UNIX/Linux ve Windows Sistemlerinde stdin, stdout ve stderr Dosyaları

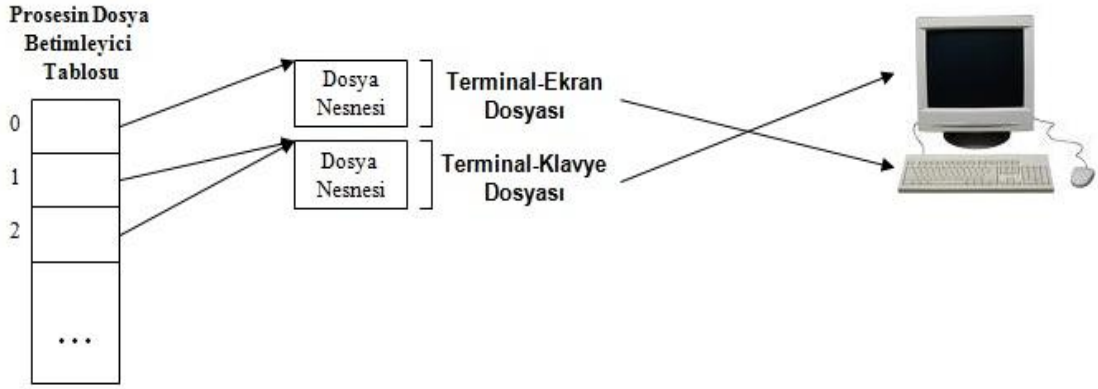
Kaan Aslan
31 Mart 2010

Yalnızca *UNIX/Linux* sistemlerinde değil modern işletim sistemlerinin çoğunda aygıtlar birer dosyaymış gibi ele alınmaktadır. Örneğin klavye ve ekran -aslında birer dosya olmadığı halde- işletim sistemi tarafından sanki birer dosyaymış gibi işleme sokulurlar. Aygıtlara ilişkin bu tür dosyalar için de birer dosya betimleyicisi (*file descriptor*) ve dosya nesnesi vardır. Bu betimleyicilerle işlem yapıldığında işletim sisteminin dosya alt sistemi aslında bu dosyaların birer aygıta ilişkin olduğunu anlar ve okuma/yazma amacıyla o aygıtlara yönelir. *UNIX* türevi sistemlerdeki çokbiçimliliği (*polymophism*) andıran bu tasarıma *Sanal Dosya Sistemi (Virtual File System)* denilmektedir.

Aygıtların birer dosyaymış gibi ele alınması bu aygıtların kullanımını kolaylaştırmaktadır. Böyle bir tasarımda bu aygıtlarla işlem yapacak programcının özel bir bilgiye sahip olmasına gerek kalmaz. Programcı normal bir dosyayla nasıl işlem yapıyorsa aygıtları temsil eden bu dosyalarla da aynı biçimde işlem yapar. Ayrıca böyle bir tasarımla aygıtlar üzerinde yönlendirme işlemleri de mümkün hale getirilmektedir. Örneğin, *stdout* dosyasını biz başka bir disk dosyasına yönlendirebiliriz. Bu durumda programın ekrana yazdırdığı şeyler aslında bu dosyaya yazılacaktır.

UNIX/Linux sistemlerinde bir proses yaratıldığında üç dosya betimleyicisi üst prosesten aktarılmaktadır. Bunlar *stdin*, *stdout* ve *stderr* dosyalarına ilişkin betimleyicilerdir. *stdin* standart giriş (*standard input*) dosyasıdır ve *default* olarak okuma amacıyla terminale (yani kalvyeye) yönlendirilmiş durumdadır. *stdout* standart çıkış (*standard output*) dosyasıdır ve *default* olarak yazma amaçlı terminale (yani ekrana) yönlendirilmiş durumdadır.[1] *stderr* ise programda oluşacak hata mesajlarının yazdırılması için kullanılan standart hata (*standard error*) dosyasıdır. Bu dosya da işin başında *default* olarak ekrana yönlendirilmiştir. *stdin*, *stdout* ve *stderr* dosyalarını programcı açmaz. Bu dosyalar proses çalışmaya başladığında zaten açık olarak üst prosesten aktarılmıştır. Programcının bu dosyaları kapatmasına gerek de yoktur. Proses sonlandığında zaten kapatma işlemi yapılacaktır.

