4

SOYUTLAMA:SÜREÇ

Bu bölümde, işletim sisteminin kullanıcılara sağladığı en temel soyutlamalardan birini tartışıyoruz: **procces(süreç).** Gayri resmi olarak bir sürecin tanımı oldukça basittir: **runing program (çalışan bir programdır)** [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 yararlı bir şeye dönüştüren işletim sistemidir.

Görünüşe göre kişi aynı anda birden fazla programı çalıştırmak istiyor; örneğin, bir web tarayıcısı, posta programı, oyun, müzik çalar vb. çalıştırmak isteyebileceğiniz masaüstü veya dizüstü bilgisayarınızı düşünün.

Aslında, tipik bir sistem görünüşte aynı anda onlarca hatta yüzlerce işlem yürütüyor olabilir. Bunu yapmak, sistemin kullanımını kolaylaştırır, çünkü bir CPU'nun kullanılabilir olup olmadığıyla asla ilgilenmeniz gerekmez; biri sadece programları çalıştırır. Dolayısıyla meydan okumamız:

SORUNUN DOĞRUSU:
BİRÇOK İŞLEMCİNİN YANLIŞI NASIL SAĞLANIR?
Kullanılabilir yalnızca birkaç fiziksel CPU olmasına rağmen, işletim sistemi, söz konusu CPU'ların neredeyse sonsuz bir arzının yanılsamasını sağlıyor mu?

İşletim sistemi, CPU'yu **virtualizing (sanallaştırarak)** bu yanılsamayı yaratır. İşletim sistemi, bir işlemi çalıştırıp ardından durdurup diğerini çalıştırarak, aslında yalnızca bir fiziksel CPU (veya birkaç tane) varken birçok sanal CPU'nun var olduğu yanılsamasını destekleyebilir. CPU'nun zaman paylaşımı olarak bilinen bu temel teknik, kullanıcıların istedikleri kadar eşzamanlı işlemi çalıştırmalarına olanak tanır; CPU (lar)ın paylaşılması gerekiyorsa her biri daha yavaş çalışacağından potansiyel maliyet performanstır. CPU'nun sanallaştırılmasını uygulamak ve iyi bir şekilde uygulamak için, işletim sisteminin hem bazı düşük seviyeli makinelere hem de bazı yüksek seviyeli zekaya ihtiyacı olacaktır. Düşük seviyeli makine **mechanisms(mekanizmaları)** diyoruz; mekanizmalar, gerekli bir işlevsellik parçasını uygulayan düşük seviyeli yöntemler veya protokollerdir. Örneğin, bir **context (bağlamı)** nasıl uygulayacağımızı daha sonra öğreneceğiz.

İPUCU: ZAMAN PAYLAŞIMINI (VE ALAN PAYLAŞIMINI) KULLANIN
Time sharing (Zaman paylaşımı) , bir işletim sistemi tarafından bir kaynağı paylaşmak için kullanılan temel bir tekniktir. Kaynağın kısa bir süre bir varlık tarafından ve daha sonra kısa 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) paylaştırıldığı **space sharing (alan paylaşımıdır).** Örneğin, disk alanı doğal olarak paylaşılan bir alan kaynağıdır; Bir blok bir dosyaya atandığında, kullanıcı orijinal dosyayı silene kadar normal olarak başka bir dosyaya imzalanmaz.

işletim sistemine bir programı çalıştırmayı durdurma ve belirli bir CPU'da başka bir programı çalıştırma yeteneği veren anahtar; bu **time-sharing (zaman paylaşım)** mekanizması tüm modern işletim sistemlerinde kullanılır.

Bu mekanizmaların üzerinde, işletim sistemindeki istihbaratın bir kısmı politikalar şeklinde bulunur. **policies (Politikalar),** işletim sistemi içinde bir tür karar vermeye yönelik algoritmalardır. Örneğin, bir CPU üzerinde çalıştırılabilecek birkaç olası program verildiğinde, işletim sistemi hangi programı çalıştırmalı? OS'deki bir **scheduling policy (zamanlama politikası),** bu kararı muhtemelen geçmiş bilgileri (örneğin, son dakikada hangi program daha çok çalıştı?), iş yükü bilgisini (örneğin, ne tür programlar çalıştırılıyor) ve performans ölçütlerini (örneğin, karar vermek için sistem etkileşimli performans için mi yoksa verim için mi optimize ediyor?)
.

