7

Scheduling: Introduction (Planlama: Giriş)

Şimdiye kadar çalışan süreçlerin düşük seviyeli **mekanizmaları (mechanisms)** (örneğin, bağlam değiştirme) açık olmalıdır; değillerse, bir veya iki bölüm geriye gidin ve bu şeylerin nasıl çalıştığına ilişkin açıklamayı tekrar okuyun. Ancak, bir işletim sistemi planlayıcısının kullandığı üst düzey **politikaları (policies)** henüz anlamadık. Şimdi, çeşitli zeki ve çalışkan insanların yıllar boyunca geliştirdikleri bir **dizi planlama politikasını (scheduling policies)** (bazen **disiplinler (disciplines)** olarak adlandırılır) sunarak tam da bunu yapacağız.

Aslında programlamanın kökenleri, bilgisayar sistemlerinden önceye dayanmaktadır; erken yaklaşımlar operasyon yönetimi alanından alınmış ve bilgisayarlara uygulanmıştır. Bu gerçek şaşırtıcı olmamalı: montaj hatları ve diğer birçok insan çabası da planlama gerektirir ve lazer benzeri bir verimlilik arzusu da dahil olmak üzere aynı endişelerin çoğu burada da mevcuttur. Ve böylece sorunumuz:

ÖNEMLİ NOKTA: PLANLAMA POLİTİKASI NASIL GELİŞTİRİLİR
Programlama politikaları 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 yelpazesine girmeden önce, bazen topluca **iş yükü (workload)** olarak adlandırılan, sistemde çalışan süreçler hakkında bir dizi basitleştirici varsayımda bulunalım. İş yükünü belirlemek, politika oluşturmanın kritik bir parçasıdır ve iş yükü hakkında ne kadar çok şey bilirseniz, politikanızda o kadar ince ayar yapılabilir.
Burada yaptığımız iş yükü varsayımları çoğunlukla gerçekçi değil, ancak (şimdilik) sorun değil, çünkü ilerledikçe onları gevşeteceğiz ve sonunda **tamamen operasyonel bir çizelgeleme disiplini1 (fully-operational scheduling discipline)** olarak adlandıracağımız şeyi geliştireceğiz.

Sistemde çalışan ve bazen **işler (jobs)** olarak adlandırılan süreçler hakkında aşağıdaki varsayımları yapacağız:

1. Her iş aynı süre boyunca çalışır.
2. Tüm işler aynı anda gelir.
3. Başladıktan sonra, her iş tamamlanana kadar devam eder.
4. Tüm işler yalnızca CPU kullanır (yani G/Ç gerçekleştirmezler)
5. Her işin çalışma zamanı biliniyor.

Bu varsayımların birçoğunun gerçekçi olmadığını söylemiştik, ancak Orwell'in Hayvan Çiftliği'nde [O45] bazı hayvanların diğerlerinden daha eşit olması gibi, bu bölümde de bazı varsayımlar diğerlerinden daha gerçekçi değil. Özellikle, her işin çalışma zamanının bilinmesi sizi rahatsız edebilir: bu, programlayıcıyı her şeyi bilen yapar, ki bu harika olsa da (muhtemelen), yakın zamanda gerçekleşmesi pek olası değildir

7.2 Planlama Metrikleri

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

Ancak şimdilik, sadece tek bir metriğe sahip olarak hayatımızı basitleştirelim: geri dönüş süresi. Bir işin geri dönüş süresi, işin tamamlandığı süre eksi işin sisteme ulaştığı süre olarak tanımlanır. Daha resmi olarak, geri dönüş süresi `Tdönüş`:

$$T_{dönüş} = T_{tamamlama} - T_{varış}$$

Çünkü tüm işlerin aynı anda geldiğini varsaydık, şimdilik `Tvarış = 0` ve dolayısıyla `Tdönüş = Ttamamlanma`. Yukarıda belirtilen varsayımları gevşettiğimizde bu gerçek değişecektir.

