

İşletim Sistemlerin Projesi

AD-SOYAD : *ODAY ABUHASHISH*

ÖĞRENCİ NO : *2020123009*

KONU : *CH5: Interlude: İşlem API'si*

Interlude: İşlem API'sı

KENAR: INTERLUDE

Interludes, işletim sistemi API'lerine ve bunların nasıl kullanılacağına özel bir odaklanma dahil olmak üzere sistemlerin daha pratik yönlerini kapsayacaktır. Pratik şeylerden hoşlanmıyorsanız, bu ara bölümleri atlayabilirsiniz. Ama pratik şeyleri sevmelisiniz, çünkü bunlar genellikle gerçek hayatta yararlıdır; örneğin şirketler sizi genellikle pratik olmayan becerileriniz için işe almazlar.

Bu arada, UNIX sistemlerinde process oluşturmayı tartışacağız. UNIX, bir çift sistem çağrısıyla yeni bir process yaratmanın en ilgi çekici yollarından birini sunar: `fork()` ve `exec()`. Üçüncü bir rutin olan `wait()`, oluşturduğu bir işlemin tamamlanmasını beklemek isteyen bir işlem tarafından kullanılabilir. Şimdi bu arayüzleri, bizi motive edecek birkaç basit örnekle daha detaylı bir şekilde sunuyoruz. Ve böylece sorumuz:

CRUX: SÜREÇLER NASIL OLUŞTURULUR VE KONTROL EDİLİR .OS, process oluşturma ve kontrol için hangi arayüzleri sunmalıdır? Bu arayüzler, güçlü işlevsellik, kullanım kolaylığı ve yüksek performans sağlamak için nasıl tasarlanmalıdır?

5.1 The `fork()` Sistem Çağrısı

The `fork()` sistem çağrısı, yeni bir process [C63] oluşturmak için kullanılır. Ancak, önceden uyarılmalıdır: Bu kesinlikle şimdiye kadar çağıracağınız en garip rutindir. Tamam, bunu kesin olarak bilmediğimizi kabul ediyoruz; Kimse bakmıyorken hangi rutinleri çağırdığını kim bilebilir? Ancak rutin arama kalıplarınız ne kadar sıra dışı olursa olsun `fork()` oldukça tuhaftır.¹ Daha spesifik olarak, kodu Şekil 5.1'de gördüğünüze benzeyen çalışan bir programınız var; kodu inceleyin veya daha iyisi yazın ve kendiniz çalıştırın!

1. `#include <stdio.h>`
2. `#include <stdlib.h>`
3. `#include <unistd.h>`

¹ Tamam, bunu kesin olarak bilmediğimizi kabul ediyoruz; Kimse bakmıyorken hangi rutinleri çağırdığını kim bilebilir? Ancak rutin arama kalıplarınız ne kadar sıra dışı olursa olsun `fork()` oldukça tuhaftır.

```

4.  int main(int argc, char * argv[]) {
5.  printf("hello world (pid:%d)\n", (int) getpid());
6.  int rc = fork();
7.  if(rc < 0){
8.  //fork başarısız
9.  printf(stderr,"fork failed\n");
10. exit(1);
11. } else if (rc == 0){
12. // child (yeni süreç)
13. Printf("hello, I am child (pid:%d)\n", (int) getpid());
14. }else{
15. //parent bu yoldan gider (main)
16. Printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
17. }
18.   Return 0;
19. }

```

şekil 5.1: **fork()** Çağırma (p1.c)

Bu programı (p1.c olarak adlandırılır) çalıştırdığınızda, aşağıdakileri göreceksiniz:

```

prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>

```

P1.c'de neler olduğunu daha ayrıntılı olarak anlayalım. İşlem ilk çalışmaya başladığında bir merhaba dünya mesajı yazdırır; bu mesaja dahil edilen, **PID** olarak da bilinen **işlem tanımlayıcısıdır**. İşlemin PID'si 29146'dır; UNIX sistemlerinde, (örneğin) çalışmasını durdurmak gibi, işlemle ilgili bir şey yapmak istendiğinde, işlemi adlandırmak için PID kullanılır. her şey yolunda.

Şimdi ilginç kısım başlıyor. İşlem, işletim sisteminin yeni bir işlem oluşturmanın bir yolu olarak sağladığı fork() sistem çağrısını çağırır. Garip kısım: oluşturulan süreç, çağırılan sürecin (neredeyse) tam bir kopyasıdır. Bu, işletim sistemi için p1 programının çalışan iki kopyası olduğu ve her ikisinin de fork() sistem çağrısından geri dönmek üzere olduğu anlamına gelir. Yeni oluşturulan süreç (oluşturan **parent** aksine **child** olarak adlandırılır) beklediğiniz gibi main()'de çalışmaya başlamaz ("merhaba, dünya" mesajı yalnızca bir kez yazdırılır); bunun yerine, fork()'un kendisini çağırması gibi hayata geçer.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>

```

```

4  #include <sys/wait.h>
5
6      int main(int argc, char *argv[]) {
7          printf("hello world (pid:%d)\n", (int) getpid());
8          int rc = fork();
9          if (rc < 0) {          // fork başarısız; exit
10             fprintf(stderr, "fork failed\n");
11             exit(1);
12         } else if (rc == 0) { // child (new process)
13             printf("hello, I am child (pid:%d)\n", (int) getpid());
14         } else { // parent goes down this path (main)
15             int rc_wait = wait(NULL);
16             printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", 17
17                    rc, rc_wait, (int) getpid());
18         }
19         return 0;
20     }

```

Figure 5.2: Calling `fork()` And `wait()` (`p2.c`)

Fark etmiş olabilirsiniz: çocuk tam bir kopya değil. Spesifik olarak, artık kendi adres alanı kopyasına (yani kendi özel belleğine), kendi kayıtlarına, kendi PC'sine vb. sahip olmasına rağmen, **fork()**'u çağırana döndürdüğü değer farklıdır. Spesifik olarak, ebeveyn yeni oluşturulan çocuğun PID'sini alırken, çocuk sıfır dönüş kodunu alır. Bu ayrım yararlıdır, çünkü o zaman iki farklı durumu ele alan kodu yazmak basittir (yukarıdaki gibi).

Şunu da fark etmiş olabilirsiniz: çıktı (`p1.c`'nin) **deterministik** değildir. Alt süreç oluşturulduğunda, sistemde artık önemsedığımız iki aktif süreç var: parent ve child. Tek bir CPU'lu bir sistemde çalıştığımızı varsayarsak (basitlik için), o zaman ya child ya da parent bu noktada çalışabilir. Örneğimizde (yukarıda), önce parent yaptı ve böylece mesajını yazdırdı. Diğer durumlarda, bu çıktı izlemeye gösterdiğimiz gibi bunun tersi de olabilir:

