Soyutlama: Süreç(The Abstraction: The Process)

Bu bölümde, işletim sisteminin kullanıcılara sağladığı en temel soyutlamalardan biri olan **süreci(process)** tartışıyoruz.Formaliteye uygun olmayarak bir sürecin tanımı oldukça basittir: **çalışan bir programdır**(**running program)**[V+65,BH70]. Programın kendisi cansız bir şeydir: diskin üzerinde öylece durur, harekete geçmeyi bekleyen bir sürü talimat (ve belki bazı statik veriler). Bu baytları alan ve çalıştıran, programı faydalı bir şeye dönüştüren işletim sistemidir.

Kişinin genellikle aynı anda birden fazla program çalıştırmak istediği ortaya çıkıyor; Örneğin, bir web tarayıcısı, posta programı, oyun, müzik çalar vb. Çalıştırmak isteyebileceğiniz masaüstünüzü veya dizüstü bilgisayarınızı düşünün. Aslında, tipik bir sistem görünüşte aynı anda onlarca hatta yüzlerce işlemi çalıştırıyor olabilir. Bunu yapmak, bir cpu'nun mevcut olup olmadığıyla asla ilgilenilmemesi gerektiğinden, sistemin kullanımını kolaylaştırır; Biri sadece programları çalıştırır. Bu yüzden meydan okumamız:

SORUNUN ÖZÜ:
BİRÇOK CPU'NUN YANILSAMASI NASIL SAĞLANIR?
Yalnızca birkaç fiziksel cpu mevcut olsa da, işletim sistemi söz konusu cpu'ların neredeyse sonsuz bir arzının ilizyonu nasıl sağlayabilir?

İşletim sistemi, CPU'yu **sanallaştırarak**(**virtualizing)** bu yanılsamayı yaratır. Bir işlemi çalıştırarak, ardından durdurarak ve diğerini çalıştırarak vb. İşletim sistemi, aslında yalnızca bir fiziksel CPU (veya birkaçı) olduğunda birçok sanal cpu'nun var olduğu yanılsamasını teşvik edebilir. Cpu'nun **zaman paylaşımı**(**time sharing)** olarak bilinen bu temel teknik, kullanıcıların istedikleri kadar eşzamanlı işlem çalıştırmasına olanak tanır; CPU'ları paylaşılması gerekiyorsa her biri daha yavaş çalışacağından, potansiyel maliyet performanstır.

Cpu'nun sanallaştırılmasını uygulamak ve iyi uygulamak için işletim sisteminin hem düşük seviyeli makinelere hem de yüksek seviyeli bir haberleşmeye ihtiyacı olacaktır. Düşük seviyeli makine **mekanizmaları(mechanisms)** diyoruz; mekanizmalar, gerekli bir işlevsellik parçasını uygulayan düşük seviyeli yöntemlerdir. Örneğin, bir **bağlamın(context)** nasıl uygulanacağını daha sonra öğreneceğiz.

TIP: ZAMAN PAYLAŞIMI (VE ALAN PAYLAŞIMI) KULLANIMI

Zaman paylaşımı(Time sharing) , bir işletim sistemi tarafından bir kaynağı paylaşmak için kullanılan temel bir tekniktir. Kaynağın bir varlık tarafından bir süre, sonra bir süre diğeri tarafından kullanılmasına vb. İzin verilerek, söz konusu kaynak (ör. CPU veya bir ağ bağlantısı) birçok kişi tarafından paylaşılabilir. Zaman paylaşımının karşılığı, bir kaynağın onu kullanmak isteyenler arasında (uzayda) bölündüğü **alan paylaşımıdır**(**space sharing**). Örneğin, disk alanı doğal olarak alan paylaşımlı bir kaynaktır; Bir dosyaya bir blok atandığında, kullanıcı orijinal dosyayı silene kadar normalde başka bir dosyaya atanmaz.

İşletim sistemine bir programı çalıştırmayı durdurma ve belirli bir cpu'da başka bir programı çalıştırmaya başlama yeteneği veren **anahtar(switch);** Bu **zaman paylaşım** (**time-sharing)** mekanizması tüm modern işletim sistemleri tarafından kullanılmaktadır.

Bu mekanizmaların üzerinde, işletim sistemindeki haberleşmenin bir kısmı **ilkeler(policies)** şeklinde bulunur. İlkeler, işletim sistemi içinde bir tür karar vermeye yönelik algoritmalardır.

Örneğin, bir CPU üzerinde çalıştırılabilecek birkaç olası program verildiğinde, işletim sistemi hangi programı çalıştırmalıdır? İşletim sistemindeki bir **zamanlama ilkesi (scheduling policy),** bu kararı muhtemelen geçmiş bilgileri (örneğin, son dakikada hangi program daha fazla çalıştı?), iş yükü bilgisini (örneğin, ne tür programlar çalıştırılıyor) ve performans ölçütlerini kullanarak verecektir. (örneğin, sistem etkileşimli performans için mi yoksa verim için mi optimize ediyor?)

- ## Soyutlama: Süreç

Çalışan bir programın işletim sistemi tarafından sağlanan soyutlama, **süreç(process)** olarak adlandıracağımız bir şeydir. Yukarıda söylediğimiz gibi, bir süreç basitçe çalışan bir programdır; Herhangi bir anda, yürütülmesi sırasında eriştiği veya etkilediği sistemin farklı parçalarının envanterini alarak bir süreci özetleyebiliriz.

Bir süreci neyin oluşturduğunu anlamak için, onun **makine durumunu(machine state)** anlamamız gerekir: bir program çalışırken ne okuyabilir veya güncelleyebilir. Herhangi bir zamanda, bu programın yürütülmesi için makinenin hangi parçaları önemlidir?

Bir süreci içeren makine durumunun bariz bileşenlerinden biri belleğidir. Talimatlar bellekte bulunur; Çalışan programın okuduğu ve yazdığı veriler de bellekte bulunur. Bu nedenle, işlemin adresleyebileceği bellek (**adres alanı(address space)** olarak adlandırılır) işlemin bir parçasıdır.

Also part of the process's machine state are *registers*; many instructions explicitly read or update registers and thus clearly they are important tothe execution of the process.
Ayrıca sürecin makine durumunun bir kısmı kayıtlardır; Birçok talimat kayıtları açıkça okur veya günceller ve bu nedenle sürecin yürütülmesi için açıkça önemlidirler.