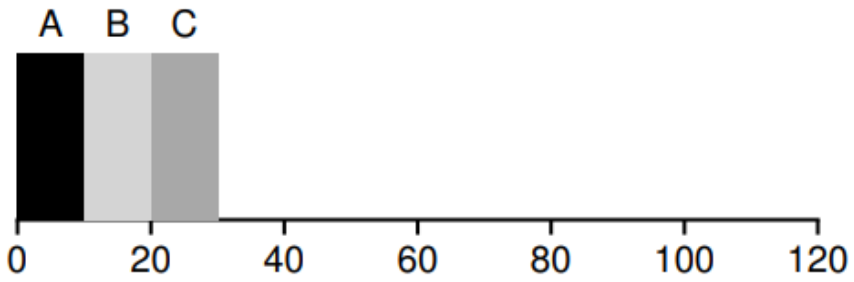
Geri dönüş süresinin, bu bölümde birincil odak noktamız olacak bir **performans (performance)** ölçütü olduğunu unutmamalısınız. Bir başka ilgi ölçüsü, (örneğin) **Jain'in Adalet Endeksi (Jain's Fairness Index)** [J91] tarafından ölçüldüğü şekliyle **adalettir (fairness)**. Performans ve adalet, planlamada genellikle çelişkilidir; örneğin bir programlayıcı, birkaç işin çalışmasını engelleme pahasına performansı optimize edebilir ve böylece adaleti azaltabilir. Bu muamma bize hayatın her zaman mükemmel olmadığını gösteriyor.

7.3 İlk Giren İlk Çıkar (First In First Out)

Uygulayabileceğimiz en temel algoritma, **İlk Giren İlk Çıkar (First In First Out)** (FIFO) planlaması veya bazen İlk Gelen, **İlk Hizmet Eder (First Come, First Served)** (FCFS) olarak bilinir.

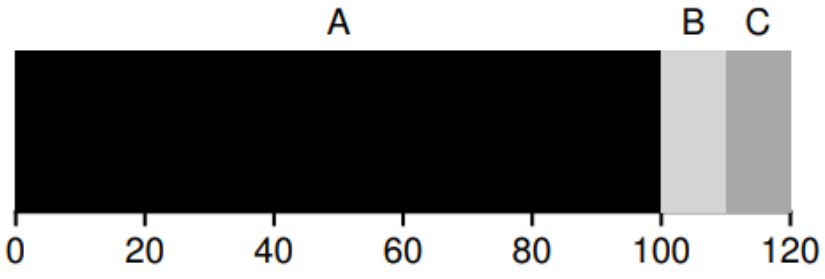
1: Aynı şekilde "Tamamen çalışır durumda bir Ölüm Yıldızı" diyeceğiniz şekilde söylendi.

FIFO'nun bir dizi olumlu özelliği vardır: açıkça basittir ve bu nedenle uygulanması kolaydır.
Varsayımlarımıza göre, oldukça iyi çalışıyor. Birlikte hızlı bir örnek yapalım. Sisteme A, B ve C olmak üzere üç işin kabaca aynı anda geldiğini hayal edin (Varış = 0). FIFO'nun önce bir iş koyması gerektiğinden, hepsi aynı anda gelirken, A'nın B'den biraz önce geldiğini ve C'den biraz önce geldiğini varsayalım. Ayrıca her işin 10 saniye sürdüğünü varsayalım. Bu işler için **ortalama geri dönüş süresi (average turnaround time)** ne olacak?



Şekil 7.1: **Basit FIFO Örneği**

Şekil 7.1'den, A'nın 10'da, B'nin 20'de ve C'nin 30'da bittiğini görebilirsiniz. Böylece, üç iş için ortalama geri dönüş süresi basitçe `(10+20+30)/3=20`'dir.
Şimdi varsayımlarımızdan birini gevşetelim. Özellikle, 1. varsayımı gevşetelim ve böylece artık her işin aynı süre boyunca çalıştığını varsaymayalım. FIFO şimdi nasıl performans gösteriyor? FIFO'nun düşük performans göstermesini sağlamak için ne tür bir iş yükü oluşturabilirsiniz?
Muhtemelen bunu şimdiye kadar anladınız, ancak her ihtimale karşı, farklı uzunluklardaki işlerin FIFO planlaması için nasıl sorunlara yol açabileceğini göstermek için bir örnek yapalım. Özellikle, yine üç işin (A, B ve C) olduğunu varsayalım, ancak bu sefer A 100 saniye, B ve C ise 10 saniye çalışıyor.



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

Şekil 7.2'de görebileceğiniz gibi, A İşi tam 100 saniye boyunca önce B veya C koşma şansı bulamadan çalışır. Bu nedenle, sistem için ortalama geri dönüş süresi yüksektir: sancılı bir 110 saniye (100+110+120/3 = 110).

