4

Soyutlama: Süreç

Bu bölümlerde, işletim sisteminin tüm bölümlerinde en temel soyutlamalardan birini tartışıyoruz: **süreci (process)** . Bir işlemin tanımı, kontrolleri olarak, hızlı kullanımları: **çalışan bir programdır(running program)[**V+65,BH70]. Programın kendisi cansız bir şeydir : sadece diskte oturur, bir sürü talimat (ve belki de bazı statik hareketler), harekete geçmeyi bekler. Bu baytları alan ve onları çalıştıran, programı yararlı bir şeye dönüştüren çalışan sistemdir.

Birinin genellikle aynı anda birden fazla program çalıştırmak istediği ortaya çıkıyor; örneğin, bir web tarayıcısını, posta programını, oyunu, müzik çaları vb. çalıştırmak isteyebileceğiniz masaüstü veya dizüstü bilgisayarınızı düşünün. Aslında, tipik bir sistem aynı anda onlarca hatta yüzlerce işlem çalıştırıyor olabilir. Bunu yapmak, sistemin kullanımını kolaylaştırır, çünkü bir CPU'nun mevcut olup olmadığı ile asla ilgilenmenize gerek yoktur; biri sadece programları çalıştırır. Bu nedenle bizim meydan okumamız:

SORUNUN ÖZÜ:
BİRÇOK CPU'NUN İLLÜZYONU NASIL SAĞLANIR?
Sadece birkaç fiziksel CPU mevcut olmasına rağmen, işletim sistemi söz konusu CPU'ların neredeyse sonsuz bir arzı illüzyonu nasıl sağlayabilir?

İşletim sistemi, CPU'yu **sanallaştırarak(virtualizing)**bu yanılsamayı (illüzyon) yaratır. Bir işlemi çalıştırarak, sonra durdurarak ve diğerini çalıştırarak, işletim sistemi, aslında yalnızca bir fiziksel CPU (veya birkaçı) varken 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; potansiyel maliyet performanstır, çünkü CPU'ların paylaşılması gerekiyorsa her biri daha yavaş çalışacaktır.

CPU'nun sanallaştırmasını uygulamak ve iyi bir şekilde uygulamak için, işletim sistemi hem bazı düşük seviyeli makinelere hem de bazı üst düzey sezgilere ihtiyaç duyacaktır. Düşük seviyeli makine **mekanizmaları(mechanisms)** diyoruz; mekanizmalar, gerekli bir işlevsellik parçasını uygulayan düşük seviyeli yöntemler veya protokollerdir. Örneğin, bir **bağlamın (context)** nasıl uygulanacağını daha sonra öğreneceğiz

İPUCU: ZAMAN PAYLAŞIMINI (VE ALAN PAYLAŞIMI) KULLANMA

Zaman paylaşımı(Time sharing), bir işletim sistemi tarafından bir kaynağı paylaşmak için kullanılan temel bir tekniktir. Kaynağın bir varlık tarafından bir süreliğine ve daha sonra bir süre başka bir varlık tarafından kullanılmasına izin vererek, söz konusu kaynak (örneğin, 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 bloğa bir blok atandıktan sonra, kullanıcı orijinal dosyayı silene kadar normalde başka bir dosyaya atanmaz.

İşletim sistemine bir programı çalıştırmayı bırakıp belirli bir CPU'da başka bir programı çalıştırmaya başlama yeteneği veren **anahtar(switch)**; Bu **zaman paylaşım(Time sharing)** mekanizması tüm modern işletim sistemleri tarafından kullanılır.

Bu mekanizmaların üstünde, işletim sistemindeki istihbaratın bir kısmı, **politikalar(policies)** biçiminde bulunur. Politikalar, işletim sistemi içinde bir tür karar vermek için kullanılan algoritmalardır. Örneğin, bir CPU'da çalışacak bir dizi olası program göz önüne alındığında, işletim sistemi hangi programı çalıştırmalıdır? İşletim sistemindeki bir **zamanlama politikası (scheduling policy)**, kararını vermek için büyük olasılıkla geçmiş bilgileri (örneğin, hangi programın son dakika içinde daha fazla çalıştığı?), iş yükü bilgisini (örneğin, ne tür programların çalıştırıldığı) ve performans ölçümlerini (örneğin, sistem etkileşimli performans veya aktarım hızı için optimize ediyor mu?) kullanarak bu kararı verecektir.

4.1 Soyutlama: Bir Süreç

Çalışan bir programın işletim sistemi tarafından sağlanan soyutlama, **süreci (process).** Yukarıda söylediğimiz gibi, bir süreç basitçe çalışan bir programdır; Herhangi bir zamanda, bir süreci, yürütülmesi sırasında eriştiği veya etkilediği sistemin farklı parçalarının bir envanterini alarak özetleyebiliriz . Bir süreci neyin oluşturduğunu anlamak için , onun **makine durumunu (machine state)** anlamamız gerekir: Bir programın çalışırken okuyabileceği veya güncelleyebileceği. Herhangi bir zamanda, makinenin hangi parçaları bu programın yürütülmesi için önemlidir? Bir süreci içeren makine durumunun belirgin bileşeni hafızasıdır (*memory*). Talimatlar hafızada yatıyor; çalışan programın okuduğu ve yazdığı veriler de bellekte bulunur. Bu nedenle, işlemin adresleyebileceği bellek **adres alanı (address space)** olarak adlandırılır) süreç (*process*) bir parçasıdır. Ayrıca işlemin makine durumunun bir parçası *da kayıtlardır (registers)*; Birçok talimat kayıtları açıkça okur veya günceller ve bu nedenle sürecin yürütülmesi için açıkça önemlidir. Bu makine durumunun bir parçasını oluşturan bazı özel kayıtlar olduğunu unutmayın. Örneğin , **program sayacı (program counter) (PC)** (bazen **talimat işaretçisi (instruction pointer)** veya **IP** olarak adlandırılır) bize bir sonraki adımda hangi program talimatının uygulanacağını söyler; benzer şekilde yığın **işaretçisi (stack)** ve ilişkili **çerçeve (associated)**

