Java ile UNIX Ortamında Süreç İzleme Çatısı

Hüseyin Pehlivan[1] Mehmet Emin Tenekeci[2]

[1,2]Bilgisayar Mühendisliği Bölümü, Karadeniz Teknik Üniversitesi, Trabzon

[1]e-posta: pehlivan@ktu.edu.tr [2] e-posta: emintenekeci@ktu.edu.tr

Özetçe

Bu çalışmada, UNIX işletim sistemlerinde süreç izleme gereksinimleri için kullanılan `/proc` dosya sisteminin Java programlama çevresine entegrasyonunu sağlayan bir çatı (framework) geliştirilmiştir. Bir sürecin izlenebilmesi işletim sistemiyle etkileşimi, yani yerel sistem fonksiyonlarının kullanımını kapsadığı için süreç izleyen programların yazımı ileri düzeyde sistem programlama deneyimi gerektirir. Diğer yandan, modern UNIX sistemlerinde süreç izleme gereksinimleri sadece C/C++ programlama çevrelerinden karşılanır. Geliştirilen çatıda Java tarafından tedarik edilen JNI (Java Yerel Arayüzü) yardımıyla C/C++ çevrelerine erişim sağlanarak süreç izleme mekanizmasının kolay bir kullanımı gerçekleştirilmiştir. Bunun sonucu olarak, her bir programcı kendi ihtiyacına göre UNIX sistem süreçlerinin farklı denetimlerini icra eden programları Java dilinde kodlayabilecektir.

1. Giriş

UNIX ve türevi işletim sistemlerinde bulunan süreç izleme (process tracing) mekanizmaları sistem saldırılarının tespiti ve programların davranışının modellenmesi gibi çok çeşitli amaçlar için yaygın olarak kullanılırlar. Birçok bilimsel çalışmada süreç eylemlerinin belirlenmesi izleme mekanizmaları yardımıyla yakalanan sistem çağrıları üzerine dayandırılır. Gerçekten sistem çağrıları dizisi ve onların verileri analiz edilerek izlenen sürecin gerçekleştireceği eylemlerin ortaya çıkarılması mümkündür. Süreçlerin yönetimi doğrudan işletim sistemleri tarafından yapıldığı için bir süreç izleme faaliyeti işletim sistemi ile işbirliğine ihtiyaç duyar. Bununla birlikte, günümüz UNIX sistemleri bu işbirliğinin sadece C/C++ programlama çevrelerinden yapılmasını desteklemektedirler.

C/C++ çevrelerinde süreç izleme mekanizmalarının kullanımı ileri düzeyde sistem programcılığı gerektirir. Özellikle işletim sistemi ile etkileşim ve sistem çekirdeğine (kernel) ait veri yapılarına erişim kodlamanın karmaşık taraflarını oluşturur. Buna ilaveten, sürecin durdurulduğu anın öğrenilmesi, hangi modlar (modes) arası geçişin (kullanıcıdan çekirdeğe yada çekirdekten kullanıcıya) yapıldığının belirlenmesi, sürecinin durum bilgilerinin alınması gibi işlemler de programlamayı zorlaştırır. Süreçlerin gözetlenmesi (process monitoring) sistem kaynaklarına doğrudan erişime ihtiyaç duyduğu için bu faaliyetin saf (pure) Java ile gerçekleştirilmesi mümkün değildir. Java çevresinden UNIX sistemleri üzerinde çalışan süreçlerin gözetlenmesinde JNI teknolojisi önemli bir rol oynar.

Bu çalışmada JNI teknolojisi yardımıyla süreç izleme mekanizmalarından `/proc` dosya sisteminin Java programlama çevresinden etkin bir yol içinde kullanımını tedarik eden bir çatı tasarlanmış ve geliştirilmiştir. Java'dan UNIX Süreçlerini İzleme (JUSİ) mekanizması olarak isimlendirilen bu çatının sunduğu programlama çevresinin ileri düzeyde programlama tecrübesi olmayan programcıların bile kolaylıkla kullanabileceği bir yapıda olması hedeflenmiştir. Günümüzde programcı gereksinimlerini en ileri düzeyde karşılayan tekil bir programlama dili bulunmamaktadır. Bu nedenle JNI gibi diller arası etkileşim arayüzleri geliştirilerek bir dile ait modern ve güçlü yapıların diğer bir dil içerisinden kullanılabilmesi mümkün hale getirilmiştir [1]. JUSİ çatısı da C/C++ programcıları için var olan bir mekanizmanın Java ortamına entegrasyonunu gerçekleştirir.