POSIX sistemlerinde *stdin* dosyasına ilişkin betimleyici *0*, *stdout* dosyasına ilişkin betimleyici *1* ve *stderr* dosyasına ilişkin betimleyici de *2* numaralı betimleyicidir. Ayrıca bu betimleyici numaraları *<unistd.h>* dosyası içerisinde *STDIN_FILENO*, *STDOUT_FILENO* ve *STDERR_FILENO* sembolik sabitleriyle *de define* edilmiştir.



Gördüğünüz gibi proses çalışmaya başladığında *stdout* betimleyicisi de *stderr* betimleyicisi de terminal ekranına yönlendirilmiş durumdadır. Yani biz *stdout* dosyasına ya da *stderr* dosyasına yazma yaptığımızda bunlar ekrana yazılacaktır. Ancak daha sonra biz *stderr* dosyasını ya da *stdout* dosyasını başka dosyalara yönlendirebiliriz ve böylece programın hata mesajlarıyla normal mesajlarını birbirinden ayırabiliriz.

C'de kullandığımız *stdin*, *stdout* ve *stderr* akım isimleri <*stdio.h*> başlık dosyasında bildirilmiş makrolardır. Bu makrolar *FILE* * türünden birer dosya bilgi göstericisi belirtirler. Örneğin bir *C* derleyicisine ilişkin <*stdio.h*> başlık dosyasında bu makrolar şöyle bildirilmiştir:

```
extern FILE (*_imp___iob)[];
...
#define stdin     (&_iob[STDIN_FILENO])
#define stdout    (&_iob[STDOUT_FILENO])
#define stderr    (&_iob[STDERR_FILENO])
```

Burada kütüphane içerisinde standart dosyalar için bir *FILE* dizisinin yaratılmış olduğunu görüyorsunuz. *stdin*, *stdout* ve *stderr* makroları bu dizinin çeşitli elemanlarının adresini veriyor.

C'nin *stdin*, *stdout* ve *stderr* akım (*stream*) isimleri ile işletim sisteminin *stdin*, *stdout* ve *stderr* dosyalarını birbirine karıştırmayınız. *C*'nin bu dosyaları birebir işletim sisteminin *stdin*, *stdout* ve *stderr* dosyalarını kullanıyor olsa da standart *C* fonksiyonlarıyla çalışmakla doğrudan sistem fonksiyonlarıyla çalışmak arasında bazı işlevsel farklılıklar söz konusudur. *C*'nin *stdin*, *stdout* ve *stderr* akımları diğer dosyalarda olduğu gibi tamponlama (*buffering*) mekanizmasını kullanırlar. Örneğin aşağıdaki iki çağırma arasında işlevsel farklılık vardır:

```
printf("Ankara\n");
write(1, "Ankara\n", 7);
```

C standartlarına göre *printf* fonksiyonu *stdout* dosyasına yazmaktadır. *UNIX/Linux* sistemlerinde standart *C* fonksiyonları *stdout* olarak *1* numaralı dosya betimleyicisini kullanırlar. Bu durumda *printf* çağırması bir biçimde *write* fonksiyonunun *1* numaralı betimleyicisi ile çağrılmasına yol açacaktır. Fakat *printf* fonksiyonu -*stdout* akımının tamponlama moduna göre- yazıyı önce tampona yazıp oradan işletim sisteminin

stdout dosyasına aktarabilir. Oysa *write* fonksiyonu yazma işlemini doğrudan yapmaktadır. Özetle *UNIX/Linux* sistemlerinde çalışıyorsak *C*'deki *stdin*, *stdout* ve *stderr* akımlarını kullandığımızda tampon aracılığı ile okuma yazma yapmış oluruz.

Programlardaki normal mesajların *stdout* dosyasına, hata mesajlarının *stderr* dosyasına yazdırılması iyi bir tekniktir. Örneğin:

```
if ((f = fopen(path, "r")) == NULL) {
    fprintf(stderr, "Cannot open file: %s", path);
    exit(EXIT_FAILURE);
}
```

Eğer hata mesajlarını *stderr* dosyasına yazdırırsak programın normal çıktısı ile hata mesajlarını yönlendirme yaparak birbirinden ayırabiliriz. Örneğin programımızın ismi *sample* olsun:

```
./sample > test.txt
```