İPUCU: AYRI POLITIKA VE MEKANIZMA
Birçok işletim sisteminde, ortak bir tasarım paradigması, üst düzey politikaları düşük düzeyli mekanizmalarından ayırmaktır [L+75]. Mekanizmayı , bir sistemle ilgili *nasıl* sorusunun cevabını vermek olarak düşünebilirsiniz; örneğin, bir işletim sistemi bağlam anahtarını *nasıl* gerçekleştirir? İlke hangi sorunun yanıtını sağlar ; örneğin, işletim sistemi şu anda hangi işlemi çalıştırmalıdır? İkisini ayırmak, mekanizmayı yeniden düşünmek zorunda kalmadan politikaları kolayca değiştirmeyi sağlar ve bu nedenle bir **modülerlik(modularity)** biçimi, genel bir yazılım tasarım ilkesidir.

İşaretçi(pointer), işlev parametreleri, yerel değişkenlikler ve dönüş adresleri için yığını yönetmek üzere kullanılır.

Son olarak, programlar genellikle kalıcı depolama aygıtlarına da erişir. Bu tür *I/O bilgileri*, işlemin şu anda açık olduğu dosyaların bir listesini içerebilir.

4.1 İşlem API'si

Gerçek bir süreç API'sinin tartışılmasını bir sonraki bölüme kadar ertelememize rağmen, burada önce bir işletim sisteminin herhangi bir arayüzüne nelerin dahil edilmesi gerektiğine dair bir fikir veriyoruz. Bu API'ler, bir biçimde, herhangi bir modern işletim sisteminde kullanılabilir .

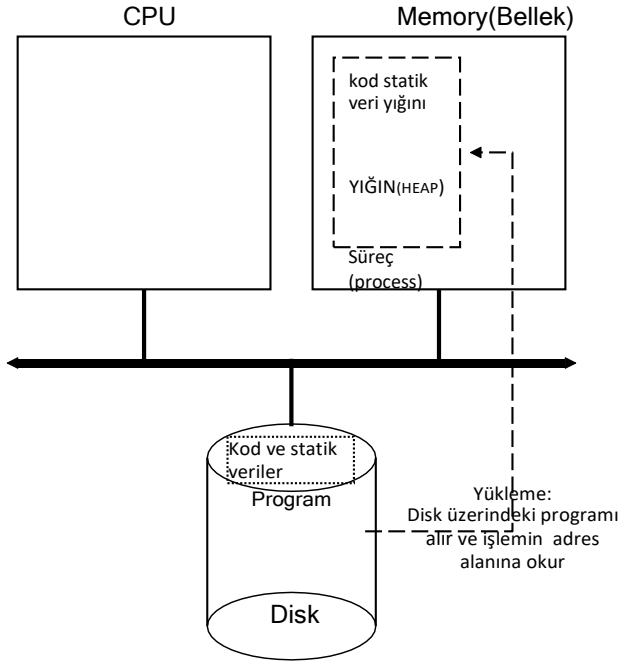
- **Yaratmak(Create):** Bir işletim sistemi, yeni işlemler oluşturmak için bazı yöntemler içermelidir. Kabuğa bir komut yazdığınızda veya bir uygulama simgesine çift tıkladığınızda, belirttiğiniz programı çalıştırmak üzere yeni bir işlem oluşturmak için işletim sistemi çağrılır.

- **Yıkmak(Destroy):** Süreç oluşturma için bir arayüz olduğu için, sistemler de süreçleri zorla yok etmek için bir arayüz sağlayın. Tabii ki, birçok süreç çalışacak ve tamamlandığında kendiliğinden çıkacaktır; Bununla birlikte, bunu yapmadıklarında, kullanıcı onları öldürmek isteyebilir ve bu nedenle kaçak bir süreci durdurmak için bir ters yüz oldukça yararlıdır.

- **Bekleme(Wait):** Bazen bir işlemin çalışmayı durdurmasını beklemek yararlıdır; bu nedenle genellikle bir tür bekleme arayüzü sağlanır.

- **Çeşitli Kontroller (Miscellaneous Control):** Bir süreci öldürmek veya beklemek dışında bazen mümkün olan başka kontroller de vardır. 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 çalıştığı veya hangi durumda olduğu gibi bir süreç hakkında da.



Şekil 4.1: **Yükleme: Programdan İşleme**

4.3 Süreç Oluşturma: Biraz Daha Ayrıntı

Maskesini biraz çözmemiz gereken bir gizem,programların süreçlere nasıl dönüştürüldüğüdür . Özellikle, 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 ve işlemin reklam elbisesi alanına **yüklemektir(load).**Programlar başlangıçta **diskte(disk)** (veya bazı modern sistemlerde **flash tabanlı SSD'lerde(flash-basedSSDs))** bir tür **yürütülebilir biçimde(executable format)** bulunur; Bu nedenle, bir programı ve statik verileri belleğe yükleme işlemi, işletim sisteminin bu baytları diskten okumasını ve belleğe bir yere yerleştirmesini gerektirir.(Şekil 4.1'de gösterildiği gibi).