Programlama dillerinin güçlü olduğu alanların kullanılmaya başlanması, dillerin birlikte işlerliğini (interoperability) sağlayan teknoloji ve yöntemlerin gelişimi sonucu ortaya çıkmıştır. Örneğin, C/C++ gibi programlama dilleri çekirdek seviyeli işlemler ile sistem aygıtlarının denetimine ait işlemleri yapabilmek için programcılara çeşitli işlevler sunarlar. Ancak, veri güvenliği veya atık toplama (garbage collection) gibi işlemlerde Java gibi daha yüksek seviyeli programlama dilleri tercih edilirler. Genel olarak, birlikte işlerlik yüksek seviyeli dillerin nispeten daha alçak seviyeli dillerde bulunan işlev ve yapıları kullanmasıyla varlık kazanır [1,2].

Programlama dillerinin birlikte kullanıldığı teknolojilere veya yapılara CORBA [3] (Ortak Nesne İstem Aracı Mimarisi), RPC [4] (Uzak Yordam Çağırma), COM [5] (Parçacıklı Nesne Modeli), JNI (Java Yerel Arayüzü) örnek olarak verilebilir. Ayrıca Microsoft.NET de dillerin ortak kullanımını sağlayacak bir alt yapıda oluşturulmuştur [6]. .NET ortamında kodlama yapılan dillerden bağımsız bir ara dile dönüşüm yapılabilmektedir. Bu yol içinde değişik dillerde yazılmış kodlar birleştirilerek programcı gereksinimini karşılayan bir program kolaylıkla üretilebilir.

2. Süreç İzleme

Süreç izleme mekanizmaları çeşitli bilimsel çalışmaların ihtiyaç duyduğu süreçlere ait durum bilgilerinin temin edilmesinde bir araç olarak görev yaparlar. Bu bölümde mevcut izleme mekanizmaları ile onların hangi amaçlar için kullanıldığı anlatılmıştır.

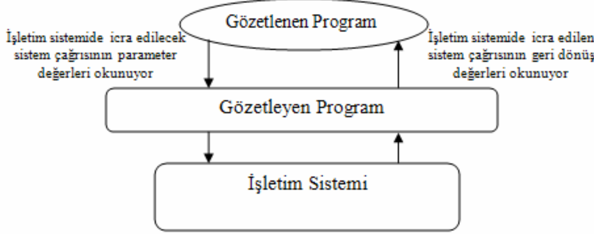
2.1. İzleme Mekanizmaları

Unix işletim sistemlerinde iki çeşit süreç gözetleme mekanizması vardır; `ptrace` sistem çağrısı [26] ve `/proc` dosya sistemi [27]. Bu mekanizmalar, bir sürecin icrasının başlangıcından sonuna kadar gerçekleştirdiği aktivitelerin gözetlenmesine izin verirler. Gözetlenen süreç adım adım koşturulabilir, bellek alanı okunup değiştirilebilir ve sistem

tarafından tutulan süreç tablosuna erişilebilir. `ptrace` sistem çağrısına göre `/proc` dosya sistemi daha gelişmiş işlevlerle donatılmıştır. Örneğin, `/proc` ile aktivitelerin küçük bir grubu için süreçler eşzamanlı olarak izlenebilirler.

`/proc` dosya sistemi, işletim sistemindeki her sürecin adres alanına erişimi sağlar. Sistemde oluşturulan bütün süreçler `/proc` dizini altında temsili dosyalar şeklinde tutulurlar. `/proc` dizininde bulunan her bir dosya, ilgili sürecin ID'si ile isimlendirilmiştir ve her bir dosyanın sahibi sürecin gerçek kullanıcı (real user) ID'si ile belirlenir. Birçok Unix programı (gdb, ps, top ve truss gibi) süreçleri kontrol etmek ve onlarla ilgili olarak bilgi toplamak için `/proc` dosya sistemini kullanır.

