5

Interlude: Süreç API'si

Kenar: INTERLUDES
Interludes, işletim sistemi API'lerine ve bunların nasıl kullanılacağına kısmi bir odaklanma da dahil olmak üzere sistemlerin daha pratik yönlerini kapsayacaktır. Pratik şeylerden hoşlanmıyorsanız, bu araları atlayabilirsiniz. Ancak pratik şeyleri sevmelisiniz, çünkü bunlar genellikle gerçek hayatta faydalıdır; Örneğin, şirketler genellikle pratik olmayan becerileriniz için sizi işe almazlar.

Bu arada, UNIX sistemlerinde süreç oluşturmayı tartışıyoruz. UNIX, bir çift sistem çağrısıyla yeni bir süreç oluşturmanın en ilgi çekici yollarından birini sunar: fork() ve exec(). Üçüncü bir rutin olan wait(), oluşturduğu bir sürecin tamamlanmasını beklemek isteyen bir süreç tarafından kullanılabilir . Şimdi bu arayüzleri bizi motive edecek birkaç basit örnekle daha ayrıntılı olarak sunuyoruz. Ve böylece, sorunumuz:

Dönüm noktası: SÜREÇLER NASIL OLUŞTURULUR VE KONTROL EDİLİR?
İşletim sistemi süreç oluşturma ve kontrol için hangi arayüzleri sunmalıdır? Bu arayüzler güçlü işlevsellik, kullanım kolaylığı ve yüksek performans sağlamak için nasıl tasarlanmalıdır?

5.1 `fork()` Sistem Çağrısı

fork() sistem çağrısı yeni bir süreç [C63] oluşturmak için kullanılır. Her nasılsa, önceden uyarılmalıdır: kesinlikle arayacağınız en garip rutindir[1]. Daha spesifik olarak, kodu görünen çalışan bir programınız var Şekil 5.1'de gördüğünüz gibi; Kodu inceleyin , veya daha iyisi, yazın ve kendiniz çalıştırın!

[1]Tamam, bunu kesin olarak bilmediğimizi itiraf ediyoruz; kimse bakmadığında hangi rutinlere dediğinizi kim bilebilir? Ancak fork(), rutin arama kalıplarınız ne kadar sıra dışı olursa olsun, oldukça gariptir.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello world (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15   } else {
16     // parent goes down this path (main)
17     printf("hello, I am parent of %d (pid:%d)\n",
18             rc, (int) getpid());
19   }
20   return 0;
21 }
22
```

Figure 5.1: **Calling `fork()` (`p1.c`)**

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

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

Ne olduğunu p1.c'de daha ayrıntılı olarak anlayalım. İlk çalışmaya başladığında, süreç bir merhaba dünya mesajı yazdırır; bu mesajda **PID** olarak da bilinen **süreç tanımlayıcısı (process identifier)** bulunur. Sürecin PID'si 29146'dır; UNIX sistemlerinde, PID, süreçle ilgili bir şeyler yapmak isterse, süreci adlandırmak için kullanılır, örneğin (örneğin) çalışmasını durdurun. Şimdiye kadar, çok iyi.

Şimdi ilginç kısım başlıyor. Süreç, işletim sisteminin yeni bir süreç oluşturmanın bir yolu olarak sağladığı fork() sistem çağrısını çağırır. Garip kısım: oluşturulan süreç, arama süreçinin (neredeyse) tam bir kopyasıdır. Bu, işletim sistemi için , şimdi p1 programının çalışan iki kopyası var gibi göründüğü anlamına gelir ve her ikisi de fork() sistem çağrısından dönmek üzeredir. Yeni oluşturulan süreç (oluşturan **ebeveynin(parent)** aksine **çocuk(child)** olarak adlandırılır) main() bölümünde çalışmaya başlamaz, beklediğiniz gibi ("merhaba, dünya" mesajının yalnızca bir kez yazdırıldığını unutmayın); daha ziyade, sanki fork() kendisini çağırmış gibi canlanır.