Bu makine durumunun bir parçasını oluşturan bazı özel kayıtlar olduğunu unutmayın. Örneğin, **program sayacı (program counter) (PC)** (bazen **komut işaretçisi** (**instruction pointer**) veya **IP** olarak adlandırılır) bize program'ın hangi komutunun bir sonraki komutu yürüteceğini söyler; benzer şekilde bir **yığın işaretçisi(stack pointer)** ve ilişkili **çerçeve(frame).**

TIP: AYRI İLKESİ VE MEKANİZMA
Birçok işletim sisteminde, ortak bir tasarım paradigması, üst düzey ilkeleri alt düzey mekanizmalarından ayırmaktır [L+75]. Mekanizmayı bir sistemle ilgili nasıl sorusuna yanıt olarak düşünebilirsiniz; örneğin, bir işletim sistemi bağlam anahtarını nasıl gerçekleştirir? İlke, hangi sorunun cevabını verir; örneğin, işletim sistemi şu anda hangi işlemi çalıştırmalı? İkisini ayırmak, mekanizmayı yeniden düşünmek zorunda kalmadan ilkelerin kolayca değiştirilmesine izin verir ve bu nedenle, genel bir yazılım tasarım ilkesi olan bir **modülerlik(modularity)** biçimidir.

İşaretçi(pointer), işlev parametreleri, yerel değişkenler ve dönüş adresleri için yığını yönetmek için kullanılır.
Son olarak, programlar genellikle kalıcı depolama aygıtlarına da erişir. Bu I / O bilgileri, işlemin o anda açık olduğu dosyaların bir listesini içerebilir.

Süreç API'sı(Process API)
Gerçek bir süreç API'sinin tartışılmasını bir sonraki bölüme kadar ertelemiş olsak da, burada önce bir işletim sisteminin herhangi bir arayüzüne nelerin dahil edilmesi gerektiğine dair bir fikir vereceğiz. Bu API'ler, bir şekilde, herhangi bir modern işletim sisteminde mevcuttur.

Oluşturma(create): : Bir işletim sistemi, yeni süreçler oluşturmak için bazı yöntemler içermelidir. Kabuğa bir komut yazdığınızda veya bir uygulama simgesine çift tıkladığınızda, belirttiğiniz programı çalıştırmak için yeni bir işlem oluşturmak üzere işletim sistemi çağrılır.
Yok Et(destroy): Süreç oluşturma için bir arayüz olduğu için sistemler de
süreçleri zorla yok etmek için bir arayüz sağlar. Elbette birçok süreç çalışacak ve tamamlandığında kendi kendine kapanacaktır; Ancak bunu yapmadıklarında, kullanıcı onları öldürmek isteyebilir ve bu nedenle kontrolden çıkmış bir süreci durdurmak için bir arayüz oldukça kullanışlıdır.
Bekle(wait): Bazen bir işlemin çalışmayı durdurmasını beklemek yararlıdır;

bu nedenle genellikle bir tür bekleme arabirimi sağlanır.

Çeşitli Kontrol(various control): Bir işlemi öldürmek veya beklemek dışında, bazen başka kontroller de mümkündür. Örneğin, çoğu işletim sistemi, bir işlemi askıya almak (bir süre çalışmasını durdurmak) ve ardından devam ettirmek (çalışmaya devam etmek) için bir tür yöntem sağlar.

Durum(status): Bazı durum bilgilerini almak için genellikle arayüzler vardır.
ne kadar süredir devam ettiği veya hangi durumda olduğu gibi bir süreç hakkında da.

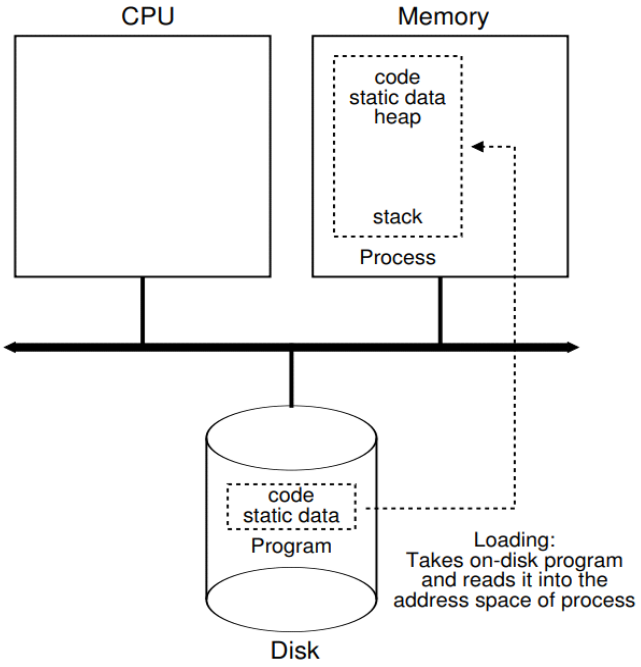

Figure 4.1: **Loading: From Program To Process**

Şekil 4.1: Yükleme: Programdan İşleme

- Süreç Oluşturma Detay

Maskesini biraz kaldırmamız gereken bir bilinmezlik, programların nasıl süreçlere dönüştürüldüğüdür. Spesifik olarak, işletim sistemi bir programı nasıl çalışır hale getirir? Süreç oluşturma gerçekte nasıl çalışır?

İşletim sisteminin bir programı çalıştırmak için yapması gereken ilk şey, kodunu ve herhangi bir statik veriyi (örneğin, başlatılmış değişkenler) belleğe, işlemin adres alanına **yüklemektir(load).** Programlar başlangıçta bir tür **yürütülebilir biçimde(executable formatta) diskte(disk**) (veya bazı modern sistemlerde **flash tabanlı SSD'lerde(flash-based SSDs)**) bulunur; bu nedenle, bir programı ve statik verileri belleğe yükleme işlemi, işletim sisteminin bu baytları diskten okumasını ve bunları bellekte bir yere yerleştirmesini gerektirir (Şekil 4.1'de gösterildiği gibi).

Erken (veya basit) işletim sistemlerinde, yükleme işlemi **istekle(eagerly),** yani programı çalıştırmadan önce birdenbire yapılır; modern işletim sistemleri, işlemi **tembel bir şekilde(lazily),** yani yalnızca program yürütme sırasında ihtiyaç duyulduklarında kod veya veri parçalarını yükleyerek gerçekleştirir. Kod parçalarının ve verilerin geç yüklenmesinin nasıl çalıştığını gerçekten anlamak için hakkında daha fazla bilgi sahibi olmanız gerekir.

Disk belleği(paging) ve **değiş tokuş(swapping)** makineleri ,gelecekte belleğin sanallaştırılmasını tartışırken ele alacağımız konular. Şimdilik, herhangi bir şeyi çalıştırmadan önce işletim sisteminin önemli program bitlerini diskten belleğe almak için bazı işler yapması gerektiğini unutmayın.