İPUCU: Önce En Kısa İş İlkesi

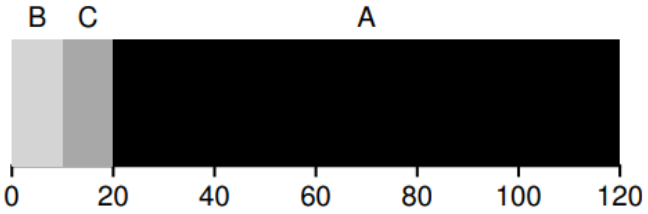
müşteri (veya bizim durumumuzda bir iş) başına algılanan geri dönüş süresinin önemli olduğu herhangi bir sisteme uygulanabilen genel bir çizelgeleme ilkesini temsil eder. Beklediğiniz herhangi bir sırayı düşünün: Söz konusu işletme müşteri memnuniyetini önemsiyorsa muhtemelen SJF'yi dikkate almıştır. Örneğin, marketlerde genellikle satın alacak çok az şeyi olan müşterilerin yaklaşan bir nükleer kışa hazırlanan ailenin gerisinde kalmamasını sağlamak için "on ürün veya daha az" bir sıra vardır.

Bu soruna genel olarak **konvoy etkisi (convoy effect)** [B+79] adı verilir, burada bir kaynağın görece kısa potansiyel tüketicileri ağır sıklet bir kaynak tüketicisinin arkasında sıraya girer. Bu planlama senaryosu size bir bakkaldaki tek bir satırı ve önünüzde üç el arabası dolusu erzak ve bir çek defteri olan kişiyi gördüğünüzde nasıl hissettiğinizi hatırlatabilir; bir süre olacak2.

Yani ne yapmalıyız? Farklı sürelerde çalışan işlerin yeni gerçekliğiyle başa çıkmak için nasıl daha iyi bir algoritma geliştirebiliriz? Önce bir düşünün; sonra okumaya devam edin.

7.4 Önce En Kısa İş (Shortest Job First)

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 çizelgeleme disiplini, **Önce En Kısa İş (SJF) (Shortest Job First)** olarak bilinir ve ilkeyi tamamen tanımladığı için adın hatırlanması kolay olmalıdır: önce en kısa işi, ardından bir sonraki en kısa işi çalıştırır vb.



Şekil 7.3: **SJF örneği**

Yukarıdaki örneğimizi ele alalım, ancak programlama politikamız olarak SJF ile. Şekil 7.3, A, B ve C'yi çalıştırmanın sonuçlarını gösterir. Umarım diyagram, SJF'nin ortalama geri dönüş süresi açısından neden daha iyi performans gösterdiğini açıkça ortaya koyar. Basitçe B ve C'yi A'dan önce çalıştırarak, SJF ortalama geri dönüşü 110 saniyeden 50 saniyeye (10+20+120 3 = 50), iki kat iyileştirmeden daha fazla düşürür.

2: Bu durumda önerilen eylem: ya hızla farklı bir hatta geçin ya da uzun, derin ve rahatlatıcı bir nefes alın. Bu doğru, nefes al, nefes ver. İyi olacak, endişelenme.

KENARA: ÖNCELİKLİ PLANLAYICILAR

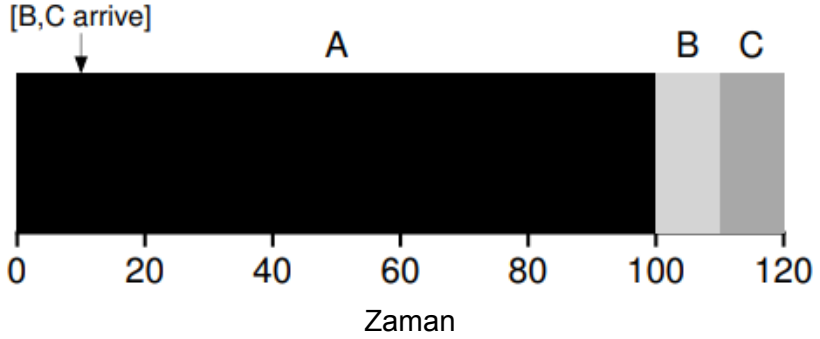
Toplu hesaplamanın eski günlerinde, bir dizi **önleyici olmayan (non-preemptive)**programlayıcı geliştirildi; bu tür sistemler, yeni bir işin çalıştırılıp çalıştırılmayacağını düşünmeden önce her işi tamamlayana kadar çalıştırır. Neredeyse tüm modern planlayıcılar **önleyicidir (preemptive)** ve bir işlemi çalıştırmak için diğerini çalıştırmak için çalışmayı durdurmaya oldukça isteklidir. Bu, zamanlayıcının daha önce öğrendiğimiz mekanizmaları kullandığı anlamına gelir; özellikle programlayıcı, çalışan bir işlemi geçici olarak durdurarak ve diğerini devam ettirerek (veya başlatarak) bir içerik geçişi gerçekleştirebilir.