Süreç gözetleme mekanizmaları, bir sürecin diğer bir sürecin sistem kaynaklarını kullanımını kolay bir şekilde takip edebilmesini sağlarlar. Bu nedenle, bir süreç gözetleme işlemi en az iki süreç içerir; gözetleyen süreç ve gözetlenen süreç. Gözetlenen süreçler bazen yeni süreçler (child processes) oluştururlar. Bu süreçlerin gözetlenmesi, onların her biri için yeni gözetleyen süreçler üreterek yapılabileceği gibi mevcut gözetleyen süreç ile de yapılabilir. Şekil 1'de gözetlenen ve gözetleyen süreçler ile işletim sistemi ilişkisi gösterilmiştir.



Şekil 1: Süreç gözetleme mimarisi.

Bir süreç gözetlenmeye başlanmadan önce hangi sistem aktiviteleri (sistem çağrıları, sinyaller, bazı donanım hataları) ile ilgileniliyor ise bu aktiviteler işletim sistemine gözetlemeyi yapan süreç tarafından bildirilir. Gözetlenen süreç çalışmaya başlaması ile gözetleyen süreç gözetlenen sürecin durdurulmasını bekler. Gözetlenen süreç bu aktivitelerden birini gerçekleştireceği zaman işletim sistemi tarafından durdurulur ve gözetleyen sürece haber verilir. Bir süreç için durdurulma noktası, icra kontrolünün işletim sistemi çekirdeğine verildiği ve geri alındığı noktadır. Diğer bir ifade ile gözetlenen sürecin sistem çağrısı yaptığı ve bu çağrıdan geri dönüldüğü noktalarda onun adres alanına erişilir. Adres alanındaki okuma ve yazma işlemleri sadece gözetlenen sürecin icrası durdurulduktan sonra gerçekleştirilebilir. Gerekli okuma işlemleri gerçekleştikten sonra icrasının devam ettirilmesi gerekmektedir. Gözetlenen sürecin durdurulması ya da icrasının başlatılması işletim sisteminin kontrolündedir.

2.2. İzleme Gereksinimleri

Süreçlerin davranışlarını sergiledikleri çalışma ortamları işletim sistemi ile süreçler arasında yer alan ve yazılımla inşa edilmiş çeşitli arayüzler tarafından oluşturulabilirler [10]. Kullanıcıların çalıştırdığı UNIX süreçleri genellikle komut yorumlayıcıların (shell) kontrol ettiği çevreler içerisinden aktivitelerini yürütürler. Bu aktivitelere örnek olarak aktif dizinin değiştirilmesi, dosya sistemi üzerinde yapılan okuma yazma gibi işlemler, yeni süreç oluşturma ve süreçler arası haberleşme verilebilir. Bütün aktiviteler işletim sistemi çekirdeğine yapılan sistem çağrıları ile çekirdek seviyesinde (modunda) gerçekleştirilir. Dolayısıyla sistem çağrılarını izleyerek süreçlerin çalışma ortamları üzerinde yürüttükleri aktiviteleri belirlemek mümkündür. JUSİ mekanizması ile süreç aktivitelerinin Java çevresinden izlenebilmesi sağlanmıştır.

Literatürde sistem çağrılarının izlenmesi, kaydedilmesi ve de analiz edilmesi çok değişik amaçlar için kullanılmıştır. Bu amaçlardan bazıları aşağıda özetlenmiştir.

- Hata Ayıklama: Bir programın yaptığı sistem çağrıları incelenerek programın davranışı hakkında bilgi edinilebilmektedir. Bunun sonucunda ilgili programın davranışında gözlemlenen anormallikler yardımıyla kod içinde hatalı alan tespit edilebilir ve gerekli düzeltmeler yapılabilir [11,12].
- Sistem Kötü Kullanımlarının ve Yetkisiz Girişlerin Algılanması: Sistemi kullanan kimselerin yaptıkları işlemler kaydedilerek bu kişilerin sistem kaynaklarını nasıl kullandıkları hakkında bilgi sahibi olunabilmektedir. Sistemde yapılan işlemler sonucu meydana gelen sistem çağrıları dizileri üzerinde sınıflandırma, tanıma algoritmaları ve veri madenciliği yöntemlerinden yaralanarak kişilerin sistemi kullanım amaçları yada niyetleri belirlenebilir. [13-17].
- Program Doğrulama: Programların sistem üzerinde gerçekleştirdiği işlemler kaydedilerek bir programın beklenen davranışı gösterip göstermediği belirlenebilir.. Ayrıca programın davranışına göre bir virüsün yada casus yazılımın (spyware) varlığı tespit edilebilir [18-20].
- Ortam Modelleme ve Yeniden Oluşturma: Programların derlenmesi esnasında yapılan sistem çağrılarının izlenmesi ile programların hangi ortamda hangi kütüphane dosyalarını kullandığı belirlenebilir [21,22].
- Arayüzlerin Anlaşılması: Programların icrası esnasında yaptığı sistem çağrıları incelenerek bir program arayüzünün kullanım biçimi ve davranışı belirlenebilir. Ayrıca belgelenmemiş sistem fonksiyonlarının veya çağrılarının nasıl çalıştığı hakkında bilgi alınabilir [23].
- Güvenli Dosya Sistemi İşlemleri: Programların dosya sistemi üzerinde yaptığı sistem çağrıları izlenerek, dosya sistemini etkileyen aktiviteler ile karşılaşıldığında dosyaların birer kopyaları alınabilir. Bu şekilde virüsler veya kullanıcı hatalarından meydana gelen hasarlar telafi edilebilir [24,25].