Burada *stdout* dosyasını *err.txt* isimli dosyaya yönlendirmiş olduk. Böylece programımızın *stdout* dosyasına yazdıkları *test.txt* dosyasına aktarılırken *stderr* dosyasına yazılanlar ekrana çıkmaya devam edecektir. Aynı işlemin tersini de şöyle yapabiliriz:

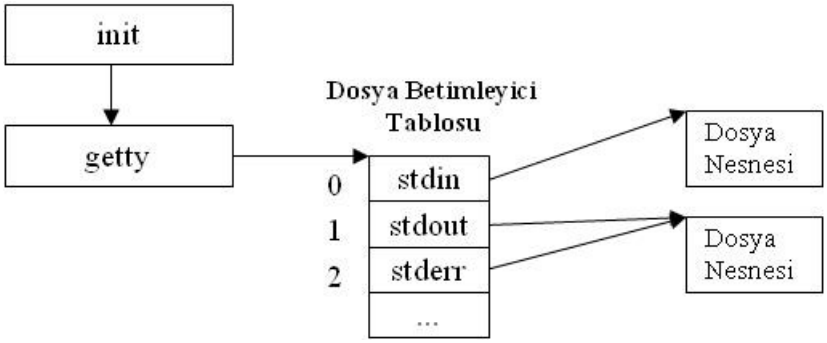
```
./sample 2> err.txt
```

Burada *2>* sentaksı *stderr* dosyasının yönlendirileceği anlamına gelir. Böylece programın *stderr* dosyasına yazdıkları *err.txt* dosyasına aktarılacak, *stdout* dosyasına yazılanlar ise yine ekrana çıkacaktır. *stdin*, *stdout* ve *stderr* dosyalarını aynı anda da yönlendirebiliriz:

```
./sample > a < b 2> c
```

Burada *stdout* dosyası *"a"* dosyasına, *stdin* dosyası *"b"* dosyasına ve *stderr* dosyası da *"c"* dosyasına yönlendirilmiştir.

UNIX/Linux sistemlerinde *stdin*, *stdout* ve *stderr* dosyalarına ilişkin betimleyiciler *fork* işlemi sırasında üst prosesten aktarılmaktadır. Yani zaten üst proseste bu dosyalar açık durumdadır. *exec* işlemleri sırasında da bu betimleyiciler kapatılmaz. Pekiyi *stdin*, *stdout* ve *stderr* dosyaları ilk kez hangi proses tarafından açılmıştır? Hikayeyi biraz geriden alarak açıklamaya çalışalım. *UNIX* türevi bir sisteme terminal yoluyla giriş yapılmak istendiğinde *init* prosesinden hareketle bir dizi işlem gerçekleştirilmektedir. Tipik olarak *init* prosesi *fork* ve *exec* fonksiyonlarını uygulayarak *getty* isimli bir programı çalıştırır. *init* prosesinin etkin kullanıcı *id*'si 0 (*root*) olduğu için *getty* prosesinin de 0 olacaktır. Daha sonra *getty* prosesi terminale ilişkin aygıt dosyasını *open* fonksiyonuyla önce okuma modunda sonra yazma modunda açar ve yazma modunda açtığı betimleyiciye *dup* fonksiyonunu uygular. Böylece *stdin*, *stdout* ve *stderr* betimleyicileri oluşturulmuş olur. Bu betimleyiciler alt proseslere aktarılacaktır.



Terminaller için birer aygıt sürücü (*device driver*) dosyası vardır. Terminal işlemleri bu aygıt sürücü dosyalarıyla yoluyla yapılmaktadır. Örneğin *"/dev/tty1"* aygıt sürücü dosyası birinci terminali temsil etmektedir. Birinci terminal kişisel bilgisayarlarımızdaki ekranımızdır. Biz bu dosyayı açarak birşeyler yazarsak birinci terminalin ekranına yazmış oluruz. Bu dosyadan okuma yaptığımızda ise klavyeden klavyeden okuma yaparız.

init prosesinin her terminal için ayrı ayrı *getty* programını çalıştırdığını vurgulayalım. Örneğin, tipik olarak *PC*'lere yüklediğimiz *UNIX* türevi sistemlerde *Ctrl+Alt+FN* tuşları ile terminaller arasında geçiş yapabiliriz. İşletim sistemi her terminal için işin başında bir *getty* programı çalıştırmaktadır. Pekiyi *getty* programları ne yapmaktadır? Bunlar genellikle kullanıcı ismini istedikten sonra *exec* işlemi ile *login* programını çalıştırırlar. Yani:

```
linux-ajw6 login: _
```

yazısını gördüğümüzde henüz *getty* programı çalışmaktadır. Ancak biz kullanıcı ismini girdiğimizde *getty* programı *login* programını aşağıdakine benzer bir biçimde çalıştırır:

```
execle("/bin/login", "login", "-p", username, (char *) 0, envp);
```

