**UNIX/Linux Sistemlerinde exec İşlemleri**

    *UNIX/Linux* sistemlerinde *fork* fonksiyonu prosesin yeni ve özdeş bir kopyasını oluşturmaktadır. Yani *fork* işlemi sonrasında alt ve üst prosesler aynı program kodunu (muhtemelen onların farklı kısımlarını) çalıştırıyor durumda olurlar. Halbuki pek çok uygulamada programcı yaratmış olduğu alt prosesin farklı bir program kodunu çalıştırmasını ister. İşte *exec* fonksiyonları prosesin başka bir program olarak çalışmaya devam etmesini sağlamaktadır. *exec* işlemleri sonrasında prosesin *id* değeri ve kontrol bloğu değişmez. Prosesin kod, *data* ve *bss* alanları çalıştırılabilen (*executable*) dosyadan alınarak yüklenir. Proses artık yaşamını başka bir program olarak sürdürür.

*exec* işlemi yapan altı farklı fonksiyonvardır. Bu fonksiyonların hepsinin isimleri *exec* ile başladığından bunlara kısaca *exec* fonksiyonları diyeceğiz. Tüm *exec* fonksiyonlarının prototipleri *<unistd.h>* dosyası içersindedir. Aşağıda bu fonksiyonların prototiplerini göreceksiniz. Bu prototiplerdeki yorum satırları kafanızı karıştırmasın. Bunları yalnızca değişken sayıda argümanlar hakkında ipucu vermek amacıyla yerleştirdik:

#include <unistd.h>

int execl(const char \*path, const char \*arg0, ... /\*,(char \*) 0 \*/);

int execv(const char \*path, char \*const argv[]);

int execle(const char \*path, const char \*arg0, ... /\*, (char \*) 0, char \*const envp[] \*/);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

int execlp(const char \* file, const char \*arg0, ... /\*, (char \*) 0 \*/);

int execvp(const char \*file, char \*const argv[]);

Fonksiyonların parametrik benzerlikleri dikkatinizi çekmiştir. *exec* fonksiyonlarının *l*’li (*execl*, *execle*, *execlp*) ve *v*’li (*execv*, *execve*, *execvp*) biçimleri vardır. Bunların *l*’li biçimleri komut satırı argümanlarını bir liste olarak, *v*’li biçimleri ise bir dizi olarak alırlar. Ayrıca bazı *exec* fonksiyon isimlerinin *‘e’* ya da *‘p’* harfiyle sonlandırıldığını göreceksiniz. Fonksiyonların *e*’li biçimleri çevre değişkenlerini (*environment variables*) de programcıdan istemektedir. *p*’li biçimler çalıştırılabilen dosyanın yerinin belirlenmesinde *PATH* çevre değişkenlerini dikkate alırlar.

*exec* fonksiyonlarıyla bir programın çalıştırılabilmesi için fonksiyonu çağıran prosesin, yol ifadesindeki tüm dizinler için ve çalıştırılabilen dosyanın kendisi için *‘x’* hakkına sahip olması gerekir. Fakat *exec* işlemleri için ilgili prosesin çalıştırılabilen dosyanın kendisi için ya da yol ifadesindeki dizinler için *‘r’* hakkına sahip olması gerekmemektedir. Dosyalar için (dizinler için değil) *‘x’* hakkı kontrolünün yalnızca *exec* fonksiyonları tarafından yapıldığını vurgulayalım. *exec* fonksiyonları çalışmakta olan programın bellek alanını boşaltırlar, sonra çalıştırılmak istenen program dosyasını prosesin bellek alanına yükleyerek ilgili programı çalıştırırlar. *exec* fonksiyonları başarı durumunda yeni bir program kodunun çalışmasını sağladıkları için zaten geri dönmezler.  Bu fonksiyonlar başarısızlık durumunda pek çok *POSIX* fonksiyonunda olduğu gibi -1 değerine geri dönmektedir. *exec* fonksiyonlarının başarısızlık durumunda*errno* değişkenine yerleştirdikleri önemli değerler şunlardır:

|  |  |  |
| --- | --- | --- |
| E2BIG | : | Komut satırı argümanları ya da çevre değişkenleri izin verilen miktardan fazladır. |
| EACCESS | : | Prosesin birinci parametrede belirtilen yol bileşenine ilişkin dizinlerden en az birine ya da çalıştırılabilen dosyanın kendisine *‘x’* hakkı yoktur ya da dosya çalıştırılabilir özellikte değildir. |
| EINVAL | : | Dosya çalıştırılabilir bir formata sahiptir fakat ilgili sistem tarafından desteklenmemektedir. |
| ENOENT | : | Fonksiyonun birinci parametresine ilişkin yol ifadesinde belirtilen bir dizin yoktur ya da bu yol ifadesine ilişkin bir yol bileşeni bir dizin girişi belirtmemektedir ya da bu yol ifadesi boş bir*string*’ten oluşmaktadır. |
| ENOEXEC | : | Prosesin dosyaya erişim hakları yeterlidir fakat dosyanın formatı belirlenememiştir. |
| ENOMEM | : | Programın çalıştırılabilmesi için gerekli olan bellek yetersizdir. |
| … | : | … |

 *exec* fonksiyonlarının nasıl çalıştığını daha iyi anlayabilmek için onları ikişerli gruplayarak ele alacağız.