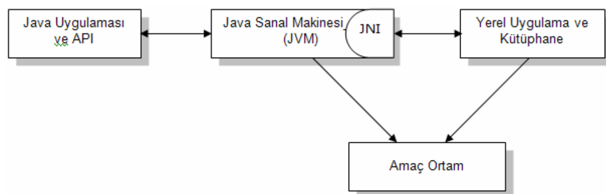
3. JNI Teknolojisi

JNI teknolojisi JUSİ mekanizmasının ortaya çıkarılmasında en önemli katkıyı yapan bileşendir. JNI, Java ile nispeten daha alçak seviyeli dillerin birlikte kullanılabilmesini sağlar. Bu sayede Java ile C/C++ dilerinin birlikte kullanılarak kodlama yapılabilmesi güçlü bir programlama çevresi oluşturacaktır. Çünkü, bu dillerin birbirlerine göre çeşitli üstünlükleri yada zenginlikleri bulunmaktadır. Örneğin, C++ dili yüksek hesaplama gerektiren uygulamalar ve bellek optimizasyonu için oldukça elverişli iken Java mümkün olduğunca taşınabilirliği destekler ve çok zengin sınıf kütüphanelerine

sahiptir [7]. Bu güçler birleştirilerek daha kullanışlı uygulamalar geliştirilebilecektir.

JNI kullanılarak geliştirilen birçok uygulama mevcuttur. Örneğin, JTUX [8] (Java-To-UNIX Package), POSIX/SUS sistem çağrılarını Java ortamında kullanabilmek için JNI ile oluşturulan bir arayüzdür. Bu arayüz sayesinde Java paketleri ile desteklenmeyen UNIX fonksiyonlarına Java ortamından erişilebilmektedir. Diğer bir JNI uygulaması da sayısal işaret işlemede Java ile C/C++ fonksiyonlarının birlikte kullanılmasıdır. Burada C/C++ fonksiyonları yüksek performans elde etmek için tercih edilmiştir [7]. Bunlara ilaveten JNIMarshall [9], Java ile C/C++ fonksiyonları arasında etkileşimi sağlayacak JNI fonksiyonlarını otomatik olarak üreten bir uygulamadır.

Bu çalışma işletim sistemi ilkellerine (primitives) ve yapılarına erişmek için JNI içerisinden C/C++ işlevlerinin kullanımını içermektedir. Özellikle süreç gözetleme ile ilgili C/C++ çevrelerine yerleştirilen fonksiyonlar ağırlıklı olarak kullanılmaktadır. İzlenen süreçlerden alınan veriler Java çevresinden işletim sistemi servislerinin yönetiminde önemli bir bilgi kaynağını oluşturur.



Şekil 2: JNI'ın Java ortamındaki rolü.

Şekil 2'de JNI bileşeninin Java çevresindeki rolü gösterilmiştir. JNI, Java sanal makinesinin (JVM) bir parçası olarak Java uygulaması ve yerel kütüphane fonksiyonları ile iki yönlü iletişimi sağlar. Bu yol içinde JNI bileşenini kullanan programlar, C/C++ dilleri ile yazılmış olan yerel uygulamalara ve kütüphanelere kolaylıkla erişebilirler.