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

Sayfalama(paging) ve değiştirme(swapping) makineleri, gelecekte belleğin sanallaştırmasını tartışırken ele alacağımız konular. Şimdilik, herhangi bir şeyi çalıştırmadan önce, işletim sisteminin önemli program bitlerini diskten belleğe almak için açıkça bazı çalışmalar yapması gerektiğini unutmayın.

Kod ve statik veriler belleğe yüklendikten sonra, işletim sisteminin işlemi çalıştırmadan önce yapması gereken birkaç şey daha vardır.Programın **çalışma zamanı yığını (run-time stack)** (veya yalnızca yığın) için bir miktar bellek ayrılmalıdır. Muhtemelen zaten bilmeniz gerektiği gibi, C programları yığını yerel değişkenler, işlev parametreleri ve dönüş adresleri için kullanır;işletim sistemi bu belleği ayırır ve işleme verir. İşletim sistemi ayrıca yığını bağımsız değişkenlerle başlatacaktır; özellikle, parametreleri main() fonksiyonuna, yani argc ve argv dizisine dolduracaktır .

İşletim sistemi ayrıca programın **yığını(heap)** için bir miktar bellek ayırabilir. C programlarında, yığın açıkça istenen dinamik olarak ayrılmış veriler için kullanılır; programlar *malloc*() öğesini çağırarak bu alanı talep eder ve *free()* öğesini çağırarak bu alanı ücretsiz olarak serbest bırakır. Yığın, bağlantılı listeler, karma tablolar, ağaçlar ve diğer ilginç veri yapıları gibi veri yapıları için gereklidir. Yığın ilk başta küçük olacak; Program çalıştıkça ve *malloc()* kitaplık API'si aracılığıyla daha fazla hafıza talep ettikçe, işletim sistemi bu tür çağrıların karşılanmasına yardımcı olmak için sürece dahil olabilir ve sürece daha fazla bellek ayırabilir .

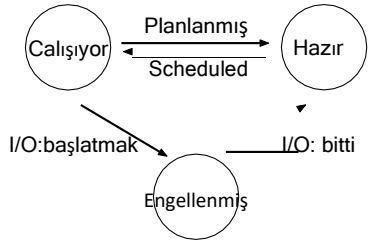
İşletim sistemi ayrıca, özellikle giriş / çıkışa (I/O) yeniden bağlandığı gibi diğer bazı başlatma görevlerini de yapacaktır.Örneğin, UNIX sistemlerinde, her işlemin varsayılan olarak standart giriş, çıkış ve hata için üç açık **dosya tanımlayıcısı(file descriptors)** vardır;Bu tanımlayıcılar, programların terminalden girişi kolayca okumasını ve çıktıyı ekrana yazdırmasını sağlar. **Kalıcılık(persistence)** hakkındaki I/O, kitabın üçüncü bölümünde , dosya gidericiler ve benzerleri hakkında daha fazla bilgi edineceğiz.

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

4.3 İşlem Durumları

Artık bir sürecin ne olduğu hakkında bir fikrimiz olduğuna göre (yine de bu kavramı rafine etmeye devam edeceğiz),(kabaca)nasıl yaratıldığı,bir sürecin belirli bir zamanda içinde olabileceği farklı **durumlar (states)** hakkında konuşalım. Bir sürecin bu durumlardan birinde olabileceği fikri, erken bilgisayar sistemlerinde ortaya çıkmıştır [DV66, V + 65].Basitleştirilmiş bir görünümde, bir işlem üç durumdan birinde olabilir:

- **Çalışıyor (Running)**: Çalışıyor durumunda, 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.



Şekil 4.2: **Süreç: Durum Geçişleri**

- **Engellenmiş (Blocked)**:Engellenmiş durumda, bir işlem , başka bir olay gerçekleşene kadar çalışmaya hazır olmamasına neden olan bir tür işlem gerçekleştirmiştir . Yaygın bir örnek: Bir işlem diske I/ isteği başlattığında, engellenir ve böylece başka bir işlem işlemciyi kullanabilir .

Bu durumları bir grafikle eşleyecek olsaydık, Şekil 4.2'deki di-agrama ulaşırdık. Diyagramda görebileceğiniz gibi, bir süreç işletim sisteminin takdirine bağlı olarak hazır ve çalışan durumlar arasında taşınabilir. Hazır olmaktan çalışmaya geçmek, sürecin **planlandığı(scheduled)** anlamına gelir; being moved from running to ready means the process **programdan(descheduled)** çıkarılmıştır. Bir işlem engellendikten sonra (e.g., başlatarak I/O işlem), işletim sistemi, bazı olaylar gerçekleşene kadar bu şekilde kalacaktır (e.g., I/O tamamlama); Bu noktada, süreç tekrar hazır duruma geçer. (ve potansiyel olarak hemen tekrar çalışmaya başlar, eğer işletim sistemi karar verirse).