```

prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147
(pid:29146)
prompt>

```

CPU zamanlayıcı, yakında ayrıntılı olarak tartışacağımız bir konu, belirli bir anda hangi sürecin çalışacağını belirler; zamanlayıcı karmaşık olduğundan, genellikle ne yapmayı seçeceği ve dolayısıyla hangi işlemin önce çalışacağı hakkında güçlü varsayımlarda bulunamayız.. Bu **nondeterminizm**, anlaşıldığı

üzere, bazı ilginç sorunlara yol açıyor, özellikle **çok iş parçacıklı programlarda**; bu nedenle, kitabın ikinci bölümünde **eşzamanlılığı** incelediğimizde çok daha fazla belirlenemezlik göreceğiz.

5.2 The wait() Sistem Çağrısı

Şimdiye kadar pek bir şey yapmadık: sadece bir mesaj yazdıran ve çıkan bir çocuk yarattık. Bazen, anlaşıldığı üzere, bir ebeveynin, bir çocuğun yapmakta olduğu şeyi bitirmesini beklemesi oldukça yararlıdır. Bu görev wait() sistem çağrısı (veya onun daha eksiksiz kardeşi waitpid()) ile gerçekleştirilir; ayrıntılar için bkz. Şekil 5.2.

Bu örnekte (p2.c), ana süreç, yürütmesini alt yürütmeyi bitirene kadar geciktirmek için wait()'i çağırır. Child bittiğinde, wait() parent döner.

Yukarıdaki koda bir wait() çağrısı eklemek çıktıyı deterministik yapar. Nedenini görebiliyor musun? Devam et, bir düşün. (düşünmenizi bekliyorum ve bitti)

Şimdi biraz düşündüğünüze göre,işte çıktı:

```
prompt> ./p2
hello world (pid:29266)
hello, I am child(pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

Bu kodla artık child her zaman önce yazdıracağını biliyoruz. Bunu neden biliyoruz? Eh, daha önce olduğu gibi önce basitçe çalışabilir ve bu nedenle parent'den önce yazdırabilir. Ancak, parent önce çalışırsa, hemen wait();'i çağırır. bu sistem çağrısı, çocuk çalışıp çıkana kadar geri dönmeyecek². Bu nedenle, önce ebeveyn çalışsa bile, kibarca çocuğun çalışmasını bitirmesini bekler, ardından wait() geri döner ve ardından parent mesajını yazdırır.

5.3 Son olarak , The exec () Sistem Çağrısı

Süreç oluşturma API'sinin son ve önemli bir parçası exec() sistem çağrısıdır³. Çağırnan programdan farklı bir program çalıştırmak istediğinizde bu sistem çağrısı kullanışlıdır. Örneğin, fork() çağrısı

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
```

² Çocuk çıkmadan önce wait() işlevinin döndüğü birkaç durum vardır; her zaman olduğu gibi daha fazla ayrıntı için kılavuz sayfasını okuyun. Ve bu kitapta yer alan "her zaman önce çocuk yazdırır" veya "UNIX dünyadaki en iyi şeydir, dondurmadan bile daha iyidir" gibi kesin ve koşulsuz ifadelere dikkat edin.

³ Linux'ta exec()'in altı çeşidi vardır: execl(), execlp(), execl(), execv(), execvp() ve execvpe(). Daha fazla bilgi edinmek için kılavuz sayfalarını okuyun.

```

5  #include <sys/wait.h>
6      int main(int argc, char *argv[]) {
7          printf("hello world (pid:%d)\n", (int) getpid());
8          int rc = fork();
9          if (rc < 0) {          // fork failed; exit
10             fprintf(stderr, "fork failed\n");
11             exit(1);
12         } else if (rc == 0) { // child (new process)
13             printf("hello, I am child (pid:%d)\n", (int) getpid());
14             char *myargs[3];
15             myargs[0] = strdup("wc"); // program: "wc" (word count)
16             myargs[1] = strdup("p3.c"); // argument: file to count
17             myargs[2] = NULL; // dizinin sonunu işaretler
18             execvp(myargs[0], myargs); // runs word count
19             printf("this shouldn't print out");
20         } else { // parent bu yola girer (main)
21             int rc_wait = wait(NULL);
22             printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", 24
23                 rc, rc_wait, (int) getpid());
24         }
25     }
26     return 0;
27 }

```

Şekil 5.3: `fork()`'u çağırma, `wait()`, ve `exec()` (p3.c)

p2.c'de, yalnızca aynı programın kopyalarını çalıştırmaya devam etmek istiyorsanız kullanışlıdır. Ancak, genellikle *farklı* bir program çalıştırmak istersiniz; `exec()` tam da bunu yapar (Şekil 5.3).

Bu örnekte, alt süreç kelime sayma programı olan `wc` programını çalıştırmak için `execvp()`'yi çağırır. Aslında, p3.c kaynak dosyasında `wc`'yi çalıştırır, böylece bize dosyada kaç satır, kelime ve bayt bulunduğunu söylerx:

```

prompt> ./p3
hello world (pid:29383)
hello, I am child(pid:29384)
      29      107      1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>

```

`fork()` sistem çağrısı garip; suç ortağı `exec()` de o kadar normal değil. Ne yapar: Yürütülebilir dosyanın adı (ör. `wc`) ve bazı bağımsız değişkenler (ör. `p3.c`) verildiğinde, bundan kod (ve statik veri) **yükler**.

İPUCU: DOĞRU OLMAK (LAMPSON YASASI)

Lampson'ın saygın "Bilgisayar Sistemleri Tasarımı İçin İpuçları" [L83] adlı eserinde belirttiği gibi, "**doğru anla**. Ne soyutlama ne de basitlik, onu doğru yapmanın yerini

tutamaz..” Bazen doğru şeyi yapmanız gerekir ve yaptığınızda alternatiflerden çok daha iyidir. Süreç oluşturma için API tasarlamamızın birçok yolu vardır; Ancak, fork() ve exec() kombinasyonu basit ve son derece güçlüdür.. Burada, UNIX tasarımcıları basitçe doğru anladılar. Ve Lampson sık sık "doğru anladığı" için, yasanın adını onun onuruna veriyoruz.

yürütülebilir ve onunla mevcut kod segmentinin (ve mevcut statik verilerin) üzerine yazar; Yığın, yığın ve programın bellek alanının diğer bölümleri yeniden başlatılır. Ardından işletim sistemi, herhangi bir bağımsız değişkeni bu işlemin argv'si olarak ileterek bu programı çalıştırır. Böylece yeni bir süreç *yaratmaz*; yerine, Halihazırda çalışan programı (eski adıyla p3) çalışan farklı bir programa (wc) dönüştürür. Child'daki exec()'ten sonra sanki p3.c hiç çalışmamış gibidir; başarılı bir exec() çağırısı asla geri dönmez.

5.4 Niye? API'yi Motive Etme

Tabii ki, aklınıza gelebilecek büyük bir soru: Yeni bir süreç yaratmanın basit eylemi olması gereken şey için neden bu kadar garip bir arayüz inşa edelim? Görünen o ki, fork() ve exec()'in ayrılması bir UNIX kabuğu oluşturmak için gereklidir, çünkü kabuğun kodu fork() çağırısından *sonra* ama exec() çağırısından *önce* çalıştırmasına izin verir; bu kod, çalıştırılmak üzere olan programın ortamını değiştirebilir ve böylece çeşitli ilginç özelliklerin kolayca oluşturulmasını sağlar.

Kabuk sadece bir kullanıcı programıdır ⁴. Size bir komut **istemi** gösterir ve ardından ona bir şey yazmanızı bekler.. Daha sonra içine bir komut (yani yürütülebilir bir programın adı ve herhangi bir bağımsız değişken) yazarsınız; çoğu durumda, kabuk daha sonra yürütülebilir dosyanın dosya sisteminde nerede bulunduğunu anlar, komutu çalıştırmak için yeni bir alt süreç oluşturmak için fork()'u çağırır, komutu çalıştırmak için exec()'in bazı türevlerini çağırır ve ardından wait() çağırılarak tamamlama komutu. Çocuk işlemi tamamladığında, kabuk wait() işlevinden geri döner ve bir sonraki komutunuz için hazır olan bir bilgi istemini yeniden yazdırır.

fork() ve exec()'in ayrılması, kabuğun bir sürü yararlı şeyi oldukça kolay bir şekilde yapmasına izin verir. Örneğin:

```
prompt> wc p3.c > newfile.txt
```

Yukarıdaki örnekte, wc programının çıktısı, newfile.txt çıktı dosyasına **yeniden yönlendirilir** (büyüktür işareti, söz konusu yeniden yönlendirmenin nasıl gösterildiğidir). Kabuğun bu görevi yerine getirme şekli oldukça basittir: çocuk oluşturulduğunda, exec() çağrılmadan önce, kabuk **standart çıktıyı**

⁴ Ve bir sürü mermi var; tcsh, bash ve zsh bunlardan birkaçıdır. Bir tanesini seçmeli, kılavuz sayfalarını okumalı ve onun hakkında daha çok şey öğrenmelisiniz; tüm UNIX uzmanları yapar.

kapatır ve newfile.txt dosyasını açar. Bunu yaparak, yakında çalışacak wc programından herhangi bir çıktı ekran yerine dosyaya gönderilir.

Şekil 5.4 (sayfa 8) tam olarak bunu yapan bir programı göstermektedir. Bu yeniden yönlendirmenin işe yaramasının nedeni, işletim sisteminin dosya tanıtıcıları nasıl yönettiğine ilişkin bir varsayımdır. Spesifik olarak, UNIX sistemleri sıfırdan ücretsiz dosya tanımlayıcıları aramaya başlar. Bu durumda, STDOUT_FILENO ilk mevcut olacak ve böylece open() çağrıldığında atanacak. Alt süreç tarafından standart çıktı dosyası tanımlayıcısına, örneğin printf() gibi rutinler tarafından yapılan müteakip yazmalar, ekran yerine şeffaf bir şekilde yeni açılan dosyaya yönlendirilecektir.

İşte p4.c programını çalıştırmanın çıktısı:

```
prompt> ./p4
prompt> cat p4.output
      32      109      846 p4.c
prompt>
```

Bu çıktı hakkında (en azından) iki ilginç haber göreceksiniz. İlk olarak, p4 çalıştırıldığında hiçbir şey olmamış gibi görünür; kabuk sadece komut istemini yazdırır ve bir sonraki komutunuz için hemen hazırdır. Ancak durum böyle değil; p4 programı gerçekten de yeni bir çocuk yaratmak için fork()'u çağırdı ve ardından wc programını bir execvp() çağrısıyla çalıştırdı. p4.output dosyasına yönlendirildiği için ekrana yazdırılan herhangi bir çıktı görmüyorsunuz. İkinci olarak, çıktı dosyasını cat'ladığımızda, wc'yi çalıştırmaktan beklenen tüm çıktının bulunduğunu görebilirsiniz. Harika, değil mi?

UNIX boruları benzer şekilde, ancak pipe() sistem çağrısıyla uygulanır. Bu durumda, bir işlemin çıktısı bir çekirdek **borusuna** (yani kuyruğa) bağlanır, ve başka bir işlemin girişi aynı boruya bağlanır; böylece, bir sürecin çıktısı sorunsuz bir şekilde bir sonrakine girdi olarak kullanılır ve uzun ve kullanışlı komut zincirleri birbirine dizilebilir. Basit bir örnek olarak, bir dosyada bir kelime aramayı ve ardından söz konusu kelimenin kaç kez geçtiğini saymayı düşünün; borular ve grep ve wc yardımcı programları ile kolaydır; sadece yaz `grep -o foo file | wc -l` komut istemine girin ve sonuca hayret edin.

Son olarak, süreç API'sini üst düzeyde kabataslak çizmiş olsak da, bu çağrılar hakkında öğrenilmesi ve sindirilmesi gereken çok daha fazla ayrıntı var; örneğin, kitabın üçüncü bölümünde dosya sistemlerinden bahsederken dosya tanıtıcılar hakkında daha çok şey öğreneceğiz. Şimdilik, fork()/exec() kombinasyonunun süreçleri oluşturmak ve değiştirmek için güçlü bir yol olduğunu söylemekle yetinelim.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
```



```

5  #include <fcntl.h>
6  #include <sys/wait.h>
7
8      int main(int argc, char *argv[]) {
9          int rc = fork();
10         if (rc < 0) {
11             // fork failed
12             fprintf(stderr, "fork failed\n");
13             exit(1);
14         } else if (rc == 0) {
15             // child: redirect standard output to a file
16             close(STDOUT_FILENO);
17             open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC,
18                 S_IRWXU);
19
20             // now exec "wc"...
21             char *myargs[3];
22             myargs[0] = strdup("wc"); // program: wc (word count)
23             myargs[1] = strdup("p4.c"); // arg: file to count
24             myargs[2] = NULL; // mark end of array
25             execvp(myargs[0], myargs); // runs word count
26         } else {
27             // parent goes down this path (main)
28             int rc_wait = wait(NULL);
29         }
30     }

```

Şekil 5.4: Yönlendirmeli Yukarıdakilerin Hepsini (p4 . c)

5.5 Process Control And Users

fork(), exec() ve wait()'in ötesinde, UNIX sistemlerindeki süreçlerle etkileşim kurmak için birçok başka arabirim vardır. Örneğin, kill() sistem çağrısı duraklatma, ölme yönergeleri ve diğer yararlı buyruklar dahil olmak üzere bir işleme **sinyaller** göndermek için kullanılır. Kolaylık sağlamak için, çoğu UNIX kabuğunda, belirli tuş vuruşu kombinasyonları, o anda çalışan işleme belirli bir sinyal iletmek üzere yapılandırılır; örneğin, kontrol-c sürece bir SIGINT (kesme) gönderir (normalde onu sonlandırır) ve kontrol-z bir SIGTSTP (durdurma) sinyali gönderir, böylece işlemi yürütmenin ortasında duraklatır (daha sonra bir komutla devam ettirebilirsiniz, örn. , birçok kabukta bulunan yerleşik fg komutu). Sinyal alt sisteminin tamamı, harici olayları süreçlere iletmek için zengin bir altyapı sağlar; bu sinyalleri ayrı süreçler içinde alma ve işleme yolları ve bireysel süreçlere olduğu kadar tüm **süreç gruplarına** sinyal gönderme yolları da dahildir. Bu iletişim biçimini kullanmak için