4.1 Soyutlama: Süreç

Çalışan bir programın işletim sistemi tarafından sağlanan soyutlama, **process(süreç)** olarak adlandıracağımız bir şeydir. Yukarıda söylediğimiz gibi, bir süreç basitçe çalışan bir programdır; herhangi bir anda, bir süreci, yürütmesi sırasında eriştiği veya etkilediği sistemin farklı parçalarının bir Bu makine durumunun bir parçasını oluşturan bazı özellikle özel kayıtlar olduğuna dikkat edin. Örneğin, program sayacı (PC) (bazen komut işaretçisi veya IP olarak adlandırılır) bize programın hangi komutunun daha sonra yürütüleceğini söyler; benzer şekilde bir yığın işaretçisi ve ilişkili çerçeve envanterini alarak özetleyebiliriz.

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

Bir işlemi içeren makine durumunun bariz bir bileşeni, belleğidir. Talimatlar hafızadadır; çalışan programın okuyup yazdığı veriler de bellekte durur. Bu nedenle, işlemin adresleyebileceği bellek (**address space (adres alanı)** olarak adlandırılır) işlemin bir parçasıdır.

Ayrıca sürecin makine durumunun bir parçası da kayıtlardır; birçok talimat, kayıtları açıkça okur veya günceller ve bu nedenle, sürecin yürütülmesi için açıkça önemlidirler. Bu makine durumunun bir parçasını oluşturan bazı özellikle özel kayıtlar olduğuna dikkat edin. Örneğin**, program counter (program sayacı) (PC)** (bazen **instruction pointer(komut işaretçisi)** veya **IP** olarak adlandırılır) bize programın hangi komutunun daha sonra yürütüleceğini söyler; benzer şekilde bir stack **pointer(yığın işaretçisi)** ve ilişkili **frame(çerçeve)**

İPUCU: AYRI BİR POLİTİKA VE MEKANİZMA
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? Politika, hangi sorunun cevabını verir; örneğin, işletim sistemi şu anda hangi işlemi çalıştırmalı? İkisini ayırmak, mekanizmayı yeniden düşünmek zorunda kalmadan politikaların kolayca değiştirilmesine izin verir ve bu nedenle, genel bir yazılım tasarım ilkesi olan bir **modularity(modülerlik)** biçimidir.

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

.2 İşlem API'sı

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:** 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 Etme:** Süreç oluşturmak için bir arayüz olduğu gibi, 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 işlemi durdurmak için bir arayüz oldukça kullanışlıdır.
- **Bekle:** Bazen bir işlemin çalışmayı durdurmasını beklemek yararlıdır; bu nedenle genellikle bir tür bekleme arabirimi sağlanır.
- **Çeşitli Kontrol:** Bir işlemi sonlandırmak 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 sunar.
- **Durum:** Genellikle bir süreç hakkında, ne kadar süredir çalıştığı veya hangi durumda olduğu gibi bazı durum bilgilerini almak için arayüzler vardır.