**execl ve execv Fonksiyonları**

*execl* fonksiyonu komut satırı argümanlarını bir liste olarak, execv ise bir dizi olarak almaktadır. Önce *execl* üzerinde duralım. *execl* fonksiyonunun prototipini yeniden inceleyiniz:

int execl(const char \*path, const char \*arg0, ...);

Fonksiyonun birinci parametresi çalıştırılabilen dosyanın yol ifadesini belirtmektedir. Yol ifadesi mutlak ya da göreli biçimde girilebilir. Fonksiyonun diğer parametreleri çalıştırılacak programın komut satırı argümanlarını belirtmektedir. Komut satırı argümanları *NULL* adres ile sonlandırılmış bir dizilim biçiminde girilmelidir. Prototipin sonundaki … (*ellipsis*) atomunun *C*ve *C++*’ta değişken sayıda argüman belirlemesi için kullanıldığını biliyor olmalısınız.

Şimdi *execl* fonksiyonunun kullanımına bir örnek verelim:

/\* exectest1.c \*/

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

int main(void)
{
    if (execl("/bin/ls", "ls", "-l", (char \*) 0) < 0) {
        perror("execl");
        exit(EXIT\_FAILURE);
    }

    printf("Tamam\n");

    return 0;
}

Derleme işlemini şöyle yapabilirsiniz:

gcc –o exectest1 exectest1.c general.c

Buradaki programda *exectest1* isimli proses çalışmasına *ls* programının kodlarıyla devam etmektedir. *execl* başarılı olduğunda geri dönmez. Dolayısıyla yukarıdaki örnekte ekrana*“Tamam”* yazısı çıkmayacaktır. Çalışmakta olan programın tüm bellek alanının boşaltılıp *exec* fonksiyonlarında belirtilen program dosyasının yükleneceğine dikkat ediniz.

*execl* fonksiyonundaki komut satırı argüman listesinin sonlandırılması hakkında bir nokta üzerinde durmak istiyoruz. Yukarıda komut satırı argümanlarının *NULL* adresle sonlandırılması gerektiğini belirtmiştik. Bildiğiniz gibi *C*’de ve *C++*’ta sıfır değerini veren tamsayı türlerine ilişkin sabit ifadeleri *NULL* adres sabiti olarak kullanılabilmektedir. (Örneğin, 0, 3 – 3, 10L – 10L gibi). Yine *C*’de (fakat *C++*’ta değil) sıfır değerini veren sabit ifadesinin *void \** türüne dönüştürülmüş hali de (yani *(void \*) 0* ifadesi) *NULL* adres sabiti olarak ele alınmaktadır. Fakat değişken sayıda argüman olan fonksiyonlarda girilen sıfır değerleri *NULL* adres sabiti olarak değil, tamsayı türlerine ilişkin sıfır değeri olarak ele alınırlar. İşte bu nedenle yukarıdaki örnekte 0 değerini *char \** türüne dönüştürmek zorunda kaldık. *execl* fonksiyonunun aşağıdaki gibi çağrılması hataya yol açma potansiyeli içermektedir:

execl("/bin/ls", "/bin/ls", "-l", 0);

Burada artık 0 değeri *int* türden 0 olarak yığına aktarılır. Bunun ise iki sakıncası söz konusu olabilir:

1. *NULL* adres sabiti bellekte sıfır numaralı adresi belirtmek zorunda değildir. Dolayısıyla burada aktarılan argüman *NULL* adres sabiti anlamına gelmeyebilir. (Tabi yaygın derleyicilerin hemen hepsinde *NULL* adres sabitinin sayısal değeri 0 olarak alındığı için bu sorun olumsuz bir etkiye sahip olmayacaktır.)

2. Adres türleriyle *int* türünün aynı uzunlukta olması zorunlu değildir. Örneğin 64 bit sistemlerin çoğunda *int* türü 4 *byte* bit iken adres türleri 8 byte uzunluğundadır. Bizim girdiğimiz *int*türden 0 değeri yığından çekildiğinde rastgele bir adres değeri haline gelebilecektir.

