# 7

# Zamanlama: Giriş

Şimdiye kadar, süreçleri çalıştırmanın düşük seviyeli **mekanizmaları** (örneğin, bağlam değiştirme) açık olmalıdır; eğer değillerse, bir veya iki bölüm geriye gidin ve bu şeylerin nasıl çalıştığının açıklamasını tekrar okuyun. Bununla birlikte, bir işletim sistemi zamanlayıcısının kullandığı üst düzey **politikaları** henüz çözmedik. Şimdi tam da bunu yapacağız, çeşitli akıllı ve çalışkan insanların yıllar içinde geliştirdiği bir dizi **zamanlama politikası** (bazen **disiplinler** olarak adlandırılır) **(scheduling policies)** sunacağız.

Zamanlamanın kökenleri, aslında, bilgisayar sistemlerinden öncedir; Operasyon yönetimi alanından erken yaklaşımlar alınmış ve bilgisayarlara uygulanmıştır. Bu gerçeklik sürpriz olmamalı: montaj hatları ve diğer birçok insan çabası da zamanlama gerektiriyor ve lazer benzeri bir verimlilik arzusu da dahil olmak üzere aynı endişelerin birçoğu orada var. Ve böylece, sorunumuz:

PÜF NOKTASI: ZAMANLAMA POLİTİKASI NASIL GELİŞTİRİLİR

Politikaları zamanlama hakkında düşünmek için temel bir çerçeveyi nasıl geliştirmeliyiz? Temel varsayımlar nelerdir? Hangi metrikler önemlidir? En eski bilgisayar sistemlerinde hangi temel yaklaşımlar kullanılmıştır?

## 7.1 İş Yükü Varsayımları

Olası politikalar aralığına girmeden önce, sistemde çalışan süreçler hakkında, bazen toplu olarak **workload**(**iş yükü)** olarak adlandırılan bir dizi basitleştirici varsayımda bulunalım. İş yükünü belirlemek, ilkeler oluşturmanın kritik bir parçasıdır ve iş yükü hakkında ne kadar çok şey bilirseniz, ilkeniz o kadar ince ayarlanmış olabilir.

Burada yaptığımız iş yükü varsayımları çoğunlukla gerçekçi değildir, ancak bu sorun değil (şimdilik), çünkü ilerledikçe onları gevşeteceğiz ve hatta sonunda ... *(dramatik duraklama) ...*

**tam operasyonel bir zamanlama disiplini(fully-operational scheduling discipline**)[1] olarak adlandırdığımız şeyi geliştireceğiz.

Sistemde çalışan ve bazen **işler(jobs)** olarak adlandırılan süreçler hakkında aşağıdaki varsayımlarda bulunacağız :

1. Her iş aynı süre boyunca çalışır.
2. Tüm işler aynı anda ulaşır.
3. Bir kez başlatıldıktan sonra, her iş tamamlanana kadar çalışır.
4. Tüm işler yalnızca CPU'yu kullanır (yani, G/Ç yapmazlar)
5. Her işin çalışma zamanı bilinir.

Bu varsayımların çoğunun gerçekçi olmadığını söylemiştik, ancak Orwell'in *Hayvan Çiftliği'nde* [O45] bazı hayvanların diğerlerinden daha eşit olması gibi, bazı varsayımlar da bu bölümdeki diğerlerinden daha gerçekçi değildir. Özellikle, her işin çalışma zamanının bilinmesi sizi rahatsız edebilir: bu, zamanlayıcıyı her şeyi bilen hale getirir, ki bu da harika olsa da (muhtemelen) yakın zamanda gerçekleşmesi muhtemel değildir.

## 7.2 Zamanlama Metrikleri

İş yükü varsayımları yapmanın ötesinde, farklı zamanlama politikalarını karşılaştırmamızı sağlayacak bir şeye daha ihtiyacımız var: bir **zamanlama metriği (scheduling metric)**. Metrik, yalnızca bir şeyi ölçmek için kullandığımız bir şeydir ve planlamada anlamlı olan birkaç farklı metrik vardır.

Ancak şimdilik, sadece tek bir metriğe sahip olarak hayatımızı da basitleştirelim: **geri dönüş süresi (turnaround time)**. Bir işin geri dönüş süresi, işin tamamlandığı zaman eksi işin sisteme geldiği süre olarak tanımlanır. Daha resmi olarak, geri dönüş süresi $T_{geri\ dönüşü}$:

$$T_{geri\ dönüşü} = T_{tamamlama} - T_{geliş} \tag{7.1}$$

Çünkü şimdilik tüm işlerin aynı anda geldiğini varsaydık. $T_{geliş} = 0$ ve dolayısıyla $T_{geri\ dönüş} = T_{tamamlama}$. Bu gerçek yukarıda belirtilen varsayımları gevşettikçe değişecektir.

Geri dönüş süresinin, bu bölümde ana odak noktamız olacak bir performans metriği olduğunu unutmayın. Bir başka ilgi ölçütü, **Jain'in Adalet Endeksi** [J91] tarafından ölçüldüğü gibi (örneğin) **adalettir**. Performans ve adalet zamanlamada genellikle çelişkilidir; Bir zamanlayıcı, örneğin, performansı optimize edebilir, ancak birkaç işin çalışmasını engelleme pahasına, böylece adaleti azaltır. Bu bilmece bize hayatın her zaman mükemmel olmadığını gösteriyor.