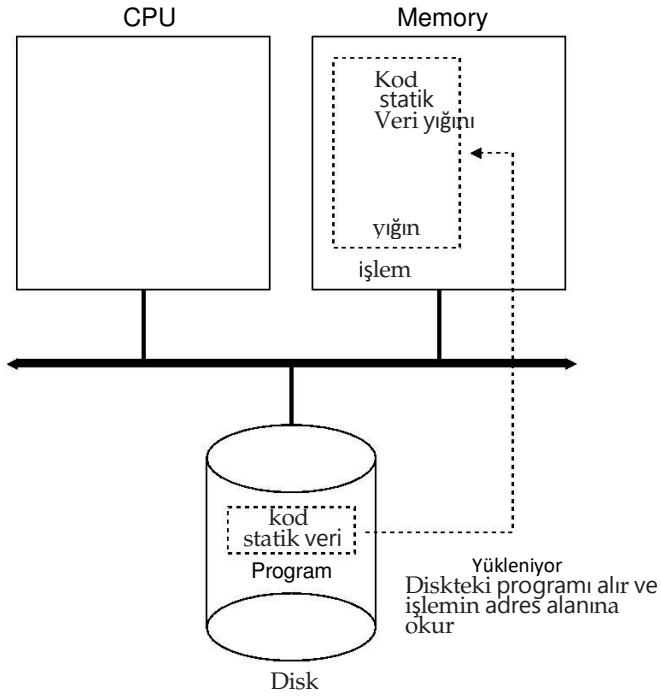

Figure 4.1: **Yükleme: Programdan Işleme**

4.3 Süreç Oluşturma: Biraz Daha Ayrıntı

Maskesini biraz kaldırmamız gereken bir muamma, programların süreçlere nasıl 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 **load(yüklemektir).** Programlar başlangıçta bir tür yürütülebilir biçimde **disk(diskte)** (veya bazı modern sistemlerde **flash-based (flash tabanlı) SSD'lerde)** 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).

InErken (veya basit) işletim sistemlerinde, yükleme işlemi **eagerly (hevesle),** yani programı çalıştırmadan önce birdenbire yapılır; modern işletim sistemleri, işlemi **lazily(tembel)** bir şekilde, 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 şey anlamanız gerekir.

paging (sayfalama) ve **swapping (değiş tokuş)** mekanizması, 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.**run-time stack (Programın çalışma zamanı yığını)** (veya sadece **stack(yığın))** için bir miktar bellek ayrılmalıdır

Muhtemelen zaten bildiğiniz gibi, C programları **heap(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 ayırabilir. C programlarında yığın, açıkça talep edilen dinamik olarak tahsis edilmiş veriler için kullanılır; programlar malloc () öğesini çağırarak böyle bir alanı talep eder ve free() öğesini 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 işleme daha fazla bellek ayırabilir.

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

Kodu ve statik verileri belleğe yükleyerek, bir yığın oluşturup başlatarak ve G/ Ç 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

4.4 Süreç Durumları

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

- **Runing (Çalışıyor):** Çalışıyor durumda, bir işlemci üzerinden bir işlem çalışmaktadır. Bu, talimatları uyguladığı anlamına gelir.
- **Ready (Hazır):** Hazır durumunda, bir işlem çalışmaya hazırdır, ancak işletim sistemi herhangi bir nedenle bu anda işlemi çalıştırmamayı seçmiştir.

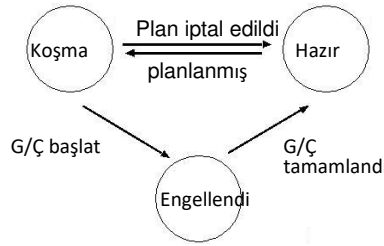

Figure 4.2: **S**üreç durum geçişleri

- Engellendi: Engellenmiş durumda, bir süreç, başka bir olay gerçekleşene kadar çalışmaya hazır olmayan bir tür işlem gerçekleştirmiştir. Yaygın bir örnek: Bir işlem bir diske G/Ç isteği başlattığında bloke olur ve bu nedenle başka bir işlem işlemciyi kullanabilir.

Bu durumları bir grafikte haritalayacak olsaydık, Şekil 4.2'deki diyagrama ulaşırdık. Diyagramda görebileceğiniz gibi, işletim sisteminin takdirine bağlı olarak bir işlem hazır ve çalışan durumlar arasında taşınabilir.
Hazır durumdan çalışır duruma geçmek, sürecin **scheduled(planlandığı)** anlamına gelir ; çalışır durumdayken hazır durumuna geçmek, işlemin zamanlamasının **descheduled(iptal edildiği)** anlamına gelir . Bir işlem bloke edildiğinde (örneğin, bir G/Ç işlemini başlatarak), işletim sistemi onu bir olay meydana gelene kadar (örneğin, G/Ç tamamlanması) olduğu gibi tutacaktır; bu noktada, süreç yeniden hazır duruma geçer (ve işletim sistemi buna karar verirse potansiyel olarak hemen yeniden çalışır).

İki sürecin bu durumlardan bazılarından nasıl geçiş yapabileceğine dair bir örneğe bakalım. İlk olarak, her biri yalnızca CPU kullanan (G/Ç yapmazlar) çalışan iki işlem hayal edin. Bu durumda, her sürecin durumunun bir izi şöyle görünebilir (Şekil 4.3).