*C* ve *C++* standartlarında eğer *argc* değeri sıfırdan büyükse programın ilk komut satırı argümanının (yani *argv[0]*’ın) program ismini göstereceği belirtilmiştir. Fakat ilgili standartlarda program isminin ifade biçimi hakkında bir belirlemede bulunulmamıştır. Örneğin program ismi yol ifadesi içerebilir ya da içermeyebilir. Tabi *C* ve *C++* standartları bunları söylüyor olsa da*UNIX/Linux*sistemlerinde biz *exec* işlemlerini uygularken programın ilk komut satırı argümanını program ismi biçiminde girmek zorunda değiliz. Fakat genel beklenti bu yönde olduğu için program isminin ilk komut satırı argümanı olarak geçirilmesi iyi bir tekniktir. Dikkat ederseniz biz de yukarıdaki örnekte *ls* programını çalıştırırken ilk argümanı *“/bin/ls”* biçiminde program ismini belirtecek biçimde oluşturduk.[1]

    *execv* fonksiyonu *execl* fonksiyonunun komut satırı argümanlarını bir dizi olarak alan biçimidir. Prototipini inceleyiniz:

int execv(const char \*path, char \*const argv[]);

Fonksiyonun birinci parametresi çalıştırılabilen dosyanın yol ifadesini, ikinci parametresi komut satırı argümanlarının yerleştirileceği dizinin başlangıç adresini alır. Yani komut satırı argümanlarına ilişkin yazıların adresleri bir gösterici dizisine yerleştirilmeli ve  bu dizinin adresi fonksiyona argüman olarak geçilmelidir. Komut satırı argümanlarının yerleştirileceği dizinin son elemanında *NULL* adres olmalıdır.[2]

Aşağıdaki örnek *execv* fonksiyonunun parametrik yapısını daha iyi anlamanıza yardımcı olacaktır:

/\* exectest2.c \*/

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

int main(void)
{
    char \*args[] = {"/bin/ls", "-l", NULL};

    if (execv("/bin/ls", args) < 0) {
        perror("execv");
        exit(EXIT\_FAILURE);
    }

    printf("Tamam\n");

    return 0;
}

Şimdi de aşağıdaki örneği inceleyiniz:

/\* exectest3.c \*/

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

int main(int argc, char \*argv[])
{
    if (argc == 1) {
        fprintf(stderr, "Yanlis sayida arguman!..\n");
        exit(EXIT\_FAILURE);
    }

    if (execv(argv[1], &argv[1]) < 0)  {
        perror("execv");
        exit(EXIT\_FAILURE);
    }

    return 0;
}

Bu program komut satırından alınan dosya ismini *execvp* fonksiyonuyla çalıştırıyor. *argv* dizisinin son elemanının *NULL* adres olduğunu anımsayınız. Programı şöyle derleyebilirsiniz:

gcc –o exectest3 exectest3.c general.c

Programı şöyle çalıştırdığımızı düşünelim:



Burada *“/bin/ls”* programı çalıştırılmış ve *“/bin/ls”* ile *“–l”* komut satırı argümanları olarak geçirilmiştir.

**execlp ve execvp Fonksiyonları**

    *execlp* fonksiyonu *execl* fonksiyonunun *PATH* çevre değişkenini dikkate alan biçimidir. Fonksiyon *execl* ile aynı parametrik yapıya sahiptir.

int execlp(const char \*file, const char \*arg0, ...);

*execlp* fonksiyonunun birinci parametresiyle belirtilen yol ifadesi ‘/’ karakteri içermiyorsa burada belirtilen dosya ismi *execlp* fonksiyonunu uygulayan prosesin *PATH* çevre değişkeninde belirtilen dizinlerde sırasıyla aranır. Örneğin *execlp* fonksiyonu şöyle uygulanmış olsun:

if (execlp("ls", "ls", "-ls", (char \*) 0) < 0) {
    perror("execlp");
    exit(EXIT\_FAILURE);
}

Prosesin *PATH* çevre değişkeninin aşağıdaki gibi olduğunu varsayalım:

PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11

*PATH* çevre değişkeni ‘:’ karakterleriyle ayrıştırılmış alanlardan oluşmaktadır. Bu alanlarda dizinlerin yol ifadeleri bulunur. *exec* fonksiyonlarının *p*’li biçimleri *PATH* çevre değişkenindeki bu alanları sırasıyla gözden geçirirler. Bu örneğimizde *“ls”* program dosyası önce “/usr/local/bin” dizini içerisinde, sonra “/usr/bin” dizini içerisinde aranacak ve arama işlemi böyle sürdürülecektir. Bu aramalar sırasında dosya bulunursa arama devam etmeyecektir. (*ls* program dosyasının /bin dizininde bulunduğunu anımsayınız.) Burada önemli olan bir noktaya dikkatinizi çekmek istiyoruz. *exec* fonksiyonlarının *p*’li biçimleri prosesin çalışma dizinine hiç bakmamaktadır. Yani *PATH* çevre değişkeninin yukarıdaki gibi bir içeriğe sahip olduğu durumda biz çalışma dizinindeki bir programı *exec* fonksiyonlarının *p*’li biçimleriyle çalıştıramayız. Fakat *PATH* çevre değişkeninde *‘.’* ifadesi ile prosesin çalışma dizini için de yol belirtilebilir. Örneğin:

PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:.