KENARA: RTFM — MAN SAYFALARINI OKUYUN

Bu kitapta birçok kez, belirli bir sistem çağrısına veya kütüphane çağrısına atıfta bulunurken, size **kılavuz sayfalarını** veya kısaca **man sayfalarını** okumanızı söyleyeceğiz. Kılavuz sayfaları, UNIX sistemlerinde bulunan belgelerin orijinal

biçimleridir; **Web** denen şey var olmadan önce yaratıldıklarını anlıyorlar. Kılavuz sayfalarını okumak için biraz zaman harcamak, bir sistem programcısının büyümesinde önemli bir adımdır; o sayfalarda tonlarca faydalı bilgi gizli. Okunması özellikle yararlı bazı sayfalar, hangi kabuğu kullanıyorsanız kullanın (örn. **tcsh** veya **bash**) ve kesinlikle programınızın yaptığı herhangi bir sistem çağrısı için (hangi dönüş değerlerinin ve hata koşullarının var olduğunu görmek için) man sayfalarıdır.

Son olarak, kılavuz sayfalarını okumak sizi biraz sıkıntıdan kurtarabilir. Meslektaşlarınıza çatalın () karmaşıklığını sorduğunuzda, basitçe şu yanıtı verebilirler: "RTFM." Bu, meslektaşlarınızın sizi nazikçe Adam sayfalarını okumaya teşvik etme şeklidir. RTFM'deki F, ifadeye biraz renk katıyor...

iletişim, bir process çeşitli sinyalleri "catch" için signal() sistem çağrısını kullanmalıdır; bunu yapmak, bir sürece belirli bir sinyal iletildiğinde, normal yürütmeyi askıya almasını ve sinyale yanıt olarak belirli bir kod parçasını çalıştırmasını sağlar. Sinyaller ve bunların birçok karmaşıklığı hakkında daha fazla bilgi edinmek için başka bir yerde [SR05] okuyun.

Bu doğal olarak şu soruyu gündeme getiriyor: Bir sürece kim sinyal gönderebilir ve kim gönderemez? Genel olarak kullandığımız sistemlerde aynı anda birden fazla kişi kullanılabilir; bu kişilerden biri keyfi olarak SIGINT (bir işlemi kesintiye uğratmak, muhtemelen sonlandırmak) gibi sinyaller gönderebilirse, sistemin kullanılabilirliği ve güvenliği tehlikeye girer. Sonuç olarak, modern sistemler güçlü bir **kullanıcı** kavramı içerir. Kullanıcı, kimlik bilgilerini oluşturmak için bir parola girdikten sonra, sistem kaynaklarına erişim elde etmek için oturum açar. Kullanıcı daha sonra bir veya daha fazla işlem başlatabilir ve bunlar üzerinde tam kontrol uygulayabilir (duraklatabilir, sonlandırabilir, vb.). Kullanıcılar genellikle yalnızca kendi processlerini kontrol edebilir; genel sistem hedeflerini karşılamak için kaynakları (CPU, bellek ve disk gibi) her kullanıcıya (ve işlemlerine) dağıtmak işletim sisteminin işidir.

5.6 Kullanışlı araçlar

Yararlı olan birçok komut satırı aracı da vardır. Örneğin, ps komutunu kullanmak, hangi işlemlerin çalıştığını görmenizi sağlar; ps'ye iletilecek bazı yararlı işaretler için **man sayfalarını** okuyun. Araç üstü, sistemin işlemlerini ve ne kadar CPU ve diğer kaynakları tükettiklerini gösterdiği için oldukça yararlıdır. Espirili bir şekilde, çoğu kez onu çalıştırdığınızda, top onun en iyi kaynak domuzu olduğunu iddia ediyor; belki de biraz egomanyaktır. kill komutu rastgele göndermek için kullanılabilir.

KENARA: SÜPER KULLANICI (KÖK)

Bir sistem genellikle, sistemi **yönetebilen** ve çoğu kullanıcının olduğu şekilde sınırlı olmayan bir kullanıcıya ihtiyaç duyar. Böyle bir kullanıcı, keyfi bir işlemi (örneğin, sistemi bir şekilde kötüye kullanıyorsa) bu kullanıcı tarafından başlatılmamış olsa bile öldürebilmelidir. Böyle bir kullanıcı kapatma gibi (şaşırtıcı olmayan bir şekilde sistemi kapatan) gibi güçlü komutları da çalıştırabilmelidir. UNIX tabanlı sistemlerde, bu özel

yetenekler **süper kullanıcıya** verilir (bazen **root** olarak adlandırılır). Çoğu kullanıcı diğer kullanıcıların işlemlerini sonlandıramazken, süper kullanıcı bunu yapabilir. Kök olmak, Örümcek Adam olmaya çok benzer: büyük güç, büyük sorumluluk getirir [QI15]. Böylece, **güvenliği** artırmak (ve maliyetli hatalardan kaçınmak) için , normal bir kullanıcı olmak genellikle daha iyidir; Kök olmanız gerekiyorsa, bilgisayar dünyasının tüm yıkıcı güçleri artık parmaklarınızın ucunda olduğundan dikkatli olun.

biraz daha kullanıcı dostu olan killall'da olduğu gibi işlemlere sinyaller. Bunları dikkatli kullandığınızdan emin olun; pencere yöneticinizi yanlışlıkla öldürürseniz, önünde oturduğunuz bilgisayarın kullanımı oldukça zor hale gelebilir.

Son olarak, sisteminizdeki yükü hızlı bir şekilde anlamak için kullanabileceğiniz birçok farklı türde CPU ölçer vardır; örneğin, Macintosh araç çubuklarımızda **MenuMeters'ı** (Raging Menace yazılımından) her zaman çalışır durumda tutuyoruz, böylece herhangi bir anda ne kadar CPU kullanıldığını görebiliriz. Genel olarak, neler olup bittiği hakkında ne kadar fazla bilgi olursa o kadar iyidir.

5.7 Summary

UNIX process oluşturma ile ilgili bazı API'leri kullanıma sunduk: fork(), exec() ve wait(). Ancak, sadece yüzeyi gözden geçirdik. Daha fazla ayrıntı için, Stevens ve Rago'yu [SR05] okuyun, tabii ki özellikle Proses Kontrolü, Proses İlişkileri ve Sinyaller ile ilgili bölümleri; Oradaki bilgelikten çıkarılacak çok şey var.

UNIX işlem API'sine olan tutkumuz güçlü olmaya devam etse de, bu pozitifliğin tek tip olmadığını da belirtmeliyiz. Örneğin, İsviçre'deki Microsoft, Boston Üniversitesi ve ETH'den sistem araştırmacıları tarafından hazırlanan yakın tarihli bir makale, fork() ile ilgili bazı sorunları ayrıntılarıyla anlatıyor ve **spawn()** [B+19] gibi diğer, daha basit işlem oluşturma API'lerini savunuyor. Bu farklı bakış açısını anlamak için onu ve atıfta bulunduğu ilgili çalışmayı okuyun. Bu kitaba güvenmek genellikle iyi olsa da, yazarların da fikirleri olduğunu unutmayın; bu görüşler (her zaman) düşündüğünüz kadar geniş çapta paylaşılmayabilir.