Time	Process0	**Process**1	**Notes**
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process1 now done

Figure 4.3: **Tracing Process State: CPU Only**

Time	Process$_0$	Process$_1$	Notes
1	Koşan	Hazır	
2	Koşan	Hazır	
3	Koşan	Hazır	işlem0 G/Ç'yi başlatır
4	Engellendi	Koşan	İşlem0 engellendi,
5	Engellendi	Koşan	böylece İşlem1 çalışır
6	Engellendi	Koşam	
7	Hazır	Koşan	G/Ç tamamlandı
8	Hazır	Koşan	İşlem1 şimdi
9	Koşan	–	tamamlandı
10	Koşan	–	Işlem0 şimdi tamamlandı

Figure 4.4: İşlem durumunu izleme**: CPU ve** G/Ç

Sonraki örnekte, ilk işlem bir süre çalıştıktan sonra bir G/Ç yayınlar. Bu noktada, süreç bloke edilir ve diğer sürece çalışma şansı verilir. Şekil 4.4 bu senaryonun izini göstermektedir.

Daha spesifik olarak, İşlem0 bir G/Ç başlatır ve tamamlanmasını beklerken bloke olur; işlemler, örneğin bir diskten okurken veya bir ağdan paket beklerken bloke olur. İşletim Sistemi, İşlem0'in CPU'yu kullanmadığını fark eder ve İşlem1'i çalıştırmaya başlar. Process1 çalışırken, G/Ç tamamlanır ve Process0 tekrar hazır konumuna geçer. Son olarak, İşlem1 biter ve İşlem0 çalışır ve sonra yapılır**.**

Bu basit örnekte bile işletim sisteminin vermesi gereken birçok karar olduğunu unutmayın. İlk olarak, İşlem0 bir G/Ç gönderirken sistemin İşlem1'i çalıştırmaya karar vermesi gerekiyordu ; bunu yapmak, CPU'yu meşgul ederek kaynak kullanımını iyileştirir. İkincisi, sistem G/Ç tamamlandığında İşlem0'a geri dönmemeye karar verdi ; Bunun iyi bir karar olup olmadığı net değil. Ne düşünüyorsun? Bu tür kararlar, gelecekte birkaç bölümde tartışacağımız bir konu olan işletim sistemi**scheduler(zamanlayıcısı)** tarafından verilir.

4.5 Veri Yapıları

İşletim sistemi bir programdır ve herhangi bir program gibi, ilgili çeşitli bilgi parçalarını izleyen bazı önemli veri yapılarına sahiptir. Örneğin, her işlemin durumunu izlemek için, işletim sistemi muhtemelen hazır olan tüm işlemler için bir tür **process list (işlem listesi)** ve o anda hangi işlemin çalıştığını izlemek için bazı ek bilgiler tutacaktır. İşletim sisteminin bir şekilde engellenen işlemleri de izlemesi gerekir; bir G/Ç olayı tamamlandığında, işletim sisteminin doğru işlemi uyandırdığından ve yeniden çalışmaya hazır olduğundan emin olması gerekir.

Şekil 4.5, bir işletim sisteminin xv6 çekirdeğindeki [CK+08] her işlem hakkında ne tür bilgileri izlemesi gerektiğini gösterir. Linux, Mac OS X veya Windows gibi "gerçek" işletim sistemlerinde benzer süreç yapıları bulunur; onlara bakın ve ne kadar karmaşık olduklarını görün. Şekilden, işletim sisteminin bir süreç hakkında izlediği birkaç önemli bilgiyi görebilirsiniz. **Register context (kayıt bağlamı)** bir süre için geçerli olacaktır.

```
 / XV6 kayıtları kaydedecek ve geri yükleyecektir
// Bir işlemi durdurmak ve ardından yeniden
/başlatmak için
struct context {
  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 {KULLANILMAMIŞ, EMBRİYO, SLEEPING,
                 KOŞABİLİR, KOŞUYOR, ZOMBİ};

// xv6'nın her işlem hakkında izlediği bilgiler
// kayıt bağlamı ve durumu dahil
struct proc {
  char *mem;                  // İşlem belleğinin başlangıcı
  uint sz;                    // İşlem belleğinin boyutu
  char *kstack.               // Çekirdek yığınının alt kısmı
                              // bu süreç için
  enum proc_state state.      // İşlem durumu
  int pid;                    // Süreç Kimliği
  struct proc *parent.        // Ebeveyn süreci
  void *chan;                 // If! zero, chan uyuyor
  int killed;                 // If! zero, öldürülmüş
  struct file *ofile[NOFILE]; // Açık dosya
  struct inode *cwd.          // Geçerli dizin
  struct context context;     // İşlemi çalıştırmak için buraya geçin
  struct trapframe *tf.       // Tuzak çerçevesi için
                              // Mevcut kesinti
};
```