Burada *execlp* fonksiyonu eğer program dosyasını diğer dizinlerde bulamazsa prosesin çalışma dizinine de bakacaktır.

*exec* fonksiyonlarının *p*’li biçimlerinde belirtilen program dosyasına ilişkin yol ifadesi herhangi bir yerinde ‘/’ karakteri içeriyorsa bu durumda *PATH* çevre değişkenine bakılmaz. Bu durumda yol ifadesi mutlaksa kök dizinden, göreliyse prosesin çalışma dizininden itibaren yer belirtir. Örneğin:

if (execlp("project/sample", "project/samples", (char \*) 0) < 0) {
    perror("execlp");
    exit(EXIT\_FAILURE);
}

Burada *PATH* çevre değişkenine hiç bakılmayacaktır. *sample* isimli program dosyası prosesin çalışma dizininin altındaki *project* dizininin altında aranacaktır.

    *execvp* fonksiyonu *execlp* fonksiyonunun komut satırı argümanlarını bir dizi ile biçiminde alan biçimidir. Prototipini yeniden inceleyiniz:

int execvp(const char \*file, char \*const argv[]);

Fonksiyonun birinci parametresi program dosyasını ikinci parametresi komut satırı argümanlarını belirtmektedir. Komut satırı argümanlarının veriliş biçimi dışında fonksiyonun *execlp*fonksiyonundan herhangi bir çalışma farklılığı farkı yoktur. Fonksiyonu aşağıdaki örnek kodla test edebilirsiniz:

/\* exectest4.c \*/

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

int main(void)
{
    char \*args[] = {"ls", "-l", NULL};

    if (execvp("ls", args) < 0) {
        perror("execvp");
        exit(EXIT\_FAILURE);
    }

    return 0;
}

    Kabuk üzerinde biz bir programı çalıştırmak istediğimizde kabuk bu programı *exec* fonksiyonlarının *p*’li biçimleriyle çalıştırmaktadır. Bu nedenle biz kabuk üzerinden bulunduğumuz dizindeki bir programı doğrudan çalıştıramayız. Çünkü *exec* fonksiyonlarının *p*’li biçimleri -eğer *PATH* çevre değişkeninde yoksa- dosyayı prosesin çalışma dizininde  aramamaktadır. Örneğin *bash* kabuğu üzerinde bulunduğumuz dizindeki *sample* isimli bir programı şöyle çalıştırmaya çalışalım:



Gördüğünüz gibi *bash* kabuğu *sample* isimli programı ya da komutu bulamadığını söylüyor. Halbuki *exec* fonksiyonlarının *p*’li biçimleri eğer dosyaya ilişkin yol ifadesinde ‘/’ karakteri varsa artık *PATH* çevre değişkenine bakmamaktadır. Dolayısıyla biz böylesi bir durumda bulunduğumuz dizindeki programı kabuk içerisinden şöyle çalıştırabiliriz:


 **execle ve execve Fonksiyonları**

*exec* fonksiyonlarının *e*’li biçimleri bize yeni bir programı çalıştırırken prosesin çevre değişkenlerini de tamamen değiştirme olanağını vermektedir. *execlp* ve *execve* fonksiyonlarının prototiplerini bir kez daha inceleyiniz:

int execle(const char \*path, const char \*arg0, ... /\*, (char \*) 0, char \*const envp[] \*/);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

Her iki fonksiyon da çevre değişkenlerini bir gösterici dizisi olarak almaktadır. Gösterici dizisinin her elemanı *“Değişken=Değer”* biçiminde bir yazıyı gösterir. Gösterici dizisinin *NULL*adresle sonlandırılması gerekir. *execle* fonksiyonu komut satırı argümanlarını liste biçiminde *execve* ise dizi biçiminde almaktadır. Şimdi bu fonksiyonların kullanımına ilişkin bir örnek verelim:

/\* dispenv. c\*/

#include <stdio.h>

extern char \*\*environ;

int main(void)
{
    int i;

    for (i = 0; environ[i] != NULL; ++i)
        puts(environ[i]);

    return 0;
}

/\*exectest5.c \*/

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

int main(void)
{
    char \*args[] = {"dispenv", NULL};
    char \*env[] = {"name=Kaan Aslan", "no=100", NULL};

    if (execve("dispenv", args, env) < 0)  {
        perror("execve");
        exit(EXIT\_FAILURE);
    }

    return 0;
}

/\*exectest6.c \*/

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

int main(void)
{
    char \*env[] = {"name=Kaan Aslan", "no=100", NULL};

    if (execle("dispenv", "dispenv", (char \*) 0, env) < 0) {
        perror("execle");
        exit(EXIT\_FAILURE);
    }

    return 0;
}

Burada üç iki ayrı program var. Üçünü de de ayrı ayrı derlemelisiniz:

gcc –o dispenv dispenv.c
gcc –o exectest5 exectest5.c general.c
gcc –o exectest6 exectest6.c general.c

*exectest5* programı *execve* fonksiyonunu, *exectest6* programı *execle* fonksiyonunu kullanarak *dispenv* programını çalıştırıyor. *dispenv* programı ise prosesin tüm çevre değişkenlerini yazdırıyor.