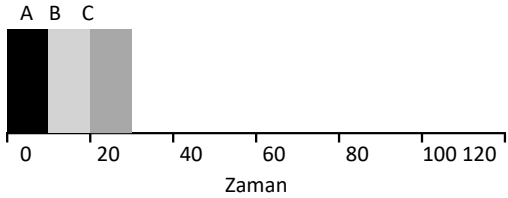
## 7.3 İlk Giren İlk Çıkar (FIFO)

Uygulayabileceğimiz en temel algoritma, **İlk Giren İlk Çıkar(FIFO)** zamanlaması veya bazen **İlk Gelen, İlk Alır** (**FCFS)** olarak bilinir.

[1] "Tam anlamıyla işleyen bir Ölüm Yıldızı" der gibi söylediniz.

FIFO'nun bir dizi olumlu özelliği vardır: açıkça basittir ve bu nedenle uygulanması kolaydır. Ve varsayımlarımız göz önüne alındığında, oldukça iyi çalışır.

Birlikte hızlı bir örnek yapalım. Sisteme üç işin geldiğini hayal edin, A, B ve C, kabaca aynı anda ($T_{geliş}$ = 0). FIFO'nun önce bir işi koyması gerektiğinden, hepsi aynı anda gelirken, A'nın B'den sadece bir kıl önce geldiğini ve B'nin C'den sadece bir kıl önce geldiğini varsayalım. Bu işler için **ortalama geri dönüş süresi (average turnaround time**) ne olacak?



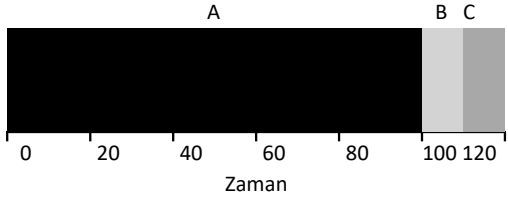

Şekil 7.1: **FIFO Basit Örneği**

Şekil 7.1'den, A 10, B 20, ve C de 30 da bittiğini görebilirsiniz.Böylece 3 işin ortalama Dönüş Süresi basitçe (10+20+30) /3= 20 olur. Geri dönüş süresini hesaplamak da bu kadar kolaydır.

Şimdi varsayımlarımızdan birini bırakalım. Özellikle, varsayım 1'i bırakalım ve böylece artık her işin aynı süre boyunca çalıştığını varsaymayalım. FIFO şimdi nasıl performans gösteriyor? FIFO'nun kötü performans göstermesi için ne tür bir iş yükü oluşturabilirsiniz?

*(Okumaya başlamadan önce bunu düşünün ... Düşünmeye devam et ... anladın mı?!)*

Muhtemelen bunu şimdiye kadar çözdünüz, ancak her ihtimale karşı, farklı uzunluklardaki işlerin FIFO planlaması için nasıl sorunlara yol açabileceğini göstermek için bir örnek yapalım. Özellikle, yine üç iş (A, B ve C) varsayalım, ancak bu sefer A 100 saniye boyunca çalışırken, B ve C her biri 10 saniye boyunca çalışır.




Şekil 7.2: **FIFO Neden O Kadar Harika Değil**

Şekil 7.2 de görebileceğiniz üzere, iş A, B veya C çalışma şansı bile bulmadan önce ilk 100 saniyede çalışır. Böylece, ortalama geri dönüş süresi sistem için yüksektir ve ağrılı 110 Saniyedir. ($\frac{100+110+120}{3}$ = 110).

Bu sorun genellikle **konvoy etkisi (convoy effect)** [B+79] olarak adlandırılır, burada Bir kaynağın göreceli olarak kısa süreli potansiyel tüketicileri sıraya alınır

İPUCU: SJF PRENSİBİ

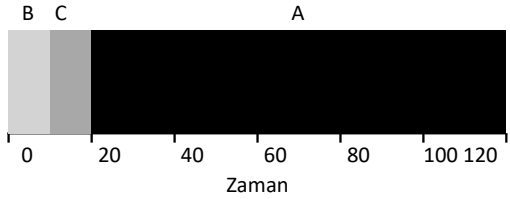
Önce En Kısa İş, müşteri başına algılanan geri dönüş süresinin (veya bizim durumumuzda bir işin) önemli olduğu herhangi bir sisteme uygulanabilecek genel bir zamanlama ilkesini temsil eder. Beklediğiniz herhangi bir sırayı düşünün: söz konusu kuruluş müşteri memnuniyetini önemsiyorsa, SJF'yi dikkate almış olmaları muhtemeldir. Örneğin, marketler, satın almak için yalnızca birkaç şeyi olan müşterilerin, yaklaşmakta olan bir nükleer kışa hazırlanan ailenin arkasında sıkışıp kalmamalarını sağlamak için genellikle "on ürün veya daha az" satırına sahiptir.

ağır bir kaynak tüketicisinin arkasında. Bu zamanlama senaryosu size bir markette tek bir sırayı ve karşınızdaki kişiyi erzak dolu üç sepet ve bir çek defteri ile gördüğünüzde nasıl hissettiğinizi hatırlatabilir; biraz zaman alacak [2].

Peki ne yapmalıyız? Farklı süreler boyunca çalışan yeni iş gerçekliğimizle başa çıkmak için nasıl daha iyi bir algoritma geliştirebiliriz? Önce bir düşünün; sonra okumaya devam edin.

## 7.4 Önce En Kısa İş (SJF)

Çok basit bir yaklaşımın bu sorunu çözdüğü ortaya çıktı; aslında yöneylem araştırmasından [C54, PV56] çalınan ve bilgisayar sistemlerindeki işlerin programlanmasına uygulanan bir fikirdir. Bu yeni zamanlama disiplini, **Önce En Kısa İş (SJF**) olarak bilinir ve adın hatırlanması kolay olmalıdır, çünkü politikayı tamamen tanımlar: önce en kısa işi , sonra bir sonraki en kısa işi çalıştırır, vb.