Figure 4.5: **xv6 Proc** Yapısı

durdurulan işlem, kayıtlarının içeriği. Bir işlem durdurulduğunda, kayıtları bu bellek konumuna kaydedilecektir; bu kayıtları geri yükleyerek (yani, değerlerini gerçek fiziksel kayıtlara geri koyarak), işletim sistemi süreci çalıştırmaya devam edebilir. **Context switch (bağlam değiştirme)** olarak bilinen bu teknik hakkında daha fazla bilgiyi gelecek bölümlerde öğreneceğiz.

Şekilden, bir işlemin çalışıyor, hazır ve engellenmiş olmanın ötesinde olabileceği başka durumlar olduğunu da görebilirsiniz. Bazen bir sistem, sürecin yaratılırken içinde olduğu bir **initial(başlangıç)** durumuna sahip olabilir. Ayrıca, bir süreç çıktığı **final(nihai)** duruma yerleştirilebilir, ancak

BİR TARAFA: VERİ YAPILARI — SÜREÇ LİSTESİ

Sistemleri, bu notlarda tartışacağımız çeşitli önemli **data structure (veri yapılarıyla)** doludur. **Process list (süreç listesi)** (**task list (görev listesi)** olarak da adlandırılır) bu türden ilk yapıdır. Daha basit olanlardan biridir, ancak aynı anda birden fazla programı çalıştırabilen herhangi bir işletim sisteminde, sistemde çalışan tüm programları takip etmek için kesinlikle bu yapıya benzer bir şey olacaktır. Bazen insanlar bir süreçle ilgili bilgileri depolayan bireysel yapıya **Process Control Block (Proses Kontrol Bloğu) (PCB)** adını verirler; bu, her süreç hakkında bilgi içeren bir C yapısından (bazen **process descriptor (süreç tanımlayıcı)** olarak da adlandırılır) bahsetmenin süslü bir yoludur.

Henüz temizlenmemiştir (UNIX tabanlı sistemlerde buna **zombie(zombi)** durumu1 denir). Bu son durum, diğer süreçlerin (genellikle süreci oluşturan **parent(ebeveyn))** sürecin dönüş kodunu incelemesine ve yeni biten sürecin başarılı bir şekilde yürütülüp yürütülmediğini görmesine izin verdiği için yararlı olabilir (genellikle, programlar UNIX tabanlı sistemlerde sıfır döndürür. bir görevi başarıyla tamamladılar ve aksi halde sıfır olmayanlar). Bittiğinde, ebeveyn, çocuğun tamamlanmasını beklemek için son bir çağrı yapacak (örneğin, wait ()) ve ayrıca OS'ye artık yok olan veri yapılarını temizleyebileceğini bildirecektir. işlem.

4.6 ÖZET

İşletim sisteminin en temel soyutlamasını tanıttık: süreç. Oldukça basit bir şekilde çalışan bir program olarak görülüyor. Bu kavramsal görüşü göz önünde bulundurarak, şimdi asıl konuya geçeceğiz: süreçleri uygulamak için gereken alt düzey mekanizmalar ve bunları akıllı bir şekilde programlamak için gereken üst düzey politikalar. Mekanizmaları ve politikaları birleştirerek, bir işletim sisteminin CPU'yu nasıl sanallaştırdığına dair anlayışımızı geliştireceğiz.

[1] Evet, Zombi durumu. Tıpkı gerçek zombiler gibi, bu zombileri öldürmek de nispeten kolaydır. Ancak, genellikle farklı teknikler önerilir.