```
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 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   } 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            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
21
```

Şekil 5.2: **fork() ve wait() (p2.c) çağrısı**

Fark etmiş olabilirsiniz: çocuk *tam* bir kopya değildir. Özellikle, artık adres alanının kendi kopyasına (yani, kendi özel belleğine), kendi kayıtlarına, kendi bilgisayarına ve benzerlerine sahip olmasına rağmen, geri döndüğü değer **fork()** öğesini arayan kişi farklıdır. Özellikle, ebeveyn yeni oluşturulan alt öğenin PID'sini alırken, alt öğe sıfır dönüş kodunu alır. Bu farklılaştırma yararlıdır, çünkü iki farklı durumu işleyen kodu yazmak basittir (yukarıdaki gibi).

Ayrıca şunu da fark etmiş olabilirsiniz: çıktı (p1.c) **deterministik (deterministic)** değildir. Alt süreç oluşturulduğunda, sistemde artık önemsediğimiz iki aktif süreç vardır: ebeveyn ve çocuk. Tek bir CPU'ya sahip bir sistemde çalıştığımızı varsayarsak (basitlik için), o zaman çocuk veya ebeveyn bu noktada çalışabilir. Örneğimizde (yukarıda), ebeveyn önce mesajını yaptı ve böylece yazdırdı. Diğer durumlarda, bu çıktı izlemesinde gösterdiğimiz gibi tam tersi olabilir:

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

Yakında ayrıntılı olarak tartışacağımız bir konu olan CPU **zamanlayıcısı (scheduler)**, hangi sürecin belirli bir zamanda çalıştığını caydırır; sched-uler karmaşık olduğu için, genellikle ne yapmayı seçeceği ve dolayısıyla hangi sürecin ilk önce çalışacağı konusunda güçlü varsayımlarda bulunamayız.

Bu **determinizm dışılık (non determinism)**, ortaya çıktığı gibi, **çok iş parçacıklı programlarda (multi-threaded programs)** kısmen bazı ilginç sorunlara yol açar; bu nedenle, kitabın ikinci bölümünde **eşzamanlılığı(concurrency)** incelediğimizde çok daha fazla determinizm görmeyeceğiz.

5.2 `wait()` Sistem Çağrısı

Şimdiye kadar fazla bir şey yapmadık: sadece bir mesajı yazdıran ve çıkan bir çocuk yarattık. Bazen, ortaya çıktığı gibi, bir ebeveynin bir çocuk sürecinin yaptığı şeyi bitirmesini beklemesi oldukça yararlıdır. Bu görev, wait() sistem çağrısıyla (veya daha eksiksiz kardeşi waitpid()) gerçekleştirilir; Ayrıntılar için Şekil 5.2'ye bakın.

Bu örnekte (p2.c), ebeveyn süreç wait() öğesini çağırarak çocuk öğe yürütmeyi bitirene kadar yürütmesini geciktirir. Çocuk tamamlandığında, wait() ebeveyne geri döner.

Yukarıdaki koda wait() çağrısı eklemek, çıktıyı deterministik hale getirir. Nedenini görebiliyor musunuz? Devam et, bir düşün.

(düşünmenizi ve bitirmenizi bekliyorum)

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 çocuğun her zaman önce yazdıracağını biliyoruz. Bunu neden biliyoruz? Eh, daha önce olduğu gibi ilk önce çalışabilir ve böylece ebeveynden önce yazdırabilir. Ancak, ebeveyn ilk önce çalışırsa, hemen wait() öğesini çağırır; Bu sistem çağrısı, çocuk çalışıp çıkana kadar geri dönmez[2]. Böylece, ebeveyn ilk önce çalıştığında bile, kibarca çocuğun çalışmasını bitirmesini bekler, sonra wait() geri döner ve ardından ebeveyn mesajını yazdırır .

5.3 Son Olarak, `exec()` Sistem Çağrısı

Süreç oluşturma API'sinin son ve önemli bir parçası exec() sistem çağrısıdır[3]. Bu sistem çağrısı, çağıran programdan farklı bir program çalıştırmak istediğinizde kullanışlıdır . Örneğin, fork() öğesini çağırmak

[2] Çocuk çıkmadan önce wait()' in geri döndüğü birkaç durum vardır; her zaman olduğu gibi daha fazla ayrıntı için adam sayfasını okuyun . Ve bu kitabın yaptığı mutlak ve niteliksiz ifadelere dikkat edin, örneğin "çocuk her zaman önce basacaktır" veya "UNIX dünyadaki en iyi şeydir, dondurmadan bile daha iyidir."

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char *argv[]) {
8    printf("hello world (pid:%d)\n", (int) getpid());
9    int rc = fork();
10   if (rc < 0) {         // fork failed; exit
11     fprintf(stderr, "fork failed\n");
12     exit(1);
13   } else if (rc == 0) { // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15     char *myargs[3];
16     myargs[0] = strdup("wc");   // program: "wc" (word count)
17     myargs[1] = strdup("p3.c"); // argument: file to count
18     myargs[2] = NULL;           // marks end of array
19     execvp(myargs[0], myargs);  // runs word count
20     printf("this shouldn't print out");
21   } else {              // parent goes down this path (main)
22     int rc_wait = wait(NULL);
23     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
24            rc, rc_wait, (int) getpid());
25   }
26   return 0;
27 }
28
```