4. JUSİ Çatısının Tasarımı ve Gerçeklemesi

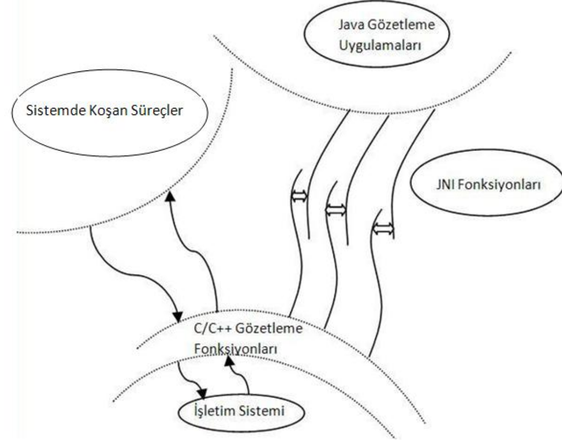
Bu bölümde JUSİ çatısının içerdiği sınıfları ve bu sınıfların arayüzünü teşkil eden metotların tasarımı ve gerçeklemesi verilmiştir. Ayrıca bu metotların örnek bir uygulaması yazılmıştır.

4.1. Çatının Yapısı

Java uygulamalarının UNIX süreçlerini gözetleyebilmeleri için geliştirilen bu çatıda Java ortamında yapılacak kodlamanın mümkün olduğu kadar işletim sistemin karmaşık alt yapısından soyutlanması amaçlanmıştır. Genel bir ilke olarak, süreçlerin gözetlenmesinde gerekli olan C++ fonksiyonlarına daha kolay bir kullanım getirilecek şekilde Java çevresi eşdeğerleri yazılmıştır. Şekil 3'de Java uygulamalarından işletim sistemine giden yol üzerinde yer alan ortamlar gösterilmiştir.

4.2. Sınıflar ve Metotları

JUSİ çatısı üç sınıftan meydana getirilmiştir: Trace, Handler, Process Sınıfları. Bu sınıflar gözetleme faaliyetini yürütmede kullanılacak JNI fonksiyonları içerirler. JNI fonksiyonları ile C++ çevresine ait işlevler çağrılarak yada süreçlerin durum bilgilerini içeren veri yapılarına erişilerek süreçlerin aktivite bilgileri Java ortamına aktarılır.



Şekil 3: Java ortamında süreç gözetl*eme*.

Trace Sınıfı
private native int executeProgram(String cmd);
private native int initEntryTraceFlags(int n, int s_entry[]);
private native int initExitTraceFlags(int n, int s_exit[]);
private native int handleStoppedProcess (int pindex);
private native int waitForSomeoneToStop ();
private native int waitForProcessToStop (int n);
private native int restartProcess (int procfd);
private native int releaseProcess (intlprocfd);
private native int shutdownTrace (int procfd);
private native int attachToProcess(int pid);

Process Sınıfı
private native int getNumOfProcs ();
private native int getProcfd (int pindex);
private native void closeProcess (int pindex);
private native int getProcWhy (int pindex);
private native int getPid (int pindex);
private native int getSysnum (int pindex);

Handler Sınıfı
private native string getSyscallParameters(pindex)
private native string getSyscallName (int pindex);
private native string getSyscallReturnValue (int pindex);
private native int changeSysCallArg(int n, int arg, String str);

Trace sınıfı içinde tanımlanan metotlar yardımıyla süreçleri gözetleme faaliyetleri yönetilir. Bu metotlardan *executeProgram*, herhangi bir UNIX programının icrasını başlatmak ve yapılan aktiviteleri izlemek için kullanılır. *attachToProcess* metodu, koşmakta olan UNIX süreçlerinin gözetimlerini gerçekleştirir. Gözetleme faaliyetlerinin sadece kullanıcının kendisine ait süreçler üzerinde yapılabileceğine dikkat edilmelidir. UNIX servis programlarını ve diğer kullanıcı aktivitelerini koşan süreçler sadece sistem yöneticisi tarafından gözetlenebilir. Gözetleme başlamadan önce *initEntryTraceFlags* ve *initExitTraceFlags* metotları ile izlenmek istenen sürecin yapacağı sistem çağrılarının hangilerinin çekirdek moduna girişte ve hangilerinin çıkışta

yakalanmak istendiği işletim sistemine bildirilir. Diğer metotlar hangi sürecin durduğunu belirlemek ve bu duran sürecin durum bilgisini adres alanından okumak ya da değiştirmek için tanımlanmıştır.