    Pek çok *UNIX* türevi sistemde tüm *exec* fonksiyonları değil yalnızca *execve* fonksiyonu bir sistem fonksiyonu olarak yazılmıştır. Geri kalan *exec* fonksiyonları normal kütüphane fonksiyonları biçiminde gerçekleştirilmiştir. Bu sistemlerde biz bir *exec* fonksiyonunu çağırdığımızda fonksiyon kendi içerisinde *execve* için gereken parametreleri oluşturur ve *execve*fonksiyonunu çağırır. Yani aslında bu sistemlerde *PATH* çevre değişkenlerine de sistem fonksiyonları değil, kullanıcı modundaki kütüphane fonksiyonları bakmaktadır. *exec* fonksiyonları için parametre düzenlemesi şöyle yapılmaktadır:

execl: Komut satırı argümanlarını bir gösterici dizisine yerleştirir. Bu diziyi ve prosesin üst prosesten aldığı çevre değişkenlerini temsil eden *environ* dizisini argüman yaparak *execve*fonksiyonunu çağırır.

execlp: Dosyanın yol ifadesinde ‘/’ karakteri kullanılmamışsa *PATH* çevre değişkenindeki her bir dizine sırasıyla bakarak belirtilen program dosyasını o dizinlerde arar, eğer dosyayı bulursa komut satırı argümanlarını bir gösterici dizisine yerleştirir, bu diziyi ve prosesin üst prosesten almış olduğu çevre değişkenlerini temsil eden *environ* dizisini argüman yaparak*execve* fonksiyonunu çağırır.

execv: Prosesin üst prosesten almış olduğu çevre değişkenlerini temsil eden *environ* dizisini de argüman yaparak *execve* fonksiyonunu çağırır.

execvp: Eğer dosyanın yol ifadesinde ‘/’ karakteri kullanılmamışsa *PATH* çevre değişkenindeki her bir dizine sırasıyla bakarak belirtilen program dosyasını o dizinlerde arar; eğer dosyayı bulursa komut satırı argümanlarıyla birlikte prosesin üst prosesten almış olduğu çevre değişkenlerini temsil eden *environ* dizisini de argüman yaparak *execve* fonksiyonunu çağırır.

execle: Komut satırı argümanlarını bir gösterici dizisine yerleştirir. Bu diziyi ve yine parametre yoluyla aldığı çevre değişken dizisini argüman yaparak *execve* fonksiyonunu çağırır.

*exec*fonksiyonlarının *execve* sistem fonksiyonunu nasıl çağırdıklarını aşağıdaki şekille de özetleyebiliriz:[3]



Son olarak da kütüphane fonksiyonlarının *execve* fonksiyonunu nasıl çağırdığını gerçek kaynak kodlar üzerinden açıklayalım. Aşağıda*glibc* (*version 2.11.2*) kütüphanesindeki *execl*fonksiyonunun kaynak kodlarını görüyorsunuz:

int execl (const char \*path, const char \*arg, ...)
{
#define INITIAL\_ARGV\_MAX 1024
    size\_t argv\_max = INITIAL\_ARGV\_MAX;
    const char \*initial\_argv[INITIAL\_ARGV\_MAX];
    const char \*\*argv = initial\_argv;
    va\_list args;

    argv[0] = arg;

    va\_start(args, arg);
    unsigned int i = 0;
    while (argv[i++] != NULL) {
        if (i == argv\_max) {
            argv\_max \*= 2;
            const char \*\*nptr = realloc(argv == initial\_argv ? NULL : argv,
                        argv\_max \* sizeof(const char \*));
            if (nptr == NULL) {
                if (argv != initial\_argv)
                    free(argv);
                return -1;
            }
            if (argv == initial\_argv)
                memcpy(nptr, argv, i \* sizeof (const char \*));
            argv = nptr;
        }
        argv[i] = va\_arg(args, const char \*);
    }
    va\_end(args);

    int ret = \_\_execve(path, (char \*const \*) argv, \_\_environ);
    if (argv != initial\_argv)
        free (argv);

    return ret;
}

Burada komut satırı argümanları için başlangıçta 1024 elemanlık bir yer ayrılmıştır. Eğer komut satırı argümanları bu değerden fazlaysa *realloc* fonksiyonu ile dizi büyütülmektedir. Değişken sayıda argümanlarının döngü içerisinde *va\_arg* makrosuyla elde edildiğini görüyorsunuz. Fonksiyonun en sonunda *\_\_execve* fonksiyonunun çağrıldığına dikkat ediniz. *\_\_execve*doğrudan *inline* sembolik makina dili koduyla *execve* sistem fonksiyonunu çağırmaktadır. *\_\_environ* global değişkeni *environ* değişkeninin kütüphane içerisindeki ismidir.