Şekil 5.3: fork(), wait() ve exec() (p3.c) çağrısı

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 sözcük sayma programı olan wc programını çalıştırmak için execvp() öğesini çağırır. Aslında, p3.c kaynak dosyasında wc çalıştırır, böylece bize dosyada kaç satır, kelime ve bayt bulunduğunu söyler:

```
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 gariptir; suç ortağı exec(), o kadar da normal değil. Ne işe yarar: yürütülebilir bir dosyanın adı (örneğin, wc) ve bazı bağımsız değişkenler (örneğin, p3.c) göz önüne alındığında , bundan kod (ve statik veriler) **yükler(loads)**.

İPUCU: DOĞRU YAPMAK (L AMPSON YASASI)

Lampson'un saygın "Bilgisayar Sistemleri Tasarımı için İpuçları" [L83] adlı eserinde belirttiği gibi, "**Doğru yapın (Get it right)**. Ne soyutlama ne de basitlik, onu doğru yapmanın yerine geçer." Bazen, sadece doğru şeyi yapmanız gerekir ve bunu yaptığınızda, alternatiflerden çok daha iyidir. Süreç oluşturma için API'ler tasarlamanın 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, yasayı onuruna adlandırıyoruz .

Yürütülebilir ve geçerli kod segmentinin (ve geçerli statik verilerinin) üzerine yazar; yığın ve yığın ve programın bellek alanının diğer bölümleri yeniden başlatılır. Daha sonra işletim sistemi basitçe bu programı çalıştırır ve herhangi bir argümanı bu sürecin argv'si olarak geçirir.Böylece, yeni bir süreç oluşturmaz; bunun yerine, şu anda çalışan programı (eski adıyla p3) farklı bir çalışan programa (wc) dönüştürür. Çocuktaki exec()'den sonra, neredeyse p3.c hiç çalışmamış gibidir; exec()'e başarılı bir çağrı asla geri dönmez..

1. Neden? API'yi Motive Etme

Tabii ki, büyük bir sorunuz olabilir: Neden yeni bir süreç yaratmanın basit eylemi olması gereken şey için bu kadar garip bir arayüz inşa edelim ? Anlaşıldığı üzere, fork() ve exec( ) öğelerinin ayrılması bir U NIX kabuğu oluşturmada esastır, çünkü kabuğun çatal() çağrısından *sonra* ancak exec() çağrısından *önce* kod 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 **istem (prompt)** gösterir ve sonra içine bir şeyler yazmanızı bekler. Daha sonra içine bir komut (yani, yürütülebilir bir programın adı ve bağımsız değişkenler) yazarsınız; çoğu durumda, kabuk daha sonra yürütülebilir dosyanın dosya sisteminde nerede bulunduğunu bulur, komutu çalıştırmak üzere yeni bir alt süreç oluşturmak için fork() öğesini çağırır, komutu çalıştırmak için exec() öğesinin bazı türevlerini çağırır ve ardından wait komutunu çağırarak komutun tamamlanmasını bekler. Alt öğe tamamlandığında, kabuk wait() öğesinden döner ve bir sonraki komutunuz için hazır olacak şekilde yeniden bir istem yazdırır.

Fork() ve exec()' in ayrılması, kabuğun bir sürü faydalı şeyi oldukça kolay bir şekilde yapmasını sağlar. Örneğin:

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

[4]Ve çok sayıda kabuk var; tcsh, bash ve zsh bunlardan birkaçıdır. Birini seçmeli, adam sayfalarını okumalı ve hakkında daha fazla bilgi edinmelisiniz; tüm UNIX uzmanları bunu yapar.

Yukarıdaki örnekte, wc programının çıktısı newfile.txt çıktı dosyasına **yönlendirilir (redirected)** (daha büyük işaret, söz konusu dirilmenin nasıl belirtildiğidir). Kabuğun bu görevi yerine getirme şekli oldukça basittir: alt öğe oluşturulduğunda, exec() öğesini çağırmadan önce, kabuk **standart çıktıyı (standard output)** kapatır ve newfile.txt dosyasını açar. Bunu yaparak, yakında çalışacak olan wc programından herhangi bir çıkış ekranı yerine dosyaya gönderilir.

Şekil 5.4 (sayfa 8) tam olarak bunu yapan bir programı göstermektedir. Bu yeniden yönlendirmenin çalışmasının nedeni , işletim sisteminin dosya tanımlayıcılarını nasıl yönettiğine ilişkin bir varsayımdan kaynaklanmaktadır. Özellikle, UNIX sistemleri sıfırda ücretsiz dosya tanımlayıcıları aramaya başlar. Bu durumda, STDOUT FILENO ilk kullanılabilir olan olacaktır ve böylece open() çağrıldığında atanacaktır. Alt süreç tarafından standart çıktı dosyası tanımlayıcısına, örneğin printf() gibi yordamlara yazılan sonraki yazılar, daha sonra ekran yerine yeni açılan dosyaya saydam bir şekilde yönlendirilir .

İşte p4.c programını çalıştırmanın çıktısı:

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

Bu çıktıyla ilgili (en azından) iki ilginç ipucu fark edeceksiniz. 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 oluşturmak için fork() öğesini çağırdı ve ardından execvp() çağrısıyla wc programını çalıştırdı. P4.output dosyasına yeniden çekildiği için ekrana yazdırılan herhangi bir çıktı görmüyorsunuz. İkincisi, çıktı dosyasını kattığımızda, wc'yi çalıştırmaktan beklenen tüm çıktıların bulunduğunu görebilirsiniz. Harika, değil mi?