Kod ve statik veriler belleğe yüklendikten sonra, işlemi çalıştırmadan önce işletim sisteminin yapması gereken birkaç şey daha vardır. Programın **çalışma zamanı yığını (run-time stack)** (veya sadece **yığını(steak))** için bir miktar bellek ayrılmalıdır. Muhtemelen zaten bildiğiniz gibi, C programları yığını yerel değişkenler, işlev parametreleri ve dönüş adresleri için kullanır; işletim sistemi bu belleği ayırır ve sürece verir. İşletim sistemi ayrıca yığını argümanlarla başlatacaktır; özellikle, main() işlevinin, yani argc ve argv dizisinin parametrelerini dolduracaktır.

İşletim sistemi ayrıca programın yığını için bir miktar **bellek(heap)** ayırabilir. C programlarında yığın, açıkça talep edilen dinamik olarak tahsis edilmiş veriler için kullanılır; programlar malloc()'u çağırarak böyle bir alanı talep eder ve free()'yi çağırarak açık bir şekilde boşaltır. Yığın, bağlantılı listeler, karma tablolar, ağaçlar ve diğer ilginç veri yapıları gibi veri yapıları için gereklidir. Yığın ilk başta küçük olacaktır; program çalışırken ve malloc() kitaplık API'si aracılığıyla daha fazla bellek talep ettikçe, işletim sistemi devreye girebilir ve bu tür çağrıları karşılamaya yardımcı olmak için sürece daha fazla bellek ayırabilir.

İşletim sistemi ayrıca, özellikle giriş/çıkış (I/O) ile ilgili olarak, diğer bazı başlatma görevlerini de yapacaktır. Örneğin, UNIX sistemlerinde, varsayılan olarak her işlemin standart girdi, çıktı ve hata için üç açık dosya tanıtıcısı vardır; bu tanımlayıcılar, programların terminalden gelen girdileri kolayca okumasına ve çıktıyı ekrana yazdırmasına olanak tanır. **Kalıcılık(persistence)** hakkındaki kitabın üçüncü I / O , **dosya tanımlayıcıları(file descriptors)** ve benzerleri hakkında daha fazla şey öğreneceğiz.

Kodu ve statik verileri belleğe yükleyerek, bir yığın oluşturup başlatarak ve I / O kurulumuyla ilgili diğer işleri yaparak, işletim sistemi şimdi (nihayet) program yürütme aşamasını hazırlamıştır. Bu nedenle son bir görevi vardır: programı giriş noktasında, yani main()'de çalıştırmak. main() rutinine atlayarak (sonraki bölümde tartışacağımız özel bir mekanizma aracılığıyla), işletim sistemi CPU'nun kontrolünü yeni oluşturulan sürece aktarır ve böylece program yürütmeye başlar.

• **Süreç Durumları**

Artık bir sürecin ne olduğu (bu kavramı geliştirmeye devam etmemize rağmen) ve (kabaca) nasıl yaratıldığı hakkında bir fikrimiz olduğuna göre, bir sürecin belirli bir zamanda içinde olabileceği farklı **durumlardan(states)** bahsedelim. Bir sürecin bu durumlardan birinde olabileceği fikri, erken bilgisayar sistemlerinde ortaya çıkmıştır [DV66,V+65]. Basitleştirilmiş bir görünümde, bir süreç üç durumdan birinde olabilir:

•**Çalışma(runing)** : Çalışan durumda, bir işlemci üzerinde bir işlem çalışıyor. Bu, talimatları uyguladığı anlamına gelir.

• **Hazır(ready):** Hazır durumda, bir süreç çalışmaya hazırdır, ancak bazıları için işletim sisteminin şu anda çalıştırmamayı seçmesinin nedeni.

- **Bloklama(Blocked):** Bloklanan durumda, bir işlem, başka bir olay gerçekleşene kadar çalışmaya hazır olmamasını sağlayan bir tür işlem gerçekleştirmiştir. Yaygın bir örnek: Bir işlem bir diske I/O isteği başlattığında bloklanır ve bu nedenle başka bir işlem işlemciyi kullanabilir.

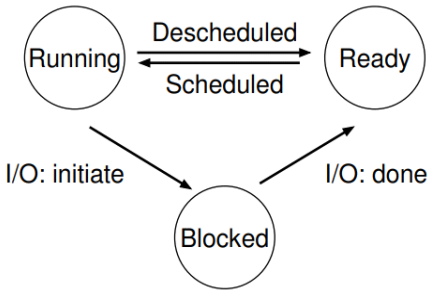

Figure 4.2: **Process: State Transitions**

Şekil 4.2: Süreç: Durum Geçişleri

Bu durumları bir grafikle haritalayacak olsaydık, Şekil 4.2'deki şemaya varırdık. Diyagramda görebileceğiniz gibi, işletim sisteminin takdirine bağlı olarak hazır ve çalışan durumlar arasında bir işlem taşınabilir. Hazırdan çalışmaya taşınmak, işlemin **programlanmış(sched- uled)** olduğu anlamına gelir; çalışmadan hazıra taşınmak, işlemin **programlanmamış**(**descheduled)**. olduğu anlamına gelir. Bir işlem engellendikten sonra (ör. Bir I / O işlemi başlatılarak), işletim sistemi bir olay gerçekleşene kadar (ör. I / O tamamlama) bu işlemi bu şekilde tutacaktır.; bu noktada, işlem tekrar hazır duruma geçer (ve işletim sistemi karar verirse potansiyel olarak hemen tekrar çalışmaya başlar).

İki sürecin bu durumlardan bazılarında nasıl geçiş yapabileceğine dair bir örneğe bakalım. İlk olarak, her biri yalnızca CPU'yu kullanan iki işlemin çalıştığını hayal edin (I / O yapmazlar). Bu durumda, her işlemin durumunun bir izi şöyle görünebilir (Şekil 4.3).