Şekil 7.3: **SJF Basit Örnek**

Yukarıdaki örneğimizi alalım ama SJF'yi zamanlama politikası olarak alalım. Şekil 7.3 A,B ve C'nin çalışma sürelerinin sonucunu gösterir. Umarız ki diyagram SJF'nin ortalama geri dönüş sürelerine göre neden daha iyi performans gösterdiğini gösterir.Basitçe B ve C'yi A'dan önce çalıştırarak SJF ortalama geri döüş süresini 110 saniyeden 50 saniyeye düşürür. ($\frac{10+20+120}{3} = 50$), iki katından daha fazla iyileştirme faktörü.

[2] Bu durumda önerilen eylem: ya hızlı bir şekilde farklı bir çizgiye geçin ya da uzun, derin ve rahatlatıcı bir nefes alın . Bu doğru**,** nefes alın, nefes verin. Tamam olacak, endişelenmeyin.

AYRI TARAF: KESİNTİLİ ZAMANLAMA

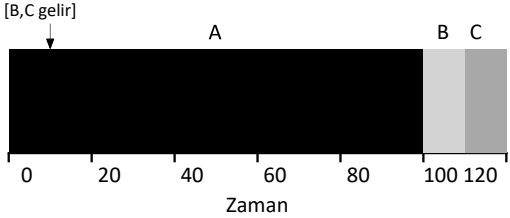
Toplu işlemin eski günlerinde, bir dizi **kesintili olmayan(non-preemptive)** zamanlayıcılar geliştirildi; Bu tür sistemler, yeni bir işin çalıştırılıp çalıştırılmayacağını düşünmeden önce her işi tamamlanana kadar çalıştırırdı. Hemen hemen tüm modern zamanlayıcılar **kesintilidir(preemptive)** ve bir süreci çalıştırmak için diğerinin çalışmasını durdurmaya oldukça isteklidir. Bu, zamanlayıcının daha önce öğrendiğimiz mekanizmaları kullandığı anlamına gelir; Özellikle, zamanlayıcı bir **bağlam anahtarı (context switch)** gerçekleştirebilir, çalışan bir prosesi geçici olarak durdurabilir ve diğerini devam ettirebilir (veya başlatabilir).

Aslında, aynı anda gelen işlerle ilgili varsayımlarımız göz önüne alındığında, SJF'nin gerçekten de optimal bir zamanlama algoritması olduğunu kanıtlayabiliriz. Ancak teori veya yöneylem araştırması değil, bir sistem sınıfındasınız; hiçbir kanıta izin verilmez.

Böylece SJF ile zamanlama konusunda iyi bir yaklaşıma ulaşıyoruz, ancak varsayımlarımız hala oldukça gerçekçi değil. Bir başkasını bırakalım. Özellikle varsayım 2'yi hedefleyebiliriz ve şimdi işlerin bir kerede değil, herhangi bir zamanda gelebileceğini varsayabiliriz. Bu ne gibi sorunlara yol açar?

*(Düşünmek için başka bir duraklama ... düşünüyor musun? Hadi, yapabilirsin)*

Burada sorunu bir örnekle tekrar örneklendirebiliriz. Bu kez, A'nın t = 0'da ulaştığını ve 100 saniye boyunca çalışması gerektiğini, B ve C'nin ise *t* =10'da ulaştığını ve her birinin 10 saniye boyunca çalışması gerektiğini varsayalım. Saf SJF ile, Şekil 7.4'te görülen programı elde ederiz.



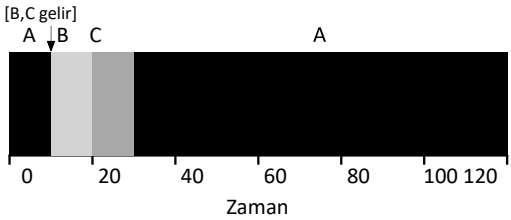

Şekil 7.4: **B ve C'den Geç Gelenlerle SJF**

Şekilden de görüleceği üzere B ve C A'dan kısa süre sonra ulaşmasına rağmen, A tamamlanana kadar beklemeye zorlanırlar. Bu nedenle aynı konvoy problemi sıkıntısı olur. Bu 3 iş için ortalama geri dönüş süresi 103.33 Saniyedir ( $\frac{100+(110-10)+(120-10)}{3}$ ). Bir zamanlayıcı ne yapar?

## 7.5 Önce En Kısa Sürelini Tamamlanması (STCF)

Bu endişeyi gidermek için, varsayım 3'ü (işlerin tamamlanması için çalışması gerektiğini) bırakmamız gerekiyor, bu yüzden bunu yapalım. Ayrıca zamanlayıcının kendisinde bazı makinelere ihtiyacımız var. Tahmin edebileceğiniz gibi, zamanlayıcı kesintileri ve içerik anahtarlama

hakkındaki önceki tartışmalarımız göz önüne alındığında, B ve C ulaştığında kesinlikle başka bir şeyler yapabilir.




Şekil 7.5: **STCF Basit Örnek**

A işini önleyebilir ve başka bir işi yürütmeye karar verebilir, belki de daha sonra A'ya devam edebilir. Bizim tanımlamamızla SJF, **kesintili olmayan(non-preemptive)** bir zamanlayıcıdır ve bu nedenle yukarıda açıklanan sorunlardan muzdariptir.