Artık *getty* programı *login* programı haline gelmiştir. Bizden parolayı isteyen *login* programıdır. Yani:

```
linux-ajw6 login: kaan
Password: _
```

ekranını gördüğümüzde artık *login* programı çalışıyor durumdadır. *login* programının kullanıcı ismini komut satırı argümanı olarak aldığına dikkat ediniz. Biraz daha ayrıntıya girersek, *login* programı *getpass* fonksiyonunu kullanarak parolayı kullanıcıdan alır. Aldığı parolayı *crypyt* fonksiyonunu kullanarak şifreleyip şifrelenmiş gerçek parola ile girilenin aynı olup olmadığına bakar. Şifelenmiş parolaların *"/etc/passwd"* ya da *"/etc/shadow"* dosyalarında tutulduklarını anımsayınız.

Açıklamalarda kullanmak için *kaan* isimli kullanıcının *login* işlemi yaptığını ve bu kullanıcıya ilişkin *"/etc/passwd"* girişinin aşağıdaki gibi olduğunu varsayalım:

```
kaan:x:1000:100::/home/study/:/bin/bash
```

login eğer parola doğrulamasını başarılı biçimde gerçekleştirirse *"/etc/passwd"* dosyasına başvurarak sırasıyla aşağıdaki işlemleri yapar:

1. Prosesin çalışma dizinini *chdir* fonksiyonuyla *"/etc/passwd"* dosyasında belirtilen dizin olacak biçimde ayarlar (örneğimizde *"/home/study"* dizini).

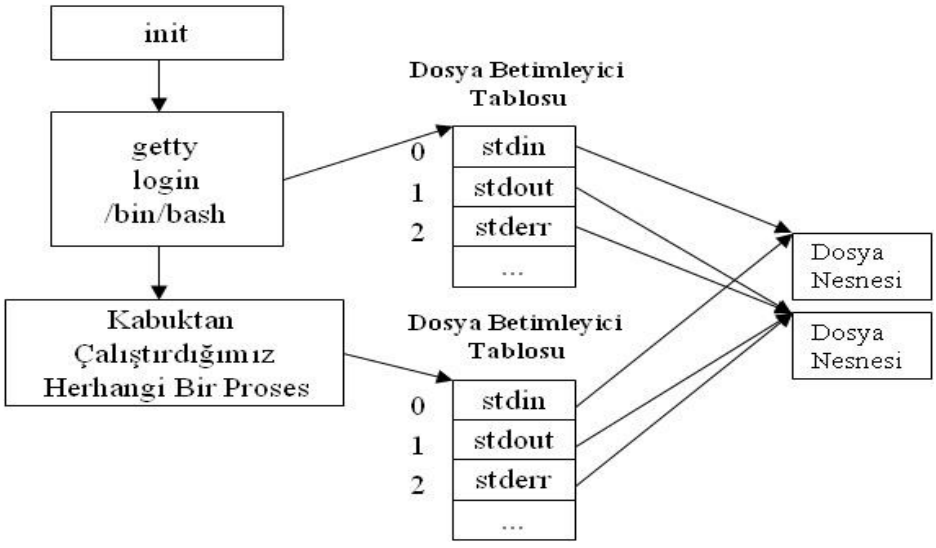
2. Prosesin grup *id*'sini *setgid* fonksiyonu ile *"/etc/passwd"* dosyasında belirtildiği gibi ayarlar (örneğimizde 100). Ayrıca *"/etc/group"* dosyasına başvurarak kullanıcıların tüm ek gruplarını elde eder ve *initgroups* fonksiyonuyla ek grupları da oluşturur.

3. HOME, *SHELL*, *USER*, *LOGNAME* ve *PATH* gibi bazı çevre değişkenlerini oluşturur.

4. *setuid* ve *setgid* fonksiyonlarıyla prosesin kullanıcı ve grup id değerlerini *"/etc/passwd"* dosyasında belirtilen değerlerle oluşturur. (Örneğimizde bu değerlerin *1000* ve *100* biçiminde olduğuna dikkat ediniz.) Burada bir şeye dikkat ediniz: *setuid* ve *setgid* fonksiyonlarını etkin kullanıcı id'si 0 olan süper kullanıcı oluşturduğuna göre prosesin yalnızca kullanıcı ve grup *id*'si değil aynı zamanda etkin kullanıcı ve grup *id*'si, saklanmış kullanıcı *id*'si ve grup *id*'si de değiştirilecektir.