UNIX boruları benzer şekilde uygulanır, ancak pipe() sistem çağrısı ile. Bu durumda, bir sürecin çıkışı çekirdek içi bir **boruya**(**pipe**) (yani kuyruğa) bağlanır ve başka bir sürecin girişi aynı boruya bağlanır ; Böylece, bir sürecin çıktısı sorunsuz bir şekilde bir sonrakine girdi olarak kullanılır ve uzun ve kullanışlı komut zincirleri bir araya getirilebilir. Basit bir örnek olarak, bir dosyada bir kelime aramayı ve ardından söz konusu kelimenin kaç kez gerçekleştiğini saymayı düşünün; borular ve grep ve wc kullanımları ile kolaydır; sadece yazın `grep -o foo file | wc -l`

Komut istemine girin ve sonuca hayret edin.

Son olarak, süreç API'sini yüksek düzeyde çizmiş olsak da, bu çağrılar hakkında öğrenilmesi ve sindirilmesi gereken çok daha fazla ayrıntı var; Örneğin, kitabın üçüncü bölümünde dosya sistemleri hakkında konuştuğumuzda dosya tanımlayıcıları hakkında daha fazla bilgi edineceğiz .Şimdilik, fork()/exec() kombinasyonunun süreçleri oluşturmak ve manipüle etmek için güçlü bir yol olduğunu söylemek yeterlidir.

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

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc (word count)
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

Figure 5.4: **Yukarıdakilerin Tümü Yeniden Yönlendirme ile (p4.c)**

1. Süreç Kontrolü ve Kullanıcılar

Fork(), exec(), ve wait()'in ötesinde, UNIX sistemlerindeki süreçlerle etkileşim kurmak için birçok başka karşılıklı yüz vardır. Örneğin, kill() sistem çağrısı, duraklatmak, ölmek ve diğer yararlı zorunluluklar için directives dahil olmak üzere bir sürece **sinyal(signals)** göndermek için kullanılır. Kolaylık sağlamak için, çoğu UNIX kabuğunda, belirli tuş vuruşu kombinasyonları o anda çalışmakta olan sürece belirli bir sinyal iletmek üzere yapılandırılır; örneğin, control-c sürece bir SIGINT (kesme) gönderir (normalde sonlandırır) ve control-z bir SIGTSTP (durdurma) sinyali gönderir ve böylece süreci duraklatır. yürütmenin ortasında (daha sonra bir komutla devam ettirebilirsiniz, örneğin, birçok kabukta bulunan fg yerleşik komutu).

Tüm sinyaller alt sistemi, bu sinyalleri bireysel süreçler içinde alma ve sürece yolları ve tüm **süreç gruplarının(process groups)** yanı sıra bireysel süreçlere sinyal gönderme yolları da dahil olmak üzere süreçlere harici olaylar sunmak için zengin bir altyapı sağlar. Bu iletişim biçimini kullanmak için

Kenara: RTFM — Klavuz Sayfalarını Okuyun

Bu kitapta çoğu zaman, belirli bir sistem çağrısına veya kütüphane çağrısına atıfta bulunurken, size **kılavuz sayfalarını (manuel pages)** veya kısaca **man sayfalarını (man pages)** okumanızı söyleyeceğiz. İnsan sayfaları, UNIX sistemlerinde var olan orijinal dokümantasyon biçimidir; **Web (the web)** denilen şey var olmadan önce yaratıldıklarını fark edin.

İnsan sayfalarını okumak için biraz zaman harcamak, bir sistem programcısının büyümesinde önemli bir adımdır; bu sayfalarda gizlenmiş tonlarca yararlı bilgi vardır. Okunması gereken bazı özellikle yararlı sayfalar, kullandığınız kabuk için (örneğin, **tcsh** veya **bash**) ve kesinlikle programınızın yaptığı herhangi bir sistem çağrısı için (hangi dönüş değerlerini görmek için) man sayfalarıdır. Ve hata koşulları mevcuttur). Son olarak, adam sayfalarını okumak sizi biraz utandırabilir. İş arkadaşlarınıza fork()ın bazı karmaşıklıklarını sorduğunuzda, basitçe şu yanıtı verebilirler: "RTFM." Bu, meslektaşlarınızın sizi nazikçe The Man sayfalarını okumaya teşvik etme şeklidir. RTFM'deki F, ifadeye biraz renk katıyor...

süreç, çeşitli sinyalleri "yakalamak" için signal() sistem çağrısını kullanmalıdır ; bunu yapmak, belirli bir sinyal bir sürece devredildiğinde , normal yürütmesini askıya almasını ve sinyale yanıt olarak kısmi bir kod parçası 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: Kim bir sürece sinyal gönderebilir ve kim gönderemez? Genel olarak, kullandığımız sistemler aynı anda onları kullanan birden fazla kişiye sahip olabilir; Bu kişilerden biri keyfi olarak SIGINT gibi sinyaller gönderebilirse (bir süreci kesintiye uğratmak, muhtemelen sonlandırmak için), sistemin kullanılabilirliği ve güvenliği tehlikeye girer.Sonuç olarak, modern sistemler bir **kullanıcı(user)** kavramının güçlü bir anlayışını içerir. Kullanıcı, kimlik bilgilerini oluşturmak için bir parola girdikten sonra, sistem kaynaklarına erişmek için oturum açar. Kullanıcı daha sonra bir veya daha fazla süreç başlatabilir ve bunlar üzerinde tam kontrol sahibi olabilir (duraklatabilir, öldürebilir, vb.). Kullanıcılar genellikle yalnızca kendi süreçlerini kontrol edebilir; Genel sistem hedeflerini karşılamak için kaynakları (CPU, bellek ve disk gibi) her kullanıcıya (ve süreçlerine) ayırmak işletim sisteminin işidir .

Faydalı Araçlar