KENARA: ANAHTAR process API ŞARTLARI

- Her işlemin bir adı vardır; çoğu sistemde bu ad, **process ID (PID)** olarak bilinen bir sayıdır.
- **Fork()** sistem çağrısı, UNIX sistemlerinde yeni bir process oluşturmak için kullanılır. Yaratıcıya **parent** denir; yeni oluşturulan süreç, **child** olarak adlandırılır. Bazen gerçek hayatta olduğu gibi [J16], çocuk süreç ebeveynin neredeyse aynı kopyasıdır.
- **wait()** sistem çağrısı, bir ebeveynin, çocuğunun yürütmeyi tamamlamasını beklemesine izin verir..
- **exec()** sistem çağrıları ailesi, bir çocuğun ebeveynine olan benzerliğinden kurtulmasına ve tamamen yeni bir program yürütmesine izin verir.
- Bir UNIX **Shell** , kullanıcı komutlarını başlatmak için genellikle fork(), wait() ve exec() kullanır; fork ve exec'in ayrılması, **giriş/çıkış** yeniden **yönlendirmesi**, **borular** ve diğer harika özellikler gibi özellikleri, çalıştırılan programlarla ilgili hiçbir şeyi değiştirmeden etkinleştirir.
- Proses kontrolü, işlerin durmasına, devam etmesine ve hatta sonlanmasına neden olabilecek **sinyaller** şeklinde mevcuttur.
- Hangi processlerin belirli bir kişi tarafından kontrol edilebileceği, **kullanıcı** kavramında özetlenmiştir; işletim sistemi birden fazla kullanıcının sisteme girmesine izin verir ve kullanıcıların yalnızca kendi processlerini kontrol edebilmelerini sağlar.
- Bir **süper kullanıcı** tüm processleri kontrol edebilir (ve aslında başka birçok şey yapabilir); bu rol nadiren ve güvenlik nedenleriyle dikkatli bir şekilde üstlenilmelidir.

Referanslar

[B+19] Andrew Baumann, Jonathan Appavoo, Orran Krieger, Timothy Roscoe tarafından yazılan “Yolda çatal()”. HotOS '19, Bertinoro, İtalya. Öfkeyle dolu eğlenceli bir kağıt. UNIX işlem API'si hakkında karşıt bir bakış açısı elde etmek için okuyun. *Sistem araştırmacılarının, topluluğu yeni yönlerle doğru itme umuduyla aşırı görüşler sunmaya gittikleri, her zaman hareketli HotOS atölyesinde sunuldu.*

[C63] Melvin E. Conway'ın “Çok İşlemcili Bir Sistem Tasarımı”. AFIPS '63 Güz Ortak Bilgisayar Konferansı, New York, ABD 1963. *Çok işlemcili sistemlerin nasıl tasarlanacağına dair ilk makale; fork() teriminin yeni processler oluşturma tartışmasında kullanıldığı ilk yer olabilir.*

[DV66] Jack B. Dennis ve Earl C. Van Horn tarafından yazılan “Çok Programlı Hesaplamalar için Programlama Semantiği”. ACM'nin İletişimi, Cilt 9, Sayı 3, Mart 1966. *Çok programlı bilgisayar sistemlerinin temellerini özetleyen klasik bir makale. Kuşkusuz Project MAC, Multics ve nihayetinde UNIX üzerinde büyük etkisi oldu.*

[J16] "İkiz olabilirler!" Phoebe Jackson-Edwards tarafından. Günlük mail. 1 Mart 2016. *Bu vurucu gazetecilik parçası, bir sürü tuhaf şekilde benzer çocuk/ebeveyn fotoğrafı gösteriyor ve açıkçası biraz büyüyeyici. Devam edin, hayatınızın iki dakikasını boşa harcayın ve kontrol edin. Ama buraya geri gelmeyi unutma! Bu, küçük bir kozmosta internette gezinme tehlikesidir.*

[L83] Butler Lampson'dan “Bilgisayar Sistemleri Tasarımı İçin İpuçları”. ACM İşletim Sistemleri İncelemesi, Cilt 15:5, Ekim 1983. *Lampson'ın bilgisayar sistemlerinin nasıl tasarlanacağına dair ünlü ipuçları. Onu hayatınızın bir noktasında ve muhtemelen hayatınızın birçok noktasında okumalısınız.*

[QI15] The Quote Investigator'dan "Büyük Güç Büyük Sorumluluk Getirir". Mevcut: <https://quoteinvestigator.com/2015/07/23/great-power>. Alıntı araştırmacısı, bu kavramın ilk kez 1793'te Fransız Ulusal Konvansiyonu'nda yapılan bir kararnameler koleksiyonunda geçtiği sonucuna varıyor. Özel alıntı: “Ils doivent envisager qu'une grande responsabilité est la suite inseparable d'un grand pouvoir”, kabaca “Büyük sorumluluğun büyük güçten ayrılmaz şekilde geldiğini düşünmeliler” anlamına gelir. Örümecek Adam'da ancak 1962'de şu sözler yer aldı: “...büyük güçle birlikte büyük sorumluluk da gelmeli!” Görünüşe göre bu seferki takdiri Stan Lee değil, Fransız Devrimi alıyor. Üzgünüm Stan.

[SR05] “UNIX Ortamında Gelişmiş Programlama”, W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. *UNIX API'lerini kullanmanın tüm nüansları ve incelikleri burada bulunur.*

Bu kitabı satın alın! Oku onu! Ve en önemlisi onu yaşa.

3. Şimdi, -t işaretini kullanarak çıkışı değiştirin (örneğin, ./fork.py -t komutunu çalıştırın). Bir dizi işlem ağacı verildiğinde, hangi eylemlerin gerçekleştirildiğini söyleyebilir misiniz?

```
ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list
ARG show_tree True
ARG just_first False
ARG local_repairs False
ARG print_style tancy
ARG solve False
```

Process Tree:

```
a
├── b
│   ├── c
│   │   ├── d
│   │   └── e
│   └── f
│       ├── g
│       └── h
└── i
    ├── j
    └── k
```

```
i@ubuntu:~/Desktop/Simulation$ python fork.py -t
```

4. Dikkat edilmesi gereken ilginç bir nokta, bir çocuk çıktığında ne olur; işlem ağacındaki çocuklarına ne olur? Bunu incelemek için belirli bir örnek oluşturalım: `/fork.py -A a+b,b+c,d,c+d,e,c-`. Bu örnekte 'a', 'b' yarat süreci vardır, bu da 'c'yi yaratır, bu da daha sonra 'd' ve 'e'yi oluşturur. Ancak, o zaman 'c' çıkar. Çıkışın sonra process ağacı sizce nasıl olmalı? Ya -R bayrağını kullanırsanız? Daha fazla bağlam eklemek için artık süreçlere kendi başınıza ne olduğu hakkında daha fazla bilgi edinin.