Bi TARAF: ANAHATR SÜREÇ ŞARTLARI

- **process (İşlem),** çalışan bir programın ana işletim sistemi soyutlamasıdır. Herhangi bir zamanda süreç, durumuyla açıklanabilir: **address space (adres alanındaki)** belleğin içeriği, CPU kayıtlarının içeriği (diğerlerinin yanı sıra **program counter(program sayacı)** ve **stack pointer(yığın işaretçisi)** dahil) ve G/Ç (okunabilen veya yazılabilen açık dosyalar gibi)

- **Process API (Süreç API'si)** , programların süreçlerle ilgili yapabileceği çağrılardan oluşur. Tipik olarak bu, oluşturma, yok etme ve diğer tam kullanım çağrılarını içerir.

- İşlemler, çalışıyor, çalışmaya hazır ve engellenmiş gibi birçok farklı **process states (işlem durumundan)** birinde bulunur. Farklı olaylar (örneğin, programlanma veya programdan çıkma veya bir G/Ç'nin tamamlanmasını bekleme) bir işlemi bu durumların birinden diğerine geçirir.
- Bir **process list (süreç listesi)** , sistemdeki tüm süreçler hakkında bilgi içerir. Her giriş, bazen yalnızca belirli bir işlem hakkında bilgi içeren bir yapı olan **process control block (işlem kontrol bloğu (PCB))** olarak adlandırılan yerde bulunur.

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 (Simulasyon)

Process-run.py adlı bu program, programlar çalışırken ve CPU'yu kullanırken (örneğin, bir toplama işlemi gerçekleştirin talimat) yada do I/O (Örneğin bi diske istek gönderin ve tamamlanmasını bekleyin). Ayrıntılar için README'ye bakın

SORULAR

1. `process-run.py` bu işaretlerle çalıştırın: `-l 5:100,5:100`. CPU kullanımı ne olmalıdır. (örneğin, CPU kullanımda olduğu süre yüzdesi) Bunu neden biliyorsun? Haklı olup olmadığını görmek için -c ve -p bayraklarını kullan.

-l bayrağı ile cpu listesini görmemize olanak sağladı. 5:100 ise öncelik ve zamanlama ayarı için yazıldı.

```
C:\Users\ulas\Desktop\aa>python3 process-run.py -
Time        PID: 0        PID: 1           CPU
  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%)
```

-c bayrağı ile CPU'nun anlık durumunu görebiliyoruz. PID:0 sürecin ilk 5 timede koşarken son 5 zamanda Done durumuna geçtiği görüldü. İkinci sürecin ise READY durumunda RUN durumuna geçtiği görüldü.
-p bayrağı ile CPU kullanım durumu ölçüldü 5:100 değerleri girdiğimiz için CPU %100 kullanıldı.

2.Şimdi bu bayraklala çalıştır: `/process-run.py -l 4:100,1:0`. Bu bayraklar, 4 yönergeli (tümü CPU'yu kullanmak için) ve yalnızca bir G/Ç 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.

```
C:\Users\ulas\Desktop\aa>python3 process-run.py
Time        PID: 0        PID: 1           CPU
  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
  7            DONE       BLOCKED
  8            DONE       BLOCKED
  9            DONE       BLOCKED
 10            DONE       BLOCKED
 11*           DONE   RUN:io_done            1

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

Python3 process-run.py -l 4:100,1:0 -c -p
Komudu ile 2 farklı iş tanımı yapıldı CPU kullanacak bir işlem ve G/Ç yapacak diğer işlem.
CPU kullanacak ilk işlem 100 önceliğe ve 100 zamanlamaya sahip olduğu için sürece öncelik verilip hemen yapılacaktır. İkinci yani G/Ç işlemi ise 0 zamanlama ve önceliğe sahip olduğu için bekleyecektir.

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

```
C:\Users\ulas\Desktop\aa>python3 process-run.py
Time        PID: 0        PID: 1           CPU
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu             1
  3        BLOCKED       RUN:cpu             1
  4        BLOCKED       RUN:cpu             1
  5        BLOCKED       RUN:cpu             1
  6        BLOCKED          DONE
  7*   RUN:io_done          DONE             1

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

Toltalde 7 time geçen sürede CPU kullanımı 85.71 ve G/Ç kullanımı ise 71.43. süreçlerin yerini değiştirmemiz ile CPU ve G/Ç kullanımı artmıştır. İlk olarak G/Ç işlemi yapan daha sonra süreci bloklayıp CPU işlemini yaptığı görülmüştür.