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

Aslında, hepsi aynı anda gelen işlerle ilgili varsayımlarımız göz önüne alındığında, SJF'nin gerçekten de **optimal** bir çizelgeleme algoritması olduğunu kanıtlayabiliriz. Ancak, teori veya yöneylem araştırması değil, sistem sınıfındasınız; hiçbir kanıta izin verilmez.
Böylece, SJF ile programlama için iyi bir yaklaşıma ulaştık, ancak varsayımlarımız hala oldukça gerçekçi değil. Diğerini rahat bırakalım. Özellikle, 2. varsayımı hedefleyebiliriz ve artık işlerin birden yerine herhangi bir zamanda gelebileceğini varsayabiliriz. Bu hangi sorunlara yol açar?

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

Burada sorunu yine bir örnekle açıklayabiliriz. Bu kez, A'nın t = 0'a vardığını ve 100 saniye koşması gerektiğini, oysa B ve C'nin t = 10'a vardığını ve her birinin 10 saniye koşması gerektiğini varsayalım. Saf SJF ile, Şekil 7.4'te görülen programı elde ederiz.

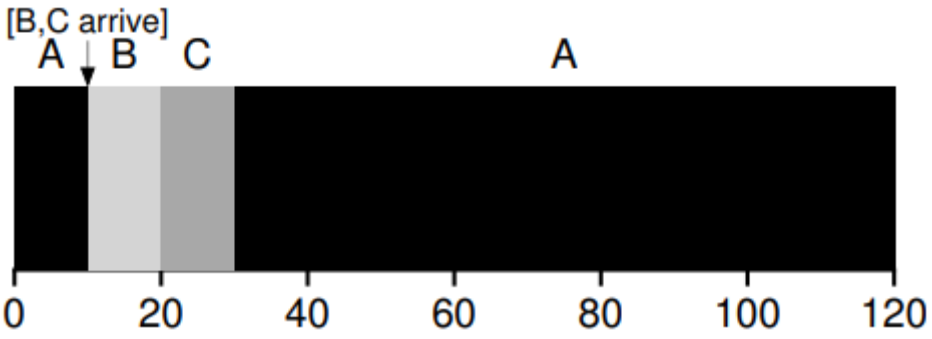


Şekil 7.4: **B ve C'den Geç Gelenlerle SJF**

Şekilden de görebileceğiniz gibi, B ve C, A'dan kısa bir süre sonra gelmelerine rağmen, yine de A'nın tamamlanmasını beklemek zorunda kalıyorlar ve bu nedenle aynı konvoy sorununu yaşıyorlar. Bu üç iş için ortalama geri dönüş süresi 103,33 saniyedir ([(100+(110−10)+(120−10)]/3). Bir zamanlayıcı ne yapabilir?

7.5 Önce En Kısa Tamamlanma Süresi (Shortest Time-to-Completion First)

Bu endişeyi gidermek için, varsayım 3'ü gevşetmemiz gerekiyor (işler tamamlanana kadar devam etmelidir), o halde bunu yapalım. Zamanlayıcının kendisinde de bazı makinelere ihtiyacımız var. Tahmin etmiş olabileceğiniz gibi, zamanlayıcı kesintileri ve bağlam değiştirme