Yararlı olan birçok komut satırı aracı da vardır. İnceleme için , ps komutunu kullanmak hangi süreçlerin çalıştığını görmenizi sağlar; ps'ye geçmek için bazı yararlı bayraklar için **man sayfalarını(manuel pages)** okuyun. Araç üstü de sistemin süreçlerini ve ne kadar CPU ve diğer kaynakları yediklerini gösterdiği için oldukça yararlıdır. Esprili bir şekilde, birçok kez çalıştırdığınızda, top onun en iyi kaynak domuzu olduğunu iddia eder; belki de biraz egomanyaktır. Kill komutu, süreçlere sinyaller göndermek için kullanılabilir.

ASIDE: THE SUPERUSER (ROOT)
A system generally needs a user who can **administer** the system, and is not limited in the way most users are. Such a user should be able to kill an arbitrary process (e.g., if it is abusing the system in some way), even though that process was not started by this user. Such a user should also be able to run powerful commands such as `shutdown` (which, unsurprisingly, shuts down the system). In UNIX-based systems, these special abilities are given to the **superuser** (sometimes called **root**). While most users can't kill other users processes, the superuser can. Being root is much like being Spider-Man: with great power comes great responsibility [QI15]. Thus, to increase **security** (and avoid costly mistakes), it's usually better to be a regular user; if you do need to be root, tread carefully, as all of the destructive powers of the computing world are now at your fingertips.

biraz daha kullanıcı dostu killall gibi. Bunları dikkatli kullandığınızdan emin olun; Yanlışlıkla pencere yöneticinizi öldürürseniz, önünde oturduğunuz bilgisayarın kullanımı oldukça zorlaşabilir. Son olarak, sisteminizdeki yükü hızlı bir şekilde anlamak için kullanabileceğiniz birçok farklı CPU ölçüm cihazı türü vardır; Örneğin , Macintosh araç çubuklarımızda her zaman **MenuMeters'ı** (Raging Menace yazılımından) çalışır durumda tutarız, böylece herhangi bir zamanda ne kadar CPU kullanıldığını görebiliriz. Genel olarak, ne olduğu hakkında daha fazla bilgi devam ediyor, daha iyi.

1. Özet

UNIX süreç oluşturma ile ilgili API'lerden bazılarını kullanıma sunduk : fork(), exec(), and wait(). Ancak, sadece yüzeyi gözden geçirdik. Daha fazla ayrıntı için, Stevens ve Rago [SR05], elbette, özellikle Süreç Kontrolü, Süreç İlişkileri ve Sinyaller ile ilgili bölümleri okuyun ; oradaki bilgelikten çıkarılacak çok şey var.

UNIX süreç API'sine olan tutkumuz güçlü kalırken, bu pozitifliğin tek tip olmadığını da not etmeliyiz. Örneğin, Microsoft, Boston Üniversitesi ve İsviçre'deki ETH'den sistem araştırmacıları tarafından yakın zamanda yapılan bir deneme, fork() ile ilgili bazı sorunları detaylandırıyor ve **spawn()** [B+19] gibi diğer, daha basit süreç 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 görüşleri olduğunu da unutmayın ; Bu görüşler (her zaman) düşündüğünüz kadar yaygın olarak paylaşılmayabilir.

ASIDE: ANAHTAR İŞLEM API TERİMLERİ

i) Her sürecin bir adı vardır; Çoğu sistemde, bu ad **süreç kimliği(Process ID)(PID)** olarak bilinen bir sayıdır.

ii) **Fork()** sistem çağrısı, UNIX sistemlerinde yeni bir süreç oluşturmak için kullanılır. Yaratıcıya **ebeveyn(parent)** denir; yeni oluşturulan sürece **çocuk(child)** denir. Bazen gerçek hayatta olduğu gibi [J16], çocuk süreci ebeveynin neredeyse aynı kopyasıdır .

iii) **wait()** sistem çağrısı, ebeveynin alt öğesinin yürütmeyi tamamlamasını beklemesine olanak tanır .

iv) **exec()** sistem çağrıları ailesi, bir çocuğun ebeveynine olan benzerliğinden kurtulmasını ve tamamen yeni bir program yürütmesini sağlar .

v) Bir UNIX **kabuğu(shell)**, kullanıcı komutlarını başlatmak için genellikle fork(), wait() ve exec() kullanır ; yükleyiciler ve yürütmenin ayrılması, **giriş / çıkış yeniden yönlendirme(input/outut redirection)**, **borular(pipes)** ve diğer harika özellikler gibi özellikleri hiçbir şeyi değiştirmeden mümkün kılar. çalıştırılan programlar hakkında.

vi) Proses kontrolü, işlerin durmasına, devam etmesine ve hatta sonlandırılmasına neden olabilecek **sinyaller(signals)** şeklinde mevcuttur.

vii) Belirli bir kişi tarafından hangi süreçlerin kontrol edilebileceği bir **kullanıcı(user)** kavramıyla kapsüllenir; işletim sistemi, sisteme birden fazla kullanıcının girmesine izin verir ve kullanıcıların yalnızca kendi süreçlerini kontrol edebilmelerini sağlar.

viii) Bir **süper kullanıcı(superuser)** tüm süreçleri kontrol edebilir (ve aslında birçok başka şey yapabilir); Bu rol, güvenlik nedenleriyle nadiren ve dikkatle üstlenilmelidir.

References