Process sınıfı, izlenen süreçler arasında durmuş olanların yapmakta oldukları aktiviteleri öğrenen metotları içerir. Örneğin, bu sınıf içinde, hangi sistem çağrısının durduğunu (*getSysnum)*, bu durmanın çekirdeğe girişte mi çıkışta mı gerçekleştiğini (*getProcWhy)* belirleyen metotlar bulunur. Süreç bir *fork* çağrısı yaparken durmuş ise bu çağrının geri dönüş değeri (child sürecin PID'si) *getPid* ile alınır.

Handler sınıfı ise sistem çağrılarının argümanlarını okumak (*getSyscallParameters*) ve *fork* dışında kalan çağrıların geri dönüş değerlerini almak için tanımlanan metotlardan (*getSyscallReturnValue*) oluşur.

4.3. Metot Gerçekleme

JUSİ çatısını oluşturan sınıfların içerdiği metotların hem JNI hem de C++ karşılıklarının gerçeklenmesi gerekir. Örnek olarak, Trace sınıfından iki metot (*initEntryTraceFlags* ve *initExitTraceFlags*) seçelim. Bu iki metodun çağırdığı JNI metotlarının gerçeklemesi aşağıda verilmiştir.

```
JNIEXPORT jint JNICALL Java_trace_initExitTraceFlags
        (JNIEnv *env, jclass obj, jint index, jintArray s_entry) {
    jint *s_ent;
    jint res;
    if ((s_ent = (*env)->GetIntArrayElements(env, s_entry,
                                                    NULL);) == NULL)
            return -1;
    JTHROW_neg1(res = initEntryTraceFlagsC(index, s_ent))
    (*env)->ReleaseIntArrayElements(env, s_exit, s_ext,
                                                    JNI_ABORT);
    return res; }

JNIEXPORT jint JNICALL Java_trace_initExitTraceFlags
        (JNIEnv *env, jclass obj, jint index, jintArray s_exit) {
    jint *s_ext;
    jint res;
    if ((s_ext = (*env)->GetIntArrayElements(env, s_exit,
                                                    NULL);) == NULL)
        return -1;
    JTHROW_neg1(res = initExitTraceFlagsC(index, s_ext))
    (*env)->ReleaseIntArrayElements(env, s_exit, s_ext,
                                                    JNI_ABORT);
    return res; }
```

Yukarıdaki JNI metotları içerisinden çağrımı yapılan iki C++ metodunun (*initEntryTraceFlagsC* ve *initExitTraceFlagsC*) gerçeklemesi de aşağıdaki gibi yapılmıştır.

```
int initEntryTraceFlagsC(int n, int *s_entry) {
  sysset_t SysCalls;
  premptyset(&SysCalls);
  for(int i = 0; s_entry[i]; i++)
    praddset (&SysCalls, s_entry[i]);
  int rc=ioctl(p->getProcfd(n),PIOCSENTRY,      &SysCalls);
  long flags = PR_FORK;
  int rc2 = ioctl(p->getProcfd(n), PIOCSET, &flags);
  return (rc || rc2); }

int initExitTraceFlagsC(int n, int *s_exit) {
  sysset_t SysCalls;
  premptyset(&SysCalls);
  for(int i = 0; s_exit[i]; i++)
    praddset (&SysCalls, s_exit[i]);
  return (ioctl(p->getProcfd(n), PIOCSEXIT, &SysCalls)); }
```

Diğer metotların JNI ve C++ karşılıkları benzer olarak gerçeklenmiştir. Süreçleri bu metotlarla izlerken ilgilenilmesi gereken önemli konulardan biri eşzamanlılıktır. Sistem çağrılarının kullanımından dolayı birbirlerinden bağımsız olarak çalışan ve sistem kaynaklarını rasgele paylaşan süreçler sistemde belirsizliğe neden olabilirler. Sistem kaynaklarının kontrolsüz paylaşımından kaynaklanan belirsizliği azaltmak için C++ dilinde tanımlanan aşağıdaki süreç tablosundan (*procT*) yararlanılır.

```
int nprocs;
struct pollfd Pollfds[MAXPROCS];
struct processTable {
   pid_t pid;
   int procfd;
   prstatus_t *pstatus;
} procT[MAXPROCS];
```

Pollfd yapısı, süreç eylemlerinin etkin kontrolünü eşzamanlı olarak gerçekleştirmek için *poll* çağrısı tarafından kullanılır. Bu şekilde tek bir program ile çalışmakta olan birçok sürecin gözetleme faaliyeti gerçekleştirilebilir. Bu yapı içerisinde sistemin kendi veri yapıları da kullanıldığı için Java tarafında oluşturulmamıştır.