Şekil 7.5: **STCF örnek**

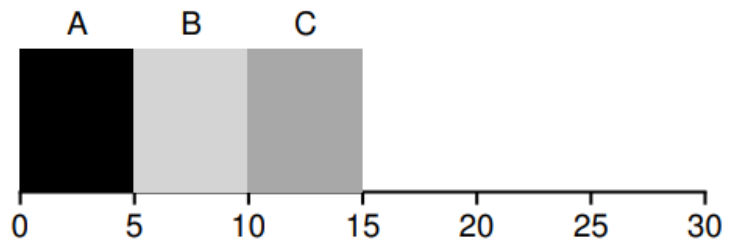
hakkındaki önceki tartışmamıza bakıldığında, zamanlayıcı B ve C geldiğinde kesinlikle başka bir şey yapabilir: A işini **önleyebilir (preempt)** ve başka bir işi çalıştırmaya karar verebilir, belki A'ya daha sonra devam edebilir. Tanımımıza göre SJF, **önleyici olmayan (non-preempt)** bir planlayıcıdır ve bu nedenle yukarıda açıklanan sorunlardan muzdariptir.
Neyse ki, tam olarak bunu yapan bir programlayıcı var: SJF'ye, Önce **En Kısa Tamamlanma Süresi (STCF) (Shortest Time-to-Completion First (STCF))** veya **Önce Öncelikli En Kısa İş (PSJF) (Preemptive Shortest Job First (PSJF))** zamanlayıcı [CK68] olarak bilinen ön alım ekleyin. Sisteme her yeni iş girdiğinde, STCF zamanlayıcı kalan işlerden hangisinin (yeni iş dahil) en az zamanı kaldığını belirler ve onu programlar. Bu nedenle, örneğimizde, STCF A'yı önceden alır ve B ve C'yi tamamlanana kadar çalıştırır; sadece bittiğinde A'nın kalan zamanı planlanır. Şekil 7.5 bir örneği göstermektedir.
Sonuç, çok gelişmiş bir ortalama geri dönüş süresidir: 50 saniye ((120−0)+(20−10)+(30−10)/3). Ve daha önce olduğu gibi, yeni varsayımlarımız göz önüne alındığında, STCF kanıtlanabilir şekilde optimaldir; Tüm işler aynı anda geliyorsa SJF'nin optimal olduğu göz önüne alındığında, muhtemelen STCF'nin optimalliğinin ardındaki sezgiyi görebilmeniz gerekir.

7.6 Yeni Metrik: Tepki Süresi

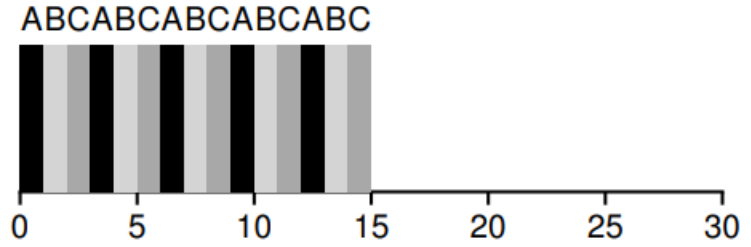
Bu nedenle, iş uzunluklarını bilseydik ve bu işler yalnızca CPU'yu kullanırdı ve tek ölçütümüz geri dönüş süresiydi, STCF harika bir politika olurdu. Aslında, bir dizi eski toplu bilgi işlem sistemi için, bu tür zamanlama algoritmaları bir anlam ifade ediyordu. Ancak, zaman paylaşımlı makinelerin tanıtılması tüm bunları değiştirdi. Artık kullanıcılar bir terminalde oturacak ve sistemden etkileşimli performans da talep edecek. Ve böylece yeni bir ölçü doğdu: **tepki süresi (response time)**.
Yanıt süresini, işin bir sisteme vardığı andan ilk planlandığı zamana3 kadar geçen süre olarak tanımlarız. Daha resmi:

$$T_{response} = T_{ilkçalıştırma} - T_{varış}$$

3: Bazıları bunu biraz farklı tanımlar, örneğin, işin bir tür "yanıt" üretmesine kadar geçen süreyi de dahil etmek için; tanımımız, esasen işin anında bir yanıt ürettiğini varsayarak, bunun en iyi durum versiyonudur.



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



Şekil 7.7: **Round Robin (Tepki Süresi Açısından İyi)**

Örneğin, Şekil 7.5'teki programa sahip olsaydık (A 0 zamanında ve B ve C 10 zamanında gelirken), her işin yanıt süresi şu şekildedir: A işi için 0, B işi için 0 ve İş için 10 C (ortalama: 3.33). Düşünmüş olabileceğiniz gibi, STCF ve ilgili disiplinler yanıt süresi için pek iyi değil. Örneğin, üç iş aynı anda gelirse, üçüncü iş yalnızca bir kez programlanmadan önce önceki iki işin bütünüyle çalışmasını beklemek zorundadır. Geri dönüş süresi için harika olsa da bu yaklaşım, yanıt süresi ve etkileşim için oldukça kötüdür. Gerçekten de, bir terminalde oturduğunuzu, yazdığınızı ve sırf önünüze başka bir iş planlandığı için sistemden bir yanıt görmek için 10 saniye beklemek zorunda kaldığınızı hayal edin: çok hoş değil. Böylece başka bir sorunla karşı karşıya kalıyoruz: Tepki süresine duyarlı bir zamanlayıcıyı nasıl oluşturabiliriz?