Zaman	Process0	Process1	Notlar
1	Çalışıyor	Hazır	
2	Çalışıyor	Hazır	
3	Çalışıyor	Hazır	
4	Çalışıyor	Hazır	Process0 şimdi bitti
5	–	Çalışıyor	
6	–	Çalışıyor	
7	–	Çalışıyor	
8	–	Çalışıyor	Process1 şimdi bitti

Figure 4.3: **Tracing Process State: CPU Only**
Şekil 4.3: İzleme İşlemi Durumu: Yalnızca CPU

Zaman	Process0	Process1	Notlar
1	Çalışıyor	Hazır	
2	Çalışıyor	Hazır	
3	Çalışıyor	Hazır	Process0 başlatılıyor/O
4	Bloklandı	Çalışıyor	Process0 is bloklama
5	Bloklandı	Çalışıyor	yani Process1 çalışır
6	Bloklandı	Çalışıyor	
7	Hazır	Çalışıyor	I/O oldu
8	Hazır	Çalışıyor	Process1 şimdi bitti
9	Çalışıyor	–	
10	Çalışıyor	–	Process0 şimdi bitti

Figure 4.4: **Tracing Process State: CPU and I/O**
Şekil 4.4: İzleme İşlemi Durumu: CPU ve I/O

Bu sonraki örnekte, ilk işlem bir süre çalıştıktan sonra bir I / O verir. Bu noktada işlem engellenir ve diğer işleme çalışma şansı verilir. Şekil 4.4, bu senaryonun bir izini göstermektedir.

Daha spesifik olarak, Process0 bir I / O başlatır ve tamamlanmasını beklerken engellenir; işlemler, örneğin bir diskten okunurken veya bir ağdan bir paket beklenirken engellenir. İşletim sistemi, İşlem0'ın CPU'yu kullanmadığını ve İşlem1'i çalıştırmaya başladığını fark eder. Process1 çalışırken, I / O işlemi tamamlayarak Process0'ı ready'ye geri taşır. Son olarak, Process1 bitirir ve Process0 çalışır ve sonra yapılır.

Bu basit örnekte bile işletim sisteminin alması gereken birçok karar olduğunu unutmayın. İlk olarak, sistem Process0 bir I / O yayınlarken Process1'i çalıştırmaya karar vermek zorunda kaldı; Bunu yapmak, CPU'yu meşgul ederek kaynak kullanımını iyileştirir. İkincisi, sistem I / O tamamlandığında Process0'a geri dönmemeye karar verdi; Bunun iyi bir karar olup olmadığı belli değil. Ne düşünüyorsunuz? Bu tür kararlar, gelecekte birkaç bölümde tartışacağımız bir konu olan işletim sistemi **zamanlayıcı (scheduler)**,tarafından alınır.

- **(Veri Yapıları)Data Structures**

İşletim sistemi bir programdır ve herhangi bir program gibi, çeşitli ilgili bilgi parçalarını izleyen bazı önemli veri yapılarına sahiptir. Örneğin, her bir işlemin durumunu izlemek için işletim sistemi, hazır olan tüm işlemler için bir tür **işlem listesi (pro-cess list)** ve şu anda hangi işlemin çalıştığını izlemek için bazı ek bilgiler tutacaktır. İşletim sistemi bir şekilde engellenen işlemleri de izlemelidir; Bir I / O olayı tamamlandığında, işletim sistemi doğru işlemi uyandırdığından ve yeniden çalışmaya hazırladığından emin olmalıdır.

Şekil 4.5, bir işletim sisteminin xv6 çekirdeğindeki her işlem hakkında ne tür bilgileri izlemesi gerektiğini göstermektedir [CK + 08]. Linux, Mac OS X veya Windows gibi "gerçek" işletim sistemlerinde de benzer işlem yapıları vardır; Onlara bakın ve ne kadar karmaşık olduklarını görün.

Şekilden, işletim sisteminin bir süreç hakkında izlediği birkaç önemli bilgiyi görebilirsiniz. **Kayıt bağlamı (register context)** , durdurulan bir işlem için kayıtlarının içeriğini tutacaktır. Bir işlem durdurulduğunda, kayıtları bu bellek konumuna kaydedilir; Bu kayıtları geri yükleyerek (yani değerlerini gerçek fiziksel kayıtlara geri yerleştirerek), işletim sistemi işlemi çalıştırmaya devam edebilir. Gelecek bölümlerde **bağlam değiştirme(context switch)** olarak bilinen bu teknik hakkında daha fazla bilgi edineceğiz.

Şekilde, bir profesyonelin koşmanın, hazır olmanın ve engellenmenin ötesinde içinde olabileceği başka durumlar da olduğunu görebilirsiniz. Bazen bir sistem, oluşturulurken işlemin içinde olduğu **bir başlangıç durumuna (initial state)** sahip olur. Ayrıca, bir süreç çıktığı **son bir duruma(final state)** yerleştirilebilir, ancak henüz temizlenmedi (UNİX tabanlı sistemlerde buna **zombi(zombie)** durumu denir). Bu son durum, diğer işlemlerin (genellikle işlemi oluşturan **üst öğe(parent))** işlemin dönüş kodunu incelemesine ve yeni tamamlanan işlemin başarıyla yürütülüp yürütülmediğine bakmasına izin verdiği için yararlı olabilir (genellikle programlar, bir görevi başarıyla tamamladıklarında UNİX tabanlı sistemlerde sıfır döndürür ve bir görevi başarıyla tamamladıklarında sıfır döndürür.).aksi takdirde sıfır). Sonlandırıldığında, ebeveyn, çocuğun tamamlanmasını beklemek için son bir çağrı yapar (ör. wait ()) ve ayrıca işletim sistemine, artık soyu tükenmiş sürece atıfta bulunan ilgili veri yapılarını temizleyebileceğini belirtir.

Zombie State(zombi durumu):Tıpkı gerçek zombiler gibi, bu zombilerin öldürülmesi nispeten kolaydır.Bununla birlikte, genellikle farklı teknikler önerilir.

```
// xv6 kayıtları kaydedilecek ve geri yüklenecektir
//  bir  işlem  yapısı  bağlamını  durdurmak  ve  ardından
yeniden başlatmak için{
  int eip; int esp; int
  ebx; int ecx; int edx;
  int esi; int edi; int
  ebp;
};

// bir sürecin numaralandırılabileceği farklı durumlar
proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

//xv6'nın her işlemle ilgili izlediği bilgiler
// kayıt bağlamı ve durum yapısı dahil
struct proc {
  char *mem;                  // İşlem belleğinin başlangıcı
  uint sz;                    // İşlem belleğinin boyutu
  char *kstack;               // Çekirdek yığınının alt kısmı
                              // bu işlem için
  enum proc_state state;      // İşlem durumu
  int pid;                    // İşlem ID
  struct proc *parent;        // Ana süreç
  void *chan;                 // egersıfırdegilse,sleeping on chan
  int killed;                 // eğer sıfır ise öldürüldü
  struct file *ofile[NOFILE]; // Açık dosyalar yapısıinode
  *cwd;                       // Geçerli dizin
  struct context context;     // İşlemi çalıştırmak için burayageç
  struct trapframe *tf;       // İçin tuzak çerçevesi
```

```
                                        // kesme
};
```