4.4. Örnek Uygulama

Yukarıda tanımlanan metotlarla basit bir uygulama yazalım. Bu uygulamada kullanıcının ismini girdiği programlar Java içerisinden çalıştırılır ve programlarla ilgilenen süreçlerin yaptığı sistem çağrıları izlenir. Uygulamanın kaynak kodu incelendiğinde programcının süreç gözetlemeyi istediği gibi yapılandırabileceği görülmektedir. Bu yapılandırmalar işletim sistemi çekirdeğine ait yapılarla doğrudan ilgilenilmeden gerçekleştirilebilmektedir.

```
public class Tracer {
    private Trace Tr, Process Ps, Handler Hnd;
    private int s_entry[254],s_exit[254];
    private int pindex, res, fd, pid, status;
    private String sysname;

    private void setAllSysCalls () {
       int i;
       for (i=2; i< N_SYSCALLS; i++)  {
           s_entry [i-2]=i;  s_exit [i-2]=i;
       }
       s_entry [i-2]=1; s_entry [i-1]=0; s_exit [i-2]=0;
    }

    private void handleProcess ()  {
       fd=Ps.getProcfd (pindex);
       pid=Ps. getPid (pindex);
       sysname = Hnd. getSyscallName(pindex);
       if (sysname != "")  {
         if (Ps.getProcPrWhy(pindex)==PR_SYSENTRY) {
           System.out.println(sysname + "(");
```

```
          System.out.println(
                    Hnd.getSyscallParameters(pindex)+“(”);
        } //end if
        else
          System.out.println(
                    Hnd.getSyscallReturnValue (pindex) + “(”);
       } //end if
    }

    public static void main(String args[]) throws Exception {
      BufferedReader jinput = new BufferedReader (
                       new InputStreamReader (System.in) );
      String program;
      Tracer tracer=new Tracer();
      Trace Tr = new Trace();
      Process Ps = new Process();
      Handler Hnd = new Handler();

      tracer.setAllcalls();
      while (true) {
        System.out.print("Enter Program Name>> ");
        program = jinput.readLine();
        if (program.length()==0) break;
        else {
          pindex = Tr. executeProgram(program);
          if (pindex < 0)
            printf("Could not attach to child.\n");
          if (Tr. initEntryTraceFlags (pindex, s_entry)< 0)
            printf("Could not set entry trace options.\n");
          if (Tr. initExitTraceFlags (pindex, s_exit)< 0)
            printf("Could not set exit trace options.\n");
          while ( Ps.getNumOfProcs() > 0 ) {
            pindex = Tr.waitForSomeoneToStop();
            res = Tr.waitForProcessToStop(pindex);
            switch (res) {
              case WPS_STOPPED:
                tracer.handleprocess();
                status = Tr.handleStoppedProcess(pindex);
                if (status == 0)
                  Tr.restartProcess(Ps.getProcfd(pindex))
                Ps.closeProcess(pindex);
                break;
              case WPS_DIED:
                Ps.closeProcess(pindex);
                break;
              case -1:
                System.out.println("ERR: WPSTOP failed");
                break;
            } //end switch
          } //end while
        } //while
      System.exit(0);
    }
}
```

5. Sonuçlar

Bu bildiride Unix ortamında çalışan bir kullanıcıya ait süreçlerin Java çevresi içerisinden gözetlenebilmesi için gerekli metotları içeren bir çatının (JUSİ) gerçeklemesi sunulmuştur. Gerçeklenen metotlar üç ayrı sınıf altında toplanmıştır. Java'da sistem kaynaklarının süreç gözetlemeye imkan verecek şekilde yapılandırılması mümkün olmadığından dolayı bu işlemler JNI teknolojisi yardımıyla erişilebilen C++ fonksiyonlarına yaptırılmıştır. Bunun sonucunda süreçler için tutulan çekirdek veri yapılarının yüksek seviyeli bir dilden daha kolay bir şekilde kontrol edilebilmesi sağlanmıştır.

Gelecek çalışmalar bu çatının çeşitli açılımları üzerine odaklanacaktır. Java çevresinden UNIX dosya sistemi işlemlerinin güvenilir yapılması ve UNIX ortamının web tabanlı bir arayüz içerisinden öğretilmesi iki açılım örneği olarak verilebilir.