[B+19] "A fork() in the road" by Andrew Baumann, Jonathan Appavoo, Orran Krieger, Timothy Roscoe. HotOS '19, Bertinoro, Italy. *A fun paper full of `fork()`ing rage. Read it to get an opposing viewpoint on the* Unix *process API. Presented at the always lively HotOS workshop, where systems researchers go to present extreme opinions in the hopes of pushing the community in new directions.*

[C63] "A Multiprocessor System Design" by Melvin E. Conway. AFIPS '63 Fall Joint Computer Conference, New York, USA 1963. *An early paper on how to design multiprocessing systems; may be the first place the term `fork()` was used in the discussion of spawning new processes.*

[DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis and Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966. *A classic paper that outlines the basics of multiprogrammed computer systems. Undoubtedly had great influence on Project MAC, Multics, and eventually* Unix.

[J16] "They could be twins!" by Phoebe Jackson-Edwards. The Daily Mail. March 1, 2016.. *This hard-hitting piece of journalism shows a bunch of weirdly similar child/parent photos and is frankly kind of mesmerizing. Go ahead, waste two minutes of your life and check it out. But don't forget to come back here! This, in a microcosm, is the danger of surfing the web.*

[L83] "Hints for Computer Systems Design" by Butler Lampson. ACM Operating Systems Review, Volume 15:5, October 1983. *Lampson's famous hints on how to design computer systems. You should read it at some point in your life, and probably at many points in your life.*

[QI15] "With Great Power Comes Great Responsibility" by The Quote Investigator. Available: `https://quoteinvestigator.com/2015/07/23/great-power`. *The quote investigator concludes that the earliest mention of this concept is 1793, in a collection of decrees made at the French National Convention. The specific quote: "Ils doivent envisager qu'une grande responsabilite´ est la suite inse´parable d'un grand pouvoir", which roughly translates to "They must consider that great responsibility follows inseparably from great power." Only in 1962 did the following words appear in Spider-Man: "...with great power there must also come–great responsibility!" So it looks like the French Revolution gets credit for this one, not Stan Lee. Sorry, Stan.*

[SR05] "Advanced Programming in the Unix Environment" by W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. *All nuances and subtleties of using* Unix *APIs are found herein. Buy this book! Read it! And most importantly,* ***live it.***

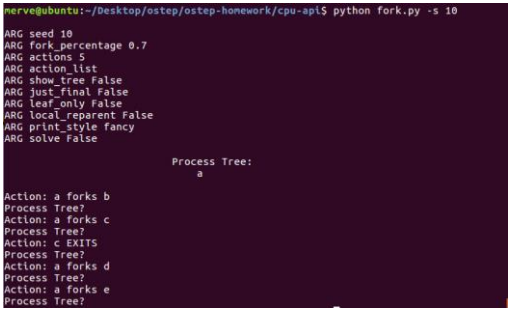
Ödev (Simülasyon)

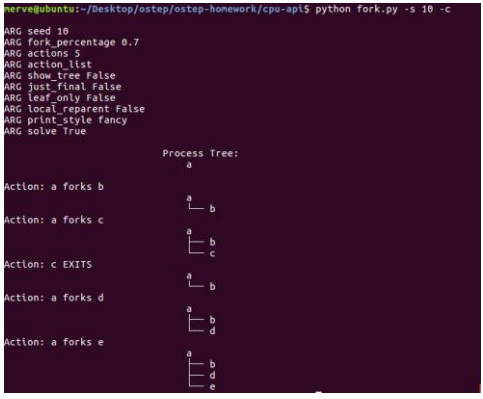
Bu simülasyon ödevi, süreçlerin tek bir "ailesel" ağaçta nasıl ilişkili olduğunu gösteren basit bir süreç oluşturma simülatörü olan fork.py'a odaklanmaktadır. Simülatörün nasıl çalıştırılacağı hakkında ayrıntılar için ilgili README'yi okuyun.

Sorular

1.) `./fork.py -s 10` komutunu çalıştırın ve hangi eylemlerin gerçekleştirildiğini görün. İşlem ağacının her adımda nasıl göründüğünü tahmin edebiliyor musunuz? Yanıtlarınızı kontrol etmek için `-c` bayrağını kullanın. Bazı farklı rastgele tohumlar `(-s)` deneyin veya onu asmak için daha fazla eylem `(-a)` ekleyin .

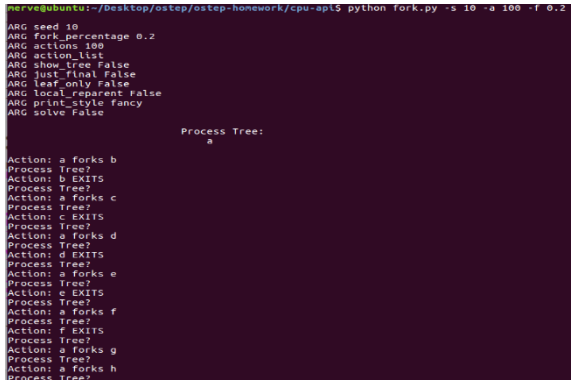
Birinci actionda a parentına b child ı ekleniyor. 2. Actionda a ya c childını da ekliyor. 3. Actionda c childini exitliyor yani ağaçta a parent çocuk olarak da b kalıyor. 4. actionda a ya d ekleniyor. 5. Actionda a ya e eklenir son görüntü a parent b, d, e child oluyor.

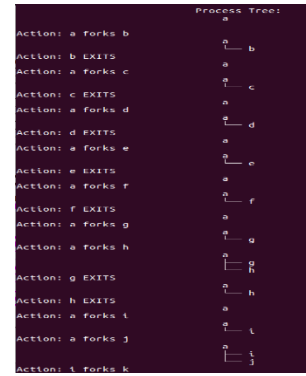




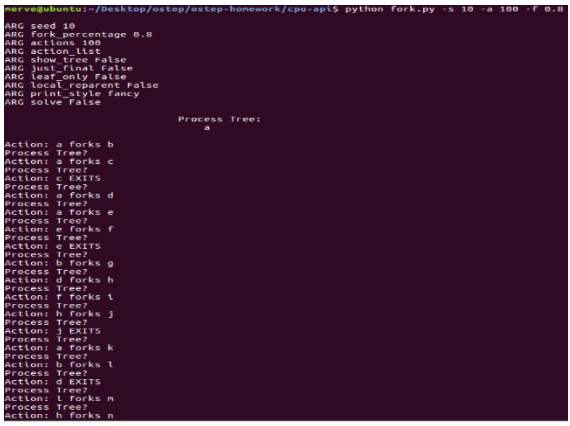
2.) Simülatörün size verdiği bir kontrol, `-f` bayrağı tarafından kontrol edilen `fork_percentage`. Ne kadar yüksek olursa, bir sonraki eylemin fork olma olasılığı o kadar artar; ne kadar düşük olursa, eylemin bir çıkış olma olasılığı o kadar yüksektir. Simülatörü çok sayıda eylemle çalıştırın (örneğin, `-a 100`) ve `fork_percentage` 0,1 ila 0,9 arasında değiştirin. Yüzdesi değiştikçe ortaya çıkan son süreç ağaçlarının nasıl görüneceğini düşünüyorsunuz? Cevabınızı -c ile kontrol edin.

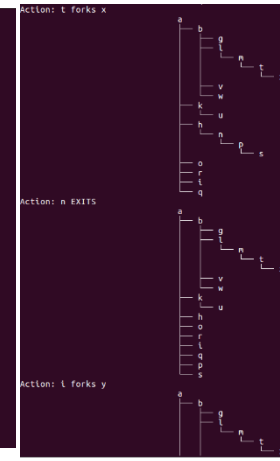
Burada ilk olarak yüzdeyi (-f değerini) düşük olarak (0.2) vermek istedim ve sonucunda exit actionların fazla olduğunu gözlemlemekteyiz. Yan ekran görüntüsünde de ağaç yapısını görmekteyiz.





Burada ise yüzdeye (-f değerine) yüksek bir değer (0.8) verdim ve sonucunda fork işlemlerinin daha fazla olduğunu gözlemlemekteyiz. Yan ekran görüntülerinde ise işlemlerin tamamamını ekleyemediğimden parçalı olarak ağaç yapısını ekledim.





3.) Şimdi, `-t` bayrağını kullanarak çıktıyı değiştirin (örneğin, `./fork.py -t`). Bir dizi süreç ağacı göz önüne alındığında, hangi eylemlerin yapıldığını söyleyebilir misiniz?