Figure 4.5: **The xv6 Proc Structure(proc yapısı)**

VERİ YAPISI - İŞLEM LİSTESİ

İşletim sistemleri, bu notlarda tartışacağımız çeşitli önemli **veri yapılarıyla(data structures)** doludur**. İşlem listesi(işlem listesi)** (**görev listesi(tasklist)** de denebilir) bu tür ilk yapıdır. Daha basit olanlardan biridir, ancak kesinlikle birden fazla programı aynı anda çalıştırabilen herhangi bir işletim sistemi, sistemdeki tüm çalışan programları takip etmek için bu yapıya benzer bir şeye sahip olacaktır. Bazen insanlar, bir süreç hakkındaki bilgileri depolayan bireysel yapıya, her süreç hakkında bilgi içeren (bazen **süreç tanımlayıcısı(process descriptor** da denebilir) bir C yapısından bahsetmenin süslü bir yolu olan bir **Süreç Kontrol Bloğu(PCB)(Process Control Block)** olarak atıfta bulunurlar.

• Özet

İşletim sisteminin en temel soyutlamasını süreçi işledik. Oldukça basit bir şekilde çalışan bir program olarak görülüyor. Bu kavramsal bakış açısını göz önünde bulundurarak, şimdi küçük ayrıntılara geçeceğiz: süreçleri uygulamak için gereken düşük seviyeli mekanizmalar ve bunları akıllı bir şekilde planlamak için gereken üst düzey ilkeler. Mekanizmaları ve ilkeleleri birleştirerek, bir işletim sisteminin CPU'yu nasıl sanallaştırdığına dair anlayışımızı geliştireceğiz.

NOT: TEMEL SÜREÇ TERİMLERİ

• **Süreç(Process),** çalışan bir programın ana işletim sistemi soyutlamasıdır. Sürecin herhangi bir noktasında, durumu şu şekilde tanımlanabilir: **adres alanındaki (address space)** bellek yapıları, CPU kayıtlarının içeriği (diğerlerinin yanı sıra program sayacı ve yığın işaretçisi dahil) ve I / O ile ilgili bilgiler (okunabilen veya okunabilen açık dosyalar gibi). yazılı).

• **Süreç API'sı(process API),** programların süreçlerle ilgili yapabileceği çağrılardan oluşur. Tipik olarak buna yaratma, yok etme ve diğer kullanım çağrıları dahildir.

• İşlemler, çalıştırma, çalıştırmaya hazır ve engellenmiş dahil olmak üzere birçok farklı **işlem durumundan(process states)** birinde bulunur. Bu durumlardan birinden diğerine farklı olaylar (örneğin, programlanmak veya programdan çıkmak veya bir I/O 'nin tamamlanmasını beklemek) bir süreci aktarır.

• Bir **süreç listesi(process list),** sistemdeki tüm süreçler hakkında bilgi içerir. Her giriş, bazen yalnızca belirli bir işlem hakkında bilgi içeren bir yapı olan **işlem kontrol bloğu (PCB)(process Control Block)** olarak adlandırılan yerde bulunur.

References

[BH70] "The Nucleus of a Multiprogramming System" by Per Brinch Hansen. Communica- tions of the ACM, Volume 13:4, April 1970. *This paper introduces one of the first **microkernels** in operating systems history, called Nucleus. The idea of smaller, more minimal systems is a theme that rears its head repeatedly in OS history; it all began with Brinch Hansen's work described herein.*

[CK+08] "The xv6 Operating System" by Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. From: https://github.com/mit-pdos/xv6-public. *The coolest real and little OS in the world. Download and play with it to learn more about the details of how operating systems actually work. We have been using an older version (2012-01-30-1-g1c41342) and hence some examples in the book may not match the latest in the source.*

[DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis, Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966 . *This paper defined many of the early terms and concepts around building multiprogrammed systems.*

[L+75] "Policy/mechanism separation in Hydra" by R. Levin, E. Cohen, W. Corwin, F. Pollack,
W. Wulf. SOSP '75, Austin, Texas, November 1975. *An early paper about how to structure operat- ing systems in a research OS known as Hydra. While Hydra never became a mainstream OS, some of its ideas influenced OS designers.*

[V+65] "Structure of the Multics Supervisor" by V.A. Vyssotsky, F. J. Corbato, R. M. Graham. Fall Joint Computer Conference, 1965. *An early paper on Multics, which described many of the basic ideas and terms that we find in modern systems. Some of the vision behind computing as a utility are finally being realized in modern cloud systems.*

ÖDEV Homework (Simulation)

Process-run.py adlı bu program, programlar çalışırken ve CPU'yu kullanırken (örn. bir ekleme talimatı gerçekleştirirken) veya I/O yaparken (örn. tamamlamak için). Ayrıntılar için README'ya bakın.

- Anlaşılması gereken en önemli seçenek, çalışan her programın (veya 'işlemin') ne yapacağını tam olarak belirten **PROCESS_LIST'dir** (-l veya --processlist bayraklarıyla belirtildiği gibi). Bir süreç talimatlardan oluşur ve her talimat iki şeyden birini yapabilir:
- CPU'yu kullanmak
- Bir I/O'nun gerçekleşmesi

Sorular

`1)process-run.py`'yi şu bayraklarla çalıştırın: `-l 5:100,5:100`. CPU kullanımı ne olmalıdır (örneğin, CPU'nun kullanımda olduğu sürenin yüzdesi?) Bunu neden biliyorsunuz? Haklı olup olmadığınızı görmek için -c ve -p bayraklarını kullanın.

- Bir işlem CPU'yu kullandığında (ve hiç IO yapmadığında), basitçe CPU üzerinde ÇALIŞIYOR(RUNİNG) veya çalışmaya HAZIR(READY) olmak arasında geçiş yapmalıdır. CPU'yu kullanıyor (IO yapmıyor).

>process-run.py -l 5:100 çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$  python process-run.py -l5:100
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu
  cpu

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

- Burada belirttiğimiz işlem "5:100" yani 5 komuttan oluşmalıdır ve her komutun CPU komutu olma olasılığı %100'dür.

Yanıtları sizin yerinize hesaplayan -c işaretini kullanarak sürece ne olduğunu görebiliriz:

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py  -l5:100 -c
Time        PID: 0          CPU          IOs
  1         RUN:cpu          1
  2         RUN:cpu          1
  3         RUN:cpu          1
  4         RUN:cpu          1
  5         RUN:cpu          1
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

- Sonuç, beklenen bir sonuçtur.RUN(çalışma) durumunda işlem basittir tüm çalışma boyunca CPU'yu kullanmasıyla CPU'yu meşgul ederek ve herhangi bir I/O yapmadan tamamlanır.