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

Time	$Process_0$	$Process_1$	Notes
1	Running	Read	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process0 now done (süreç tamamlandı)
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process1 now done (süreç tamamlandı)

Şekil 4.3: **İşlem Durumunu İzleme: Yalnızca CPU**

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ başlatmak I/O
4	Blocked	Running	$Process_0$ engellendi,
5	Blocked	Running	so $Process_1$ Çalışır
6	Blocked	Running	
7	Ready	Running	I/O Bitti
8	Ready	Running	$Process_1$ şimdi bitti
9	Running	–	
10	Running	–	$Process_0$ şimdi bitti

Şekil 4.4: **İşlem Durumunu İzleme: CPU ve I/O**

Bu sonraki örnekte, ilk işlem bir süre çalıştıktan sonra bir I/O verir. Bu noktada, işlem engellenir ve diğer sürece çalışma şansı verilir. Şekil 4.4'te bu senaryonun izi gösterilmektedir.

Daha spesifik olarak, $İşlem_0$ bir I/O başlatır ve tamamlanmasını beklerken engellenir; işlemler engellenir, örneğin, bir diskten okurken veya bir ağdan bir paket beklerken engellenir. İşletim sistemi, $İşlem_{0'ın}$ CPU'yu kullanmadığını kabul eder ve İşlem 1'i çalıştırmaya başlar. İşlem $_1$ çalışırken, I/O tamamlanır ve $İşlem_{0'ı}$ tekrar hazır konuma getirir. Son olarak, İşlem $_1$ biter ve İşlem $_0$ çalışır ve sonra tamamlanır.

Bu basit örnekte bile işletim sisteminin vermesi gereken birçok karar olduğunu unutmayın. İlk olarak, Sistem $Süreç_{1'i}$ çalıştırmaya karar vermek zorundaydı, $Süreç_0$ ise bir I/O yayınladı; bunu yapmak, CPU'yu meşgul tutarak kaynak kullanımını geliştirir.İkincisi, sistem I/O tamamlandığında $Proses_{0'a}$ geri dönmemeye karar verdi; bunun iyi bir sapma olup olmadığı belli değil. Ne düşünüyorsun? Bu tür kararlar, gelecekte birkaç bölümde tartışacağımız bir konu olan işletim sistemi **zamanlayıcısı (scheduler)** tarafından verilir.

4.2 VERİ YAPILARI

İşletim sistemi bir programdır ve herhangi bir program gibi, çeşitli ilgili bilgi parçalarını izleyen bazı önemli veri yapılarına sahiptir. Örneğin, her işlemin durumunu izlemek için, işletim sistemi muhtemelen hazır olan tüm işlemler için bir tür **süreç listesi (process list)** ve hangi işlemin şu anda çalıştığını izlemek için bazı ek bilgiler tutacaktır. İşletim sistemi ayrıca bir şekilde engellenen işlemleri izlemelidir; Bir I/O olayı tamamlandığında, işletim sistemi doğru işlemi uyandırdığından ve yeniden çalışmaya hazır olduğundan emin olmalıdır.Şekil 4.5,bir işletim sisteminin xv6 çekirdeğindeki [CK + 08] her işlem hakkında ne tür bilgileri izlemesi gerektiğini göstermektedir.Benzer süreç yapıları Linux, Mac OS X veya Windows gibi "gerçek" işletim sistemlerinde de mevcuttur; Onlara bakın ve ne kadar karmaşık olduklarını görün.

Şekilden, işletim sisteminin bir süreç hakkında izlediği birkaç önemli bilgi parçasını görebilirsiniz . **Kayıt bağlamı (register context)** geçerli olacaktır,

```
// xv6 kayıtları kaydedecek ve geri yükleyecektir
// Bir işlem yapısı bağlamını durdurmak ve daha sonra yeniden
başlatmak için{
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// Bir sürecin içinde olabileceği farklı durumlar
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// xv6 her işlemle ilgili bilgileri izler
// kayıt bağlamı ve durumu 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 Kimliği
  struct proc *parent;        // Ebeveyn süreci
  void *chan;                 // If !zero, uyumak chan
  int killed;                 // If !zero, öldürüldü
  struct file *ofile[NOFILE]; // Dosyaları açma
  struct inode *cwd;          // Geçerli dizin
  struct context context;     // İşlemi çalıştırmak için buraya geçin
  struct trapframe *tf;       // Tuzak çerçevesi
                              // akım kesintisi
};
```

Şekil 4.5: **xv6 Proc Yapısı**

durdurulmuş bir işlem için, kayıtlarının içeriği. Bir işlem durdurulduğunda, kayıtları bu bellek konumuna kaydedilir ;Bu kayıt isterlerini 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 .

Ayrıca, şekilden, koşmanın, hazır olmanın ve engellenmenin ötesinde, bir işlemin içinde olabileceği başka durumlar da olduğunu görebilirsiniz. Bazen bir sistem, oluşturulduğu sırada işlemin içinde bulunduğu bir **başlangıç(initial)** durumuna sahip olacaktır.Ayrıca, bir süreç çıktığı **son (final)** bir duruma yerleştirilebilir , ancak

BİR KENARA: VERİ YAPISI-SÜREÇ LİSTESİ

İşletim sistemleri, bu notlarda tartışacağımız çeşitli önemli **veri yapılarıyla (data structures)** doludur. **Süreç listesi (process list)** (**görev listesi (task list)** olarak da adlandırılır) bu tür ilk yapıdır. Daha basit olanlardan biridir, ancak kesinlikle aynı anda birden fazla programı çalıştırma yeteneğine sahip herhangi bir işletim sistemi, sistemdeki tüm çalışan programları takip etmek için bu yapıya benzer bir şeye sahip olacaktır . Bazen insanlar , bir süreç hakkında bilgi depolayan bireysel yapıya bir **Süreç Kontrol Bloğu (Process Control Block) (PCB)**, her süreç hakkında bilgi içeren bir C yapısı hakkında konuşmanın süslü bir yolu (bazen **süreç tanımlayıcısı (process descriptor)** olarak da adlandırılır) .