7.7 Yuvarlak Sıralama (Round-Robin (RR))

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

RR'yi daha ayrıntılı anlamak için bir örneğe bakalım. Üç A, B ve C işinin sisteme aynı anda geldiğini ve her birinin 5 saniye boyunca çalışmak istediğini varsayalım. Bir SJF zamanlayıcısı, bir başkasını çalıştırmadan önce her işi tamamlanana

İPUCU: Amortisman, MALİYETLERİ DÜŞÜREBİLİR

Amortismanın (amortize) genel tekniği, bazı işlemlerin sabit bir maliyeti olduğu sistemlerde yaygın olarak kullanılır. Bu maliyeti daha az sıklıkta gerçekleştirerek (yani, işlemi daha az kez gerçekleştirerek), sistemin toplam maliyeti azalır. Örneğin, zaman dilimi 10 ms'ye ayarlanmışsa ve bağlam değiştirme maliyeti 1 ms ise, zamanın kabaca %10'u bağlam değiştirmeye harcanır ve bu nedenle boşa harcanır. Bu maliyeti amortize etmek istiyorsak, zaman dilimini örneğin 100 ms'ye yükseltebiliriz. Bu durumda, bağlam değiştirme için harcanan zamanın %1'inden daha azı harcanır ve bu nedenle zaman dilimlemenin maliyeti amortismana tabi tutulur.

kadar çalıştırır (Şekil 7.6). Buna karşılık, 1 saniyelik bir zaman dilimine sahip RR, işler arasında hızlı bir şekilde geçiş yapar (Şekil 7.7)

RR'nin ortalama yanıt süresi: 0+1+2 3 = 1; SJF için ortalama yanıt süresi: 0+5+10 3 = 5. Gördüğünüz gibi, zaman diliminin uzunluğu RR için kritiktir. Ne kadar kısa olursa, yanıt süresi metriği altında RR'nin performansı o kadar iyi olur. Ancak, zaman dilimini çok kısa yapmak sorunludur: birdenbire bağlam değiştirmenin maliyeti genel performansa hakim olacaktır. Bu nedenle, zaman diliminin uzunluğuna karar vermek, bir sistem tasarımcısına bir değiş tokuş sunar; bu, sistemin artık yanıt vermeyeceği kadar uzun sürmeden geçiş maliyetini **amorti etmeye (amortize)** yetecek kadar uzun olmasını sağlar. Bağlam değiştirme maliyetinin, yalnızca işletim sisteminin birkaç kaydı kaydetme ve geri yükleme eylemlerinden kaynaklanmadığına dikkat edin. Programlar çalıştığında, CPU önbelleklerinde, TLB'lerde, şube tahmincilerinde ve diğer çip üstü donanımlarda büyük miktarda durum oluştururlar. Başka bir işe geçmek, bu durumun temizlenmesine ve o anda çalışmakta olan işle ilgili yeni durumun getirilmesine neden olur, bu da dikkate değer bir performans maliyetine neden olabilir [MB91]. Makul bir zaman dilimine sahip RR, bu nedenle, yanıt süresi tek ölçütümüz ise mükemmel bir programlayıcıdır. Peki ya eski dostumuz geri dönüş süresi? Yukarıdaki örneğimize tekrar bakalım. Her biri 5 saniyelik çalışma sürelerine sahip A, B ve C aynı anda gelir ve RR, (uzun) 1 saniyelik bir zaman dilimine sahip programlayıcıdır. Yukarıdaki resimden A'nın 13'te, B'nin 14'te ve C'nin 15'te ortalama 14'te bittiğini görebiliyoruz. Oldukça korkunç! O halde, geri dönüş süresi bizim ölçütümüzse, RR'nin gerçekten de en kötü politikalardan biri olması şaşırtıcı değildir. Sezgisel olarak, bu mantıklı olmalıdır: RR'nin yaptığı şey, her işi bir sonrakine geçmeden önce yalnızca kısa bir süre çalıştırarak olabildiğince uzatmaktır. Geri dönüş süresi yalnızca işlerin ne zaman biteceğini umursadığından, RR neredeyse kötümser, hatta birçok durumda basit FIFO'dan bile daha kötü.