>process-run.py -l 5:100,5:100

İki işlem çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ettim

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l5:100,5:100
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu
  cpu

Process 1
  cpu
  cpu
  cpu
  cpu
  cpu

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

- Bu işlemde ikisi de CPU'yu kullanan iki farklı işlem çalışır

-c bayrağı bayrağı ile işletim sistemi bunları çalıştırdığındaki ekran görüntüsü:

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l5:100,5:100 -c
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:cpu         READY             1
  2         RUN:cpu         READY             1
  3         RUN:cpu         READY             1
  4         RUN:cpu         READY             1
  5         RUN:cpu         READY             1
  6            DONE       RUN:cpu             1
  7            DONE       RUN:cpu             1
  8            DONE       RUN:cpu             1
  9            DONE       RUN:cpu             1
 10            DONE       RUN:cpu             1
```

- Ekran görüntüsünde de olduğu gibi önce PID'si 0 olan işlem çalışır(run),PID:1 olan çalışmaya hazır(ready) ancak 0 tamamlanana kadar bekler.PID:0 işlemi bittiğinde DONE durumuna geçer ve 1 işlemi bittiğinde tüm işlemler tamamlanmıştır.

-p bayrağını kullanarak istatiskleri yazdırdım aşağıdaki ekran görüntüsünü elde ettim

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l5:100,5:100 -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:cpu         READY             1
  2         RUN:cpu         READY             1
  3         RUN:cpu         READY             1
  4         RUN:cpu         READY             1
  5         RUN:cpu         READY             1
  6            DONE       RUN:cpu             1
  7            DONE       RUN:cpu             1
  8            DONE       RUN:cpu             1
  9            DONE       RUN:cpu             1
 10            DONE       RUN:cpu             1

Stats: Total Time 10
Stats: CPU Busy 10 (100.00%)
Stats: IO Busy  0 (0.00%)
```

- E.G'den de anlaşıldığı gibi CPU %100 meşgulken ;IO meşgul değildir.

2) Şimdi şu bayraklarla çalıştırın: `./process-run.py -l 4:100,1:0` Bu bayraklar, 4 yönergeli (tümü CPU'yu kullanmak için) ve yalnızca bir I/O yayınlayan ve bunun yapılmasını bekleyen bir işlemi belirtir. Her iki işlemin tamamlanması ne kadar sürer? Haklı olup olmadığınızı öğrenmek için -c ve -p tuşlarını kullanın.

>`./process-run.py -l 4:100,1:0` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l 4:100,1:0
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu

Process 1
  io
  io_done

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

-
- Bu işlemde işlemin birisi cpu kullanırken diğeri IO yapıyor.

-c bayrağı bayrağı ile işletim sistemi bunları çalıştırdığındaki ekran görüntüsü:

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l 4:100,1:0 -c
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:cpu         READY             1
  2         RUN:cpu         READY             1
  3         RUN:cpu         READY             1
  4         RUN:cpu         READY             1
  5            DONE        RUN:io             1
  6            DONE       BLOCKED                           1
  7            DONE       BLOCKED                           1
  8            DONE       BLOCKED                           1
  9            DONE       BLOCKED                           1
 10            DONE       BLOCKED                           1
 11*           DONE    RUN:io_done             1
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

- Ekran görüntüsünde de olduğu gibi önce PID'si 0 olan işlem çalışır(run),PID:1 olan çalışmaya hazır(ready) ancak 0 tamamlanana kadar bekler.PID:0 işlemi bittiğinde DONE durumuna geçer ve 1 işlemi I/O yaptığında bloklama başlar ve I/O Done durumunda işlemi tamamlar.

-p bayrağını kullanarak istatiskleri yazdırdım aşağıdaki ekran görüntüsünü elde ettim

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l 4:100,1:0 -p -c
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:cpu         READY             1
  2         RUN:cpu         READY             1
  3         RUN:cpu         READY             1
  4         RUN:cpu         READY             1
  5            DONE        RUN:io             1
  6            DONE       BLOCKED                           1
  7            DONE       BLOCKED                           1
  8            DONE       BLOCKED                           1
  9            DONE       BLOCKED                           1
 10            DONE       BLOCKED                           1
 11*           DONE   RUN:io_done             1

Stats: Total Time 11
Stats: CPU Busy 6 (54.55%)
Stats: IO Busy  5 (45.45%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

- E.G'den de anlaşıldığı gibi CPU %54 meşgulken ;IO %45 meşguldur.

3) İşlemlerin sırasını değiştirin: -l 1:0,4:100. Şimdi ne olacak? Düzeni değiştirmek önemli mi? Neden? Niye? (Her zaman olduğu gibi, haklı olup olmadığınızı görmek için -c ve -p kullanın)

-

`>./process-run.py -l 1:0, 4:100 -p -c` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l 1:0,4:100 -p -c
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu             1             1
  3        BLOCKED       RUN:cpu             1             1
  4        BLOCKED       RUN:cpu             1             1
  5        BLOCKED       RUN:cpu             1             1
  6        BLOCKED          DONE                           1
  7*   RUN:io_done          DONE             1

Stats: Total Time 7
Stats: CPU Busy 6 (85.71%)
Stats: IO Busy  5 (71.43%)
```

- Artık 1. işlem, 0. işlem I/O'nin tamamlanmasını beklerken çalışır. Düzeni değiştirmek önemli çünkü birinci düzende 11. Zamanda işlemi tamamlarken ikinci düzende 7.zamanda işlemi tamamlar.
- Ve ikinci düzende I/O ve CPU nun daha meşgul olduğu gayet açık görünüyor.

4)Şimdi diğer bayraklardan bazılarını keşfedeceğiz. Önemli bir bayrak
-S, bir işlem bir I/O yayınladığında sistemin nasıl tepki vereceğini belirler. Bayrak SWITCH ON END olarak ayarlandığında sistem, biri I/O yaparken başka bir işleme GEÇMEYECEK, bunun yerine işlem tamamen bitene kadar bekleyecektir. Aşağıdaki iki işlemi çalıştırdığınızda ne olur (-l 1:0,4:100
-c -S SWITCH ON END), biri I/O yapıyor ve diğeri CPU işi yapıyor mu?

`>./process-run.py -l 1:0, 4:100 -c -S SWITCH_ON_END` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l 1:0,4:100 -c -S SWITCH_ON_END
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        BLOCKED         READY                           1
  3        BLOCKED         READY                           1
  4        BLOCKED         READY                           1
  5        BLOCKED         READY                           1
  6        BLOCKED         READY                           1
  7*   RUN:io_done         READY             1
  8           DONE       RUN:cpu             1
  9           DONE       RUN:cpu             1
 10           DONE       RUN:cpu             1
 11           DONE       RUN:cpu             1
```

- Proecss 1, process 0 I/O'yu beklerken çalışmaz.