Neyse ki, tam olarak bunu yapan bir zamanlayıcı var: En Kısa Tamamlanma Süresi İlk (**STCF**) veya **İlk Önce Kesintili En Kısa İş** ( **PSJF** ) zamanlayıcısı [CK68] olarak da bilinen SJF'ye kesinti ekler. Sisteme her yeni iş girdiğinde, STCF zamanlayıcısı yeniden ana işlerden hangisinin (yeni iş dahil) en az zamana sahip olduğunu belirler ve bunu zamanlar. Bu nedenle, örneğimizde, STCF A'yı önleyecek ve B ve C'yi tamamlamak için çalıştıracaktır; Sadece bittiğinde A'nın kalan süresi planlanacaktır. Şekil 7.5'te bir örnek gösterilmektedir.

Sonuç, çok daha iyi bir ortalama geri dönüş süresidir: 50 saniye ( $\frac{(120-0)+(20-10)+(30-10)}{3}$ ). Ve eskisi gibi verilen yeni varsayımlarımız STCF kanıtlanabilir şekilde optimaldir. Bütün işler aynı anda ulaşırsa Verilen bu SJF optimaldir, muhtemelen STCF optimalliğinin arkasındaki önseziyi görebilmeniz gerekir.

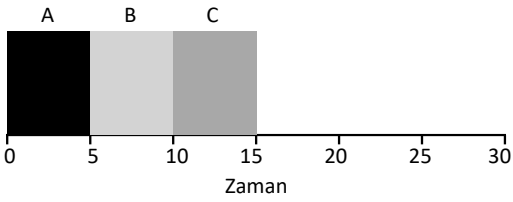
## 7.6 Yeni Bir Metrik: Yanıt Süresi

Bu nedenle, iş uzunluklarını bilseydik ve işlerin yalnızca CPU'yu kullandığını ve tek metriğimizin geri dönüş süresi olduğunu bilseydik , STCF harika bir politika olurdu. Aslında, bir dizi erken toplu hesaplama sistemi için, bu tür zamanlama algoritmaları bir anlam ifade ediyordu. Ancak, zaman paylaşımlı makinelerin piyasaya sürülmesi tüm bunları değiştirdi. Artık kullanıcılar bir terminalde oturacak ve etkileşimli performansı sistemden de alacaklardı. Ve böylece, yeni bir metrik doğdu: **tepki süresi (response time)**.

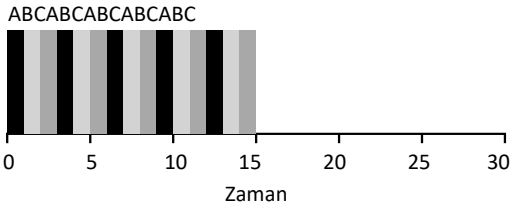
Yanıt süresini, işin bir sisteme ulaştığı andan itibaren sistem ilk kez programlandığı zamana kadar geçen süre olarak tanımlıyoruz. [3]. Daha resmi olarak:

$$T_{yanıt} = T_{ilkçalışma} - T_{ulaşma} \tag{7.2}$$

[3] Bazıları bunu biraz farklı tanımlar, örneğin, işin bir tür "yanıt" üretene kadar geçen süreyi de dahil etmek; tanımımız, esasen işin anında bir yanıt ürettiğini varsayarsak, bunun en iyi versiyonudur.




Şekil 7.6: **SJF Tekrar (Yanıt Süresi Açısından Kötü )**




Şekil 7.7: **Round Robin (Yanıt Süresi İçin İyi)**

Örneğin, Şekil 7.5'teki programa sahip olsaydık (A 0 zamanında ve B ve C 10 zamanında geldi), her işin yanıt süresi aşağıdaki gibidir:A işi için 0, B için 0 ve C için 10 (ortalama: 3.33).

Düşündüğünüz gibi, STCF ve ilgili disiplinler yanıt süresi için kısmen iyi değildir. Örneğin, üç iş aynı anda gelirse, üçüncü işin yalnızca bir kez zamanlanmadan önce önceki iki işin *tamamen* çalışmasını beklemesi gerekir. Geri dönüş süresi için harika olsa da, bu yaklaşım yanıt süresi ve etkileşim için oldukça kötüdür. Gerçekte, bir terminalde oturduğunuzu, yazdığınızı ve sistemden bir yanıt görmek için 10 saniye beklemek zorunda kaldığınızı hayal edin, çünkü başka bir iş önünüzde planlandı: çok hoş değil.

Bu nedenle, başka bir sorunla karşı karşıyayız: yanıt süresine duyarlı bir zamanlayıcıyı nasıl oluşturabiliriz?

## 7.7 Round Robin

Bu sorunu çözmek için, klasik olarak **Round-Robin (RR)** zamanlaması [K64] olarak adlandırılan yeni bir zamanlama algoritması sunacağız. Temel fikir basittir: RR, işleri tamamlanana kadar çalıştırmak yerine, bir işi bir **zaman dilimi** (bazen **zamanlama kuantumu(scheduling quantum)** olarak adlandırılır) için çalıştırır ve ardından çalıştırma kuyruğundaki bir sonraki işe geçer. İşler bitene kadar bunu tekrar tekrar yapar. Bu nedenle, RR bazen **zaman dilimleme(time-slicing)** olarak adlandırılır. Bir zaman diliminin uzunluğunun, zamanlayıcı-kesme süresinin bir katı olması gerektiğini unutmayın; bu nedenle, zamanlayıcı her 10 milisaniyede bir kesilirse, zaman dilimi 10, 20 veya 10 ms'nin başka bir katı olabilir .

RR'yi daha ayrıntılı olarak anlamak için bir örneğe bakalım. A, B ve C olmak üzere üç işin sisteme aynı anda geldiğini ve

İPUCU: AMORTİSMAN MALİYETİ DÜŞÜREBİLİR

Genel Bakış **amortisman(amortization)** tekniği, bazı operasyonlar için sabit bir maliyet olduğunda sistemlerde yaygın olarak kullanılır. Bu maliyete daha az sıklıkta maruz kalarak (yani, işlemi daha az kez gerçekleştirerek), sisteme toplam maliyet azalır. Örneğin, zaman dilimi 10 ms'ye ayarlanırsa ve bağlam değiştirme maliyeti 1 ms,zamanın %10'u bağlam değiştirmek için harcanır ve bu nedenle boşa harcanır. Bu maliyeti amortismana tabi tutmak istiyorsak, zaman dilimini örneğin 100 ms'ye yükseltebiliriz. bu durumda, zamanın% 1'inden azı bağlam değiştirmek için harcanır ve böylece zaman dilimleme maliyeti *amortismana* tabi tutulur.

her biri 5 saniye çalışmak istiyor. SJF zamanlayıcısı, her işi başka bir işi çalıştırmadan önce tamamlanana kadar çalıştırır (Şekil 7.6). Buna karşılık, 1 saniye zaman dilimine sahip RR, işler arasında hızlı bir şekilde geçiş yapacaktır (Şekil 7.7).

Burada RR'nin ortalama yanıt süresi: $\frac{0+1+2}{3}$ = 1; SJF için, ortalama yanıt süresi: $\frac{0+5+10}{3}$ = 5.

Gördüğünüz gibi, zaman diliminin uzunluğu RR için kritik öneme sahiptir. Ne kadar kısa olursa, yanıt süresi metriği altındaki RR performansı o kadar iyi olur. Bununla birlikte, zaman dilimini çok kısa yapmak sorunludur: aniden bağlam değiştirmenin maliyeti genel performansa hakim olacaktır. Bu nedenle, zaman diliminin uzunluğuna karar vermek, bir sistem imzalayana bir takas sunar ve bunu yapmadan anahtarlama maliyetini **amorti** etmek için yeterince uzun hale getirir.O kadar uzun süre ki sistem artık yanıt vermiyor.

Bağlam değiştirme maliyetinin yalnızca işletim sisteminin birkaç register kaydetme ve değiştirme eylemlerinden kaynaklanmadığını unutmayın. Programlar çalıştığında, CPU önbelleklerinde, TLB'lerde, dal tahmincilerinde ve diğer yonga üstü donanımlarda büyük miktarda durum oluştururlar. Başka bir işe geçmek, bu durumun temizlenmesine ve şu anda çalışmakta olan işle ilgili yeni durumun getirilmesine neden olur, bu da önemli bir performans maliyeti [MB91] belirleyebilir.

Makul bir zaman dilimine sahip RR, bu nedenle yanıt süresi tek ölçütümüzse mükemmel bir planlayıcıdır. Peki ya eski arkadaşımız geri dönüş zamanı? Yukarıdaki örneğimize tekrar bakalım. Her biri 5 saniyelik çalışma sürelerine sahip A, B ve C aynı anda gelir ve RR 1 saniyelik(uzun) bir zaman dilimine sahip zamanlayıcıdır. Yukarıdaki resimden A'nın 13'te, B'nin 14'te ve C'nin t 15'te ortalama 14'te bittiğini görebiliriz. Oldukça korkunç!

Öyleyse, geri dönüş süresi metriğimizse, RR'nin gerçekten *de en kötü* politikalardan biri olması şaşırtıcı değildir. Sezgisel olarak, bu mantıklı olmalıdır: RR'nin yaptığı şey, bir sonrakine geçmeden önce her işi yalnızca kısa bir süre çalıştırarak, her işi olabildiğince uzatmaktır. Geri dönüş süresi sadece işler bittiğinde umursadığından, RR neredeyse kötümserdir, çoğu durumda yalın FIFO'dan bile daha kötüdür.

Daha genel olarak, adil olan, yani CPU'yu aktif işlemler arasında küçük bir zaman ölçeğinde eşit olarak bölen herhangi bir politika (RR gibi ), geri dönüş süresi gibi metriklerde kötü performans gösterecektir. Aslında, bu doğal bir ödünleşimdir: haksız olmaya istekliyseniz,

tamamlanması için daha kısa işler çalıştırabilirsiniz, ancak yanıt süresi pahasına; bunun yerine adalete değer veriyorsanız,

İPUCU: ÖRTÜŞME DAHA YÜKSEK KULLANIM SAĞLAR

Mümkün olduğunda, **overlap**(örtüşme)sistemlerin kullanımını en üst düzeye çıkarır. Örtüşme, disk G/Ç'yi oluştururken veya uzak makinelere mesaj gönderirken de dahil olmak üzere birçok farklı alanda yararlıdır; Her iki durumda da, işlemi başlatmak ve ardından diğer işlere geçmek iyi bir fikirdir ve sistemin genel kullanımını ve verimliliğini artırır.

yanıt süresi düşürülür, ancak geri dönüş süresi pahasına. Bu tür bir **değiş tokuş(trade-off)** sistemlerde yaygındır; pastanı hem saklayıp hem yiyemezsin [4].

İki tür zamanlayıcı geliştirdik. İlk tür (SJF, STCF) geri dönüş süresini optimize eder, ancak yanıt süresi açısından kötüdür. İkinci tür (RR) yanıt süresini optimize eder, ancak geri dönüş için kötüdür. Ve hala gevşetilmesi gereken iki varsayımımız var: varsayım 4 (işlerin G/Ç yapmadığı) ve varsayım 5 ( her işin çalışma zamanının bilindiği).
Şimdi bu varsayımları ele alalım.

## 7.8 G/Ç Ekleme

İlk önce varsayım 4'ü bırakacağız- elbette tüm programlar G/Ç gerçekleştirir. Herhangi bir girdi almayan bir program düşünün: her seferinde aynı çıktıyı üretecektir. Çıktısı olmayan birini hayal edin: atasözündeki ormanda düşen ağaç gibidir. Onu görecek kimse yoktur; çalışmış olması önemli değil.

Bir iş bir G/Ç isteği başlattığında zamanlayıcının açıkça vermesi gereken bir karar vardır, çünkü şu anda çalışan iş G /Ç sırasında CPU'yu kullanmaz; G/Ç'nin tamamlanması beklenirken engellenir. G/Ç bir sabit disk sürücüsüne gönderilirse, sürücünün geçerli G/Ç yüküne bağlı olarak işlem birkaç milisaniye veya daha uzun süre engellenebilir. Böylece, zamanlayıcı muhtemelen o sırada CPU'da başka bir iş planlamalıdır.

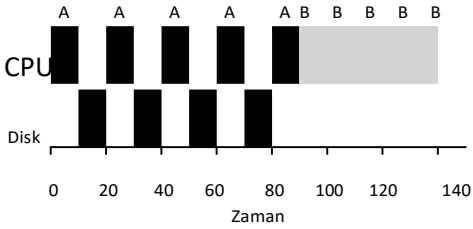
Zamanlayıcı ayrıca G/Ç tamamlandığında bir karar vermek zorundadır. Bu gerçekleştiğinde, bir kesme oluşturulur ve işletim sistemi çalışır ve G/Ç'yi veren işlemi engellenmiş durumdan hazır duruma geri taşır. Tabii ki, bu noktada işi yürütmeye bile karar verebilir. İşletim sistemi her işi nasıl ele almalıdır?

Bu sorunu daha iyi anlamak için, her biri 50 ms CPU süresine ihtiyaç duyan A ve B olmak üzere iki işimiz olduğunu varsayalım. Bununla birlikte, bariz bir fark vardır: A 10 ms boyunca çalışır ve daha sonra bir G/Ç isteği yayınlar (burada G/Ç'lerin her birinin 10 ms aldığını varsayalım), oysa B sadece CPU'yu 50 ms için kullanır ve G/Ç gerçekleştirmez. Zamanlayıcı önce A'yı, sonra B'yi çalıştırır (Şekil 7.8).

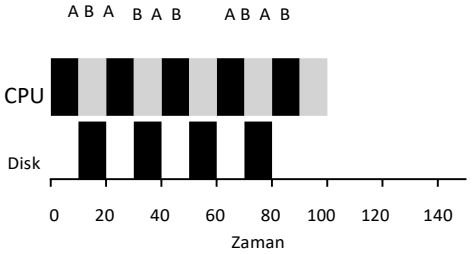
Bir STCF zamanlayıcısı oluşturmaya çalıştığımızı varsayalım. Böyle bir zamanlayıcı, A'nın 5 adet 10 ms'lik alt işe ayrıldığı gerçeğini nasıl açıklamalıdır,

[4] İnsanların İnsanların kafasını karıştıran bir söz, çünkü "Pastanı hem saklayıp hem de

yiyemezsin" olmalıdır(ki bu çok açıktır, hayır?). Şaşırtıcı bir şekilde, bu sözle ilgili bir Wikipedia sayfası var; daha da şaşırtıcı bir şekilde, [W15]okumak biraz eğlencelidir. İtalyanca'da dedikleri gibi, Avere la *botte piena e la moglie ubriaca* yapamazsınız.




Şekil 7.8: **Kaynakların Kötü Kullanımı**




Şekil 7.9: **Örtüşme Kaynakların Daha İyi Kullanılmasına İzin Veriyor**

B ise sadece tek bir 50 ms'lik CPU talep mi eder? Açıkçası, G/Ç'nin nasıl dikkate alınacağını düşünmeden sadece bir işi ve ardından diğerini çalıştırmak çok az anlam ifade ediyor.

Yaygın bir yaklaşım, A'nın her 10 ms'lik alt işini bağımsız bir iş olarak ele almaktır. Bu nedenle, sistem başladığında, seçimi 10 ms'lik bir A mı yoksa 50 ms'lik bir B mi zamanlanacağıdır. STCF ile seçim açıktır: daha kısa olanı seçin, bu durumda A. Ardından, A'nın ilk alt işi tamamlandığında, yalnızca B kalır ve çalışmaya başlar. Daha sonra A'nın yeni bir alt işi gönderilir ve B'nin önüne geçer ve 10 ms boyunca çalışır. Bunu yapmak, başka bir işlemin G/Ç'sinin tamamlanmasını beklerken CPU'nun bir işlem tarafından kullanılmasıyla **örtüşme(overlap)** izin verir; böylece sistem daha iyi kullanılır (bkz. Şekil 7.9).