Daha genel olarak, **adil (fair)** olan, yani CPU'yu küçük bir zaman ölçeğinde etkin işlemler arasında eşit olarak bölen herhangi bir politika (RR gibi), geri dönüş süresi gibi ölçütlerde kötü performans gösterecektir. Aslında, bu doğal bir değiş tokuştur: Eğer adaletsiz olmaya istekliysen, tamamlamak için daha kısa işler çalıştırabilirsin, ama yanıt verme süresi pahasına; bunun yerine adalete değer verirseniz

İPUCU: OVERLAP DAHA YÜKSEK KULLANIMI SAĞLAR

Mümkün olduğunda, sistemlerin kullanımını en üst düzeye çıkarmak için işlemleri **çakıştırın (overlap)**. Örtüşme, disk I/O oluştururken veya uzak makinelere mesaj gönderirken; Her iki durumda da, operasyonu başlatmak ve ardından diğer işe geçmek iyi bir fikirdir ve sistemin genel kullanımını ve verimliliğini artırır.

yanıt süresi azaltılır, ancak geri dönüş süresi pahasına. Bu tür değiş tokuş sistemlerde yaygındır; pastanı alıp onu da yiyemezsin4 .

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

7.8 I/O'yu dahil etme

İlk önce varsayım 4'ü gevşeteceğiz - elbette tüm programlar I/O gerçekleştirir. Herhangi bir girdi almayan bir program düşünün: her seferinde aynı çıktıyı üretecekti. Çıktısı olmayan birini hayal edin: Bu, kimsenin görmediği, ormanda düşen meşhur ağaçtır; koştuğu önemli değil.

Bir iş bir I/O talebini başlattığında, bir planlayıcının vereceği bir karar olduğu açıktır, çünkü şu anda çalışmakta olan iş I/O sırasında CPU'yu kullanmayacaktır; I/O tamamlanmasını beklerken **engellenir**. I/O bir sabit disk sürücüsüne gönderilirse, sürücünün geçerli I/O yüküne bağlı olarak işlem birkaç milisaniye veya daha uzun süre engellenebilir. Bu nedenle, programlayıcı muhtemelen o sırada CPU üzerinde başka bir iş programlamalıdır.

Zamanlayıcının ayrıca I/O tamamlandığında bir karar vermesi gerekir. Bu gerçekleştiğinde, bir kesme yükseltilir ve işletim sistemi çalışır ve I/O'yu veren işlemi bloke durumundan hazır durumuna geri taşır. Tabii ki, o noktada işi yürütmeye bile karar verebilir. İşletim sistemi her işi nasıl ele almalı?

Bu konuyu daha iyi anlamak için, her biri 50 ms CPU süresine ihtiyaç duyan A ve B olmak üzere iki işimiz olduğunu varsayalım. Ancak, bariz bir fark vardır: A 10 ms çalışır ve ardından bir I/O isteği gönderir (burada I/O'ların her birinin 10 ms sürdüğünü varsayın), oysa B yalnızca CPU'yu 50 ms kullanır ve I/O gerçekleştirmez. Zamanlayıcı önce A'yı, ardından B'yi çalıştırır (Şekil 7.8).

Bir STCF zamanlayıcı oluşturmaya çalıştığımızı varsayalım. Böyle bir programlayıcı, A'nın 5 adet 10 ms'lik alt işe ayrıldığı gerçeğini nasıl hesaba katmalıdır?

4A, insanların kafasını karıştıran bir söz, çünkü "Pastanı tutup da yiyemezsin" (ki bu çok açık, değil mi?). Şaşırtıcı bir şekilde, bu sözle ilgili bir wikipedia sayfası var; daha da şaşırtıcı olanı, [W15] okumak biraz eğlenceli. İtalyancada dedikleri gibi, Avere la botte piena e la moglie ubriaca olamazsın

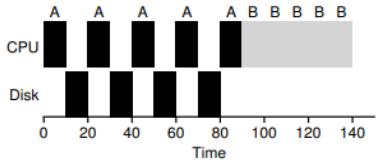
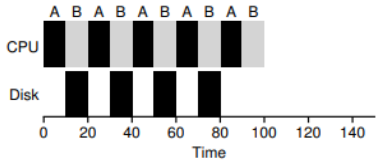


Figure 7.8: **Poor Use Of Resources**



Şekil 7.9: Örtüşme Kaynakların Daha İyi Kullanılmasını Sağlar

oysa B yalnızca 50 ms'lik tek bir CPU talebi mi? Açıkça, sadece birini çalıştırıyorum
I/O'nun nasıl dikkate alınacağını düşünmeden iş ve sonra diğeri
pek mantıklı değil.