```
merve@ubuntu:~/Desktop/ostep/ostep-homework/cpu-api$ python fork.py -s 10 -t

ARG seed 10
ARG fork_percentage 0.7
ARG actions 5
ARG action_list
ARG show_tree True
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve False

                        Process Tree:
                            a

Action?

                            a
                            └─ b

Action?

                            a
                            ├─ b
                            └─ c

Action?

                            a
                            └─ b

Action?

                            a
                            ├─ b
                            └─ d

Action?

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

Bu sorunun cevabı:
Action: a forks b
Action: a forks c
Action: c exits
Action: a forks d
Action: a forks e

4.) Unutulmaması gereken ilginç bir şey, bir çocuk çıktığında ne olduğudur; süreç ağacındaki çocuklarına ne olur? Bunu incelemek için belirli bir örnek oluşturalım: `./fork.py -A a+b,b+c,c+d,c+e,c-`.Bu örnekte 'a' 'b' oluşturma işlemi vardır, bu da 'c' oluşturur ve ardından 'd' ve 'e ' oluşturur. Ancak, o zaman, 'c' çıkar. Sizce süreç ağacı çıkıştan sonra nasıl olmalı? -R bayrağını kullanırsanız ne olur? Daha fazla bağlam eklemek için artık işlemlere kendi başınıza ne olduğu hakkında daha fazla bilgi edinin. `-R` bayrağını kullanırsanız ne olur? Daha fazla bağlam eklemek için yetim işlemlere kendi başınıza ne olduğu hakkında daha fazla bilgi edinin.

`Fork.py -A a+b,b+c,c+d,c+e,c-` işleminde + işaretleri aslında fork anlamına geçiyor. Yani a dan b yi, b den c yi, c den d yi ve e yi oluşturuyor. En sonunda c yi exitliyor. Bunun sonucunda a ilk parent olduğu için c nin çocukları olan d ve e yi a nın çocukları olarak yapıyor.

```
merve@ubuntu:~/Desktop/ostep/ostep-homework/cpu-api$ python fork.py -A a+b,b+c,c+d,c+e,c- -c

ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list a+b,b+c,c+d,c+e,c-
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True

                        Process Tree:
                            a

Action: a forks b
                            a
                            └─ b
Action: b forks c
                            a
                            └─ b
                               └─ c
Action: c forks d
                            a
                            └─ b
                               └─ c
                                  └─ d
Action: c forks e
                            a
                            └─ b
                               └─ c
                                  ├─ d
                                  └─ e
Action: c EXITS
                            a
                            ├─ b
                            ├─ d
                            └─ e
```

Ama buna -R flag ini eklersek c nin parenti olan b ye d ve e yi çocuk olarak yapacaktır.

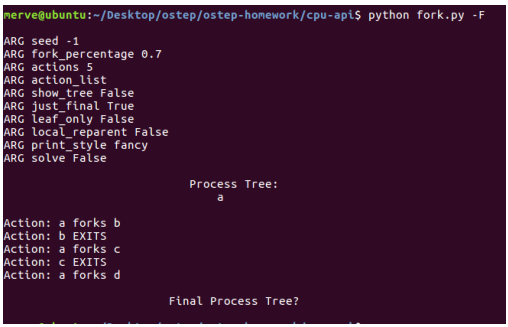
```
merve@ubuntu:~/Desktop/ostep/ostep-homework/cpu-api$ python fork.py -R -A  a+b,b+c,c+d,c+e,c- -c

ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list a+b,b+c,c+d,c+e,c-
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent True
ARG print_style fancy
ARG solve True

                        Process Tree:
                            a

Action: a forks b
                            a
                            └── b
Action: b forks c
                            a
                            └── b
                                └── c
Action: c forks d
                            a
                            └── b
                                └── c
                                    └── d
Action: c forks e
                            a
                            └── b
                                └── c
                                    ├── d
                                    └── e
Action: c EXITS
                            a
                            └── b
                                ├── d
                                └── e
```

Çıktıda da gördüğümüz gibi c slindikten sonra c nin çocukları olan d ve e b nin çocukları olmuştur.

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

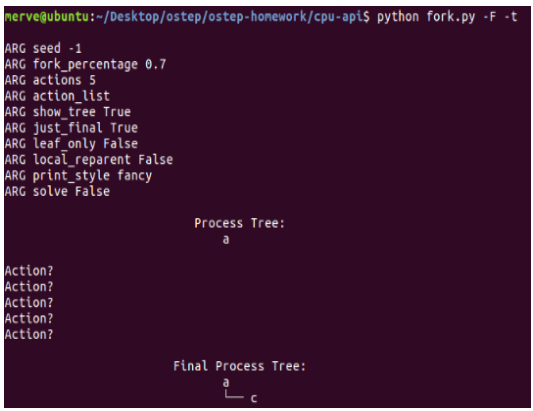
```
merve@ubuntu:~/Desktop/ostep/ostep-homework/cpu-api$ python fork.py -F

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 forks c
Action: b forks d
Action: b forks e
Action: b forks f

                    Final Process Tree?
```

Birinci yazışımda bu actionlar çıktı. Bunun final süreç ağacı a dan b ve b den de c, d, e, f türemiş şeklinde olacaktır.



Bunda ise önce a dan b yi oluşturuyor. Sonra b yi siliyor. A dan c yi oluşturuyor. C yi siliyor. En son a dan d yi oluşturuyor. Yani son süreç ağacımızda a dan sadece d oluşmuş şekilde göreceğiz.

6.) Son olarak, hem -t hem de -F'yi birlikte kullanın. Bu, son işlem ağacını gösterir, ancak daha sonra gerçekleşen eylemleri doldurmanızı ister. Ağaca bakarak, gerçekleşen eylemleri tam olarak belirleyebilir misiniz? Hangi durumlarda söyleyebilirsiniz? Hangisinde söyleyemezsin? Bu soruyu araştırmak için baz ı farklı rastgele tohumlar deneyin



Burada 5 tane bilmediğimiz eylemler var. Son süreç ağacına baktığımızda görüyoruz ki bu eylemlerin bir tanesi a forks c eylemi. Fakat diğer 4 eylem farklı şekillerde olabilir. Bir tane tahmin etmek istersem : **Action**: a forks b

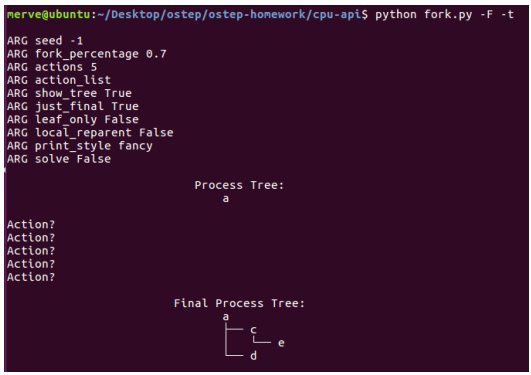
Actions: b EXITS

Action: a forks c

Actions: a forks d ya da c forks d olabilir

Action: d EXITS

Şeklinde olsa yine aynı sonucu görecektik



Burada ise kesin olarak bildiğimiz eylemler

Action: a forks c

Action: a forks d

Action: c forks e

Geriye kalan iki eylem için şöyle bi tahmin yapabiliriz: **Action**: a forks b

Action: b EXITS şeklinde olabilir .