[illegible]

ör (1)

```
lingumbuntu:/Desktop/Simulations python fork.py -A a:b,b:c,c:d,c:e,e:-R-  
ARG seed -1  
ARG fork_percentage 0.7  
ARG actions 5  
ARG action_list a:b,b:c,c:d,c:e,  
ARG show_flow False  
ARG just_final False  
ARG leaf_only False  
ARG local_request True  
ARG print_style fancy  
ARG solve True
```

Process Tree:

```
a  
├── b  
│   ├── c  
│   │   └── d  
│       └── e  
└── b  
    └── c  
        └── d  
            └── e
```

Action: a forks b

Action: b forks c

Action: c forks d

Action: c forks e

Action: c EXITS

```
b  
├── c  
└── c  
    └── d  
        └── e
```

```
lingumbuntu:/Desktop/Simulations █
```

ör(2)

- 1.örnek : 'c' çıktığında, herhangi bir derinliğe sahip dolaylı alt öğeler (örneğin c'nin çocuklarının çocukları) dahil olmak üzere c'nin tüm çocukları, bu durumda d ve e, işlem ağacındaki en yüksek atalarının çocukları olurlar. Bu durumda d ve e, işlem ağacındaki en yüksek ataları olan a'nın ebeveyni olur. 2.örnek:-R bayrağını kullanırken, yetim süreçler en yakın atalarının çocukları haline gelir.

5. Keşfedilecek son bir bayrak, ara adımları atlayan ve yalnızca son işlem ağacını doldurmayı isteyen -F bayrağıdır. ./fork.py -F komutunu çalıştırın ve oluşturulan eylemler dizisine bakarak son ağacı yazıp yazamayacağınıza bakın. Bunu birkaç kez denemek için farklı rastgele tohumlar kullanın.

```
is@ubuntu:~/Desktop/Simulation$ python fork.py -F
ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list
ARG show_tree False
ARG just_final True
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve False

Process Tree:
a

Action: a forks b
Action: b EXITS
Action: a forks c
Action: a forks d
Action: a forks e

Final Process Tree?
is@ubuntu:~/Desktop/Simulation$
```

6. Son olarak -t ve -F'yi birlikte kullanın. Bu, son işlem ağacını gösterir, ancak ardından gerçekleşen eylemleri doldurmanızı ister. Ağaca bakarak tam olarak gerçekleşen eylemleri belirleyebilir misiniz? Hangi durumlarda söyleyebilirsiniz? Hangisinde söyleyemezsiniz? Bu soruyu derinlemesine incelemek için bazı farklı rastgele tohumlar deneyin.

```
is@ubuntu:~/Desktop/Simulation$ python fork.py -F -t
ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list
ARG show_tree True
ARG just_final True
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve False

Process Tree:
a

Action?
Action?
Action?
Action?
Action?

Final Process Tree:
a
├── b
├── d
└── e
is@ubuntu:~/Desktop/Simulation$
```


KENARA: ÖDEVLERİ KODLAMAK

Kodlama ödevleri, bazı temel işletim sistemi API'leri ile biraz deneyim kazanmak için gerçek bir makinede çalışacak kod yazdığınız küçük alıştırımlardır. Ne de olsa, (muhtemelen) bir bilgisayar bilimcisisiniz ve bu nedenle kodlamayı sevmelisiniz, değil mi? Bunu yapmazsanız, her zaman CS teorisi vardır, ancak bu oldukça zordur. Tabii ki, gerçekten uzman olmak için, makineyi hacklemek için birazdan fazla zaman harcamanız gerekir; aslında, bazı kodlar yazmak ve nasıl çalıştığını görmek için bulabildiğiniz her bahaneyi bulun. Vakit harcayın ve olabileceğinizi bildiğiniz bilge usta olun.

Ödev (Kod)

Bu ödevde, biraz önce okuduğunuz process yönetimi API'lerine biraz aşinalık kazanacaksınız. Endişelenme - görüldüğünden daha eğlenceli! Biraz kod yazmak için mümkün olduğunca çok zaman bulursanız genel olarak çok daha iyi durumda olacaksınız, öyleyse neden şimdi başlamıyorsunuz?

Sorular

1. `fork()` çağırın bir program yazın. `fork()`'u çağırmadan önce, ana sürecin bir değişkene (örn. `x`) erişmesini sağlayın ve değerini bir değere ayarlayın (örn. 100). Alt prosessteki değişkenin değeri nedir? Hem çocuk hem de ebeveyn `x`'in değerini değiştirdiğinde değişkene ne olur?

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(void){
    int x = 10;
    x = 100;
    int rc = fork();
    if(rc < 0)
        printf("failed to fork\n");
    else if(rc == 0){
        x = 99;
        printf("child process get x: %d\n", x);
        exit(0);
    }else{
        x = 999;
        printf("parent process set x to: %d\n", x);
    }

    return 0;
}
```

```
is@ubuntu: ~/Desktop
is@ubuntu:~$ cd Desktop/
is@ubuntu:~/Desktop$ gcc 01.c -o 01
is@ubuntu:~/Desktop$ ./01
parent process set x to: 999
child process get x: 99
is@ubuntu:~/Desktop$
```

- Fork çağırısı iki kez geri döner ve child işlem, parent işlemin tüm adres alanı içeriğini adres alanına kopyalayacaktır (aslında yazma üzerine kopyala kullanılır). kopyalanırsa, child işlem fork çağırıldıktan sonra talimattan çalışmaya başlayacaktır.
- Yazma üzerine kopyalama mekanizması nedeniyle, child işlem ve parent process değiştirme değişkeni birbirini etkilemeyecektir.
- Değişkenden farklı olarak, child işlem, disk dosyasını (child işlem) parent işlemle paylaşacaktır; child işlem ve parent işlem, dosya tanıtıcılarını paylaşır.

2. Bir dosyayı açan (open()) sistem çağrısıyla ve ardından fork()'u çağırarak yeni bir process oluşturan bir program yazın. Open() tarafından döndürülen dosya tanıtıcıya hem çocuk hem de ebeveyn erişebilir mi? Dosyaya aynı anda, yani aynı anda yazdıklarında ne olur?