henüz temizlenmemiştir (UNIX tabanlı sistemlerde buna **zombi** durumu[1] denir). Bu son durum, diğer işlemlere izin verdiği için yararlı olabilir.(genellikle işlemi oluşturan **üst öğe(parent)**) İşlemin dönüş kodunu incelemek ve henüz bitmiş işlemin başarıyla yürütülüp yürütülmediğini görmek için(Genellikle, programlar bir görevi başarıyla yerine getirdiklerinde UNIX tabanlı sistemlerde sıfır döndürür ve aksi takdirde sıfır olmayan). Cezalandırıldığında, ebeveyn son bir çağrı yapacaktır. (e.g., `wait()`) Çocuğun tamamlanmasını beklemek ve ayrıca işletim sistemine, şu anda soyu tükenmiş sürece atıfta bulunan ilgili veri yapılarını temizleyebileceğini belirtmek.

4.3 ÖZET

İşletim sisteminin en temel soyutlamasını tanıttık: süreç. Oldukça basit bir şekilde çalışan bir program olarak görülür. Bu kavramsal görüşü akılda tutarak, şimdi nitty-gritty'ye geçeceğiz: süreçleri uygulamak için gereken düşük seviyeli mekanizmalar ve bunları akıllı bir şekilde programlamak için gereken üst düzey politikalar . Mekanik anizmleri ve politikaları birleştirerek, işleyen bir sistemin CPU'yu nasıl sanallaştırdığına dair anlayışımızı geliştireceğiz .

[1] Evet, zombi devleti. Tıpkı gerçek zombiler gibi, bu zombilerin öldürülmesi nispeten kolaydır . Bununla birlikte, genellikle farklı teknikler önerilir..

BİR KENARA: ANAHTAR SÜREÇ TERİMLERİ

- **Süreç (process),**çalışan bir programın ana işletim sistemi soyutlamasıdır. Zamanın herhangi bir noktasında, süreç durumuyla tanımlanabilir: **adres alanındaki (address space)** belleğin bileşenleri , CPU kayıtlarının içeriği (diğerlerinin yanı sıra **program sayacı (program counter)** ve **yığın (stack) işaretçisi** dahil) ve G/Ç hakkındaki bilgiler (okunabilen veya yazılabilen açık dosyalar gibi).
- **İşlem API'si(process API)**, programların işlemlerle ilgili yapabileceği çağrılardan oluşur. Tipik olarak, bu oluşturma, imha ve diğer yararlı çağrıları içerir
- İşlemler, çalıştırılan, çalışmaya hazır ve engellenmiş dahil olmak üzere birçok farklı **işlem durumundan (process states)** birinde bulunur. Farklı etkinlikler (e.g., zamanlanmış veya programlanmış hale getirilmiş veya bir I/O'nin tamamlanmasını beklemek) bu durumlardan birinden diğerine bir süreç aktarmak
- **İşlem (Süreç) listesi (process list)**, sistemdeki tüm işlemler hakkında bilgi içerir. Her giriş, bazen **işlem kontrol bloğu (process control block) (PCB)** olarak adlandırılan şeyde bulunur ; bu, aslında yalnızca belirli bir süreç hakkında bilgi içeren bir yapıdır. .

References

[BH70] "The Nucleus of a Multiprogramming System" by Per Brinch Hansen. Communications 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 operating 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 (Simülasyon)

Bu program, `process-run.py`, programlar çalıştıkça işlem durumlarının nasıl değiştiğini görmenizi sağlar ve CPU'yu kullanır (e.g., ekleme talimatı gerçekleştirme) veya I/O yapmak (e.g., diske istek gönderme ve diskin tamamlanmasını bekleme). Ayrıntılar için README'ye bakın.

Sorular

1. *process-run.py*'i aşağıdaki bayraklarla çalıştırın : *-l 5:100,5:100*. CPU kullanımı ne olmalıdır? (e.g., CPU'nun kullanımda olduğu sürenin yüzdesi?) Bunu neden biliyorsun? Haklı olup olmadığınızı görmek için -c ve -p bayraklarını kullanın .

" ./process-run.py -l 5:100 " Süreci çalıştırıldığında aşağıdaki ekran görüntüsünü alıyoruz.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
5:100
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)
```

Yukardaki ekran görüntüsündeki "5:100" ifadesinin anlamı bu süreç 5 talimattan oluşur ve her talimatın bir cpu talimatı olma olasılığı %100 dür. (IO YAPMAZ)

" ./process-run.py -l 5:100 -c " komutu çalıştırılınca aşağıdaki ekran görüntüsünü elde ederiz.
-c bayrağı kullanıldığında ise süreç içerisinde neler olduğunu görebiliriz.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
5:100 -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
```

Yukardaki ekran görüntüsünde 5 komut boyunca cpu kullanılmaktadır ve ekran görüntüsünde görüldüğü gibi I/O işlemi yapılmamıştır.