6. Kaynakça

[1] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, Kevin Stoodley. Inlining Java Native Calls At Runtime Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments VEE '05, June 2005.

[2] Rodrigo Vivanco, Nicolino Pizzi, Computational Performance of Java and C++ in Processing fMRI Datasets, April 2002, Pages: 183 - 192

[3] Todd Scallan. a corba primer. http://www.omg.org/news/whitepapers/seguecorba.pdf, June 3 2002.

[4] Andrew D. Birrell ve Bruce Jay Nelson. Implementing remote procedure calls. ACM Transactions on Computer Systems, 2(1):39{59, 1984}.Bb

[5] Microsoft Inc Com: Component object model technologies. http://www.microsoft.com/com/default.mspx.

[6] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. ACM SIGPLAN Notices, 36(3):248{260, March 2001.

[7] V. P. Shah and N. H. Younan , T. Alford and A. skjellum, An Advanced Signal Processing Toolkit For Java Applications, proceedings of the 2nd international conference on Principles and practice of programming in Java PPPJ '03, June 2003

[8] Rochind, Marc J., Jtux – Java to Unix Package, Nov. 2005, http://www.basepath.com/aup/jtux/

[9] Gerald Dueck, JniMarshall A Java Native Interface Generator, October 2, 2006

[10] Diomidis Spinellis, Trace: A Tool for Logging Operating System Call Transactions, Operating Systems Review, Vol. 28 Num. 4 {56,63}, 1994

[11] Richard M. Stallman. The GNU source-level debugger. Distributed by the Free Software Foundation, January 1989.

[12] Bert Beander. VAX DEBUG: An interactive, symbolic, multilingual debugger. In M.S. Johnson, editor, Proceedings of the Software Engineering Symposium on High-Level Debugging, pages 173–179. ACM SIGSOFT/SIGPLAN, March 1983.

[13] M. Bernaschi, E. Gabrielli, and L. Mancini, “Operating System Enhancements to Prevent the Misuse of System Calls,” Proc. ACM Conf. Computer and Comm. Security, pp. 174-183, 2000.

[14] T. Garfinkel, “Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools,” Proc. Network and Distributed Systems Security Symp., Feb. 2003.

[15] T. Garfinkel, B. Pfaff, and M. Rosenblum, “Ostia: A Delegating Architecture for Secure System Call

Interposition," Proc. Network and Distributed Systems Security Symp., Feb. 2004.

[16] N. Provos, "Improving Host Security with System Call Policies," Proc. 12th USENIX Security Symp., pp. 257-272, Aug. 2003.

[17] Eskin, E.; Wenke Lee; Stolfo, S.J.: Modeling system calls for intrusion detection with dynamic window sizes. Proceedings of the 2001 DARPA Information Survivability Conference & Exposition II. DISCEX '01, Vol.1 (2001) 165 -175

[18] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," Proc. IEEE Symp. Security and Privacy, pp. 144-155,2001

[19] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proc.* 11th USENIX Security Symposium, August 2002

[20] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," Proc. Eighth European Symp. Research in Computer Security (ESORICS '03), pp. 326-343, 2003

[21] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. Software — Practice and Experience, 13:671–685, 1983.

[22] Diomidis Spinellis. v08i002: A C execution profiler for MS-DOS. Posted in the Usenet newsgroup comp.sources.misc, August 1989

[23] Gary R. Ignatin. Let the hackers hack: Allowing the reverse engineering of copyrighted computer programs to achieve compatibility. *University of Pennsylvania Law Review*, 140:1999–2050, 1992

[24] Holyer, I., and Pehlivan, H., "A recovery mechanism for shells", *The Computer Journal, Vol. 4, 2000, No. 3, 1-9.*

[25] Hüseyin Pehlivan, Mehmet Emin Tenekeci, Unix İşletim Sistemleri İçin Akıllı Geri Dönüşüm Kutusu Tasarımı Ve Gerçekleştirilmesi, Eleco2006 Bursa

[26] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and* Implementation of the 4.3BSD Unix Operating System, page 104. Addison-Wesley, 1988.

[27] T. S. Killian. Processes as files. In *Proceedings of the USENIX Summer 84 Conference*, pages 203–207. USENIX Association, 1984