4.Şimdi diğer bayraklardan bazılarını inceleyeceğiz. Önemli bir işaret, `-S` bir işlem bir G/Ç yayınladığında sistemin nasıl tepki vereceğini belirleyen işaretidir. Bayrak olarak ayarlandığında SWITCH_ON_END, sistem G/Ç yaparken başka bir işleme GEÇMEYECEKTİR, bunun yerine işlem tamamen bitene kadar bekleyecektir. Biri G/Ç yapan, diğeri CPU işi yapan aşağıdaki iki işlemi çalıştırdığınızda ne olur? (`-l 1:0,4:100 -c -S SWITCH_ON_END`)

```
C:\Users\ulas\Desktop\aa>python3 process-run.py -l 1:0,4
Time        PID: 0        PID: 1           CPU
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu            1
  3        BLOCKED       RUN:cpu            1
  4        BLOCKED       RUN:cpu            1
  5        BLOCKED       RUN:cpu            1
  6        BLOCKED          DONE
  7*   RUN:io_done          DONE             1

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

-S SWITCH_ON_IO kodu ile G/Ç önceliğini değiştirir. Önce G/Ç işlemlerini yaptığı görüldü ardından G/Ç işlemlerinin blokclandığı görüldü. Sonrasında CPU işlemleri yapıldığı görüldü.

5. Şimdi, aynı işlemleri çalıştırın, ancak anahtarlama davranışı, G/Ç (`-l 1:0,4:100 -c -S SWITCH_ON_IO`) için BEKLENİYOR olduğunda başka bir işleme geçmek üzere ayarlanmış olarak. Ne oluyor?

```
C:\Users\ulas\Desktop\aa>python3 process-run.py -l 1:0
Time        PID: 0        PID: 1           CPU
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu             1
  3        BLOCKED       RUN:cpu             1
  4        BLOCKED       RUN:cpu             1
  5        BLOCKED       RUN:cpu             1
  6        BLOCKED          DONE
  7*   RUN:io_done          DONE             1
```

-S SWITCH_ON_IOVarsayılan ayar olduğu gibi bu tam olarak Soru 3'tür. 2 işlemin tamamlanması artık 5 tıklama alıyor, çünkü ikinci işlem ilk GÇ'yi beklerken çalışabiliyor. Simülatör, 6 tik yapmak için son bir "boş tik" ekler.

6.Diğer bir önemli davranış, bir G/Ç tamamlandığında ne yapılacağıdır. ile -I IO_RUN_LATER, bir G/Ç tamamlandığında, onu yayınlayan işlemin hemen çalıştırılması gerekmez; bunun yerine, o sırada çalışan her ne ise, çalışmaya devam eder. Bu işlem kombinasyonunu çalıştırdığınızda ne olur? (. `/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?

```
C:\Users\ulas\Desktop\aa>python3 process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_I
Time        PID: 0        PID: 1        PID: 2        PID: 3           CPU           IO
  1         RUN:io         READY         READY         READY             1
  2        BLOCKED       RUN:cpu         READY         READY             1
  3        BLOCKED       RUN:cpu         READY         READY             1
  4        BLOCKED       RUN:cpu         READY         READY             1
  5        BLOCKED       RUN:cpu         READY         READY             1
  6        BLOCKED       RUN:cpu         READY         READY             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
 20        BLOCKED          DONE          DONE          DONE
 21        BLOCKED          DONE          DONE          DONE
 22        BLOCKED          DONE          DONE          DONE
 23        BLOCKED          DONE          DONE          DONE
 24*   RUN:io_done          DONE          DONE          DONE             1
 25         RUN:io          DONE          DONE          DONE             1
 26        BLOCKED          DONE          DONE          DONE
 27        BLOCKED          DONE          DONE          DONE
 28        BLOCKED          DONE          DONE          DONE
 29        BLOCKED          DONE          DONE          DONE
 30        BLOCKED          DONE          DONE          DONE
 31*   RUN:io_done          DONE          DONE          DONE             1

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

İlk işlem IO'da bekleyecek ve sonraki 3 işlem birbiri ardına tamamlanmaya çalışacaktır. Toplam çalışma süresi, sonunda fazladan bir "boş tik" ile 26 tik olacaktır. Diğer işlemler CPU kullanırken ilk işlem tüm IO işlemlerini çalıştırıyor olabileceğinden, bu, kaynakların etkin bir şekilde kullanılması değildir .