Ve böylece bir zamanlayıcının G/Ç'yi nasıl kapsayabileceğini görüyoruz. Zamanlayıcı, her CPU ani artışını bir iş olarak ele alarak, "etkileşimli" olan işlemlerin sık sık çalıştırılmasını sağlar. Bu etkileşimli işler G/Ç gerçekleştirirken, CPU yoğun diğer işler çalıştırır, böylece işlemciyi daha iyi kullanır.

## 7.9 Artık Kehanet Yok

G/Ç'ye temel bir yaklaşımla, nihai varsayımımıza geliyoruz: zamanlayıcının her işin uzunluğunu bildiği. Daha önce de söylediğimiz gibi, bu muhtemelen yapabileceğimiz en kötü varsayımdır. Aslında, genel amaçlı bir işletim sisteminde (önemsediklerimiz gibi), işletim sistemi genellikle her işin uzunluğu hakkında çok az şey bilir. Dolayısıyla, böyle bir

*Öncül* bilgi olmadan SJF/STCF gibi davranan bir yaklaşımı nasıl inşa edebiliriz? Ayrıca, RR zamanlayıcı ile gördüğümüz bazı fikirleri nasıl birleştirebiliriz, böylece yanıt süresi de oldukça iyidir?

## 7.10 Özet

Zamanlamanın arkasındaki temel fikirleri tanıttık ve iki yaklaşım ailesi geliştirdik. Birincisi, kalan en kısa işi çalıştırır ve böylece geri dönüş süresini optimize eder; ikincisi tüm işler arasında geçiş yapar ve böylece yanıt süresini optimize eder. Her ikisi de diğerinin iyi olduğu yerde kötüdür, ne yazık ki, sistemlerde yaygın olan doğal bir takas. G/Ç'yi resme nasıl dahil edebileceğimizi de gördük, ancak işletim sisteminin geleceği görememesinin temel yetersizliği sorununu hala çözemedik . Kısaca, geleceği tahmin etmek için yakın geçmişi kullanan bir zamanlayıcı oluşturarak bu sorunun üstesinden nasıl gelineceğini göreceğiz. Bu zamanlayıcı **çok düzeyli geri bildirim kuyruğu (**multi-level feedback queue) olarak bilinir ve bir sonraki bölümün konusudur.

## Referanslar

[B+79] “The Convoy Phenomenon” by M. Blasgen, J. Gray, M. Mitoma, T. Price. ACM Operating Systems Review, 13:2, April 1979. Perhaps the first reference to convoys, which occurs in databases as well as the OS.

[C54] “Priority Assignment in Waiting Line Problems” by A. Cobham. Journal of Operations Research, 2:70, pages 70–76, 1954. The pioneering paper on using an SJF approach in scheduling the repair of machines.

[K64] “Analysis of a Time-Shared Processor” by Leonard Kleinrock. Naval Research Logistics Quarterly, 11:1, pages 59–73, March 1964. May be the first reference to the round-robin scheduling algorithm; certainly one of the first analyses of said approach to scheduling a time-shared system.

[CK68] “Computer Scheduling Methods and their Countermeasures” by Edward G. Coffman and Leonard Kleinrock. AFIPS ’68 (Spring), April 1968. An excellent early introduction to and analysis of a number of basic scheduling disciplines.

[J91] “The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling” by R. Jain. Interscience, New York, April 1991. The standard text on computer systems measurement. A great reference for your library, for sure.

[O45] “Animal Farm” by George Orwell. Secker and Warburg (London), 1945. A great but depressing allegorical book about power and its corruptions. Some say it is a critique of Stalin and the pre-WWII Stalin era in the U.S.S.R; we say it’s a critique of pigs.

[PV56] “Machine Repair as a Priority Waiting-Line Problem” by Thomas E. Phipps Jr., W. R. Van Voorhis. Operations Research, 4:1, pages 76–86, February 1956. Follow-on work that generalizes the SJF approach to machine repair from Cobham’s original work; also postulates the utility of an STCF approach in such an environment. Specifically, “There are certain types of repair work, ... involving much dismantling and covering the floor with nuts and bolts, which certainly should not be interrupted once undertaken; in other cases it would be inadvisable to continue work on a long job if one or more short ones became available (p.81).”

[MB91] “The effect of context switches on cache performance” by Jeffrey C. Mogul, Anita Borg. ASPLOS, 1991. A nice study on how cache performance can be affected by context switching; less of an issue in today’s systems where processors issue billions of instructions per second but context-switches still happen in the millisecond time range.

[W15] “You can’t have your cake and eat it” by Authors: Unknown.. Wikipedia (as of December 2015). http://en.wikipedia.org/wiki/You can’t have your cake and eat it. The best part of this page is reading all the similar idioms from other languages. In Tamil, you can’t “have both the moustache and drink the soup.”

## Ödev (Simülasyon)

Bu program scheduler.py, yanıt süresi, geri dönüş süresi ve toplam bekleme süresi gibi zamanlama metrikleri altında farklı zamanlayıcıların nasıl performans gösterdiğini görmenizi sağlar. Ayrıntılar için README'ye bakın.

### Sorular

1. SJF ve FIFO zamanlayıcılarıyla 200 uzunluğundaki üç işi çalıştırırken yanıt süresini ve geri dönüş süresini hesaplayın.