5)Şimdi, aynı işlemleri çalıştırın, ancak I/O için BEKLENİYOR(READY) olduğunda (-l 1:0,4:100 -c -S SWITCH ON IO) başka bir işleme geçmek için anahtarlama davranışı ayarlı olarak çalıştırın. Şimdi ne olacak? Haklı olduğunuzu onaylamak için -c ve -p tuşlarını kullanın.

`>./process-run.py -l 1:0, 4:100 -c -S SWITCH_ON_IO` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l1:0,4:100 -c -S SWITCH_ON_IO
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu             1             1
  3        BLOCKED       RUN:cpu             1             1
  4        BLOCKED       RUN:cpu             1             1
  5        BLOCKED       RUN:cpu             1             1
  6        BLOCKED          DONE                           1
  7*   RUN:io_done          DONE             1
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l3:0,5:100,5:100,5:100 -S SWITCH
```

6) Diğer bir önemli davranış, bir I/O tamamlandığında ne yapılacağıdır–`I IO RUN LATER` ile, bir I/O tamamlandığında, onu yayınlayan işlemin hemen çalıştırılması gerekmez; bunun yerine, o sırada çalışan her ne ise, çalışmaya devam eder. Bu işlem kombinasyonunu çalıştırdığınızda ne olur? (Çalıştır(RUN) `./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH ON IO -I IO RUN LATER-c -p`) Sistem kaynakları etkin bir şekilde kullanılıyor mu?

`/process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p
Time        PID: 0        PID: 1        PID: 2        PID: 3           CPU           IOs
  1         RUN:io         READY         READY         READY             1
  2        BLOCKED       RUN:cpu         READY         READY             1             1
  3        BLOCKED       RUN:cpu         READY         READY             1             1
  4        BLOCKED       RUN:cpu         READY         READY             1             1
  5        BLOCKED       RUN:cpu         READY         READY             1             1
  6        BLOCKED       RUN:cpu         READY         READY             1             1
  7*         READY          DONE       RUN:cpu         READY             1
  8          READY          DONE       RUN:cpu         READY             1
  9          READY          DONE       RUN:cpu         READY             1
 10          READY          DONE       RUN:cpu         READY             1
 11          READY          DONE       RUN:cpu         READY             1
 12          READY          DONE          DONE       RUN:cpu             1
 13          READY          DONE          DONE       RUN:cpu             1
 14          READY          DONE          DONE       RUN:cpu             1
 15          READY          DONE          DONE       RUN:cpu             1
 16          READY          DONE          DONE       RUN:cpu             1
 17    RUN:io_done          DONE          DONE          DONE             1
 18         RUN:io          DONE          DONE          DONE             1
 19        BLOCKED          DONE          DONE          DONE                           1
 20        BLOCKED          DONE          DONE          DONE                           1
 21        BLOCKED          DONE          DONE          DONE                           1
 22        BLOCKED          DONE          DONE          DONE                           1
 23        BLOCKED          DONE          DONE          DONE                           1
 24*   RUN:io_done          DONE          DONE          DONE             1
 25         RUN:io          DONE          DONE          DONE             1
 26        BLOCKED          DONE          DONE          DONE                           1
 27        BLOCKED          DONE          DONE          DONE                           1
 28        BLOCKED          DONE          DONE          DONE                           1
 29        BLOCKED          DONE          DONE          DONE                           1
 30        BLOCKED          DONE          DONE          DONE                           1
 31*   RUN:io_done          DONE          DONE          DONE             1

Stats: Total Time 31
Stats: CPU Busy 21 (67.74%)
Stats: IO Busy  15 (48.39%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

- İşlem 0, ilk I/O'yi çalıştırır, ardından kalan I/O'lari çalıştırmak için diğer işlemin yapılmasını bekler. Sistem kaynakları etkin bir şekilde kullanmıyor.

-

7)Şimdi aynı işlemleri çalıştırın, ancak I/O veren işlemi hemen çalıştıran `–I IO RUN IMMEDIATE` ayarlı olarak. Bu davranış nasıl farklıdır? Bir I/O'yi henüz tamamlamış bir işlemi yeniden çalıştırmak neden iyi bir fikir olabilir?

- `/process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_IMMEDIATE -c -p` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -l3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_IMMEDIATE  -c -p
Time        PID: 0        PID: 1        PID: 2        PID: 3           CPU           IOs
  1         RUN:io         READY         READY         READY             1
  2        BLOCKED       RUN:cpu         READY         READY             1             1
  3        BLOCKED       RUN:cpu         READY         READY             1             1
  4        BLOCKED       RUN:cpu         READY         READY             1             1
  5        BLOCKED       RUN:cpu         READY         READY             1             1
  6        BLOCKED       RUN:cpu         READY         READY             1             1
  7*   RUN:io_done          DONE         READY         READY             1
  8         RUN:io          DONE         READY         READY             1
  9        BLOCKED          DONE       RUN:cpu         READY             1             1
 10        BLOCKED          DONE       RUN:cpu         READY             1             1
 11        BLOCKED          DONE       RUN:cpu         READY             1             1
 12        BLOCKED          DONE       RUN:cpu         READY             1             1
 13        BLOCKED          DONE       RUN:cpu         READY             1             1
 14*   RUN:io_done          DONE          DONE         READY             1
 15         RUN:io          DONE          DONE         READY             1
 16        BLOCKED          DONE          DONE       RUN:cpu             1             1
 17        BLOCKED          DONE          DONE       RUN:cpu             1             1
 18        BLOCKED          DONE          DONE       RUN:cpu             1             1
 19        BLOCKED          DONE          DONE       RUN:cpu             1             1
 20        BLOCKED          DONE          DONE       RUN:cpu             1             1
 21*   RUN:io_done          DONE          DONE          DONE             1

Stats: Total Time 21
Stats: CPU Busy 21 (100.00%)
Stats: IO Busy  15 (71.43%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

- Artık diğer işlemler, işlem 0 I/O beklerken çalışabilir. Daha adil ve yanıt süresini azalttı.

8) Şimdi rastgele oluşturulmuş bazı işlemlerle çalıştırın: `-s 1 -l 3:50,3:50` veya `-s 2 -l 3:50,3:50` veya `-s 3 -l 3:50,3:50`. İzin nasıl sonuçlanacağını tahmin edip edemeyeceğinize bakın. `-I IO RUN IMMEDIATE` ya karşı `-I IO RUN LATER` bayrağını kullandığınızda ne olur? `-S SWITCH ON IO` 'ya karşı `-S SWITCH ON END` kullandığınızda ne olur?

`>./process-run.py -s 3 -l 3:50, 3:50 -c -p` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -s 3 -l 3:50,3:50 -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2         RUN:io         READY             1
  3        BLOCKED        RUN:io             1             1
  4        BLOCKED       BLOCKED                           2
  5        BLOCKED       BLOCKED                           2
  6        BLOCKED       BLOCKED                           2
  7        BLOCKED       BLOCKED                           2
  8*   RUN:io_done       BLOCKED             1             1
  9*       RUN:cpu         READY             1
 10           DONE   RUN:io_done             1
 11           DONE        RUN:io             1
 12           DONE       BLOCKED                           1
 13           DONE       BLOCKED                           1
 14           DONE       BLOCKED                           1
 15           DONE       BLOCKED                           1
 16           DONE       BLOCKED                           1
 17*          DONE   RUN:io_done             1
 18           DONE       RUN:cpu             1

Stats: Total Time 18
Stats: CPU Busy 9 (50.00%)
Stats: IO Busy  11 (61.11%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

`>./process-run.py -s 3 -l 3:50, 3:50 -I IO_RUN_IMMEDIATE -c -p` çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -s 3 -l 3:50,3:50 -I IO_RUN_IMMEDIATE -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2         RUN:io         READY             1
  3        BLOCKED        RUN:io             1             1
  4        BLOCKED       BLOCKED                           2
  5        BLOCKED       BLOCKED                           2
  6        BLOCKED       BLOCKED                           2
  7        BLOCKED       BLOCKED                           2
  8*   RUN:io_done       BLOCKED             1             1
  9*         READY   RUN:io_done             1
 10          READY        RUN:io             1
 11        RUN:cpu       BLOCKED             1             1
 12           DONE       BLOCKED                           1
 13           DONE       BLOCKED                           1
 14           DONE       BLOCKED                           1
 15           DONE       BLOCKED                           1
 16*          DONE   RUN:io_done             1
 17           DONE       RUN:cpu             1

Stats: Total Time 17
Stats: CPU Busy 9 (52.94%)
Stats: IO Busy  11 (64.71%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

>./process-run.py -s 3 -l 3:50, 3:50 -I IO_RUN_LATER -c -p çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -s 3 -l 3:50,3:50 -I IO_RUN_LATER -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2         RUN:io         READY             1
  3        BLOCKED        RUN:io             1             1
  4        BLOCKED       BLOCKED                           2
  5        BLOCKED       BLOCKED                           2
  6        BLOCKED       BLOCKED                           2
  7        BLOCKED       BLOCKED                           2
  8*   RUN:io_done       BLOCKED             1             1
  9*       RUN:cpu         READY             1
 10           DONE   RUN:io_done             1
 11           DONE        RUN:io             1
 12           DONE       BLOCKED                           1
 13           DONE       BLOCKED                           1
 14           DONE       BLOCKED                           1
 15           DONE       BLOCKED                           1
 16           DONE       BLOCKED                           1
 17*          DONE   RUN:io_done             1
 18           DONE       RUN:cpu             1

Stats: Total Time 18
Stats: CPU Busy 9 (50.00%)
Stats: IO Busy  11 (61.11%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

-I IO RUN IMMEDIATE ya karşı -I IO RUN LATER Kullandığımızda yukarıdaki 2 ekran görüntüsünde de gördüğümüz gibi ımmedıatede total time 17 iken later da total time 18 yani ımmedıate daha iyi diyebiliriz.

>./process-run.py -s 3 -l 3:50, 3:50 -S SWITCH_ON_IO -c -p çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -s 3 -l 3:50,3:50 -S SWITCH_ON_IO -c -
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2         RUN:io         READY             1
  3        BLOCKED        RUN:io             1             1
  4        BLOCKED       BLOCKED                           2
  5        BLOCKED       BLOCKED                           2
  6        BLOCKED       BLOCKED                           2
  7        BLOCKED       BLOCKED                           2
  8*   RUN:io_done       BLOCKED             1             1
  9*       RUN:cpu         READY             1
 10           DONE   RUN:io_done             1
 11           DONE        RUN:io             1
 12           DONE       BLOCKED                           1
 13           DONE       BLOCKED                           1
 14           DONE       BLOCKED                           1
 15           DONE       BLOCKED                           1
 16           DONE       BLOCKED                           1
 17*          DONE   RUN:io_done             1
 18           DONE       RUN:cpu             1

Stats: Total Time 18
Stats: CPU Busy 9 (50.00%)
Stats: IO Busy  11 (61.11%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

>./process-run.py -s 3 -l 3:50, 3:50 -S SWITCH_ON_END -c -p çalıştığında aşağıdaki ekran görüntüsünü elde ettim.