7. Şimdi aynı işlemleri çalıştırın, ancak -I `IO_RUN_IMMEDIATE` G/Ç'yi veren işlemi hemen çalıştıran set ile. Bu davranış nasıl farklıdır? Bir G/Ç'yi henüz tamamlamış bir işlemi yeniden çalıştırmak neden iyi bir fikir olabilir?

```
C:\Users\ulas\Desktop\aa>python3 process-run.py -l 3:0,5:100,5:100,5:100 -S
Time        PID: 0        PID: 1        PID: 2        PID: 3           CPU
  1         RUN:io         READY         READY         READY             1
  2        BLOCKED       RUN:cpu         READY         READY             1
  3        BLOCKED       RUN:cpu         READY         READY             1
  4        BLOCKED       RUN:cpu         READY         READY             1
  5        BLOCKED       RUN:cpu         READY         READY             1
  6        BLOCKED       RUN:cpu         READY         READY             1
  7*   RUN:io_done          DONE         READY         READY             1
  8         RUN:io          DONE         READY         READY             1
  9        BLOCKED          DONE       RUN:cpu         READY             1
 10        BLOCKED          DONE       RUN:cpu         READY             1
 11        BLOCKED          DONE       RUN:cpu         READY             1
 12        BLOCKED          DONE       RUN:cpu         READY             1
 13        BLOCKED          DONE       RUN:cpu         READY             1
 14*   RUN:io_done          DONE          DONE         READY             1
 15         RUN:io          DONE          DONE         READY             1
 16        BLOCKED          DONE          DONE       RUN:cpu             1
 17        BLOCKED          DONE          DONE       RUN:cpu             1
 18        BLOCKED          DONE          DONE       RUN:cpu             1
 19        BLOCKED          DONE          DONE       RUN:cpu             1
 20        BLOCKED          DONE          DONE       RUN:cpu             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%)
```

Bu durumda, ilk süreç kendi G/Ç işlemlerini başlatmak için diğer süreçleri kesintiye uğratacak ve bu da hesaplama süresinin 18 tiklik azalmasıyla sonuçlanacaktır. Kaynak kullanımı, CPU kullanımı %100 olduğunda benzer şekilde iyileşir.

8. Şimdi rastgele oluşturulmuş bazı işlemlerle çalıştırın, örneğin, `-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` vs kullandığınızda ne olur `-I IO_RUN_LATER`? `-S SWITCH_ON_IO` vs kullandığınızda ne olur? `-S SWITCH_ON_END`

```
:\Users\ulas\Desktop\aa>python3 process-run.py -s 2 -l 3:50,3:50 -S SWITCH
roduce a trace of what would happen when you run these processes:
rocess 0
 io
 io_done
 io
 io_done
 cpu

rocess 1
 cpu
 io
 io_done
 io
 io_done

mportant behaviors:
 System will switch when the current process is FINISHED
 After IOs, the process issuing the IO will run LATER (when it is its turn
```

SWITCH_ON_IOher zaman daha hızlı çalışma süreleriyle sonuçlanır SWITCH_ON_END. Verilen 3 örnek için, -I IO_RUN_IMMEDIATEile tam olarak aynı yürütme sırasını verir -I IO_RUN_LATER. Her 3 durumda da, IO işlemlerinin uzunluğuna göre (5 tik) birkaç olası CPU talimatı (her işlem en fazla 3 talimat içerebilir, bu nedenle her biri en fazla 3 CPU talimat alabilir), süreçlerin çoğunu harcadığı anlamına gelir. WAITINGIO'da geçirdikleri süre , dolayısıyla IO_RUN_IMMEDIATEşu anda çalışan bir işlemi tahliye etme fırsatı yoktur.