   Öncelikle SJF zamanlayıcısıyla çalıştırdığımızda sonuç aşağıdaki gibi olur. En kısa işin önce bittiği ve yanıt süresinin 200 ms geri dönüş süresinin 400 ms olduğunu görüyoruz.

```
** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
  [ time 200 ] Run job 1 for 200.00 secs ( DONE at 400.00 )
  [ time 400 ] Run job 2 for 200.00 secs ( DONE at 600.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
  Job   1 -- Response: 200.00  Turnaround 400.00  Wait 200.00
  Job   2 -- Response: 400.00  Turnaround 600.00  Wait 400.00

  Average -- Response: 200.00  Turnaround 400.00  Wait 200.00
```

   FIFO ile çalıştırdığımızda işlerin sırayla çalıştırılıp sırasına göre tamamlandığını görüyoruz.

```
** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
  [ time 200 ] Run job 1 for 200.00 secs ( DONE at 400.00 )
  [ time 400 ] Run job 2 for 200.00 secs ( DONE at 600.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
  Job   1 -- Response: 200.00  Turnaround 400.00  Wait 200.00
  Job   2 -- Response: 400.00  Turnaround 600.00  Wait 400.00

  Average -- Response: 200.00  Turnaround 400.00  Wait 200.00
```

2. Şimdi aynısını yapın, ancak farklı uzunluklardaki işlerle: 100, 200 ve 300.

```
** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
  [ time 100 ] Run job 1 for 200.00 secs ( DONE at 300.00 )
  [ time 300 ] Run job 2 for 300.00 secs ( DONE at 600.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
  Job   1 -- Response: 100.00  Turnaround 300.00  Wait 100.00
  Job   2 -- Response: 300.00  Turnaround 600.00  Wait 300.00

  Average -- Response: 133.33  Turnaround 333.33  Wait 133.33
```

Öncelikle SJF ile çalıştırdığımızda yanıt sürelerinin 0, 100, 300 ve geri dönüş sürelerinin 100,300, 600 olduğunu görüyoruz. Ortalama yanıt süresinin 133.33 geri dönüş süresinin ise 333.33 olduğunu görüyoruz. Daha sonra FIFO ile çalıştırdığımızda da yine yanıt ve geri dönüş sürelerinin aynı olduğunu görüyoruz.

```
** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
  [ time 100 ] Run job 1 for 200.00 secs ( DONE at 300.00 )
  [ time 300 ] Run job 2 for 300.00 secs ( DONE at 600.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
  Job   1 -- Response: 100.00  Turnaround 300.00  Wait 100.00
  Job   2 -- Response: 300.00  Turnaround 600.00  Wait 300.00

  Average -- Response: 133.33  Turnaround 333.33  Wait 133.33
```

3. Şimdi aynısını yapın, ancak RR zamanlayıcısı ve 1'lik bir zaman dilimi ile de yapın.
RR zamanlayıcısıyla çalıştırdığımızda geri dönüş süresinin 465.67 yanıt süresinin 1 olduğunu görüyoruz

```
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 298.00  Wait 198.00
  Job   1 -- Response: 1.00  Turnaround 499.00  Wait 299.00
  Job   2 -- Response: 2.00  Turnaround 600.00  Wait 300.00

  Average -- Response: 1.00  Turnaround 465.67  Wait 265.67
```

4. SJF, ne tür iş yükleri için FIFO ile aynı geri dönüş sürelerini sunar?

   İşlerin süreleri giriş sıraların göre sıralanırsa geri dönüş süreleri aynı olur.

5. SJF ne tür iş yükleri ve kuantum uzunlukları için RR ile aynı yanıt sürelerini sunar?
   İşlerin sürelerinin eşit olması ve quantum uzunluklarının da eşit olması gerekir.

6. İş süreleri arttıkça SJF ile yanıt süresine ne olur? Simülatörü trendi göstermek için kullanabilir misiniz?
   Yanıt süresi iş süreleri arttıkça artacaktır. Örneğin scheduler.py örneğinde iş sürelerini sırasıyla 100, 200, 300 yaparsak aşağıdaki yanıt sürelerini alırız.İş süresi 100 olduğunda;

```
** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
  [ time 100 ] Run job 1 for 100.00 secs ( DONE at 200.00 )
  [ time 200 ] Run job 2 for 100.00 secs ( DONE at 300.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
  Job   1 -- Response: 100.00  Turnaround 200.00  Wait 100.00
  Job   2 -- Response: 200.00  Turnaround 300.00  Wait 200.00

  Average -- Response: 100.00  Turnaround 200.00  Wait 100.00
```

İş süresi 200 olduğunda;

```
** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
  [ time 200 ] Run job 1 for 200.00 secs ( DONE at 400.00 )
  [ time 400 ] Run job 2 for 200.00 secs ( DONE at 600.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
  Job   1 -- Response: 200.00  Turnaround 400.00  Wait 200.00
  Job   2 -- Response: 400.00  Turnaround 600.00  Wait 400.00

  Average -- Response: 200.00  Turnaround 400.00  Wait 200.00
```

İş süresi 300 olduğunda;

```
** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 300.00 secs ( DONE at 300.00 )
  [ time 300 ] Run job 1 for 300.00 secs ( DONE at 600.00 )
  [ time 600 ] Run job 2 for 300.00 secs ( DONE at 900.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 300.00  Wait 0.00
  Job   1 -- Response: 300.00  Turnaround 600.00  Wait 300.00
  Job   2 -- Response: 600.00  Turnaround 900.00  Wait 600.00

  Average -- Response: 300.00  Turnaround 600.00  Wait 300.00
```

7. Kuantum uzunlukları arttıkça RR ile tepki süresine ne olur? *N* işleri göz önüne alındığında, en kötü durum yanıt zamanını veren bir denklem yazabilir misiniz?
Yanıt süresi artacaktır. N'inci işin yanıt süresi (N-1)*q ve ortalama yanıt süresi (N-1)*q/2