Kenar: CODING EV ÖDEVLERI

Kodlama ödevleri, bazı temel işletim sistemi API'leri hakkında biraz deneyim kazanmak için gerçek bir makinede çalıştırmak üzere kod yazdığınız küçük alıştırmalardır . Sonuçta, (muhtemelen) bir bilgisayar bilimcisisiniz ve bu nedenle kodlamak istemelisiniz, değil mi? Bunu yapmazsanız, her zaman CS teorisi vardır, ancak bu oldukça zordur. Tabii ki, gerçekten bir uzman olmak için, makineyi hacklemek için biraz zaman harcamanız gerekir; Gerçekten de, biraz kod yazmak ve nasıl çalıştığını görmek için elinizden gelen her bahaneyi bulun. Zaman harcayın ve olabileceğinizi bildiğiniz bilge usta olun.

Ödev (Kod)

Bu ödevde, az önce okuduğunuz süreç yönetimi API'lerine biraz aşinalık kazanacaksınız. Endişelenmeyin - göründüğünden daha eğlenceli! Genel olarak, biraz kod yazmak için mümkün olduğunca fazla zaman bulursanız çok daha iyi durumda olacaksınız, o zaman neden şimdi başlamıyorsunuz?

Sorular

a. fork()'ı çağıran bir program yazın. Fork() öğesini çağırmadan önce, ana işlemin bir değişkene (örneğin, x) erişmesini ve değerini bir şeye (örneğin, 100) ayarlamasını sağlayın. Alt işlemdeki değişken hangi değerdir? Hem alt hem de üst öğe x değerini değiştirdiğinde değişkene ne olur?

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

int
main(int argc, char *argv[])
{
    int x = 100;
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        //x = 50;
        printf("child x = %d\n", x);
    } else {
        // parent goes down this path (original process)
        printf("parent x = %d\n", x);
    }
    return 0;
}
```

mainde x e 100 değerini verdik ve bir child oluşturduk. Ilk başta aşağıdaki gibi ikisinde de 100 gözükecektir

```
merve@ubuntu:~/Desktop/kodlar$ gcc -c soru1.c
merve@ubuntu:~/Desktop/kodlar$ gcc soru1.o -o soru1
merve@ubuntu:~/Desktop/kodlar$ ./soru1
parent x = 100
child x = 100
```

ikinci olarak child x ini 50 yaptığımızda parentte bir değişiklik olmadığını gözlemleriz.

```
merve@ubuntu:~/Desktop/kodlar$ gcc -c soru1.c
merve@ubuntu:~/Desktop/kodlar$ gcc soru1.o -o soru1
merve@ubuntu:~/Desktop/kodlar$ ./soru1
parent x = 100
child x = 50
```

aynı şekilde parent x ini 40 yaptığımızda child x inde bi değişiklik olmadığını görürüz.

```
merve@ubuntu:~/Desktop/kodlar$ gcc soru1.o -o soru1
merve@ubuntu:~/Desktop/kodlar$ ./soru1
parent x = 40
child x = 50
```

b. Bir dosyayı açan (open() sistem çağrısıyla) ve ardından yeni bir işlem oluşturmak için fork() öğesini çağıran bir program yazın . Hem çocuk hem de ebeveyn, open() tarafından döndürülen dosya tanımlayıcısına erişebilir mi? Dosyaya eşzamanlı, yani aynı anda yazarken ne olur?

c. Fork() kullanarak başka bir program yazın. Alt işlem "merhaba" yazdırmalıdır; ana işlem "hoşçakal" yazdırmalıdır. Alt işlemin her zaman önce yazdırıldığından emin olmaya çalışmalısınız; bunu ebeveynde **wait()** *çağırmadan* yapabilir misiniz ?

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

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("merhaba\n");
    } else {
        // parent goes down this path (original process)
        printf("hoscakal\n");
    }
    return 0;
}
```

Kodu çalıştırdığımda her zaman parent önce çalıştı

```
merve@ubuntu:~/Desktop/kodlar$ ./soru3
hoscakal
merhaba
merve@ubuntu:~/Desktop/kodlar$ ./soru3
hoscakal
merhaba
merve@ubuntu:~/Desktop/kodlar$ ./soru3
hoscakal
merhaba
```

d. fork() öğesini çağıran ve ardından /bin/ls programını çalıştırmak için bir tür exec() çağıran bir program yazın. (Linux'ta) execl(), execle(), execlp(), execv(), execvp() ve execvpe()) dahil olmak üzere exec()'nin tüm varyantlarını deneyip deneyemeyeceğinize bakın. Aynı temel çağrının neden bu kadar çok çeşidi olduğunu düşünüyorsunuz ?

e. Şimdi alt işlemin üst öğede bitmesini beklemek için wait() kullanan bir program yazın. wait() ne döndürür? Çocukta wait() kullanırsanız ne olur?

a. Önceki programın küçük bir değişikliğini yazın, bu sefer waitpid() yerine wait(). waitpid() ne zaman işe yarar?

b. Alt işlem oluşturan bir program yazın ve sonra alt öğede standart çıktıyı (STDOUT FILENO) kapatın. Çocuk tanımlayıcıyı kapattıktan sonra bazı çıktıları yazdırmak için printf() öğesini çağırırsa ne olur?

c. İki alt öğe oluşturan ve pipe() sistem çağrısını kullanarak birinin standart çıkışını diğerinin standart girişine bağlayan bir program yazın.