./process-run.py -l 5:100,5:100 çalıştırıldığında :
2 süreç olarak çalıştırdığımızda aşağıdaki ekran görüntüsü oluşmaktadır.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
5:100,5:100
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)
```

./process-run.py -l 5:100,5:100 -c çalıştırıldığında aşağıdaki ekran görüntüsü oluştu.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
5:100,5:100 -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
```

* PID 0 olan işlem süreç önce çalışırken (RUN) PID 1 olan işlem Ready (hazır) bekledi .PID 0 olan işlem bitince(done) PID 1 olan işlem çalışıtı ve böylelikle 2 süreçte tamamlandı.

./process-run.py -l 5:100,5:100 -c -p çalıştırıldığında aşağıdaki ekran görüntüsü oluştu.Bazı istatistikleri bize verdi.(sadece -p bayrağını çalıştırır isek istediğimiz görüntüyü alamayız)

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
5:100,5:100 -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%)
```

* Sürecin toplam çalışma zamanı 10(clock ticks) dır.
* CPU %100 meşgul iken IO %0 meşgul dur.

2. Şimdi şu bayraklarla çalıştırın: *./process-run.py -l 4:100,1:0*. Bu bayraklar, 4 talimat (hepsi CPU'yu kullanmak için) içeren bir işlem ve sadece bir I/O yayınlayan ve bunun tamamlanmasını bekleyen bir işlem belirtir. Her iki işlemin tamamlanması ne kadar sürer? Haklı olup olmadığınızı öğrenmek için *-c* ve *-p* bayraklarını kullanın.

./process-run.py -l 4:100,1:0 çalıştırıldığında :
2 süreç olarak çalıştırdığımızda aşağıdaki ekran görüntüsü oluşmaktadır.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 p
rocess-run.py -l 4:100,1:0
Produce a trace of what would happen when you run these proces
ses:
Process 0
  cpu
  cpu
  cpu
  cpu

Process 1
  io
  io_done

Important behaviors:
  System will switch when the current process is FINISHED or I
SSUES AN IO
  After IOs, the process issuing the IO will run LATER (when i
t is its turn)
```

* Yukardaki ekran görüntüsündeki "4:100,1:0" ifadesinin anlamı 2 süreçten oluşur ve ilk süreç için 4 talimattan oluşur ve her talimatın bir cpu talimatı olma olasılığı %100 dür diğer süreç içinse 1 talimattan oluşur ve her talimatın bir cpu talimatı olma olasılığı %0 dır.(IO çalıştırır)

./process-run.py -l 4:100,1:0 -c -p çalıştırıldığında aşağıdaki ekran görüntüsü oluştu.
* -c bayrağı kullanıldığında ise süreç içerisinde neler olduğunu görebiliriz.
* -p bayrağı kullanıldığında ise bize bazı istatistikleri verir .

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
4:100,1:0 -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            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%)
```

*Yukardaki ekran görüntüsünde PID 0 olan işlem önce çalıştı ve CPU'yu meşgul etti o sırada PID 1 olan işlem hazırda bekledi ve PID 0 olan işlem bitince PID 1 olan işlem io'yu çalıştırdı.(ve bunu işlemek için CPU' kullandı)süreç bloklanmış duruma geçti PID1 olan IO'yu meşgul ediyor ve o sırada CPU boştadır son olarak IO tamamlanmasını işlemek için CPU'yu kullandı.
*CPU meşgululiyeti %54.55 iken IO meşgululiyeti %45.45 dir.toplam çalışma zamanı 11 (clock ticks) dir.

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

/.process-run.py -l 1:0,4:100 süreci çalıştırıldığında aşağıdaki ekran görüntüsü oluşuyor.
Sadece süreçlerin(process) yeri değişiyor.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
1:0,4:100
Produce a trace of what would happen when you run these processes:
Process 0
  io
  io_done

Process 1
  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)
```

/.process-run.py -l 1:0,4:100 -c -p süreci çalıştırıldığında aşağıdaki ekran görüntüsü oluşuyor.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
1:0,4:100 -c -p
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%)
```

* PID 0 olan işlem io çalıştırırken PID1 olan işlem hazırda bekledi.PID0 olan işlem bloklanınca PID1 olan işlem çalışmaya başladı ve cpu ile io birlikte meşgul edildi.PID1 olan işlem 6. Zamnda bitti. PID 0 7.zamanda bitti ve IO' yu işlemek için cpu'yu kullandı.
*CPU meşguliyeti 6 clock ticks , IO meşguliyeti 5 clock ticks dir fakat toplam 7 clock ticks sürmüştür sebebi ise cpu ile ıo aynı anda meşgul edildiği zamanlar olmuştur.

NOT: I/O, başlatma ve tamamlamayı işlemek için cpu eylemi gerçekleştirir.

4. Şimdi diğer bayraklardan bazılarını keşfedeceğiz.Önemli bir bayrak -S, bir süreç bir I/O verdiğinde sistemin nasıl tepki verdiğini belirler.Bayrak SWITCH_ON_END olarak ayarlandığında, sistem I/O yaparken başka bir işleme GEÇMEYECEKTİR, bunun yerine işlem tamamen bitene kadar bekleyecektir. Aşağıdaki iki işlemi çalıştırdığınızda hangi hap kalemleri (-l 1:0,4:100 -c -S SWITCH_ONE_END), biri I/O ve diğeri CPU çalışması mı yapıyor?

/.process-run.py -l 1:0,4:100 -c -p -S SWITCH_ON_END süreci çalıştırıldığında aşağıdaki ekran görüntüsü oluşuyor.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
1:0,4:100 -c -p -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

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

* -S Bayrağını SWITCH ON END olarak ayarlanınca Sistem I/O yaparken başka bir işleme geçmiyor bunun yerine işlem tamamen bitene kadar bekliyor.

*CPU meşguliyeti 6 zaman , IO meşguliyeti 5 zamandır. Toplam çalışma zamanı tekrardan 11 oldu eğer -S SWITCH_ON_END kullanmasaydık toplam çalışma zamanı bir önceki sorudaki gibi 7 olurdu.

5. Şimdi, aynı işlemleri çalıştırın, ancak anahtarlama davranışı, biri I/O için BEKLEDİĞİNDE başka bir işleme geçecek şekilde ayarlanmış olsun (*-l 1:0,4:100 -c -S SWITCH_ON_IO*). Şimdi ne olacak? Haklı olduğunuzu doğrulamak için *-c* ve *-p* kullanın.

/.process-run.py -l 1:0,4:100 -c -p -S SWITCH_ON_IO işlemi çalıştırılınca aşağıdaki ekran görüntüsü oluşuyor -S SWITCH_ON_IO işlemi çalıştırılınca görüldüğü gibi IO yaparken başka işleme geçiyor.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l
1:0,4:100 -c -p -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

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

6. Bir Bir diğer önemli davranış da bir I/O tamamlandığında ne yapılacağıdır.-I IO_RUN_LATER ile, bir I/O tamamlandığında, işlem hemen çalıştırılması gerekmez; bunun yerine, her ne o sırada çalışıyordu, çalışmaya devam ediyor. Ne zaman olur bu süreç kombinasyonunu çalıştırabilir mi? (Çalıştır./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 çalıştırıldığında aşağıdaki ekran görüntüsü oluşur.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_O
N_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%)
```