```

Open ▾
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int fd = open("./check.txt", O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);
    int pid = fork();
    if (pid < 0) {
        printf("fork error\n");
        exit(1);
    } else if (pid == 0) {
        char *buf = "child\n";
        int error = write(fd, buf, sizeof(char) * strlen(buf));
        printf("child error: %d\n", error == -1 ? 1 : 0);
    } else {
        char *buf = "parent\n";
        int error = write(fd, buf, sizeof(char) * strlen(buf));
        printf("child error: %d\n", error == -1 ? 1 : 0);
    }

    int rc = wait(NULL);
    close(fd);
    return 0;
}

Is@ubuntu: ~/Desktop
Is@ubuntu:~/Desktop$ gcc 01.c -o 01
Is@ubuntu:~/Desktop$ ./01
parent process set x to: 999
child process get x: 99
Is@ubuntu:~/Desktop$ gcc 02.c -o 02
Is@ubuntu:~/Desktop$ ./02
Is@ubuntu:~/Desktop$ gcc 02.c -o 02
Is@ubuntu:~/Desktop$ ./02
Is@ubuntu:~/Desktop$ gcc 02.c -o 02
Is@ubuntu:~/Desktop$ ./02
child error: 0
child error: 0
Is@ubuntu:~/Desktop$

```

Hem child işlem hem de parent işlem fd'ye erişebilir. Bir yarış durumu var, fd aynı anda kullanılamaz ama sonunda ikisi de başarılı bir şekilde yazar

3. fork() kullanarak başka bir program yazın. Child işlem "hello" yazmalıdır; ana işlem "goodbye" yazmalıdır. Child işlemin her zaman önce yazdığından emin olmaya çalışmalısınız; bunu parent'te wait() çağırmadan yapabilir misiniz?

```

#include "chapter5.h"

int main(void) {
    int rc;

    rc = fork();

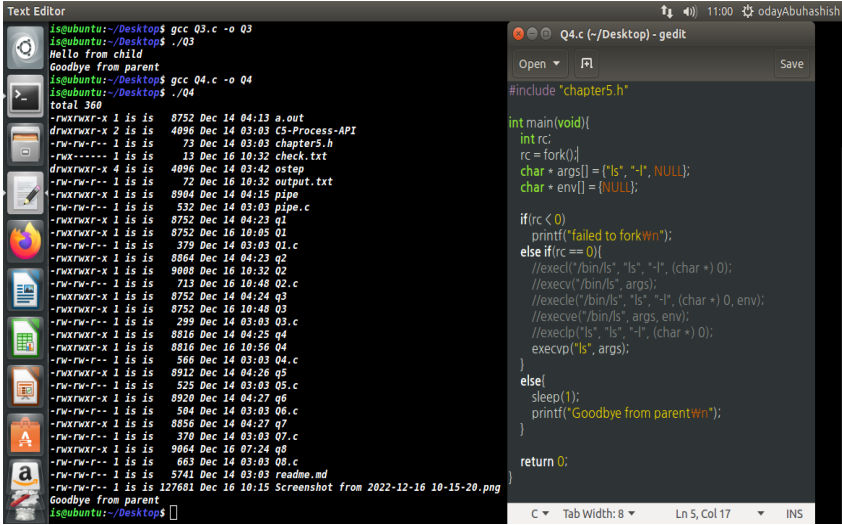
    if (rc < 0)
        printf("failed to fork\n");
    else if (rc == 0) {
        printf("Hello from child\n");
        exit(0);
    } else {
        sleep(3);
        printf("Goodbye from parent\n");
    }

    return 0;
}

Is@ubuntu: ~/Desktop
Is@ubuntu:~/Desktop$ gcc 02.c -o 02
Is@ubuntu:~/Desktop$ ./02
Is@ubuntu:~/Desktop$ gcc 02.c -o 02
Is@ubuntu:~/Desktop$ ./02
child error: 0
child error: 0
Is@ubuntu:~/Desktop$ gcc 03.c -o 03
Is@ubuntu:~/Desktop$ ./03
Hello from child
Goodbye from parent
Is@ubuntu:~/Desktop$

```

4. `/bin/lis` programını çalıştırmak için önce `fork()`'u sonra da bir tür `exec()`'i çağıran bir program yazın. (Linux'ta) `execl()`, `execle()`, `execlp()`, `execv()`, `execvp()` ve `execvpe()` dahil `exec()`'in tüm türevlerini deneyip deneyemeyeceğinize bakın. Aynı temel çağırmanın neden bu kadar çok çeşidi olduğunu düşünüyorsunuz?



```

isqubuntu:~/Desktop$ gcc 03.c -o 03
isqubuntu:~/Desktop$ ./03
Hello from child
Goodbye from parent
isqubuntu:~/Desktop$ gcc 04.c -o 04
isqubuntu:~/Desktop$ ./04
total 360
-rwxrwxr-x 1 is is 8752 Dec 14 04:13 a.out
drwxrwxr-x 2 is is 4096 Dec 14 03:03 CS-Process-API
-rw-rw-r-- 1 is is 73 Dec 14 03:03 chapter5.h
-rw-rw-r-- 1 is is 13 Dec 16 10:32 check.txt
drwxrwxr-x 4 is is 4096 Dec 14 03:42 cstep
-rw-rw-r-- 1 is is 72 Dec 16 10:32 output.txt
-rwxrwxr-x 1 is is 8904 Dec 14 04:15 pipe
-rw-rw-r-- 1 is is 532 Dec 14 03:03 pipe.c
-rwxrwxr-x 1 is is 8752 Dec 14 04:23 q1
-rwxrwxr-x 1 is is 8752 Dec 16 10:05 q1
-rw-rw-r-- 1 is is 379 Dec 14 03:03 Q1.c
-rwxrwxr-x 1 is is 8864 Dec 14 04:23 q2
-rwxrwxr-x 1 is is 9008 Dec 16 10:32 Q2
-rw-rw-r-- 1 is is 713 Dec 16 10:48 Q2.c
-rwxrwxr-x 1 is is 8752 Dec 14 04:24 q3
-rwxrwxr-x 1 is is 8752 Dec 16 10:48 Q3
-rw-rw-r-- 1 is is 299 Dec 14 03:03 Q3.c
-rwxrwxr-x 1 is is 8816 Dec 14 04:25 q4
-rwxrwxr-x 1 is is 8816 Dec 16 10:56 Q4
-rw-rw-r-- 1 is is 566 Dec 14 03:03 Q4.c
-rwxrwxr-x 1 is is 8912 Dec 14 04:26 q5
-rw-rw-r-- 1 is is 525 Dec 14 03:03 Q5.c
-rwxrwxr-x 1 is is 8920 Dec 14 04:27 q6
-rw-rw-r-- 1 is is 584 Dec 14 03:03 Q6.c
-rwxrwxr-x 1 is is 8856 Dec 14 04:27 q7
-rw-rw-r-- 1 is is 370 Dec 14 03:03 Q7.c
-rwxrwxr-x 1 is is 9064 Dec 16 07:24 q8
-rw-rw-r-- 1 is is 663 Dec 14 03:03 Q8.c
-rw-rw-r-- 1 is is 5741 Dec 14 03:03 readme.md
-rw-rw-r-- 1 is is 12760 Dec 16 10:15 Screenshot from 2022-12-16 10-15-20.png
Goodbye from parent
isqubuntu:~/Desktop$