    *exec* fonksiyonları ile bir programı çalıştırdığımızda kendi programımızın sonlandırıldığını ve prosesin yaşamına başka bir program olarak devam ettiğini gördük. Fakat uygulamada çoğu kez programcılar hem kendi programının devam etmesini hem de yeni bir programın çalıştırılmasını isterler. İşte bu nedenle *exec* fonksiyonları genellikle tek başına değil *fork*fonksiyonuyla birlikte kullanılmaktadır. Tipik olarak programcı önce bir kez *fork* yaparak yeni bir proses yaratır, alt proses akışında da *exec* fonksiyonlarından birini uygulayarak alt prosesin prosesin başka bir programı çalıştırmasını sağlar. Örneğin:

pid\_t pid;
int status;
...
if ((pid = fork()) < 0)  {
    perror("fork");
    exit(EXIT\_FAILURE);
}

if (pid == 0)
    if (execl("sample", "sample", (char \*) 0) < 0) {
        perror("fork");
        \_exit(EXIT\_FAILURE);
    }

/\* Üst proses çalışmasına devam ediyor ... \*/

if (waitpid(pid, &status, 0) < 0) {
    perror("waitpidfork");
    exit(FAILURE);
}

Örneğimizde alt proseste *execl* uygulandığını görüyorsunuz. Eğer alt proseste uygulanan *exec* işlemi başarısızsa alt proses sonlandırılmıştır. Sonlandırmanın *\_exit*fonksiyonuyla yapılmış olmasına takılmayınız. Eğer *<stdio>* fonksiyonlarına ilişkin bir tampon aktarım sorunu yoksa sonlandırmanın *\_exit* ya da *exit* fonksiyonu ile yapılmasının da bir önemi yoktur. Örneğimizin sonunda alt prosesin yine *wait* fonksiyonlarıyla beklendiğine dikkat ediniz. Böylece alt proses sonlandığında hortlak olmaktan kurtarılmıştır. Aşağıda *fork* ve *exec* işlemlerinin birlikte uygulandığı basit bir örnek görüyorsunuz:

/\*exectest7.c \*/

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

void err\_sys(const char \*msg)
{
    perror(msg);
    fflush(stdout);
    exit(EXIT\_FAILURE);
}

int main(int argc, char \*argv[])
{
    pid\_t pid;
    int status;

    if (argc != 2)
        err\_usr("Yanlis sayida arguman girildi!\n");

    if ((pid = fork()) < 0)
        err\_sys("fork");

    if (pid == 0 && execl(argv[1], argv[1], (char \*) 0) < 0)
        err\_sys("execl");

    if (waitpid(pid, &status, 0) < 0)
        err\_sys("waitpid");

    if (WIFEXITED(status))
         printf("\nProssin cikis kodu: %d\n", WEXITSTATUS(status));

    return 0;
}

Program komut satırı ile aldığı programı çalıştırıyor, alt prosesin çalışması bittiğinde onun çıkış kodunu alarak ekrana yazdırıyor. Programı şöyle derleyebilirsiniz:

gcc –o exectest7 exectest7.c

Aşağıdaki gibi çalıştırabilirisniz:

./exectest7 /bin/ls

    Şimdi de *exec* işlemleri sonrasında alt prosesin dosya betimleyici tablosunun durumu üzerinde durmak istiyoruz. *fork* işlemi sonrasında prosesin betimleyici tablosunun alt prosese kopyalandığını anımsayınız. Burada küçük bir pürüz söz konusu olabilmektedir. Biz *exec* fonksiyonlarıyla tamamen farklı bir programı çalıştırdığımızda artık daha önce açık olan dosyaların yeni program için bir anlamı kalacak mıdır? Yeni program zaten kendi açtığı dosyalar için yeni betimleyiciler elde edecek ve bunları kullanacaktır. Üst prosesten gelen betimleyicilerle genellikle yeni çalıştırılan programın bir ilgisi olmaz. O halde mademki *exec* yapıldığında artık üst prosesteki bu bu betimleyicilerin alt prosesteki yeni çalıştırılan program için bir anlamı kalmamaktadır, o halde bu betimleyicilerin kapatılması uygun olabilir. İşte *UNIX* türevi sistemler *exec* işlemleri sırasında üst prosesten gelmiş olan betimleyicilerin otomatik olarak kapatılmasına olanak vermektedir. İzleyen praragrafta bunun nasıl yapıldığını göreceksiniz.