* -I IO_RUN_LATER ile, bir I/O tamamlandığında, işlem hemen çalıştırılması gerekmez; bunun yerine,o sırada ne çalışıyorsa çalışmaya devam eder (eğer IO_RUN_IMMEDIATE kullanılsaydı daha kısa sürede biterdi ve kaynaklar daha fazla meşgul olurdu.)
* kaynakların daha iyi kullanılması için tüm cihazları meşgul tutmak isteriz. Bu işlemde görüldüğü gibi CPU meşguliyeti %67.74, IO meşguliyeti %48,39 dur. Toplam çalışma süresi 31 (clock ticks)zamandır.

7. Şimdi aynı işlemleri -I IO_RUN_IMMEDIATE ayarıyla çalıştırın, bu da I/O'yu veren işlemi hemen çalıştırır. Nasıl olur da bu davranış farklı mı? Yeni tamamlanmış bir süreci çalıştırmak neden tekrar bir I/O iyi bir fikir olabilir mi?

*./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_IMMEDIATE -c -p çalıştırıldığında aşağıdaki ekran görüntüsü oluşur.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_O
N_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%)
```

* "IO_RUN_IMMEDIATE" I/O'yu veren işlemi hemen çalıştırır bu sayede süreçleri daha kısa sürede bitirir.
*kaynakların daha iyi kullanılması için tüm cihazları meşgul tutmak isteriz burada önceki sorudan daha iyi şekilde kullanıldığını görüyoruz.
*CPU %100 meşgul durumda , IO %71.43 meşgul durumdadır
Sürecin toplam çalışma süreci 21(clock ticks) zamandır.

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. İzlemenin nasıl sonuçlanacağını tahmin edip edemeyeceğinize bakın.-I IO_RUN_IMMEDIATE ile -I IO_RUN_LATER bayraklarını kullandığınızda ne olur? -S SWITCH_ON_IO ,-S SWITCH_ON_END kullandığınızda ne olur?

*./ process-run.py -s 1 -l 3:50,3:50 -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -s 1 -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:cpu             1             1
  4         BLOCKED       RUN:cpu             1             1
  5         BLOCKED       RUN:cpu             1             1
  6         BLOCKED          DONE                           1
  7         BLOCKED          DONE                           1
  8*    RUN:io_done          DONE             1
  9          RUN:io          DONE             1
 10         BLOCKED          DONE                           1
 11         BLOCKED          DONE                           1
 12         BLOCKED          DONE                           1
 13         BLOCKED          DONE                           1
 14         BLOCKED          DONE                           1
 15*    RUN:io_done          DONE             1

Stats: Total Time 15
Stats: CPU Busy 8 (53.33%)
Stats: IO Busy  10 (66.67%)
```

*./ process-run.py -s 2 -l 3:50,3:50 -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -s 2 -l 3:50,3:50 -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1          RUN:io         READY             1
  2         BLOCKED       RUN:cpu             1             1
  3         BLOCKED        RUN:io             1             1
  4         BLOCKED       BLOCKED                           2
  5         BLOCKED       BLOCKED                           2
  6         BLOCKED       BLOCKED                           2
  7*    RUN:io_done       BLOCKED             1             1
  8          RUN:io       BLOCKED             1             1
  9*        BLOCKED   RUN:io_done             1             1
 10         BLOCKED        RUN:io             1             1
 11         BLOCKED       BLOCKED                           2
 12         BLOCKED       BLOCKED                           2
 13         BLOCKED       BLOCKED                           2
 14*    RUN:io_done       BLOCKED             1             1
 15         RUN:cpu       BLOCKED             1             1
 16*           DONE   RUN:io_done             1

Stats: Total Time 16
Stats: CPU Busy 10 (62.50%)
Stats: IO Busy  14 (87.50%)
```

*./ process-run.py -s 3 -l 3:50,3:50 -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 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%)
```

Yukardaki çıktılara baktığımızda kaynakları daha iyi kullanan "process-run.py -s 2 -l 3:50,3:50 -c -p" dir. Çünkü kaynakların daha iyi kullanılması için tüm cihazları meşgul tutmak isteriz. En hızlı bitiren ise "./ process-run.py -s 1 -l 3:50,3:50" işlemidir.