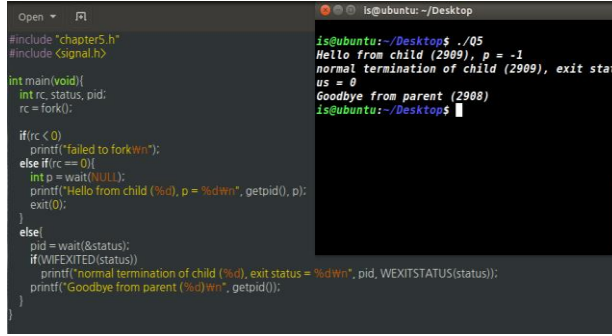
```

Birden fazla `exec` varyantı farklı işlevsellik sağlar,

`exec` işlev ailesinde, `exec`'e `l`, `v`, `p`, `e` son ekleri eklendikten sonra, belirtilen işlevin bazı operasyonel yetenekleri olacaktır:

- **I:** Virgülle ayrılmış bir parametre listesi almayı umuyoruz, liste bir `NULL` işaretçisi ile bitiyor
- **V:** `NULL` ile sonlandırılmış dizelerden oluşan bir diziye bir işaretçi almayı bekler
- **P:** `NULL` ile sonlandırılmış bir dize dizisi işaretçisidir, işlev kendi kendine program dosyasını bulmak için `DOS`'un `PATH` değişkenini kullanabilir
- **e** işlevi, belirtilen evlat edinme numarası `envp`'yi (ortam değişkeni) iletir ve alt sürecin ortamını değiştirmeye izin verir, son ek yoktur
- **E:** Evet, alt process mevcut program ortamını kullanır
- **c** dilinin varsayılan parametre sözdizimi yoktur ve yalnızca birden çok değişken uygulayabilir

5. Şimdi, alt sürecin ebeveynde bitmesini beklemek için wait()’i kullanan bir program yazın. wait() ne döndürür? Child’da wait() kullanırsanız ne olur??



```

Open  ▾  [🔍]  is@ubuntu: ~/Desktop

#include "chapter5.h"
#include <signal.h>

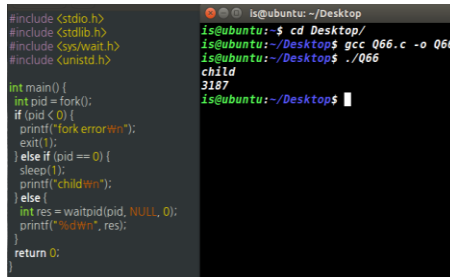
int main(void){
    int rc, status, pid;
    rc = fork();

    if(rc < 0){
        printf("failed to fork\n");
    } else if(rc == 0){
        int p = wait(NULL);
        printf("Hello from child (%d), p = %d\n", getpid(), p);
        exit(0);
    } else{
        pid = wait(&status);
        if(WIFEXITED(status))
            printf("normal termination of child (%d), exit status = %d\n", pid, WEXITSTATUS(status));
        printf("Goodbye from parent (%d)\n", getpid());
    }
}

is@ubuntu:~/Desktop$ ./Q5
Hello from child (2909), p = -1
normal termination of child (2909), exit status = 0
Goodbye from parent (2908)
is@ubuntu:~/Desktop$
  
```

- Wait() child işlem kimliğini başarıyla döndürür ve yürütme başarısız olursa -1 döndürür.
- child işlem beklemeyi çağırır, yürütme başarısız olur ve -1 döndürür.

6. Önceki programda küçük bir değişiklik yazın, bu kez wait() yerine waitpid() kullanarak. waitpid() ne zaman yararlı olur?



```

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int pid = fork();
    if (pid < 0) {
        printf("fork error\n");
        exit(1);
    } else if (pid == 0) {
        sleep(1);
        printf("child\n");
    } else {
        int res = waitpid(pid, NULL, 0);
        printf("%d\n", res);
    }
    return 0;
}

is@ubuntu:~/Desktop$
is@ubuntu:~/Desktop$ cd Desktop/
is@ubuntu:~/Desktop$ gcc Q66.c -o Q66
is@ubuntu:~/Desktop$ ./Q66
child
3187
is@ubuntu:~/Desktop$
  
```

waitpid, wait'in engellenmeyen bir sürümünü sağlamak gibi daha fazla işlem sağlar

7. Bir child process oluşturan ve ardından çocukta standart çıktı (STDOUT_FILENO) oluşturan bir program yazın. Çocuk tanımlayıcıyı kapattıktan sonra bazı çıktıları yazdırmak için printf()'i çağırırsa ne olur?

```
#include "chapter5.h"
#include <signal.h>

int main(void)
{
    int rc, status, pid;
    rc = fork();

    if(rc < 0)
        printf("failed to fork\n");
    else if(rc == 0)
    {
        close(STDOUT_FILENO);
        printf("child (%d) print to standard output.\n", getpid());
        exit(0);
    }
    else
        printf("Goodbye from parent (%d)\n", getpid());
}
```

```
is@ubuntu: ~/Desktop
is@ubuntu:~/Desktop$ gcc 07.c -o 07
is@ubuntu:~/Desktop$ ./07
Goodbye from parent (3330)
is@ubuntu:~/Desktop$
```

printf konsola yazdırmıyor

8. Pipe() sistem çağrısını kullanarak iki çocuk oluşturan ve birinin standart çıkışı diğerinin standart girdisine bağlayan bir program yazın.

```
#include "chapter5.h"
#include <signal.h>
#define MAXLINE 1024
Q8

int main(void)
{
    int rc, status, pid;
    int fd[2];
    char line[MAXLINE];
    rc = fork();

    if(pipe(fd) < 0)
        printf("pipe error\n");

    if(rc < 0)
        printf("failed to fork\n");
    else if(rc == 0)
    {
        if(fork() == 0)
        {
            close(fd[1]);
            printf("receive message from another child (%d)\n", getpid());

            int n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line, n);
            exit(0);
        }
        else
        {
            close(fd[0]);
            write(fd[1], "hello from one child\n", 21);
            exit(0);
        }
    }
}
```

```
is@ubuntu:~/Desktop$ gcc pipe.c -o pipe
is@ubuntu:~/Desktop$ ./pipe
pipe fd[0]: 3
pipe fd[1]: 4
hello world
is@ubuntu:~/Desktop$ gcc 08.c -o 08
is@ubuntu:~/Desktop$ ./08
receive message from another child (348
3)
hello from one child
is@ubuntu:~/Desktop$
```

```
#include "chapter5.h"
#define MAXLINE 1024

int main(void)
{
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];

    if(pipe(fd) < 0)
        printf("pipe error");
    printf("pipe fd[0]: %d\n", fd[0]);
    printf("pipe fd[1]: %d\n", fd[1]);
    if((pid = fork()) < 0)
        printf("fork error");
    else if(pid > 0)
    {
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    }
    else
    {
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

- pipe(int fildes[2]); yarı çift yönlü; fd[1] çıkışı, fd[0] için girdidir.
- Geçerli işlemin şu anda boşta olan iki dosya tanıttıcısını (en küçük tanımlayıcı numarasına sahip ikisi, varsayılan olarak > 2, çünkü 0, 1 ve 2 stdin, stdout, stderr'e tahsis edilmiştir) fd[]'ye depolamak için boruyu çağırın.
- Borunun her iki ucunun da sırasıyla fd[0] ve fd[1]'i kapatması gerekir.