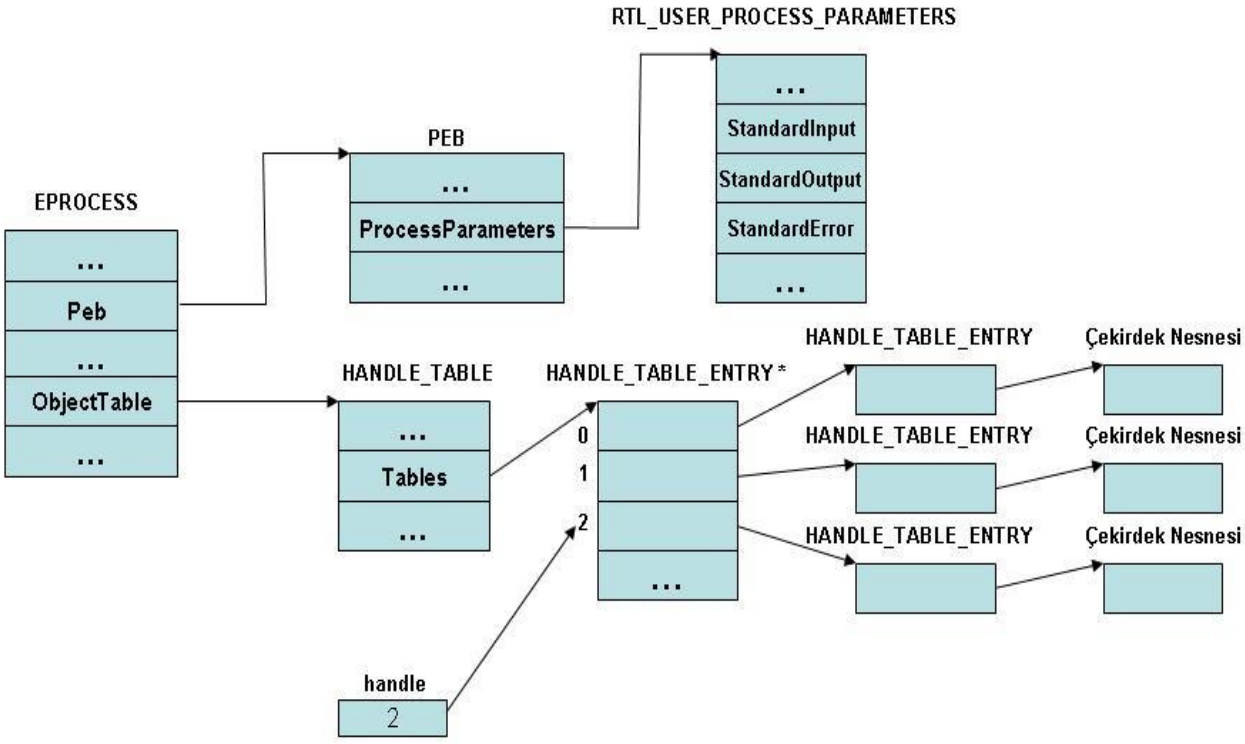
5. *exec* fonksiyonları ile *"/etc/passwd"* dosyasında belirtilen programı çalıştırır. (örneğimizde *"/bin/bash"*)

Gördüğünüz gibi tıpkı *getty* programının yaptığı gibi *login* programı da *fork* ve *exec* ile değil yalnızca *exec* ile kabuk programını çalıştırmaktadır. Yani *login* prosesi kabuk programı olarak yaşamına devam eder. O halde kabuk programının üst prosesi *init* prosesidir. Çünkü *getty* için yaratılandan başka hiçbir proses yaratılmamıştır. *getty* prosesi önce *login* programı olarak sonra da kabuk programı olarak yaşamını devam etmiştir. Biz kabuk altında bir programı çalıştırdığımızda kabuk bu programı *fork* ve *exec* fonksiyonlarıyla çalıştırır. Böylece kabuğun *stdin*, *stdout* ve *stderr* dosyaları *fork* işlemi çalıştırdığımız prosese aktarılmış olur. Bu durumu şekilsel olarak şöyle gösterebiliriz:



Windows sistemlerinde *stdin*, *stdout* ve *stderr* dosyaları *UNIX/Linux* sistemlerindekine benzer bir biçimde kullanılmaktadır. Bu sistemlerde *stdin*, *stdout* ve *stderr* dosyalarının *handle* değerleri üst prosesten alt prosese doğrudan aktarılmaz, proses *CreateProcess API* fonksiyonuyla yaratılırken belirlenir.

