0 at PRIORITY 1 [ TICKS 17 ALLOT 1 TIME 17 (of 100) ]
[ time 83 ] Run JOB 0 at PRIORITY 1 [ TICKS 16 ALLOT 1 TIME 16 (of 100) ]
[ time 84 ] Run JOB 0 at PRIORITY 1 [ TICKS 15 ALLOT 1 TIME 15 (of 100) ]
[ time 85 ] Run JOB 0 at PRIORITY 1 [ TICKS 14 ALLOT 1 TIME 14 (of 100) ]
[ time 86 ] Run JOB 0 at PRIORITY 1 [ TICKS 13 ALLOT 1 TIME 13 (of 100) ]
[ time 87 ] Run JOB 0 at PRIORITY 1 [ TICKS 12 ALLOT 1 TIME 12 (of 100) ]
[ time 88 ] Run JOB 0 at PRIORITY 1 [ TICKS 11 ALLOT 1 TIME 11 (of 100) ]
[ time 89 ] Run JOB 0 at PRIORITY 1 [ TICKS 10 ALLOT 1 TIME 10 (of 100) ]
[ time 90 ] Run JOB 0 at PRIORITY 1 [ TICKS 9 ALLOT 1 TIME 9 (of 100) ]
[ time 91 ] Run JOB 0 at PRIORITY 1 [ TICKS 8 ALLOT 1 TIME 8 (of 100) ]
[ time 92 ] Run JOB 0 at PRIORITY 1 [ TICKS 7 ALLOT 1 TIME 7 (of 100) ]
[ time 93 ] Run JOB 0 at PRIORITY 1 [ TICKS 6 ALLOT 1 TIME 6 (of 100) ]
[ time 94 ] Run JOB 0 at PRIORITY 1 [ TICKS 5 ALLOT 1 TIME 5 (of 100) ]
[ time 95 ] Run JOB 0 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 4 (of 100) ]
[ time 96 ] Run JOB 0 at PRIORITY 1 [ TICKS 3 ALLOT 1 TIME 3 (of 100) ]
[ time 97 ] Run JOB 0 at PRIORITY 1 [ TICKS 2 ALLOT 1 TIME 2 (of 100) ]
[ time 98 ] Run JOB 0 at PRIORITY 1 [ TICKS 1 ALLOT 1 TIME 1 (of 100) ]
[ time 99 ] IO_START by JOB 0
IO DONE
[ time 99 ] Run JOB 1 at PRIORITY 1 [ TICKS 99 ALLOT 1 TIME 1 (of 2) ]
[ time 100 ] IO_DONE by JOB 0
[ time 100 ] Run JOB 0 at PRIORITY 1 [ TICKS 99 ALLOT 1 TIME 0 (of 100) ]
[ time 101 ] FINISHED JOB 0
[ time 101 ] Run JOB 1 at PRIORITY 1 [ TICKS 98 ALLOT 1 TIME 0 (of 2) ]
[ time 102 ] FINISHED JOB 1

Final statistics:
  Job  0: startTime   0 - response   0 - turnaround 101
  Job  1: startTime   0 - response  99 - turnaround 102

  Avg  1: startTime n/a - response 49.50 - turnaround 101.50
```

Çözüm çok uzun olduğu için kodun sadece başını ve sonunu ekledim.

5. En üst kuyruğunda 10 ms bir kuantum uzunluğu olan bir sistemde, tek bir uzun süren (ve muhtemelen açlıktan ölen) işin CPU'nun en az %5'ini almasını garanti etmek için en yüksek öncelik düzeyine işleri ne sıklıkla teşvik etmeniz gerekir (-B bayrağı ile)?

Maksimum kuyruğu 10 ms olan bir sistemde, uzun süredir devam eden (ve potansiyel olarak aç olan) bir işin CPU'nun en az %5'ini almasını sağlamak için işler ne sıklıkla en yüksek öncelik seviyesine (-B) geri itilebilir?

Toplam N sürecin aynı anda geldiğini, ilk sürecin uzun soluklu (ve potansiyel olarak aç) bir süreç olduğunu ve geri kalan N-1 sürecin

İlk olarak, uzun süreçler 10 ms boyunca en yüksek öncelikli kuyrukta çalışır, ardından bir sonraki kuyruk seviyesine düşer
N-1 süreç en yüksek öncelikli kuyrukta her biri 10 ms çalışır (iş uzunluğu 10'dan azsa daha kısa) ve ardından bir sonraki kuyruk seviyesine düşer
N-1 <= 19, gereksinim en yüksek öncelik seviyesine geri itilmeden karşılanabilir
N-1 > 19, o zaman "kuyruğu atlamanız" ve her 20 ms'de bir BOOST yapmanız gerekir

```
emre@EMRE:~/Desktop/ostep/ostep-homework/cpu-sched-mlfq$ ./mlfq.py -n 2 -q 10 -l 0,40,0:0,20,0:0,20,0:0,20,0:0,20,0:0,20,0:0,20,0:0,20,0:0,20,
Here is the list of inputs:
OPTIONS jobs 20
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  40 - ioFreq   0
  Job  1: startTime   0 - runTime  20 - ioFreq   0
  Job  2: startTime   0 - runTime  20 - ioFreq   0
  Job  3: startTime   0 - runTime  20 - ioFreq   0
  Job  4: startTime   0 - runTime  20 - ioFreq   0
  Job  5: startTime   0 - runTime  20 - ioFreq   0
  Job  6: startTime   0 - runTime  20 - ioFreq   0
  Job  7: startTime   0 - runTime  20 - ioFreq   0
  Job  8: startTime   0 - runTime  20 - ioFreq   0
  Job  9: startTime   0 - runTime  20 - ioFreq   0
  Job 10: startTime   0 - runTime  20 - ioFreq   0
  Job 11: startTime   0 - runTime  20 - ioFreq   0
  Job 12: startTime   0 - runTime  20 - ioFreq   0
  Job 13: startTime   0 - runTime  20 - ioFreq   0
  Job 14: startTime   0 - runTime  20 - ioFreq   0
  Job 15: startTime   0 - runTime  20 - ioFreq   0
  Job 16: startTime   0 - runTime  20 - ioFreq   0
  Job 17: startTime   0 - runTime  20 - ioFreq   0
  Job 18: startTime   0 - runTime  20 - ioFreq   0
  Job 19: startTime   0 - runTime  20 - ioFreq   0
```

```
[ time 400 ] FINISHED JOB 19
[ time 400 ] Run JOB 0 at PRIORITY 0 [ TICKS 9 ALLOT 1 TIME 19 (of 40) ]
[ time 401 ] Run JOB 0 at PRIORITY 0 [ TICKS 8 ALLOT 1 TIME 18 (of 40) ]
[ time 402 ] Run JOB 0 at PRIORITY 0 [ TICKS 7 ALLOT 1 TIME 17 (of 40) ]
[ time 403 ] Run JOB 0 at PRIORITY 0 [ TICKS 6 ALLOT 1 TIME 16 (of 40) ]
[ time 404 ] Run JOB 0 at PRIORITY 0 [ TICKS 5 ALLOT 1 TIME 15 (of 40) ]
[ time 405 ] Run JOB 0 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 14 (of 40) ]
[ time 406 ] Run JOB 0 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 13 (of 40) ]
[ time 407 ] Run JOB 0 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 12 (of 40) ]
[ time 408 ] Run JOB 0 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 11 (of 40) ]
[ time 409 ] Run JOB 0 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 10 (of 40) ]
[ time 410 ] Run JOB 0 at PRIORITY 0 [ TICKS 9 ALLOT 1 TIME 9 (of 40) ]
[ time 411 ] Run JOB 0 at PRIORITY 0 [ TICKS 8 ALLOT 1 TIME 8 (of 40) ]
[ time 412 ] Run JOB 0 at PRIORITY 0 [ TICKS 7 ALLOT 1 TIME 7 (of 40) ]
[ time 413 ] Run JOB 0 at PRIORITY 0 [ TICKS 6 ALLOT 1 TIME 6 (of 40) ]
[ time 414 ] Run JOB 0 at PRIORITY 0 [ TICKS 5 ALLOT 1 TIME 5 (of 40) ]
[ time 415 ] Run JOB 0 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 4 (of 40) ]
[ time 416 ] Run JOB 0 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 3 (of 40) ]
[ time 417 ] Run JOB 0 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 2 (of 40) ]
[ time 418 ] Run JOB 0 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 1 (of 40) ]
[ time 419 ] Run JOB 0 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 0 (of 40) ]
[ time 420 ] FINISHED JOB 0

Final statistics:
  Job  0: startTime   0 - response   0 - turnaround 420
  Job  1: startTime   0 - response  10 - turnaround 220
  Job  2: startTime   0 - response  20 - turnaround 230
  Job  3: startTime   0 - response  30 - turnaround 240
  Job  4: startTime   0 - response  40 - turnaround 250
  Job  5: startTime   0 - response  50 - turnaround 260
  Job  6: startTime   0 - response  60 - turnaround 270
  Job  7: startTime   0 - response  70 - turnaround 280
  Job  8: startTime   0 - response  80 - turnaround 290
  Job  9: startTime   0 - response  90 - turnaround 300
  Job 10: startTime   0 - response 100 - turnaround 310
  Job 11: startTime   0 - response 110 - turnaround 320
  Job 12: startTime   0 - response 120 - turnaround 330
  Job 13: startTime   0 - response 130 - turnaround 340
  Job 14: startTime   0 - response 140 - turnaround 350
  Job 15: startTime   0 - response 150 - turnaround 360
  Job 16: startTime   0 - response 160 - turnaround 370
  Job 17: startTime   0 - response 170 - turnaround 380
  Job 18: startTime   0 - response 180 - turnaround 390
  Job 19: startTime   0 - response 190 - turnaround 400

 Avg 19: startTime n/a - response 95.00 - turnaround 315.50
```

Çözüm uzun olduğu için kodun sadece başını ve sonunu ekledim.

6. Planlama sırasında ortaya çıkan bir soru, I / O'yu tamamlayan bir işin kuyruğun hangi ucuna ekleneceğidir; -I bayrağı, bu planlama simülatörü için bu davranışı değiştirir. Bazı iş yükleriyle oynayın ve bu bayrağın etkisini görebileceğinizi görün.

Basit bir örnek olarak, ilk süreç bir IO talep eder ve bir zaman dilimi boyunca çalışan ve ardından önceliğini düşüren ikinci sürece geçer, ilk süreç IO'sunu tamamladığında ve önceliğini düşürdüğünde (ikinci süreçten sonra), sonuç İŞ 0 – İŞ 1 - İŞ 1 - İŞ 0 olur.

2 öncelik kuyruğu
Her öncelik kuyruğundaki her işlemin bir zaman dilimi kotası vardır
Zaman dilimi uzunluğu 5
IO süresi 5
İlk süreç (İŞ 0) 0 ms'de gelir, 10 ms boyunca çalışır ve her 5 ms'de bir IO talep eder
İkinci süreç (JOB 1) 0 ms'de gelir, 10 ms boyunca çalışır ve IO istekleri gerçekleştirmez

```
emre@EMRE:~/Desktop/ostep/ostep-homework/cpu-sched-mlfq$ ./mlfq.py -n 2 -q 5 -l 0,10,5:0,10,0 -c
Here is the list of inputs:
OPTIONS jobs 2
OPTIONS queues 2
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is   5
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is   5
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False


For each job, three defining characteristics are given:
  startTime : at what time does the job enter the system
  runTime   : the total CPU time needed by the job to finish
  ioFreq    : every ioFreq time units, the job issues an I/O
              (the I/O takes ioTime units to complete)

Job List:
  Job  0: startTime   0 - runTime  10 - ioFreq   5
  Job  1: startTime   0 - runTime  10 - ioFreq   0


Execution Trace:

[ time 0 ] JOB BEGINS by JOB 0
[ time 0 ] JOB BEGINS by JOB 1
[ time 0 ] Run JOB 0 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 9 (of 10) ]
[ time 1 ] Run JOB 0 at PRIORITY 1 [ TICKS 3 ALLOT 1 TIME 8 (of 10) ]
[ time 2 ] Run JOB 0 at PRIORITY 1 [ TICKS 2 ALLOT 1 TIME 7 (of 10) ]
[ time 3 ] Run JOB 0 at PRIORITY 1 [ TICKS 1 ALLOT 1 TIME 6 (of 10) ]
[ time 4 ] Run JOB 0 at PRIORITY 1 [ TICKS 0 ALLOT 1 TIME 5 (of 10) ]
[ time 5 ] IO_START by JOB 0
IO DONE
[ time 5 ] Run JOB 1 at PRIORITY 1 [ TICKS 4 ALLOT 1 TIME 9 (of 10) ]
[ time 6 ] Run JOB 1 at PRIORITY 1 [ TICKS 3 ALLOT 1 TIME 8 (of 10) ]
[ time 7 ] Run JOB 1 at PRIORITY 1 [ TICKS 2 ALLOT 1 TIME 7 (of 10) ]
[ time 8 ] Run JOB 1 at PRIORITY 1 [ TICKS 1 ALLOT 1 TIME 6 (of 10) ]
[ time 9 ] Run JOB 1 at PRIORITY 1 [ TICKS 0 ALLOT 1 TIME 5 (of 10) ]
[ time 10 ] IO_DONE by JOB 0
[ time 10 ] Run JOB 1 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 4 (of 10) ]
[ time 11 ] Run JOB 1 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 3 (of 10) ]
[ time 12 ] Run JOB 1 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 2 (of 10) ]
[ time 13 ] Run JOB 1 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 1 (of 10) ]
[ time 14 ] Run JOB 1 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 0 (of 10) ]
[ time 15 ] FINISHED JOB 1
[ time 15 ] Run JOB 0 at PRIORITY 0 [ TICKS 4 ALLOT 1 TIME 4 (of 10) ]
[ time 16 ] Run JOB 0 at PRIORITY 0 [ TICKS 3 ALLOT 1 TIME 3 (of 10) ]
[ time 17 ] Run JOB 0 at PRIORITY 0 [ TICKS 2 ALLOT 1 TIME 2 (of 10) ]
[ time 18 ] Run JOB 0 at PRIORITY 0 [ TICKS 1 ALLOT 1 TIME 1 (of 10) ]
[ time 19 ] Run JOB 0 at PRIORITY 0 [ TICKS 0 ALLOT 1 TIME 0 (of 10) ]
[ time 20 ] FINISHED JOB 0

Final statistics:
  Job  0: startTime   0 - response   0 - turnaround  20
  Job  1: startTime   0 - response   5 - turnaround  15

  Avg  1: startTime n/a - response 2.50 - turnaround 17.50
```

I komutu ile, IO'yu talep eden süreç IO'yu tamamladıktan sonra kuyruğun en üstüne taşınır ve öncelik verilir, bu nedenle İŞ 0 - İŞ 1 – İŞ 0 - İŞ 1 şeklinde sıralanır.