*./ process-run.py -s 1 -l 3:50,3:50 -I IO_RUN_IMMEDIATE -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -s 1 -l 3:50,3:50 -I IO_RUN_IMMEDIAT
E -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2         RUN:io         READY             1
  3        BLOCKED       RUN:cpu             1             1
  4        BLOCKED       RUN:cpu             1             1
  5        BLOCKED       RUN:cpu             1             1
  6        BLOCKED          DONE                           1
  7        BLOCKED          DONE                           1
  8*   RUN:io_done          DONE             1
  9         RUN:io          DONE             1
 10        BLOCKED          DONE                           1
 11        BLOCKED          DONE                           1
 12        BLOCKED          DONE                           1
 13        BLOCKED          DONE                           1
 14        BLOCKED          DONE                           1
 15*   RUN:io_done          DONE             1

Stats: Total Time 15
Stats: CPU Busy 8 (53.33%)
Stats: IO Busy  10 (66.67%)
```

*./ process-run.py -s 1 -l 3:50,3:50 -I IO_RUN_LATER -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -s 1 -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:cpu             1             1
  4         BLOCKED       RUN:cpu             1             1
  5         BLOCKED       RUN:cpu             1             1
  6         BLOCKED          DONE                           1
  7         BLOCKED          DONE                           1
  8*   RUN:io_done          DONE             1
  9          RUN:io          DONE             1
 10         BLOCKED          DONE                           1
 11         BLOCKED          DONE                           1
 12         BLOCKED          DONE                           1
 13         BLOCKED          DONE                           1
 14         BLOCKED          DONE                           1
 15*   RUN:io_done          DONE             1

Stats: Total Time 15
Stats: CPU Busy 8 (53.33%)
Stats: IO Busy  10 (66.67%)
```

*./ process-run.py -s 1 -l 3:50,3:50 –S SWITCH_ON_END -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.
CPU meşguliyeti azalırken yüzdesel olarak,IO meşguliyeti arttı.
Toplam işlem(clock ticks) süresi arttı.
(-S Bayrağını SWITCH_ON_END olarak ayarlanınca Sistem I/O yaparken başka bir işleme geçmiyor bunun yerine işlem tamamen bitene kadar bekliyor.)

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -s 1 -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:io         READY             1
 10         BLOCKED         READY                           1
 11         BLOCKED         READY                           1
 12         BLOCKED         READY                           1
 13         BLOCKED         READY                           1
 14         BLOCKED         READY                           1
 15*   RUN:io_done         READY             1
 16            DONE       RUN:cpu             1
 17            DONE       RUN:cpu             1
 18            DONE       RUN:cpu             1

Stats: Total Time 18
Stats: CPU Busy 8 (44.44%)
Stats: IO Busy  10 (55.56%)
```

./ process-run.py -s 1 -l 3:50,3:50 –S SWITCH_ON_IO -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.
–S SWITCH_ON_IO bayrağı kullanılınca değişim olmuyor çünkü işlemi çalıştırılınca görüldüğü gibi IO yaparken başka işleme geçiyor.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -s 1 -l 3:50,3:50 -S SWITCH_ON_IO -c
 -p
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:cpu         READY             1
  2          RUN:io         READY             1
  3         BLOCKED       RUN:cpu             1             1
  4         BLOCKED       RUN:cpu             1             1
  5         BLOCKED       RUN:cpu             1             1
  6         BLOCKED          DONE                           1
  7         BLOCKED          DONE                           1
  8*   RUN:io_done          DONE             1
  9          RUN:io          DONE             1
 10         BLOCKED          DONE                           1
 11         BLOCKED          DONE                           1
 12         BLOCKED          DONE                           1
 13         BLOCKED          DONE                           1
 14         BLOCKED          DONE                           1
 15*   RUN:io_done          DONE             1

Stats: Total Time 15
Stats: CPU Busy 8 (53.33%)
Stats: IO Busy  10 (66.67%)
```

./ process-run.py -s 1 -l 3:50,3:50 –S SWITCH_ON_IO -I IO_RUN_LATER -c -p işlem çalıştırıldığında aşağıdaki ekran görüntüsü çıktı.

```
use@ubuntu:~/Desktop/ostep/ostep-homework/cpu-intro$ python3 process-run.py -s 1 -l 3:50,3:50 -S SWITCH_ON_IO -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:cpu             1             1
  4         BLOCKED       RUN:cpu             1             1
  5         BLOCKED       RUN:cpu             1             1
  6         BLOCKED          DONE                           1
  7         BLOCKED          DONE                           1
  8*   RUN:io_done          DONE             1
  9          RUN:io          DONE             1
 10         BLOCKED          DONE                           1
 11         BLOCKED          DONE                           1
 12         BLOCKED          DONE                           1
 13         BLOCKED          DONE                           1
 14         BLOCKED          DONE                           1
 15*   RUN:io_done          DONE             1

Stats: Total Time 15
Stats: CPU Busy 8 (53.33%)
Stats: IO Busy  10 (66.67%)
```