```
karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python process-run.py -s 3 -l 3:50,3:50 -S SWITCH_ON_END -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2         RUN:io         READY             1
  3        BLOCKED         READY                           1
  4        BLOCKED         READY                           1
  5        BLOCKED         READY                           1
  6        BLOCKED         READY                           1
  7        BLOCKED         READY                           1
  8*   RUN:io_done         READY             1
  9        RUN:cpu         READY             1
 10           DONE        RUN:io             1
 11           DONE       BLOCKED                          1
 12           DONE       BLOCKED                          1
 13           DONE       BLOCKED                          1
 14           DONE       BLOCKED                          1
 15           DONE       BLOCKED                          1
 16*          DONE   RUN:io_done             1
 17           DONE        RUN:io             1
 18           DONE       BLOCKED                          1
 19           DONE       BLOCKED                          1
 20           DONE       BLOCKED                          1
 21           DONE       BLOCKED                          1
 22           DONE       BLOCKED                          1
 23*          DONE   RUN:io_done             1
 24           DONE       RUN:cpu             1

Stats: Total Time 24
Stats: CPU Busy 9 (37.50%)
Stats: IO Busy  15 (62.50%)

karaca@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$
```

-S SWITCH ON IO 'ya karşı -S SWITCH ON END i kullandığımızda yukarıdaki 2 ekran görüntüsünden de anlaşılacağı gibi IO da total time 18 iken END de total time 24 yani IO daha iyi diyebiliriz.