     *UNIX* türevi sistemler her açılan dosya için proses kontrol bloğunda bir *exec kapatma bayrağı (close on exec flag)* tutarlar. Dosya normal olarak *open* fonksiyonuyla açıldığında *exec*kapatma bayrağı kapalı durumdadır. Bu bayrak kapalıyken *exec* işlemleri sırasında ilgili dosyaya ilişkin betimleyici açık olarak kalır. Eğer bayrak açık duruma getirilirse (*set* edilirse) bu durumda *exec* işlemleri sırasında ilgili dosya betimleyicisi otomatik olarak kapatılır. Dosyaların *exec* kapatma bayrakları *fcntl* fonksiyonuyla açılabilir ya da kapatılabilir. *fcntl*fonksiyonunu 3. Bölümden biliyorsunuz. Bu fonksiyonun ikinci parametresinin *F\_SETFD* girilirse üçüncü parametresinin betimleyici bayrakları biçiminde girilmesi gerektiğini anımsayınız.*Exec* kapatma bayrağı betimleyici bayraklarında *FD\_CLOEXEC* ile tamsil edilmektedir. Bu durumda *fd* numaralı betimleyicinin *exec* kapatma bayrağını şöyle açabilirsiniz:

fcntl(fd, F\_SETFD, fcntl(fd, F\_GETFD) | FD\_CLOEXEC);

Betimleyicilerin *exec* kapatma bayraklarının varsayılan durumunun kapalı olduğunu bir kez daha anımsatalım. Benzer biçimde *dup*, *dup2* gibi fonksiyonlar da yeni bir betimleyici oluşturduklarında -eski betimleyicilerin durumu ne olursa olsun- yeni betimleyicilerin *exec* kapatma bayrağını kapatmaktadır. Yani bu fonksiyonlarla elde ettiğimiz betimleyiciler *exec*işlemleri sırasında yeni programda açık olmaya devam ederler.

    Son olarak *exec* fonksiyonlarının ilginç bir özelliğinden söz etmek istiyoruz. Bu fonksiyonlar bizim doğrudan betik (*script*) programlarını çalıştırmamıza da izin veriyorlar. *exec*fonksiyonları program dosyasını açtığında eğer ilgili dosya çalıştırılabilir bir formatta değilse onun ilk satırına bakarak aşağıdaki özel işlemi uygularlar:

1. *exec* yapılmak istenen dosyanın bir metin dosyası olduğunu varsayarak onun ilk satırını bakarlar. Normal olarak ilk satırdaki karakterlerin aşağıdaki kalıba uygun olması gerekir:

!# <yol ifadesi> [argüman]

Gördüğünüz gibi *exec* fonksiyonları açtıkları metin dosyasının ilk iki karakterine bakarlar,  eğer bu ilk iki karakter #! biçimindeyse bu karakterlerden sonraki boşluk karakterlerini atarak orada belirtilen yol ifadesini ve argüman listesini elde elderler. Buradaki yol ifadesi mutlaak ya da göreli olabilir. Yol ifadesini isteğe bağlı olarak bir argüman listesinin izleyebildiğine dikkat ediniz.

2. *exec* fonksiyonları açtıkları metin dosyasını değil, onun ilk satırından elde ettikleri yol ifadesinde belirtilen programı çalıştırırlar.

3. Çalıştırdıkları programa, *exec* işleminde ve betik dosyasının ilk satırında belirtilen komut satırı argümanlarını geçirirler.

Örneğin *testscript* isimli bir metin dosyasının içeriği şöyle olsun:

!# /bin/bas

for x in istanbul ankara izmir
do
    echo $x
done

Sonra dosyanın *‘x’* haklarını aşağıdaki gibi oluşturalım:

chmod +x testscript

Bu durumda yukarıdaki *“testscript”* dosyasına *exec* uyguladığımızda *exec* fonksiyonları bu dosyayı açacak ve onun ilk satırında belirtilen *“/bin/bash”* programını çalıştıracaklardır. Yani bu işlemin sonucunda *“testscript”* programı değil, *“/bin/bash”* programı çalıştırılmış olacaktır.

Şimdi komut satırı argümanlarının aktarılmasına ilişkin bazı ayrıntılar üzerinde durmak istiyoruz. e*xec* fonksiyonları ile *“dispargs”* isimli bir programı *“testscript”* isimli bir dosya yoluyla dolaylı biçimde çalıştırmak isteyelim. *“testscript”* dosyasının içeriği aşağıdaki gibi olsun:

#! dispargs ali veli selami

*“testscript”* dosyasını da *execscript* isimli programla *exec* işlemine sokalım:

/\* execscript \*/

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

int main(void)
{
    if (execl("testscript", "testscript", "ali", "veli", "selami", (char \*) 0) < 0) {
        perror("execve");
        exit(EXIT\_FAILURE);
    }

    return 0;
}

*dispargs* ise komut satırı argümanlarını ekrana yazdıran basit bir program:

/\* dispargs \*/

#include <stdio.h>

int main(int argc, char \*argv[])
{
    while (\*argv != NULL)
        puts(\*argv++);

    return 0;
}

Bu örnek için *dispargs*ve *execscript*programlarını aşağıdaki gibi ayrı ayrı derlemelisiniz:

gcc –o dispargs dispargs.c
gcc –o execscript execscript.c

*execscript* programını çalıştırdığımızda elde edilen sonuç şöyledir:

argv[0]:dispargs
argv[1]:ayse fatma zeynep
argv[2]:testscript
argv[3]:ali
argv[4]:veli
argv[5]:selami

Burada verdiğimiz örneği şöyle genelleştirebiliriz: *a* programı *b* isimli betik dosyasını çalıştırıyor olsun ve bu betik dosyası içerisinde *c* isimli program dosyasının belirtildiğini düşünelim. Bu durumda *c* dosyasına aktarılan komut satırı argümanları şöyle olacaktır:

argv[0]: Betik dosyasında belirtilen yol ifadesi (örneğimizde *c*)

argv[1]: Betik dosyasındaki komut satırı argümanları. (Yani *b* dosyasında *c* yol ifadesinin yanındaki argümanlar).Tüm argümanlar ayrı ayrı değil tek bir yazı biçiminde geçirilmektedir.

argv[2]: *exec* fonksiyonlarının birinci parametresinde belirtilen betik dosyasının yol ifadesi (Örneğimizde *b*)

argv[3] – argv[argc – 1]: *exec* fonksiyonlarında belirtilen komut satırı argümanları her biri ayrı bir argüman olarak

Son olarak bir nokta üzerinde durmak istiyoruz. Betik dosyalarını herhangi bir *exec* fonksiyonuyla çalıştırabiliriz. (Zaten pek çok sistemde diğer *exec* fonksiyonlarının dolaylı olarak*execve* fonksiyonunu çağırdığını biliyorsunuz.) *exec* fonksiyonlarının *p*’li biçimleri ile bir betik dosyasını çalıştırdığımızda betik dosyasının kendisi *PATH* çevre değişkeni ile belirtilen dizinlerde aranır. Betik dosyası içerisinde belirtilen dosyanın aranması sırasında *PATH* çevre değişkenine bakılmamaktadır. Betik dosyasının içerisinde belirtilen dosya yol ifadesinin mutlak ya da göreli olarak belirtilip belirtilmediğine göre kök dizinden ya da prosesin çalışma dizininden itibaren aranacaktır. Eğer betik dosyasını *exec* fonksiyonlarının *e*’li biçimiyle çalıştırırsak bu fonksiyonlarda belirttiğimiz çevre değişkeni betik dosyasında belirtilen programa geçirilecektir.

﻿

[1]*C*’nin *<stdio.h>* ve diğer bazı başlık dosyalarında bildirilen *NULL* sembolik sabiti *NULL* adres sabitini belirtmek zorundadır. *NULL* sembolik sabiti *C*’de aşağıdaki iki biçimden biri olarak define edilmiş olabilir:

#define NULL      0

#define NULL      ((void \*) 0)

[2]Fonksiyonun ikinci parametresindeki *const* niteleyicisinin konumu size tuhaf gelebilir. Bu durum göstericinin gösterdiği gösterici dizisinin *const* olduğunu belirtmektedir. Yani fonksiyonun ikinci parametresi bir gösterici dizisini göstermektedir ve fonksiyon bu gösterici dizisinin elemanlarında bir değişikilik yapmayacaktır. Yukarıdaki prototipin sektaks bakımından eşdeğeri şöyledir:

int execv(const char \*path, char \* const \*argv);

Fonksiyonun ikinci parametresi*const char \*\*argv* biçiminde olmadığına dikkat ediniz. Eğer bu parametre *const \*\*argv*biçiminde olsaydı, biz bu parametreye *const* olmayan bir gösterici dizisinin adresini argüman olarak geçiremezdik. Konuyu biraz daha açarsak, *C*’de ve *C++*’ta *T* bir tür belirtmek üzere *T \** türünden *const T \** türüne doğrudan dönüştürme olduğu halde, *T \*\** türünden *const T\*\** türüne doğrudan dönüştürme yoktur. Eğer böyle bir dönüştürme olsaydı aşağıdaki gibi hileli bir yöntemle *const* bir nesnenin içeriği değiştirilebilirdi:

const int x = 10;

int \*pi;

const int \*\*ppi;

ppi = &pi;          /\* geçersiz ama geçerli olduğunu varsayalım \*/

\*ppi = &x;          /\* geçerli (const int \* türü const int \* türüne atanıyor) \*/

Buradaki son işlemle aslında biz *pi*’nin içerisine *x*’in adresini atıyoruz. Artık *pi* yoluyla *x*’i değiştirebiliriz:

\*pi = 20;           /\* geçerli, x değiştiriliyor \*/