Yaygın bir yaklaşım, A'nın her 10 ms'lik alt işini bağımsız bir iş olarak ele almaktır. Bu nedenle, sistem başladığında, seçimi programlanıp planlanmayacağıdır.
10 ms A veya 50 ms B. STCF ile seçim açıktır: daha kısa olanı seçin
bir, bu durumda A. Ardından, A'nın ilk alt işi tamamlandığında, yalnızca
B kaldı ve koşmaya başladı. Ardından A'nın yeni bir alt işi gönderilir,
ve B'yi engeller ve 10 ms çalışır. Bunu yapmak, **overlap (üst üste binmeye**) izin verir ve bir işlem tarafından diğerinin I/O'sunu beklerken kullanılan CPU
tamamlanması gereken süreç; sistem böylece daha iyi kullanılır (bkz. Şekil 7.9)

Ve böylece bir zamanlayıcının I/O'yu nasıl dahil edebileceğini görüyoruz. tedavi ederek
her CPU bir iş olarak patladığında, zamanlayıcı "etkileşimli" süreçlerin sık sık çalıştırılmasını sağlar. Bu etkileşimli işler yapılırken
I/O, diğer CPU yoğun işler çalışır, böylece işlemciden daha iyi yararlanılır.

7.9 Daha Fazla Oracle Yok

I/O'ya yönelik temel bir yaklaşımla, son varsayımımıza geliyoruz: programlayıcı her işin uzunluğunu bilir. 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 a priori bilgi olmadan SJF/STCF gibi sahip olunan bir yaklaşımı nasıl inşa edebiliriz? Ayrıca, yanıt süresinin de oldukça iyi olması için RR zamanlayıcı ile gördüğümüz bazı fikirleri nasıl dahil edebiliriz?

7.10 Özet

Programlamanın arkasındaki temel fikirleri tanıttık ve iki yaklaşım ailesi geliştirdik. İlki, 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 kötü, ne yazık ki, sistemlerde yaygın olan doğal bir değiş tokuş. Ayrıca I/O'yu resme nasıl dahil edebileceğimizi de gördük, ancak işletim sisteminin geleceği görememesi sorununu hala çözemedik. Kısaca, geleceği tahmin etmek için yakın geçmişi kullanan bir zamanlayıcı oluşturarak bu sorunu nasıl aşacağımızı göreceğiz. Bu programlayıcı, **multi-level feedback queue (çok düzeyli geri bildirim kuyruğu)** olarak bilinir ve bir sonraki bölümün konusudur.

References

[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)

Scheduler.py adlı bu program, yanıt süresi, geri dönüş gibi zamanlama ölçütleri altında farklı programlayıcıların nasıl performans gösterdiğini görmenizi sağlar.
süresi ve toplam bekleme süresi. Ayrıntılar için README'ya bakın.

Sorular

1. SJF ve FIFO programlayıcılarla 200 uzunluğunda üç iş yürütürken yanıt süresini ve geri dönüş süresini hesaplayın.

cevap:

$ ./scheduler.py -p SJF -l 200,200,200 -c
response time: 0, 200, 400
turnaround time: 200, 400, 600

$ ./scheduler.py -p FIFO -l 200,200,200 -c
response time: 0, 200, 400
turnaround time: 200, 400, 600

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

cevap:

$ ./scheduler.py -p SJF -l 100,200,300 -c
response time: 0, 100, 300
turnaround time: 100, 300, 600

$ ./scheduler.py -p FIFO -l 100,200,300 -c
response time: 0, 100, 300
turnaround time: 100, 300, 600

3. Şimdi aynısını yapın, ancak aynı zamanda RR zamanlayıcı ve 1'lik bir zaman dilimi ile yapın.

cevap:

$ ./scheduler.py -p RR -q 1 -l 100,200,300 -c
response time: 0, 1, 2
turnaround time: 298, 499, 600

4. SJF ne tür iş yükleri için FIFO ile aynı geri dönüş sürelerini sağlıyor mu?

cevap:

İşler, uzunluğa göre artan sıradadır.

5. SJF, hangi tür iş yükleri ve kuantum uzunlukları için RR ile aynı yanıt sürelerini sağlıyor?

cevap:

İşlerin uzunluğu aynıdır ve kuantum uzunluğu iş uzunluğuna eşittir.

6. İş uzunlukları arttıkça SJF ile yanıt süresi ne olur? Eğilimi göstermek için simülatörü kullanabilir misiniz?

Response time will increase.

$ ./scheduler.py -p SJF -l 200,200,200 -c
$ ./scheduler.py -p SJF -l 300,300,300 -c
$ ./scheduler.py -p SJF -l 400,400,400 -c

7. Kuantum uzunlukları arttıkça RR ile yanıt süresine ne olur? N iş verildiğinde en kötü durum yanıt süresini veren bir denklem yazabilir misiniz?

Yanıt süresi artacaktır.
n'inci işin yanıt süresi = (n - 1) * q
Ortalama yanıt süresi = (n - 1) * q / 2