Bilindiği gibi *Windows* sistemlerinde proses kontrol bloğu *EPROCESS* (*Executive Process*) yapısıyla temsil edilmektedir. Bu sistemlerde betimleyici (*descriptor*) terimi yerine *handle* terimi kullanılır ve yalnızca dosyaların değil tüm çekirdek nesnelerinin (*kernel objects*) birer *handle* değeri vardır. Açılmış çekirdek nesnelerinin adresleri *Proses Handle Tablosu (Process Handle Table)* denilen bir listede doğrudan ya da dolaylı olarak tutulmaktadır. *Handle* değerleri tıpkı *UNIX/Linux* sistemlerinde olduğu gibi prosesin *handle* tablosunda birer indeks belirtir.[2] Aşağıda *ReactOS* sistemlerindeki konu ile ilgili veri yapısını veriyoruz. Çekirdek veri yapısının *Windows* sistemlerindeki tasarımı ile *ReactOS* sistemlerindeki tasarımının birbirine çok benzer olduğunu vurgulayalım. *ReactOS* kaynak kodları gerçek *Windows* sistemleri hakkında bize oldukça iyi bir fikir verebilmektedir:



Burada *EPROCESS* yapısı proses kontrol bloğunu, *PEB* yapısı prosesin ikincil bilgilerinin saklandığı prosesin çevresel bloğunu temsil ediyor[3]. *HANDLE_TABLE* yapısının *Tables* isimli elemanı *HANDLE_TABLE_ENTRY* ** türündendir. Yani bu eleman bir gösterici dizisini göstermektedir. İşte çekirdek nesnelerinin *handle* değerleri de bu gösterici dizisinde bir indeks belirtirler. *HANDLE_TABLE_ENTRY* ise çekirdek nesnelerine ilişkin çeşitli açış ve erişim haklarını içeren bir yapıdır. Şeklin aşağısında örnek bir *handle* değeri verdik. *Windows* sistemlerindeki *handle* değerlerinin -*HANDLE* bir adres türü olsa da- aslında bir indeks belirttiğine dikkat ediniz. Ayrıca *Windows* sistemlerinde *stdin*, *stdout* ve *stderr* dosyalarına ilişkin *handle* değerlerinin *UNIX/Linux* sistemlerinde olduğu gibi önceden bilinen ve değişmez değerler olmadığını da söyleyelim. Bu nedenle *stdin*, *stdout* ve *stderr handle*

değerlerinin proses kontrol bloğu yoluyla erişilebilen bir alanda tutulması uygun görülmüştür. Yukarıdaki şekilden de gördüğünüz gibi *ReactOS*'ta *stdin*, *stdout* ve *stderr* dosyalarına ilişkin *handle* değerleri *PEB* yapısının *ProcessParameters* elemanının gösterdiği yapıda tutulmaktadır. *Microsoft* belirli bir zamandan sonra *Windows* sistemlerinde *handle* tablosunu 3 kademeli hale getirmiştir. Bu 3 kademeli *handle* tablosu *X86* sistemlerindeki sayfalama mekanizmasındaki sayfa dizini (*page directory*) ve sayfa tablosu (*page table*) tasarımına benzemektedir.

Windows sistemlerinde proses yaratılırken *CreateProcess* fonksiyonunda *stdin*, *stdout* ve *stderr* dosyalarına ilişkin *handle* değerleri belirtilebilir. *CreateProcess* fonksiyonunun parametrik yapısını anımsatalım:

```
#include <windows.h>

BOOL WINAPI CreateProcess(
    LPTSTR lpCommandLine,
    LPCTSTR lpApplicationName,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

Fonksiyonun 9'uncu parametresi *STARTUPINFO* isimli bir yapı türündendir. Bu yapıyı inceleyiniz:

```
#include <windows.h>

typedef struct _STARTUPINFO {
    DWORD cb;
    LPTSTR lpReserved;
    LPTSTR lpDesktop;
    LPTSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
    DWORD dwXSize;
    DWORD dwYSize;
    DWORD dwXCountChars;
    DWORD dwYCountChars;
    DWORD dwFillAttribute;
    DWORD dwFlags;
    WORD wShowWindow;
    WORD cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFO,  *LPSTARTUPINFO;
```

Eğer yapının *dwFlags* elemanı *STARTF_USESTDHANDLES* bayrağını içeriyorsa yapının *hStdInput*, *hStdOutput* ve *hStdError* elemanları alt prosesin kullanacağı *handle* değerlerini belirtir. Şüphesiz bu *handle* değerlerinin alt prosese geçirilebilir (*inheritable*) olması gerekmektedir. Yoksa burada belirtilen *handle* değerleri alt prosese aktarılmazsa anlamsız bir durum oluşur. Eğer yapının *hStdInput*, *hStdOutput* ve *hStdError* elemanlarına atama yapılmamışsa bu dosyaların *handle* değerleri *NULL* durumdadır.

Programcı *stdin*, *stdout* ve *stderr* dosyalarına ilişkin dosya nesnelerine ilişkin *handle* değerlerini *GetStdHandle API* fonksiyonuyla elde edebilir:

```
#include <windows.h>

HANDLE WINAPI GetStdHandle(DWORD nStdHandle);
```

Fonksiyonun parametresi hangi standart dosyaya ilişkin *handle* değerinin elde edileceğini belirtir. Bu parametre aşağıdakilerden biri olarak girilmelidir:

```
STD_INPUT_HANDLE
STD_OUTPUT_HANDLE
STD_ERROR_HANDLE
```

Fonksiyon başarı durumunda ilgili dosyanın *handle* değeri ile başarısızlık durumunda *INVALID_HANDLE_VALUE* değeri ile geri döner. Standart dosyaların *handle* değerleri benzer biçimde *SetStdHandle* fonksiyonuyla değiştirilebilir:

```
#include <windows.h>

BOOL WINAPI SetStdHandle(DWORD nStdHandle, HANDLE hHandle);
```

Fonksiyonun birinci parametresi standart dosyayı belirten *GetStdHandle* fonksiyonunda açıkladığımız değerlerden birini alır. İkinci parametre ise ilgili standart dosyaya yeni atanacak *handle* değeridir. Örneğin biz *CreateFile* fonksiyonuyla bir dosya açıp elde ettiğimiz handle değerini *bu fonksiyonla stdout* olarak atayabiliriz. Böylece artık *stdout* dosyasına yazılanlar bizim açtığımız dosyaya yazılacaktır.

---

[1] Terminal kavramı hem ekran ve klavyenin birleşiminden oluşan bir kavram olarak kullanılmaktadır.

[2] *Windows* sistemlerinde *HANDLE* türü *void ** olarak *typedef* edilmiştir. Ancak *HANDLE* türünden bir gösterici içerisinde ilgili çekirdek nesnesinin adresi değil prosesin *handle* tablosundaki bir indeks tutulmaktadır.

[3]*EPROCESS* yapısı çekirdek alanında (*kernel space*), *PEB* yapısı ise kullanıcı alanında (*user space*) tutulur. Böylece prosesin çok kullanılan bu çevresel değişkenlere erişimi kolaylaştırılmıştır.

Kaynaklar

Aslan, K. (2002). *UNIX/Linux Sistem Programlama Kurs Notları*. Istanbul: C ve Sistem Programcıları Derneği.

Aslan, K. (2001). *Win32 Sistem Programlama Kurs Notları*. Istanbul: C ve Sistem Programcıları Derneği.

Bovet, D. and Cesati, M. (2005). *Understanding the Linux Kernel*. Oreilly & Associates Inc.

Mauerer W. (2008). *Professional Linux Kernel Architecture*. Indianapolis: Wiley Publishing, Inc.

Pietrek, M. (1995). *Windows 95 System Programming SECRETS*. California: IDG Books Worldwide, Inc.

Richter, J. M. (1999). *Programming Applications for Microsoft Windows with Cdrom. 4th*. *Edition.* Redmond, Washinton: Microsoft Press.

Russinovich M. E., Solomon D. A. (2009). *Windows Internals. 5th Edition*. Redmond, Washington: Microsoft Press.

Stevens, R., Rago, S. A. (2005). *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional