KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

UNIX SİSTEM ÇAĞRILARININ İZLENMESİ VE ÖĞRENİLMESİ

YÜKSEK LİSANS TEZİ

Bilgisayar Müh. Çağlar GÜMRÜKÇÜ

MART 2009
TRABZON

KARADENİZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

UNIX SİSTEM ÇAĞRILARININ İZLENMESİ VE ÖĞRENİLMESİ

Bilgisayar Mühendisi Çağlar GÜMRÜKÇÜ

Karadeniz Teknik Üniversitesi Fen Bilimleri Enstitüsünce
"Bilgisayar Yüksek Mühendisi"
Unvanı Verilmesi İçin Kabul Edilen Tezdir.

Tezin Enstitüye Verildiği Tarih : 12.02.2009
Tezin Savunma Tarihi : 19.03.2009

Tez Danışmanı : Yrd. Doç. Dr. Hüseyin PEHLİVAN
Jüri Üyesi : Yrd. Doç. Cemal KÖSE
Jüri Üyesi : Prof. Dr. Sefa AKPINAR

Enstitü Müdürü : Prof. Dr. Salih TERZİOĞLU

Trabzon 2009

ÖNSÖZ

Sistem çağrılarının izlenmesi ve çağrıların çalışma mantığının çok iyi yorumlanmasıyla, süreçlerin işletim sistemiyle olan etkileşimini anlamak daha kolay olmaktadır. Özellikle sistem programlamaya yeni başlayan, işletim sistemini araştıran kullanıcılar için karmaşık süreçleri incelemek zor ve anlaşılması güç olmaktadır. Bu çalışmamızda sistem çağrılarının gözetlenmesini, işletim sisteminde hangi görevi yerine getirdiğini, süreçlerin koşarken sistem çağrılarını nasıl kullandığını gösterebilecek bir çatı oluşturulmuştur. Geliştirmiş olduğumuz proje tamamen web tabanlı olup, süreçlerin izlenebilmesi UNIX ortamından bağımsız hale getirilmiştir.

Bu çalışmamda danışmanlığımı üstlenen değerli hocam Yrd. Doç. Dr. Hüseyin Pehlivan' a ilgi, alaka ve yardımlarından dolayı teşekkürü bir borç bilirim. Ayrıca her türlü destekleriyle her zaman yanımda olan aileme ve çok değerli arkadaşlarım Mustafa SADIK, Murat AYKUT, Mehmet Emin TENEKECİ, Necati PEYNİRCİ, Ramazan FIRIN'a teşekkür ederim.

Çağlar GÜMRÜKÇÜ
Trabzon 2009

İÇİNDEKİLER

Sayfa No

ÖZET

İşletim sisteminde 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. UNIX sistemlerinde çeşitli süreç izleme mekanizmaları mevcuttur. Fakat bu çalışmada süreçlerin izlenebilmesi için farklı bir program yazılmış olup standart UNIX izleme mekanizmaları kullanılmamıştır. Süreç izlemede en önemli rolü işletim sistemi üstlenir. Bu sebeple geliştirilen izleme mekanizmaları işletim sistemiyle işbirliğine ihtiyaç duyar.

Bu çalışmanın amacı sistem çağrılarının herhangi bir web ara yüzünden koşularak izlenmesi, çağrıların çalışma mantığının grafiksel olarak gösterilmesi ve kullanıcılara öğretilmesidir. Çalışmamız temelde iki bölümden oluşmaktadır. Birincisi UNIX sistem üzerinde koşan çağrıları analiz eden bir C programı ve bir web servisi, ikinci kısım ise izlenecek sistem çağrılarının girilerek sonuçların gösterildiği web ara yüzüdür. Burada önemli olan nokta çağrıların izlenebilmesi için kullanıcıların fiilen bir UNIX makineye ihtiyaçlarının olmamasıdır. Web ara yüzünden izlenmek istenen çağrılar programla UNIX sistem üzerinde koşan web servisine aktarılır, servis UNIX tarafında geliştirdiğimiz çağrı izleme programı yardımıyla sonuçları üreterek istemci tarafa gönderir. İstemci tarafta, dönen sonuçlara göre süreçler bazında koşulan çağrıların listelenmesi, gruplanması, grafiksel olarak yorumlanması sağlanmaktadır. Genel olarak sistem çağrıları Bellek, Disk, Sistem, Giriş/Çıkış (I/O) gibi kategorize edilmiş ve grafiksel gösterim bu kategorizasyona dayandırılmıştır.

Sistem çağrıların izlenmesi, çağrıların çalışma mantığının anlaşılması, işletim sisteminin süreçleri nasıl kullandığının yorumlanması, sistem programlamayla ilgilenen kullanıcılara yardımcı olabilmesi amacıyla bir web uygulaması yazılmıştır.

Anahtar Kelimeler: Süreç İzleme, Sistem Çağrıları, Süreç Davranışları, Kullanıcı Öğrenmesi

.

SUMMARY

Tracing and Learning UNIX System Calls

Process activities in operating systems are based on system calls which can be captured by monitoring mechanisms. It is possible to determine kernel-based activities of processes, by analyzing system calls and its data. In UNIX systems, there are some process monitoring mechanisms, but in this thesis a different computer program is written to monitor process, standard UNIX monitoring tools are not used. Since operating systems have a significant role in process monitoring, monitoring mechanisms needs to cooperate with operating systems.

The aim of this work is, to run system programs using a web interface, monitoring the related processes and graphically presenting their kernel operations to the user in a system call-based fashion. The work is organized in two parts. First, a C program and web service, which analyses system calls of processes running on UNIX operation system, second, a web interface where you can enter some system calls to analyze and see the results. The key point is that, users do not need any UNIX machine to monitor system calls. System calls are transferred via a program to the web service which runs on a UNIX machine, the service generates results by means of system call monitoring program on UNIX side and sends results to the client side. According to the returned results, process based system calls are listed, grouped, and graphically displayed on client so that it can be analyzed and interpreted. Generally, system calls are categorized as; memory, disk, system, I/O, and graphical representation is based on this categorization.

A web application is written to monitor system calls, to understand the logic behind the calls, to interpret how operating system manages processes, to help those who deal with system programming.

Key Words: Process Tracing, System Calls, Process Behaviour, User Learning

ŞEKİLLER DİZİNİ

Sayfa No

TABLOLAR DİZİNİ

Sayfa No

SEMBOLLER DİZİNİ

CoW	Yazarken Kopyala (Copy On Write)
CPU	Merkezi İşlem Birimi (Central Processing Unit)
PID	Süreç ID (Process ID)
TSS	Görev Durumu Bölmesi (Task State Segment-TSS)

1. GENEL BİLGİLER

1.1. Giriş

Unix işletim sisteminde, sistem çağrılarının izlenmesi, kaydedilmesi ve davranışlarının modellenmesi için değişik uygulamalar geliştirilmiştir. Bazı Unix türevleriyle gelen hazır izleme mekanizmaları ise süreç ve çağrı izlemeye yönelik tüm işlemleri bir arada yapamamaktadır. İzleme işlemleri genellikle Unix makinelere bağlı kalınarak gerçekleştirilmekte olup farklı platformlara uygulanmamıştır. Sistem çağrılarının izlenmesi ileri düzeyde sistem programcılığı gerektirir. Özellikle işletim sistemi ile etkileşim ve sistem çekirdeğine ait veri yapılarına erişim kodlamanın karmaşık tarafını oluşturur. İşte bu noktada, sistem programlamayla ilgilenen, işletim sisteminin süreç yönetimini nasıl yaptığı araştıran ya da programların davranışlarını belirlemeye çalışan kullanıcılar için bu konu önemli bir referans olacaktır.

1.2. Süreçler

1.2.1. Süreç Kavramı

Bir süreç (process) içinde bir veya daha çok iş parçacığının (thread) çalıştığı ve o iş parçacıkları için gerekli sistem kaynaklarının yer aldığı bir adres alanıdır. Genel olarak, bir süreç çalışmakta olan bir program olarak değerlendirilebilir.

Unix gibi çok görevli (multi-tasking) bir işletim sistemi çok sayıda programın aynı anda çalışmasına izin verir. Çalışmakta olan programın her bir gerçekleşmesi ayrı bir süreç ortaya koyar.

Çok kullanıcılı bir sistem olarak, Unix çok sayıda kullanıcının sisteme aynı anda erişmesine izin verir. Her kullanıcı çok sayıda programı, hatta aynı programın çok sayıda gerçekleşmesini aynı anda çalıştırabilir. Sistem kaynaklarını yönetmek ve kullanıcı erişimini kontrol etmek için, aynı zamanda sistemin kendisi de başka programlar çalıştırır.

Çalışmakta olan bir program veya süreç; program kodu, veriler, değişkenler, dosya tanımlayıcıları, ve ortam‘dan oluşur. Genellikle, Unix sistemi kod ve sistem kütüphanelerini süreçler arasında ortak kullandırarak bir anda bellekte kodun yalnız bir kopyasının bulunmasını sağlar.

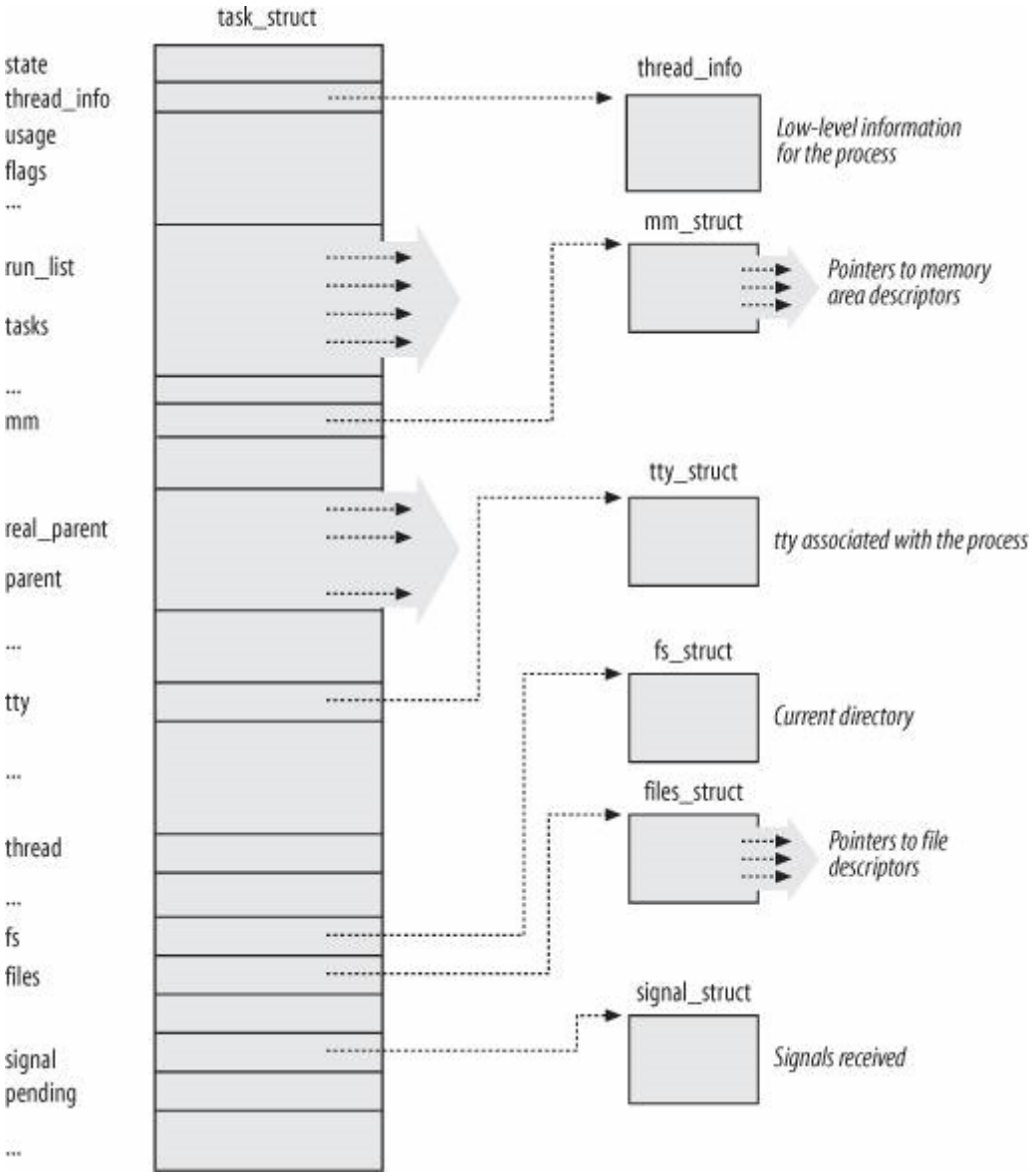
UNIX' te süreçlerin kullanıcı ve çekirdek olmak üzere iki çalışma modu vardır [3]. Bir sürecin sistem çağrısı yapmasıyla veya bir kesme geldiğinde modlar arası geçiş yapılır (kullanıcı moddan → çekirdek moda). Kullanıcı moddaki süreçler kendi emir ve veri alanlarına erişebilirler. Çekirdeğin veya başka süreçlerin veri alanlarına erişemezler. Ayrıca bazı ayrıcalıklı emirlerin kullanıcı modda çalışmaları hata üretir.

Çekirdek moddaki süreçler çekirdek veri alanlarına erişebilirler. Sürecin çekirdek modda koşması, çekirdeğin o süreç adına istenen işi yapıyor olması gibi düşünülebilir. Kaynaklara çekirdek erişir, çeşitli işlemleri yapar, süreç tekrar kullanıcı modda çalışmaya bırakılır. Daha sonra süreç çeşitli sistem çağrılarıyla tekrar çekirdek moda geçmeyi isteyebilir [1].

İki ayrı mod kaynaklara erişim konusunda bir karışıklık çıkmasını engellemektedir. Bir süreç her zaman kaynaklara erişim isteğinde bulunmayacağı için farklı bir modda beklemesi uygundur. Kaynaklara erişimi kontrol etmek, işletim sistemlerinin temel görevlerindendir.

1.2.2. Süreç Tanımlayıcı

Süreçleri yönetmek için çekirdek, her sürecin ne yaptığına ilişkin net bir resme sahip olmalıdır. Örneğin süreç önceliğini, CPU üzerinde çalışır durumda olduğunu ya da bir olay nedeni ile tıkanmış olduğunu, hangi adres alanının ona atanmış olduğunu, hangi dosyalara erişmesine izin verildiğini bilmek zorundadır. Bu süreç tanımlayıcının, yani tek bir sürece ilişkin bütün bilgileri içeren *task_struct* tipinde bir yapının rolüdür. Şekil 1.1.'de süreç tanımlayıcı modellenmiştir.



Şekil 1.1. Süreç Tanımlayıcı

1.2.2.1. Süreç Durumu

Süreç tanımlayıcının durum alanı süreç içerisinde o anda ne olduğunu tarif eder. Her biri süreç durumunu tarif eden bir dizi bayraktan oluşur. Unix'te bu durumlar karşılıklı olarak birbirlerini hariç tutar ve böylece kümenin yalnız bir bayrağı aktif diğerleri pasif olur.

Mümkün süreç durumları aşağıda listelenmiştir.

- *task_running* – süreç CPU üzerinde çalışıyor yada çalıştırılmak için bekliyor.

- *task_interruptible* – bir koşul gerçekleşinceye kadar süreç askıdadır. Bir donanım kesintisini başlatmak, sürecin beklediği bir sistem kaynağını serbest bırakmak veya bir işaret teslim etmek bir süreci uyandıran koşullara örnektir. Yani sürecin durumunu *task_running*'e getirirler.
- *task_uninterruptible* – bir önceki durum gibidir. Yalnız uyuyan sürece bir işaret teslim edilişi durumu değiştirmez. Bu süreç nadir olark kullanılır. Yine de, bir sürecin öngörülmüş bir olay oluncaya kadar kesintiye uğramadan beklemesi gerektiği belirli özel koşullar altında değerlidir. Örneğin, bir süreç bir cihaz dosyasını açtığında ve cihaz sürücüsü donanımı kullanmaya başladığında bu durum kullanılabilir. Cihaz sürücüsü kullanma tamamlanıncaya kadar kesintiye uğramamak zorundadır ya da donanım belirsiz bir halde kalabilir.
- *task_stopped* – süreç çalıştırılması durduruldu. Süreç bu duruma bir *sigstop*, *sigtstp*, *sigttin* veya *sigttou* işareti aldıktan sonra girer. Bir süreç bir diğeri tarafından izlendiğinde herhangi bir işaret süreci *task_stopped* durumuna getirir.
- *task_zombie* – süreç çalıştırılması bitirildi. Fakat ana süreç, ölü süreç hakkında bilgi vermek için henüz *wait* benzeri sistem çağrısı çalıştırmadı. *wait* benzeri çağrısı çalıştırılmadan önce, çekirdek ölü sürecin tanımlayıcısının içerdiği veriyi çöpe atamaz. Çünkü ebeveyn ona ihtiyaç duyabilir.

1.2.2.2. Süreci Tanımlama

Unix süreçleri kendilerine ait çekirdek veri yapılarınının önemli bir kısmını paylaşabildikleri halde her süreç kendine ait süreç tanımlayıcısına sahiptir. Lightweight süreçler bir kullanıcı düzeyinde (user-level) kütüphane tarafından ele alınan ve farklı bir çalıştırma akışı olan, user-mode thread'ler ile karıştırılmamalıdır. Süreç ile süreç tanımlayıcı arasında birebir uyuşma, 32-bit süreç tanımlayıcı adresini, süreci tanımlamak için kullanışlı bir araç yapar. Bu adreslere süreç tanımlayıcı işaretçileri (process descriptor pointers) olarak değinilir. Çekirdeğin yarattığı süreçlere olan referansların çoğu süreç tanımlayıcı işaretçileri aracılığı iledir.

Öte yandan, herhangi bir UNIX-benzeri sistem, kullanıcılarının süreçleri Süreç Kimlik No'su (PID) adı verilen bir sayı aracılığıyla belirleyişine müsade eder. PID süreç tanımlayıcının pid alanında tutulan 32-bit işaretsiz bir sayıdır. PID'ler sıra ile sayılandırılmıştır. Yeni yaratılan bir sürecin PID'si normalde bir öncekinin PID'sinin bir

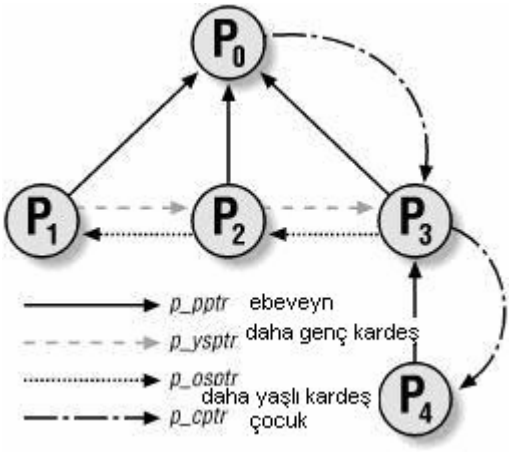
fazlasıdır. Çekirdek sistemdeki 32768'inci süreci yarattığı zaman, önceki kullanılmamış PID'leri yeniden kullanmaya başlamalıdır.

1.2.2.3. Süreçler Arasında Ebeveynlik İlişkileri

Bir program tarafından yaratılan süreçler ebeveyn/çocuk ilişkisine sahiptirler. Bir süreç birden çok sayıda çocuk yaratabileceği için, bunlar kardeşlik ilişkisine sahip olurlar. Bu ilişkileri temsil etmek için bir süreç tanmlayıcıda çok sayıda alan kullanılması gerekir. Süreç 0 ve 1 çekirdek tarafından yaratılır. Süreç 1 diğer bütün süreçlerin atasıdır.

Bir P sürecinin tanımlayıcısı aşağıdaki alanları içerir:

- p_opptr (doğal ebeveyn) (original parent)
- p_pptr (ebeveyn) (parent)
- p_cptr (çocuk) (child)
- p_ysptr (daha genç kardeş) (younger sibling)
- p_osptr (daha yaşlı kardeş) (older sibling)



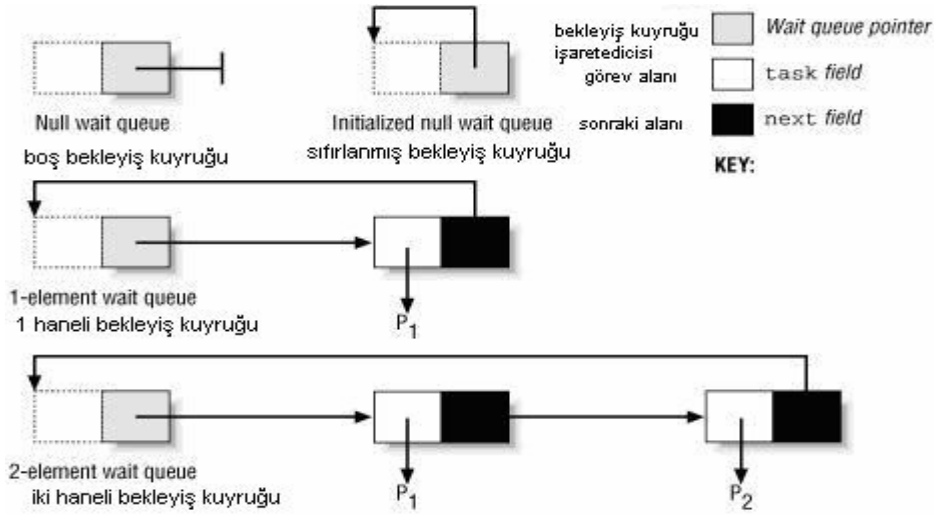
Şekil 1.2. Beş süreç arasında ebeveynlik ilişkileri

1.2.2.4. Bekleme Kuyrukları

Çalışma kuyruğu listesi *task_running* halindeki bütün süreçleri biraraya getirir. Başka durumlardaki süreçler için farklı opsiyonlar geçerlidir:

- *task_stopped* veya *task_zombie* durumundaki süreçler belirli listelere bağlanmaz. Bunları gruplamaya gerek yoktur. Çünkü çocuk sürece erişmek için ebeveyn süreç tarafından PID yada süreç ebeveynlik ilişkileri kullanılabilir.
- *task_interruptible* veya *task_uninterruptible* durumundaki süreçler her biri belirli bir olaya denk düşen çok sayıda alt sınıfa ayrılabilir. Bu durumda, süreç durumu süreci elde etmek için yeterli bilgi sağlamaz. Bu yüzden ek süreç listeleri kullanmak gerekir. Bu ek listelere bekleme listeleri adı verilir.

Bekleme listeleri çekirdek içinde bir çok kullanımlara sahiptir. Bir süreç genellikle bir olayın gerçekleşmesini beklemek zorundadır. Örneğin bir disk işleminin bitişi, bir sistem kaynağının serbest bırakılışı, veya sabit aralık zamanın doluşu gibi. Bekleme kuyrukları olaylara bağlı koşullu bekleyişleri gerçekleştirir. Belirli bir olayı beklemek isteyen bir süreç kendini bir bekleme kuyruğuna yerleştirir ve kontrolü serbest bırakır. Bu yüzden, bir bekleyiş kuyruğu bir koşul sağlandığında çekirdek tarafından uyandırılan bir grup uyuyan süreci temsil eder.



Şekil 1.3. Bekleme Kuyruğu Yapısı

1.2.3. Süreç Geçişi

Süreçlerin çalışmasını kontrol etmek için, çekirdek CPU üzerinde çalışmakta olan sürecin çalışmasını askıya alabilmek ve daha önce askıya alınmış başka bir sürecin çalışmasını kaldığı yerden devam ettirebilmek zorundadır. Bu faaliyet süreç geçişi, görev geçişi, ya da içerik geçişi olarak adlandırılır.

Aşağıdaki kısımlar süreç geçişinin unsurlarını tarif etmektedir:

• Donanım içeriği

• Donanım desteği

• Kod

• Kayan nokta yazmaçlarının saklanması

1.2.3.1 Donanım İçeriği

Her süreç kendine ait bir adress alanına sahip olabildiği gibi, bütün süreçler CPU yazmaçlarını (register) paylaşmak zorundadırlar. Dolayısıyla bir sürecin çalışmasına kaldığı yerden devam etmeden önce, çekirdek bunun gibi her yazmacın, süreç askıya alınmadan önce sahip olduğu değerle yeniden yüklendiğini garantiye almalıdır.

Süreç CPU üzerinde çalışımasına kaldığı yerden devam etmeden önce yazmaçlara yüklenmek zorunda olan veri grubuna, donanım içeriği adı verilir. Donanım içeriği süreç çalışması için ihtiyaç duyulan bütün bilgileri içeren, süreç çalışma bağlamının bir alt kümesidir. Unix'te, bir sürecin donanım içeriğinin bir kısmı TSS bölmesine, kalan kısmı da çekirdek modu yığınına saklanır.

Süreç geçişleri sık sık olduğundan, donanım içeriğini saklamak ve yeniden yüklemek için harcanan zamanı en aza indirmek önemlidir.

1.2.3.2. Görev Durumu Bölmesi

Intel 80x86 mimarisi, donanım içeriğini saklamak için, Görev Durumu Bölmesi (Task State Segment-TSS) adı verilen özel bir bölme türü içerir. Her süreç en az 104 byte uzunluğunda kendi TSS bölmesini içerir. Donanım tarafından otomatik olarak kaydedilmemiş yazmaçları ve G/Ç izin bitlerini (I/O Permission Bitmap) depolamak için işletim sistemi ek byte'lara ihtiyaç duyar [5].

1.2.3.3. switch_to Makrosu

switch_to makrosu bir süreç açış işlemini icra eder. Önceki (prev) ve sonraki (next) ile belirtilen iki parametre kullanır: İlki askıya alınacak sürecin süreç tanımlayıcı işaretçisi, ikincisi CPU üzerinde çalıştırılacak sürecin süreç tanımlayıcısı işaretçisidir. Bu makro *schedule* fonksiyonunca CPU üzerinde yeni bir süreç çalıştırmak için plan yapılmak amacı

ile harekete geçirilir. *switch_to* makrosu çekirdeğin en çok donanım bağımlı alt programlarından biridir.

1.2.3.4. Kayan Nokta Yazmaçlarının Saklanması

Aritmetik Kayan Nokta Birimi (Arithmetic Floating Point Unit-FPU) CPU içine 80x86 işlemciler ile başlayarak birleşik inşa edildi. Matematiksel ikinci işlemci (mathematical coprocessor) kayan nokta hesaplarının pahalı özel amaçlı yongalarla uygulandığı günlerin anısı için hala kullanılmaktadır. Eski modellerle uyumluluğu sürdürmek için yine de, kayan nokta aritmetiği fonksiyonları *escape* komutlarını kullanmaya başvurarak icra edilmektedir. *escape* komutları, baştaki (prefix) byteları 0xd8 ve 0xdf arasında değişen komutlardır. Bu komutlar CPU'a dahil edilen bir küme kayan nokta kayıtı üzerinde etkili olurlar. Açıkçası, eğer bir süreç *escape* komutları kullanıyorsa, kayan nokta yazmaçlarının içerikleri onun donanım içeriğine aittir. Intel 80x86 mikroişlemcileri kayan nokta yazmaçlarını TSS içine otomatik olarak saklamazlar. Yine de, çekirdeklerin bu kayıtları yalnız gerekli olduğunda saklamalarına olanak sağlayan bazı donanım destekleyicilerini içerirler.

1.2.4. Süreçlerin Yaratılması

Unix işletim sistemi kullanıcılarını memnun etmek için yüklü bir şekilde süreç yaratmaya başvurur. Örnek olarak, kabuk (shell) süreci kullanıcı ne zaman bir komut verirse kabuğun bir başka kopyasını çalıştıran yeni bir süreç yaratır. Geleneksel Unix sistemleri bütün süreçlere aynı şekilde muamele eder. Ebeveyn süreç tarafından sahip olunan kaynaklar katlanır, ve bir kopyası çocuk sürece verilir.

Bu yaklaşım, süreç yaratılışını çok yavaş ve verimsiz kılar. Çünkü ebeveyn sürecinin bütün adres alanını kopyalamayı gerektirir. Çocuk süreç, ebeveynine ait kaynakların tümünü okumak ya da değiştirmek ihtiyacını nadiren duyar. Bir çok durumda, hemen bir *execve* komutu verir ve o kadar dikkatli olarak yaratılmış adres alanını siliverir. Çağdaş Unix çekirdekleri bu sorunu üç ayrı mekanizma ile çözer:

•Yazarken Kopyala (Copy On Write) tekniği ebeveyn ve çocuğun aynı fiziki sayfaları okuyuşlarına olanak verir. Ne zaman biri fiziki sayfaya yazmağa kalkışırsa, çekirdek onun içeriğini yazan sürece atanmış olan yeni bir fiziksel sayfaya kopyalar.

• Hafif süreçler hem ebeveynin hem de çocuğun süreç başına yaratılan bir çok çekirdek veri yapısını paylaşmalarına olanak sağlar. Sayfalama (paging) tabloları ve açık dosya tabloları gibi.

• *fork* sistem çağrısı ebeveyninin bellek adres alanını paylaşan bir süreç yaratır.

Çocuğun ihtiyaç duyduğu verinin ebeveyn tarafından üstüne yazılmaması için, ebeveynin çalıştırılması çocuk bitinceye ya da çocuk yeni bir program çalıştırıncaya kadar engellenir.

1.2.4.1. Süreç Yaratma Kavramları

Süreçler fork sistem çağrısı ile oluşturulurlar. fork ile yaratılan bir alt süreç orjinal üst süreçin bir kopyasıdır, sadece kendisine ait süreç kimliği farklıdır.

Bir alt sürec dalladıktan sonra, üst ve alt süreçler normal çalışmalarına devam ederler. Eğer yazılımımızın devam etmeden önce alt süreçleri çalışmalarını bitirinceye kadar beklemesini istiyorsak, bunu dallanma işleminden hemen sonra wait veya waitpid işlevlerini çağırarak açıkça yapmanız gerekir. Bu işlevler alt sürecin neden sonlandırıldığı hakkında sınırlı bilgi verirler. Örneğin, çıkış durum kodu gibi.

Yeni dallanan bir alt süreç aynı yazılımı, fork çağrısının döndüğü noktada, üst süreci olarak çalıştırmaya devam eder. Fork işlevinin dönüş değerini yazılımın üst süreçte mi yoksa alt süreçte mi çalıştığını söylemek için kullanabilirsiniz.

Aynı yazılımı çalıştıran çeşitli süreçlerin olması ara sıra kullanışlıdır. Fakat alt süreç exec işlevlerinden birini kullanarak bir başka yazılımı da çalıştırabilir; Sürecin çalıştırdığı yazılıma süreç görüntüsü denir. Yeni yazılımın çalıştırılmasının başlatılması, sürecin, önceki süreç görüntüsü hakkındaki herşeyi unutmasına sebep olur; yeni yazılım sonlandığında, önceki süreç görüntüsüne dönülmez, süreç de sonlanır.

1.2.4.2. Süreç Yaratma Yöntemleri

1. *pid_t fork(void)*: fork işlevi yeni bir süreç oluşturur. Süreç oluşturma başarılıysa, hem üst hem de alt süreçler çalışır ve her ikisi de fork işlevinin dönüş değerini görür, ancak bu değerler farklıdır: fork işlevi, alt süreçte 0 değerini ve üst süreçte alt sürecinin süreç kimliğini döndürür. Süreç oluşturma başarısızsa, fork işlevi üst süreçte -1 değerini döndürür. Aşağıdaki errno hata durumları fork işlevi için tanımlanmıştır:

- *eagain*: Başka süreç oluşturmak için yeterli sistem kaynağı yok ya da kullanıcının zaten çok fazla süreci çalışmakta. Bu RLIMIT_NPROC kaynak sınırının aşılmakta olduğu anlamına gelir, bu genellikle artırılabilir.
- *enomem:*Süreç sistemin sağlayabileceğinden fazla yere ihtiyaç duymaktadır.

Alt süreci üstünden farklılaştıran özellikleri:

- Alt sürecin kendi süreç kimliği vardır.
- Alt sürecin üst süreç kimliği üst sürecinin süreç kimliğidir.
- Alt süreç, üst sürecin açık dosya tanımlayıcılarının kendine ait kopyalarını alır. Böylece üst süreçteki dosya tanımlayıcısının özelliklerinin değiştirilmesi alttaki dosya tanımlayıcıları etkilemez, bu tersi için de geçerlidir.Ancak, her tanımlayıcıyla ilişkilendirilmiş olan dosya konumu her iki süreç tarafından paylaşılır. Alt süreçler için biten işlemci süreleri sıfırlanır.
- Alt süreç üstü tarafından kurulmuş dosya kilitlerini miras almaz.
- Alt süreç üstü tarafından kurulmuş uyarıları miras almaz.
- Alt süreç için bekleyen sinyal kümesi temizlenir. (Alt süreç baskılanmış sinyallerin maskesini ve sinyal hareketlerini üst sürecinden miras alır.)

2. pid_t vfork(void): vfork işlevi fork gibidir fakat bazı sistemlerde daha verimlidir; fakat, güvenli kullanımı için bazı kısıtlar vardır. Fork çağıran sürecin adres alanının tam bir kopyasını alıp hem alt hem de üst sürecin bağımsız olarak çalışmasını sağlarken, vfork bu kopyayı yapmaz. Bunun yerine vfork ile oluşturulan alt süreç _exit veya exec işlevleri çağrılana kadar üst sürecin adres alanını paylaşır. Bu arada, üst süreç çalışmayı askıya alır. vfork ile oluşturulan alt sürecin evrensel (global) verileri, hatta üstüyle paylaştığı yerel değişkenleri bile değiştirmesine izin vermemek konusunda dikkatli olmalısınız. Bundan başka, alt süreç vfork'u çağıran işlevden dönemez. Bu üst sürecin kontrol bilgisini karıştırabilir. Bazı işletim sistemleri gerçek anlamda vfork'u uygulamazlar. GNU C kütüphanesi vfork'u bütün sistemlerde kullanmanıza izin verir, ancak aslında vfork yoksa fork çalıştırır. Eğer vfork kullanımında önlemlerinizi alırsanız, yazılımınız, sistem onun yerine fork'u kullansa da çalışır.

1.2.5. Süreçleri Yok etmek

Süreçlerin çoğu koşturmak zorunda oldukları kodun çalışmasını bitirmek anlamında ölürler. Bu olduğunda, çekirdek, bu sürecin sahibi olduğu kaynakları serbest bırakmak için haberdar edilmelidir. Bu bellek, açık dosyalar ve semafor gibi başka kaynakları içerir. Bir sürecin alışılagelmiş bitiş şekli exit sistem çağrısını harekete geçirmektir. Bu sistem çağrısı programcı tarafından doğrudan konabilir.

Bu tipik olarak, sürecin elde edemeyeceği veya yok sayamayacağı bir işaret aldığında veya çekirdek bir süreç için çalışırken, çekirdek modunda kurtarılamayacak bir CPU istisnası ilan edilirse ortaya çıkar.

1.2.5.1. Süreç Kaldırılması

Unix işletim sistemi bir sürecin ebeveyninin PID'sini temin etmek için ya da çocuklarından herhangi birinin çalışma durumunu temin etmek için o sürecin çekirdeği sorgulayışına izin verir. Bir süreç, örneğin belirli bir görev için bir çocuk süreç yaratır ve sonra wait benzeri bir sistem çağrısı icra ederek çocuğun bitip bitmediğini kontrol eder. Eğer çocuk bitmiş ise, bitiş kodu ebeveyn sürece görevin başarı ile yapılıp yapılmadığını anlatır. Bu tasarım seçeneklerine uymak için, Unix çekirdeklerinin bir süreç bittikten hemen sonra süreç tanımlayıcı alanı içine dahil edilen verileri çöpe atmalarına izin verilmez. Böyle yapmalarına yalnız ebeveyn süreç bir wait benzeri sistem çağrısını yaptıktan sonra izin verilir. task_zombie durumu işte bu yüzden kullanılmıştır. Süreç teknik açıdan ölü olsa bile, tanımlayıcısı ebeveyn süreci haberdar edilinceye kadar saklanmalıdır.

Eğer ebeveyn süreç çocuklarından önce sona ererse ne olur? Böyle bir durumda, sistem var olan görev hanelerini kullanıp tüketebilen zombie süreçler ile dolup taşabilir. Daha önce bahsedildiği gibi, bu problem bütün sahipsiz süreçlerin init sürecinin çocukları haline getirilişi ile çözülür.

Bu şekilde, init süreci bir wait benzeri sistem çağrısı ile kendi meşru çocuklarından birini kontrol ederken zombie süreçleri yok eder.

1.2.5.2. Süreç İzleme Mekanizmasının Kullanıldığı Uygulamalar

Süreç izleme mekanizmaları çeşitli bilimsel çalışmaların ihtiyaç duyduğu süreçlere ait durum bilgilerinin temin edilmesinde bir araç olarak görev yaparlar.

Süreçlerin davranışlarını sergiledikleri çalışma ortamları işletim sistemi ile süreçler arasında yer alan ve yazılımla inşa edilmiş çeşitli arayüzler tarafından oluşturulabilirler [6]. 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.

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

1. 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 [5,9].
2. 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 yararlanarak kişilerin sistemi kullanım amaçları yada niyetleri belirlenebilir. [8-10].
3. 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 [11-12].
4. Ortam Modelleme ve Yeniden Oluşturma: Programların derlenmesi esnasında yapılan sistem çağrılarının izlenmesi ile programların hangi ortamda hangi kütüphane dosyalarını kullandığı belirlenebilir [13, 15].

5. 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 [16].

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 [14-17].

1.2.6. - Borular ve FIFO'lar

Bir boru süreçler arası haberleşme mekanizmasıdır. Bir süreç tarafından boruya yazılan veri başka bir süreç tarafından okunabilir. Veri ilk giren, ilk çıkar (FIFO) sırasıyla ele alınır. Borunun adı yoktur; bir kullanımlık oluşturulur ve her iki uç, boruyu oluşturan süreç tarafından erişilebilir olmalıdır.

Bir FIFO özel dosyası boru ile aynıdır, fakat anonim, geçici bağlantı olmak yerine, bir FIFO'nun bir adı veya diğer dosyalar gibi isimleri vardır. Süreçler FIFO'yu üzerinden haberleşmek için açarlar.

Boru veya FIFO'nun her iki ucu aynı anda açılmalıdır. Eğer herhangi bir sürecin üzerine yazmadığı bir boru veya FIFO dosyasından okuma yapıyorsanız (belki hepsi dosyayı kapatmış veya çıkmış olabilir), okuma sonucunda dosya-sonu (EOF) döner. Üzerinde okuma işlemi olmayan bir boru veya FIFO'ya yazmak hata durumu olarak karşılanır; bir SIGPIPE sinyali üretir ve eğer sinyal yakalanıyor ya da bloklanıyorsa EPIPE hata koduyla sonlanır.

Ne borular ne de FIFO özel dosyaları dosya içinde konumlamaya izin vermez. Okuma ve yazma işlemleri sırayla gerçekleşir; dosyanın başından okunur ve sonuna yazılır.

1.2.6.1. Bir Borunun Oluşturulması

Boru oluşturmak için en ilkel işlev pipe işlevidir. Bu borunun okuma ve yazma uçlarının her ikisini de oluşturur. Tek bir sürecin kendisiyle konuşması için boru kullanımı pek kullanışlı değildir. Tipik kullanım şekli, bir işlemin bir veya daha fazla alt süreci oluşturmadan önce boruyu oluşturmasıdır. Bundan sonra boru üst ve alt süreç arasında veya iki alt süreç arasında haberleşme için kullanılır.

pipe işlevi unistd.h başlık dosyası içinde tanımlıdır. *int pipe (int dosyatnm[2])* pipe işlevi boruyu oluşturur ve borunun okuma ve yazma uçları için dosya tanımlayıcıları (sırasıyla) dosyatnm[0] ve dosyatnm[1] içine koyar.

Girdi ucunun önce geldiğini hatırlamanın kolay bir yolu dosya tanımlayıcı 0'ın standart girdi, ve dosya tanımlayıcı 1'in standart çıktı olmasıdır.

Başarı halinde pipe, 0 değerini döndürür. Başarısızlık halinde ise -1 döndürülür. Aşağıdaki errno hata durumları bu işlev için tanımlanmıştır:

- *emfile:*Sürecin çok sayıda açık dosyası var.
- *enfile:*Sistemde çok sayıda açık dosya var. Bu hata GNU sisteminde hiçbir zaman oluşmaz.

1.2.6.2. Borunun G/Ç Bütünlüğü

Yazılan verinin miktarı PIPE_BUF değerinden büyük olmadığı sürece borudan okuma ve yazma işlemi *atomik* bir işlemdir. Bu veri aktarımının anlık bir birim olarak göründüğü anlamına gelir, bu nedenle sistemdeki hiçbir şey tamamlanmış halini gözlemleyemez. Atomik G/Ç hemen başlayamayabilir (tampon alanı veya veri için beklemesi gerekebilir), fakat başladı mı hemen biter.

Büyük miktarda veri okumak veya yazmak atomik olmayabilir; örneğin, dosya tanımlayıcısını paylaşan diğer süreçlerin çıktı verisi araya serpiştirilmiş olabilir. Aynı zamanda, bir kere PIPE_BUF'a karakterler yazıldığında, okuma yapılıncaya kadar başka yazımlar durdurulur.

1.3. Sistem Çağrıları

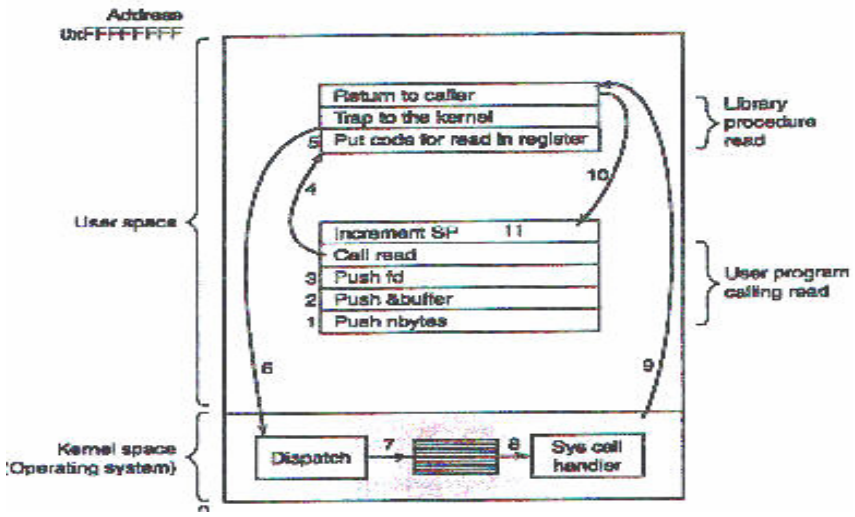
İşletim sistemi ile kullanıcı programları arasında, işletim sistemi tarafından sağlanan sistem çağrıları aracılığıyla bir arayüz tanımlanır. İşletim sisteminin gerçekte ne yaptığını anlamak için, bu arayüzü açıkça incelememiz gerekir. Sistem çağrıları işletim sisteminden işletim sistemine farklı arayüzler kullanabilir.

Bir sistem çağrısı yapmanın gerçek mekaniği iyi bir makineye bağlıdır ve çoğunlukla çağrılar assembly kodunda belirtilmelidir. Assembly dilinin haricinde C programlarından ve buna benzer diğer dillerden de sistem çağrıları yapmak mümkündür.

Sistem çağrıları bir sonraki işlemi akılda tutmak için çok yararlıdır. Herhangi bir tek işlemcili bilgisayar aynı anda sadece bir komut çalıştırabilir. Eğer kullanıcı modunda bir

kullanıcı programı bir işlem gerçekleştiriyorsa ve bir dosyadan veri okuma gibi bir sistem hizmetine gereksinim duyarsa, trap komutunu veya kontrolü işletim sistemine devretmek için bir sistem çağrısını çalıştırmak zorundadır. İşletim sistemi sonra sistem çağrısının parametrelerini gözden geçirerek işlemin neyi istediğini anlar. Daha sonra sistem çağrısı gerçekleştirilir ve kontrol bir sonraki sistem çağrısına döndürülür. Bir sistem çağrısı yapmak özel bir prosedür çağrısı yapmak gibidir, aradaki fark sadece sistem çağrıları kernel'a girer, prosedür çağrıları ise giremez [2].

Bir sistem çağrısı üç tane parametreye sahiptir. Birincisi dosyayı bildirir, ikincisi buffer belleğe dosyanın yerini bildirir ve üçüncüsü ise verinin kaç byte uzunluğa sahip olduğunu bildirir.



Şekil 1.4. Sistem Çağrısı Çalışması

C programından bir sistem çağrısı aşağıdaki şekilde yapılabilir :

count = read(fd, buffer, nbytes)

Eğer sistem çağrısı yanlış bir parametre ya da disk hatası nedeniyle çalıştırılamıyorsa, sayaç (count) -1'e ayarlanır ve hata numarası errno global değişkenine yerleştirilir. Programlar, eğer bir hata meydana geldiyse bu hatayı görmek için her zaman sistem çağrılarının sonuçlarını kontrol etmelidirler.

Sistem çağrısı bir dizi adımlarla gerçekleştirilir. Read sistem çağrısını gerçekleştirmek için yapılması gereken ilk şey Şekil 1.4.' da 1 ile 3. adımlar arasında görüldüğü gibi stack

üzerine parametreleri yerleştirmektir. Birinci ve üçüncü parametreler bir değerdir, fakat ikinci parametre buffer'ın adresidir, içeriği değildir. 4. adımda kütüphane prosedürüne asıl sistem çağrısı gelir. Bu komut tüm prosedür çağrılarında kullanılan normal bir prosedür çağrısıdır.

İmkan dahilinde assembly dilinde yazılan kütüphane prosedürü, tipik olarak işletim sisteminin kaydedici gibi olmasını istediği bir yere sistem çağrı numarasını yerleştirir(5.Adım). Daha sonra, işletim sistemi kullanıcı moddan kernel moda geçmek için TRAP komutunu çalıştırır ve kernel içerisinde belirlenmiş adresler üzerinde çalışma başlatılır (6.Adım).

Kernel sistem çağrı numarasını inceler ve sonra doğru sistem çağrısını gönderir (7.Adım). Burada sistem çağrısı çalışır (8.Adım). Sistem çağrısının çalışması sona erdiğinde, kontrol kullanıcı-alan kütüphane prosedürüne dönebilir (9.Adım). Bu prosedür daha sonra her zaman prosedür çağrılarının dönmesi gibi kullanıcı programa döner (10. Adım). İşin sona ermesi için, kullanıcı programı stack'i boşaltmalıdır (11.Adım). POSIX, 100 civarında prosedür çağrısına sahiptir. Bu çağrılar, işlem yönetimi, dosya yönetimi, dizin ve dosya sistem yönetimi ve çeşitli çağrılar olarak gruplandırılabilir.

1.3.1. İşlem Yönetimi Sistem Çağrıları

Fork, UNIX'te yeni bir işlem oluşturmak için kullanılan sadece bir yoldur. Fork, tüm dosya tanımlayıcıları ve kaydedicilerin dahil olduğu orijinal işlemin bir kopyasını oluşturur. Fork'tan sonra, orijinal işlem ve kopyası (ana ve çocuk işlem) farklı yollara giderler.

Tablo 1.1. İşlem Yönetimi Sistem Çağrıları

Çağrı	Tanımı
pid = fork ()	Ana işlemle hemen hemen aynı olan çocuk işlemi oluşturur.
pid = waitpid (pid,&statloc,options)	Bir çocuk işlemi sonlandırmak için bekler.
exit(status)	Çalışan işlemi sonlandırır ve başlangıç durumuna döner.

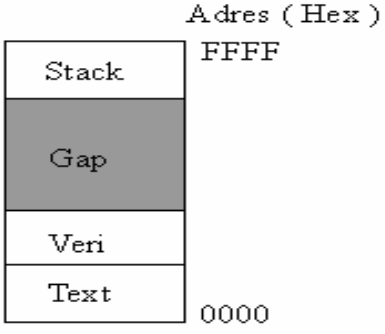
Fork çağrısı boyunca hemen hemen tüm değişkenler aynı değerlere sahiptir, fakat ana işlemin verileri child işlem oluşturmak için kopyalandığında, ana işlem veya çocuk işlemde yapılan bir değişiklik diğerini etkilemeyecektir. Fork sistem çağrısı, çocuk işlemde sıfır olan ve çocuk işlemin tanımlayıcısına veya ana işlemde PID değerine sahip bir değer döndürür. PID kullanılarak değer döndürüldüğünde, biri ana işlem ve diğeri de çocuk işlem olmak üzere iki işlem görülebilir. (Ana ve çocuk işlem)

Birçok durumda, bir fork çağrısından sonra, çocuk işlem ana işlemden farklı kodlar çalıştırmak isteyecektir. Bu durumda shell, terminalden bir komut okur, bu sırada çocuk işlem komutu çalıştırmak için bekler ve sonra çocuk işlem sona erdiğinde, shell terminalden bir sonraki komutu okur. Çocuk işlemin sona ermesi için beklerken, ana işlem sadece çocuk işlem sona erene kadar varolan waitpid sistem çağrısını çalıştırır. Waitpid sistem çağrısı, ilk parametreyi -1'e ayarlayarak özel bir çocuk işlem veya herhangi çocuk işlem için bekler. Waitpid çağrısı tamamlandığında, ikinci parametre olan ve çocuk işlemin çıkış durumunu (normal veya normal olmayan sonlandırma ve çıkış değeri) ayarlayacak olan statloc ile adres işaretlenir. Ayrıca üçüncü parametre yoluyla bir takım seçeneklerde sağlanır.

Bir komut yazıldığında shell yeni işlemi ikiye ayırır. (Çocuk ve ana işlem) Bu çocuk işlem kullanıcı komutunu çalıştırmalıdır. Çocuk işlem execve sistem çağrısını kullanarak bu komutunu çalıştırır [4].

Bir çok durumda execve sistem çağrısı genel olarak üç parametreye sahiptir: çalıştırılacak dosyanın ismi, ikincisi işlem dizisi için bir işaretçi ve üçüncüsü de bulunulan ortama ilişkin bir işaretçidir. Execve'nin genel formatı:

s = execve (name, argv, environp)



Şekil 1.5. Bir İşlemin Sahip Olduğu Bölümler

Bir başka işlem yönetimiyle ilgili sistem çağrısı exit'tir. Exit sadece bir parametreye sahiptir ve bu parametre waitpid sistem çağrısındaki statloc komutu ile ana işleme dönmeyi sağlayan çıkış durumudur.

UNIX işletim sisteminde işlemler belleği üç segmente bölmüşlerdir: text segment, data segment ve stack segment. Şekilde de görüldüğü gibi data segment yukarıya doğru, stack segment ise aşağıya doğru büyür. Bu ikisinin arasında kullanılmamış adres alanı olan gap yer alır. Stack segment, ihtiyaç duyduğu için gap bölgesine doğru otomatik olarak büyür, fakat data segmentinin genişlemesi brk denilen bir sistem çağrısıyla yapılır.

1.3.2. Dosya Yönetimi Sistem Çağrıları

Tablo 1.2. Dosya Yönetimi Sistem Çağrıları

Çağrı	Tanımı
fd = open (file, how,.....)	Bir dosyayı okuma, yazma ya da her ikisi için açar.
s = close (fd)	Açık olan bir dosyayı kapatır.
n = read (fd,buffer,nbytes)	Buffer belleğin içerisine dosyadan veri okur.
n = write (fd,buffer,nbytes)	Bir dosyanın içerisine buffer bellekten yazar.
position = iseek(fd,offset,whence)	Dosya işaretçisini taşır.
s = stat (name, &buf)	Bir dosyanın durum bilgisini alır.

Dosya sistemiyle ilgili olarak UNIX'te bir çok sistem çağrısı bulunmaktadır. Bir dosyayı okumak ya da onun üzerine yazmak için ilk önce dosya open çağrısı ile açılmalıdır. Bu çağrı dosyanın doğrudan ismini verir ve sadece okuma, sadece yazma veya her ikisini de sağlayan O_RONLY, O_WRONLY veya O_RDWR kodlarından biri kullanılır. Yeni bir dosya oluşturmak için O_CREAT çağrısı kullanılır. Dosya tanımlayıcısı geri dönerek, dosya okuma veya yazma için kullanılabilir [7]. Daha sonra dosya close komutuyla kapatılabilir. Close çağrısı açık olan bir dosya için dosya tanımlayıcısı yapabilir

Dosya yönetimi için sistem çağrılarının çoğunluğu okuma ve yazma için kullanılır. Bunlardan en önemlileri de read ve write çağrılarıdır. Bu iki çağrı da benzer parametrelere sahiptir.

Birçok program dosya okumak ve yazma için kullanılmasına karşın, bazı uygulama programları random olarak dosyanın herhangi bir parçasına girmek isterler. Her dosya bir pointer ile ilişkilidir ki bu pointer dosyanın bulunduğu pozisyonu belirtmektedir. Ardışık okuma olduğu zaman, pointer normal olarak bir sonraki byte'ı gösterecektir. Iseek çağrısı, pointer pozisyonunun değerini değiştirir. Böylece dosyanın herhangi bir yerindeki okuma ya da yazma işlemi başlatılabilir.

Lseek çağrısı üç tane parametreye sahiptir: birincisi dosya için dosya tanımlayıcısıdır, ikinci olarak dosya pozisyonu ve üçüncü olarak dosyanın başlangıcı ile bulunulan pozisyonu karşılaştırarak dosya sonu olup olmadığını anlar. Pointer değiştikten sonra, dosyanın bulunduğu pozisyonu belirtmek için Iseek tarafından bir değer döndürülür.

UNIX her dosya için, dosya modunu (normal dosya, özel dosya, directory vb.), boyutunu, son yapılan değişikliğin zamanını ve diğer bilgileri tutar. Birinci parametre incelenecek dosyaları bildirir, ikinci parametre bilginin yerleştirildiği yeri bildiren pointer'dır.

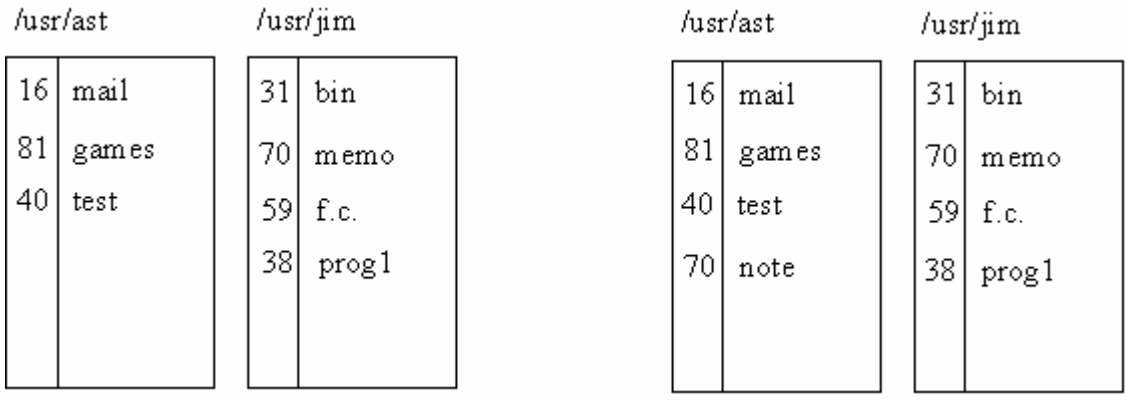
1.3.3. Dizin Yönetimi Sistem Çağrıları

Tablo 1.3. Dizin Yönetimi Sistem Çağrıları

Çağrı	Tanımı
s = mkdir (name, mode)	Yeni bir dizin oluşturur.
s = rmdir (name)	Boş olan bir dizini siler.
s = link (name1, name2)	Name2 adında yeni bir dosya oluşturur ve name1'e bağlar.
s = unlink (name)	Boş olan bir dizini siler.
s = mount (special, name, flag)	Bir dosya sistemi ekler
s = unmount (special)	Bir dosya sistemi siler.

İlk olarak boş bir klasör oluşturmak için veya silmek için kullanılan mkdir ve rmdir olmak üzere iki çağrı vardır. Bir sonraki çağrı ise link'tir. Link çağrısının amacı farklı directorylerde bulunan iki veya daha çok isim altında gözüken dosyalara izin verir. Tipik bir kullanımı ortak bir dosyayı paylaşmak için aynı programlama grubunun üyelerine izin vermesidir. Onların her biri farklı isimler altında olması mümkün olan kendi dizinlerinde

gözüken dosyalara sahiptir. Paylaşma grubun her üyesine özel bir kopya vermeye benzemez, çünkü paylaşılan bir dosyaya sahip olmanın anlamı takımın üyelerinden herhangi birinin yaptığı değişikliğin anında diğerlerine gözükmesidir. Bir dosyanın kopyası yapıldığı zaman, sonradan yapılan değişiklikler bir diğer dosyayı etkilemez.



Şekil 1.6. Ast'ın dizinine /usr/jim/memo dosyası aktarılmadan önceki durum ve aktarılma işleminden sonra iki dizinin aldığı benzer durum

Şekildeki durumda bir link'in nasıl çalıştığı görülmektedir. Burada iki kullanıcı vardır, ast ve jim, her biri birkaç dosyayla birlikte kendi directory'lerine sahiptir. Eğer ast aşağıdaki sistem çağrısını içeren bir programı çalıştırırsa,

Link("/usr/jim/memo","/usr/ast/note");

Jim'in directory'sinde bulunan memo dosyası Ast'ın directory'si içindeki note ismi altında girilir. Bundan sonra, */usr/jim/memo* ve */usr/ast/note* dosyaları aynı dosyalar olacaktır. Bir ek bilgi olarak, kullanıcı dizinlerinin /usr, /user, /home veya başka dizinlerde tutulup tutulmadığı yerel bir kullanıcı tarafından basitçe bir kararla yapılır.

UNIX'de her dosya tek bir numaraya sahiptir ki bu numara onu tanıtan i-numarasıdır. Bu i-numarası her dosya başına, i-düğümlerinin bir tablosunu içeren indekslerdir. Onun disk bloklarının bulunduğu dosyaya kimin sahip olduğunu söyler. Bir directory bir çift eleman içeren basitçe bir dosyadır.

UNIX'in ilk versiyonlarında, her bir directory i-numarası için 2 – 16 byte ve isim için 14 byte'lık bir giriş vardır. Şekilde mail 16 i-numarasına sahiptir. Link çağrısı basitçe var olan bir dosyanın i-numarasını kullanarak, yeni bir isim girerek yeni bir directory oluşturur. Yine şekilde iki giriş aynı i-numarasına sahiptir ve bu nedenle ikisi de aynı dosyadır. Eğer daha sonra unlink sistem çağrısını kullanarak ikisinden biri silinirse diğeri

kalır. Eğer her ikisi de silinirse, UNIX diskte var olan dosya silindiği için herhangi bir dosya göremez.

Mount sistem çağrısı iki dosyayı bir dosya olarak birleştirmeye izin verir. Genel bir durum root dosya sistemi genel komutların ve hard disk'te diğer çok kullanılan komutların çalıştırılabilir versiyonlarını içermelidir. Kullanıcı bir disk yerleştirdiği zaman floopy disk sürücüsündeki verileri okuyabilir.

1.3.4. Diğer Sistem Çağrıları

Tablo 1.4. Diğer Sistem Çağrıları

Çağrı	Tanımı
s = chdir (name)	Aktif dizini değiştirir.
s = chmod (name,mode)	Dosyanın koruma bitini değiştirir.
s = kill (pid,signal)	İşleme bir sinyal gönderir.
seconds = time (&seconds)	Sistemin zamanını gösterir.

chdir aktif directory'i değiştirmek için kullanılır. Bu sayede uzun dosya isimleri yazmaksızın istenilen dosyaya kolayca ulaşılabilir.

UNIX'te her dosya koruma için bir moda sahiptir. Mod, kendisi, grup ve diğerleri için okuma, yazma, çalıştırma bitlerine sahiptir. *chmod* sistem çağrısı dosyanın modunu değiştirmeyi gerçekleştirir. Örneğin, bir dosya sahibinden başka herkes için sadece okunur yapılabilir.

kill sistem çağrısı kullanıcılara ve kullanıcı işlemlerine sinyaller gönderen bir yoldur. POSIX zamanla ilgili olan birkaç prosedür tanımlamıştır. Örneğin, zaman sadece aktif zamanda saniyelerde ilerler. 32 bitlik bilgisayarlarda maksimum zaman değeri 2^{32} -2 saniyedir. Bu değer 136 yılın biraz fazlasına denk gelir. Bu nedenle 2145 yılında, 32 bitlik UNIX sistemleri karışacak, meşhur Y2K problemi ortaya çıkacaktır. Eğer 32 bitlik bir UNIX sisteme sahipseniz, 2145 yılından önce 64 bitlik bir sisteme geçmenizde fayda vardır.

1.4. /proc Dosya Sistemi

/proc dosya sistemi, çekirdek ve çekirdek modüllerinin programlara bilgi vermek için oluşturulmuş bir yöntemdir. Sanal bir dosya sistemi olan proc, çekirdeğin iç veri yapıları hakkında bilgi almak, sistemde çalışmakta olan programlar hakkında kullanışlı bilgiler edinmek ve çekirdeğin parametrelerini değiştirerek, çalışmakta olan sistemde yapılandırmalar yapılmasını sağlamaktadır. Diğer dosya sistemleri diskte iken proc, geçici bellekte yeralmaktadır. mount komutu gibi, sistemde bağlanmış tüm dosya sistemlerini listeleyen /proc/mounts dosyasına bir gözatılırsa, aşağıdaki gibi bir satır görülür:

grep proc /proc/mounts

/proc /proc proc rw 0 0

/proc çekirdek tarafından denetlemekte ve bir aygıta bağlı değildir. Genellikle durum bilgileri içerdiğinden, proc dosya sistemini çekirdeğin yönettiği geçici belleğe yerleştirmek en mantıklı seçimdir. /proc'da 'ls -l' komutunu çalıştırırsanız, çoğu dosyanın 0 byte büyüklüğünde olduğu göreceksiniz. Eğer, dosya içeriğine bakacak olursanız, içinde birçok bilginin yeraldığını göreceksiniz. Bu nasıl olmaktadır? Bunun nedeni, /proc dosya sistemi, diğer sıradan dosya sistemleri gibi, kendisini Sanal Dosya Sistemi (SDS) (VMS) katmanına kaydettirmektedir. Ancak, SDS dosya veya dizinler hakkında i-node'lar ile ilgili sistem çağrıları kullandığında, /proc dosya sistemi bu dosya veya dizinleri, çekirdek içerisindeki bilgilerden yaratmaktadır.

1.4.1. /proc Dosya Sistemi Bağlanması

Eğer, daha önceden bağlanmamış ise, proc dosya sistemini aşağıdaki komutu kullanarak bağlayabilirsiniz:

mount -t proc proc /proc

Yukarıdaki komut başarılı bir şekilde bağlama işlemi yerine getirecektir.

1.4.2. /proc'taki Dosyaların Görüntülenmesi

/proc'taki dosyalardan, bilgisayarınızın özellikleri, çekirdeğin ve çalışan programların durumları gibi bilgiler edinebilirsiniz. /proc'taki birçok dosya, sisteminizin donanımsal ortamı hakkındaki en son kullanım durumunu yansıtmaktadır. Her nekadar, /proc'taki

dosyalar sanal olsalar da, dosyaların içeriğini, çeşitli kelime işlemcileri veya 'more', 'less', 'cat' gibi komutlarla görüntüleyebilirsiniz [6]. Herhangi bir kelime işlemci burada yer alan bir dosyayı açmaya kalkıştığında, dosya çekirdek bilgilerinden otomatik olarak oluşturulmaktadır. Örnek sonuç aşağıdaki gibidir:

$ ls -l /proc/cpuinfo -r--r--r-- 1 root root 0 Dec 25 11:01 /proc/cpuinfo

$ file /proc/cpuinfo
/proc/cpuinfo: empty

$ cat /proc/cpuinfo

processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 8
model name : Pentium III (Coppermine)
stepping : 6
cpu MHz : 1000.119
cache size : 256 KB
fdiv_bug : no
hlt_bug : no
sep_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 mmx fxsr xmm
bogomips : 1998.85

processor : 3
vendor_id : GenuineIntel
cpu family : 6
model : 8
model name : Pentium III (Coppermine)
stepping : 6
cpu MHz : 1000.119
cache size : 256 KB
fdiv_bug : no
hlt_bug : no
sep_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 mmx fxsr xmm
bogomips : 1992.29

Yukarıdaki alanların çoğu kendi kendini açıklamakta ve sistemdeki donanım hakkında bilgiler sağlamaktadır. /proc'ta yer alan bazı bilgiler kodlanmış olarak yer almakta ve bunları bizlerin anlayabileceği bir şekile sokmak için çeşitli araçlar yazılmıştır. Bu araçlardan bazıları 'top', 'ps', 'apm' vs. dir.

1.4.3. Yararlı Sistem ve Çekirdek Bilgilerine Ulaşmak

Proc dosya sistemi, sistem ve çekirdek hakkında yararlı bilgilere ulaşmak için kullanılabilir. Bazı önemli dosyalar aşağıda listelenmiştir:

- /proc/cpuinfo - işlemci hakkındaki bilgiler (modeli, ailesi, cache büyüklüğü vs.)
- /proc/meminfo - gerçek bellek, swap alanı vs.
- /proc/mounts - tüm bağlanmış dosya sistemlerini listeler

- /proc/devices - var olan aygıtların listesi
- /proc/filesystems - desteklenen dosya sistemleri
- /proc/modules - yüklenmiş modüllerin listesi
- /proc/version - çekirdek sürümü
- /proc/cmdline - açılış sırasında çekirdeğe verilen parametreler

Yukarıda listelenenden çok daha fazla sayıda dosya /proc'ta yeralmaktadır. İlgili bir kullanıcı, /proc dizininde yeralan tüm dosyaları 'more' komutuyla görüntülemeli ve burada bulunan dosyalar hakkındaki daha fazla bilgiyi referanslarda verilen ilk bağlantıdan okumalıdır. kcore gibi bazı dosyalar çok büyük olabileceklerinden ve /proc'taki dosyalar hakkında bilgi sahibi olmadan önce, dosyaları görüntülerken 'cat' yerine 'more' komutunu kullanmak daha mantıklıdır.

1.4.4. Çalışan Programlar Hakkında Bilgiler

/proc dosya sistemi, çalışmakta olan tüm programlar hakkında bilgi sahibi olmanızı sağlamaktadır. /proc dizininde isimleri numara olan dizinler vardır. Herbir numara bir program numarasına (PID) karşılık gelmektedir [3]. Dolayısıyla, her çalışan program için /proc'ta kendi program numarasını taşıyan bir dizin yaratılmıştır. Bu dizinler içerisinde, programın durumu ve ortamı hakkında bilgiler veren dosyalar vardır. Çalışan bir programı araştırırsak:

$ ps -aef | grep mozilla
root 32558 32425 8 22:53 pts/1 00:01:23 /usr/bin/Mozilla

Yukarıdaki komuta göre mozilla 32558 PID'sine sahip olduğunu göstermektedir. Buna karşılık /proc'ta 32558 adlı bir dizin olması gerekir.

$ ls -l /proc/32558
total 0
-r--r--r-- 1 root root 0 Dec 25 22:59 cmdline
-r--r--r-- 1 root root 0 Dec 25 22:59 cpu
lrwxrwxrwx 1 root root 0 Dec 25 22:59 cwd -> /proc/
-r-------- 1 root root 0 Dec 25 22:59 environ

```
lrwxrwxrwx    1 root  root    0 Dec 25 22:59 exe -> /usr/bin/mozilla*
dr-x------    2 root  root    0 Dec 25 22:59 fd/
-r--r--r--    1 root  root    0 Dec 25 22:59 maps
-rw-------    1 root  root    0 Dec 25 22:59 mem
-r--r--r--    1 root  root    0 Dec 25 22:59 mounts
lrwxrwxrwx    1 root  root    0 Dec 25 22:59 root -> //
-r--r--r--    1 root  root    0 Dec 25 22:59 stat
-r--r--r--    1 root  root    0 Dec 25 22:59 statm
-r--r--r--    1 root  root    0 Dec 25 22:59 status
```

"cmdline" dosyasında, programın çalıştırma komutu yeralmaktadır. "environ" dosyasında programın ortam değişkenleri yeralmaktadır. "status" dosyasında, kullanıcı numarası (UID), grup numarası (GID), programı çalıştırmış olan ana programın program numarası (PPID) ve programın o anda "uyumakta" veya "çalışmakta" olup olmadığını belirten bilgi gibi, program hakkında durum bilgileri yeralmaktadır. Her programa ait dizinde birkaç adet sembolik bağlantı vardır. "cwd", programın çalışmakta olduğu dizine, "exe", programın kendisine, "root", programın kendi kök dizini olarak gördüğü dizine, ki genellikle bu "/" dizinidir, "fd", programın kullanmakta olduğu dosya tanımlayıcılarına bağlanmıştır. "cpu" ise, sadece Linux'un SMP, yani birden fazla işlemciyi destekleyen çekirdeklerde var olup, programın işlemci kullanım bilgilerini içermektedir.

/proc/self, bir programın kendisi hakkındaki bilgilere ulaşmak için kullanılabilecek ilginç bir dizindir. /proc/self, /proc dizinine ulaşan programın /proc'ta yeralan dizinine bir sembolik bağlantıdır.

1.4.5. Çekirdekle /proc Sayesinde Etkileşimde Bulunmak

Yukarıda sözünü ettiğimiz ve /proc'ta yeralan dosyaların çoğu sadece okunabilirdir. Ancak, çekirdekle etkileşimi sağlamaya yarayan /proc'ta hem okunabilir ve hem de yazılabilir dosyalar da vardır. Bu tür dosyaların içeriğini değiştirmek, çekirdeğin o andaki durumunu etkileyeceği için, dikkatli olunmasında yarar vardır. /proc/sys içeriği değiştirilebilinir tüm dosyaları içermektedir.

/proc/sys/kernel - Çekirdeğin genel çalışmasını etkileyen bilgileri içermektedir. /proc/sys/kernel/{domainname, hostname} alan adı ve bilgisayar adını tutmaktadır. Bu dosyaları ilgili bilgileri değiştirmek için kullanabilirsiniz.

$ hostname
machinename.domainname.com

$ cat /proc/sys/kernel/domainname
domainname.com

$ cat /proc/sys/kernel/hostname
machinename

$ echo "new-machinename" > /proc/sys/kernel/hostname

$ hostname
new-machinename.domainname.com

/proc/sys/kernel/ dizini içerisinde yapılandırılabilecek birçok dosya bulunmaktadır. Burada hepsini listelemek pek akıllıca olmaz. En iyisi kullanıcının dosyalara bakıp ayrıntıları kendisinin öğrenmesi olacaktır. Yapılandırma yapılabilecek bir dizin de /proc/sys/net dir. Buradaki dosyaları, sisteminizin bilgisayar ağlarındaki özelliklerini değiştirmek için kullanabilirsiniz. Sözgelimi, yapılacak basit bir değişiklik ile, bilgisayar ağ'da gizlenebilir:

$ echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_all

Bu komut sayesinde, icmp_echos paketlerinine bilgisayarınız cevap vermeyeceği için, ağ'da yeralan diğer bilgisayarlardan gelen ping sorguları cevapsız kalacaktır.

$ ping machinename.domainname.com
no answer from machinename.domainname.com

Eski haline getirmek için:

$ echo 0 > /proc/sys/net/ipv4/icmp_echo_ignore_all

Çekirdeğin özelliklerini değiştiren ve /proc/sys dizininde bulunan birçok dizin bulunmaktadır.Sonuç olarak; /proc dosya sistemi Linux'un iç yapısına yönelik dosya tabanli bir arayüz oluşturmaktadır. Sistemdeki aygıt ve süreçlerin belirlenmesi ve denetiminde yardımcı görev yapmaktadır.

1.5. Kullanılan Teknolojiler

1.5.1. AJAX (Asynchronous JavaScript and XML)

AJAX teknolojisi, web yazılıma yeni bir boyut katan araçlardan biridir. Bunun için hemen hemen herkesin ilgisini çekmektedir. AJAX aslında yeni bir olay değil sadece var olan bir kaç teknolojinin bir araya gelmesi ile oluşturulmuş bir yöntemdir. Ajax kullanılarak birçok uygulama yapılmıştır. Ajax sunucuda çalışacak betiğin sayfa yenilenmeden javascriptle tetiklenmesinden ibarettir. Yani aslında yeni olan araçlar değil araçların kullanma şeklidir. Bu tetiklemeyi tarayıcı tarafında Javascript ile yapılabilmesine imkan tanır [21,22]. Klasik bir istemci-sunucu (client-server) uygulamasında kullanıcı tarafından yapılan her işlem sunucuya *Http Request* olarak gönderilir, yorumlanır ve veriler işletildikten sonra sonuçlar istemciye gönderilir. Bu da çok büyük bir ağ trafiğine neden olur. Şekil 1.7.'de bu durum şematize edilmiştir [25].

Şekil 1.7. AJAX kullanılmayan web sayfalarının çalışma şeması

Ajax kullanıldığında ise istekler *Http Request* yerine *XmlHttpRequest* olarak gönderilir. Burada sunucu ve istemci arasında taşınan veriler sıkıştırılmış xml formatındadır. Bu sıkıştırılmış yapı istemci tarafında açılır böylece sunucu ve istemci arasındaki bant genişliği boş yere işgal edilmemiş olur. Ajax ile sağlanan en büyük özellik de *Parçalı Güncelleme* dir. Bu sayade bütün bir sayfanın istemci ile sunucu arasında gidip gelmesiyle doğan ağ trafiğinin önüne geçilmiş olur. Şekil 1.8.'de bu durum şematize edilmiştir.

Şekil 1.8. AJAX kullanılan web sayfalarının çalışma şeması

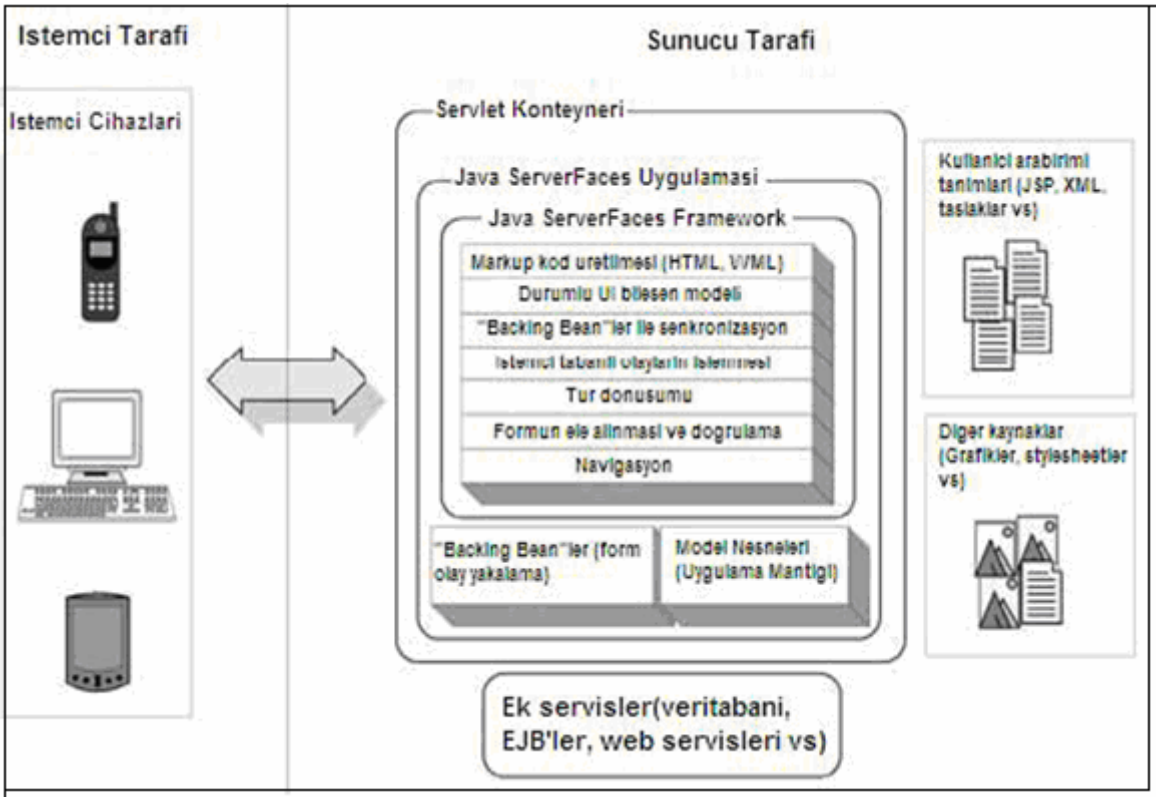
1.5.2. JSF (Java Server Faces) ve Richfaces

Java Server Faces güçlü ve dinamik web uygulamaları geliştirmeyi kolaylaştıran web tabanlı ara yüzler hazırlamak için bir framework'tür . JSF *swing* gibi bir takim standart bileşenler (butonlar, linkler, secim kutuları) ya da özelleştirilmiş bileşenler oluşturmak için model sunmasının yanında istemci taraflı olayların (event) işlenmesi için yöntemler sunar ve ayrıca araç(tool) kullanma desteği vardır. Java için birçok web ara yüz frameworkler'i bulunmasına karşın Java Server Faces birçok sebepten dolayı öne çıkmaktadır. JSF "java community" standardıdır. JSF, piyasadaki birçok framework'ün iyi yönleri alınarak tasarlanmıştır. Şekil 1.9.'de bu durum şematize edilmiştir.

Richfaces, ajax desteğine sahip oldukça gelişmiş bir JSF kütüphanesidir. Yazılımcılar tarafından hazır olarak kullanılabilecek, ajax desteğine sahip oldukça fazla sayıda bileşene (component) sahiptir. JSP-java nesneleri arası iletişim, sayfalar arası gezinme, hazır temalar ile tüm bileşenlerin görüntüsünün değiştirilebilmesi, doğrulama

(validation), nesneler arası dönüşüm (convertion) gibi hazır olarak kullanılabilecek özellikleri mevcuttur [26,27].

Richfaces deki ajax desteği standart yapıdan biraz farklıdır. Bileşenler üzerinde olaylar tanımlanır [23]. Bu olaylar server tarafında çalıştırılır. Kod çalıştırıldıktan sonra jsp sayfa üzerinde istenilen kısımlar yenilenir (render) ve bu işlemler ajax teknololisi ile yapılır [24]. Şekil 1.8. JSF kullanımına genel bir örnek gösterilmiştir.



Şekil 1.9. JSF kullanımına genel bir örnek

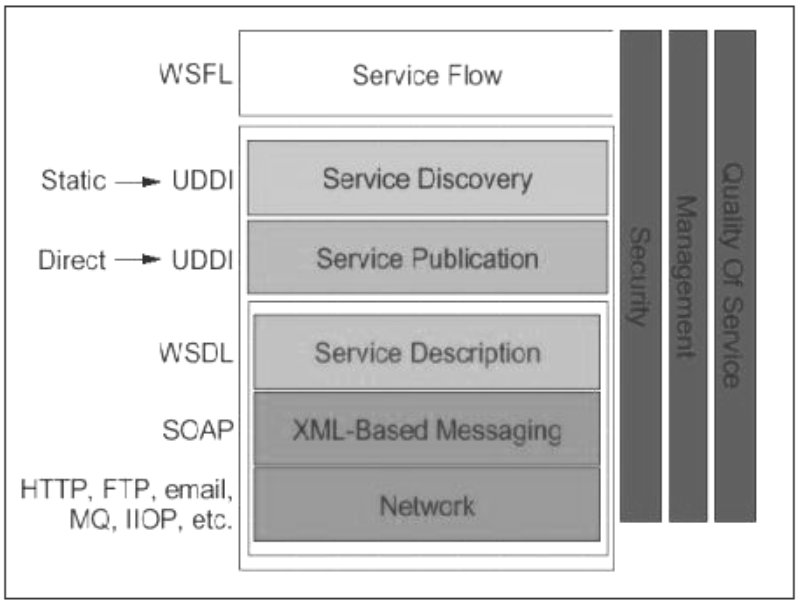
1.5.3. Web Servisleri

Web servislerinin ortaya çıkmasıyla internet ve web teknolojileri için yeni bir devrim başlatmıştır. Web servisleri web ortamında yayınlanabilen, aranıp bulunabilen ve çağrılarak erişilebilen modüler uygulama fonksiyonlarıdır. Bu fonksiyonlar değişik kurumsal iş süreçlerini gerçekleştirebilme özelliğine sahiptir. Web servisleri Haziran 2000'de ortaya çıkan bir çok yazılım firması tarafından yoğun bir destek bulan bir modeldir. Web servisleri açık internet standartlarına dayanır. Henüz gelişme ve olgunlaşma

aşamasında olan bu modelle ilgili olarak bu aşamada ortaya çıkan ve kullanılan çekirdek standartları SOAP, WSDL ve UDDI'dır.

Web servisleri modelini destekleyen Microsoft, IBM, Sun, HP, Oracle ve daha bir çok firma bu konuda yoğun bir şekilde çalışmakta ve web servisleri yazılım ve uygulama geliştirme araçlarını geliştiricilere sunmaktadırlar.

Şekil 1.10.'de web servisi mimarisindeki temel katmanlar gösterilmiştir. Bu katmanlarda belirtilen güvenlik, iş akışı, servis kalitesi ve yönetim gibi konulardaki web servisi standartları henüz araştırma aşamasındadır. Bunların yanında bir takım temel çekirdek standartlar oluşmaya başlamıştır [20].



Şekil 1.10. Web servisi mimarisi katmanları

1.5.4. WSDL (Web Services Description Language)

Çalışmamızda, Unix sunucu üzerinde koşan web servisi WSDL dili kullanarak geliştirilmiştir. Bir uygulamanın bir web servisini kullanması için web servisinin nasıl çağırılacağının, ara yüzünün, hangi protokollerin ve kodlama standartlarının belirtilmesi gerekir. WSDL web servisini tanımlayan bir XML belgesidir. Bir anlamda dağıtık programlamada kullanılan IDL'e (Interface Definition Language – Arayüz Tanımlama Dili) benzer. Web servisi tanımı işlemler, giren ve çıkan mesaj formatları, ağ ve port adresleri gibi bilgileri tanımlar. Şekil 1.10.'den de görüldüğü gibi genel mimaride WSDL'ın yeri servis tanımlamalarının (Service Description) olduğu katmandır [18,19].

2. YAPILAN ÇALIŞMALAR

2.1. Giriş

Bu çalışmada, sistem çağrılarının herhangi bir web ara yüzünden koşularak izlenmesi, çağrıların çalışma mantığının grafiksel olarak gösterilmesi ve kullanıcılara öğretilmesi için bir uygulama geliştirilmiştir.

2.2. UNIX Sunucu Üzerinde Çağrı İzleme Programının Geliştirilmesi

2.2.1. Tracer Programının Yazılması

Tracer programı Unix ortamında C dili fonksiyonları kullanılarak geliştirilmiştir. Temelde çağrıların izlenmesi ve sonuçların web servisine aktarılması işlemi Tracer ile gerçeklenmiştir. Tracer C fonksiyonlarını içerdiği için sistem çağrısı arayüzünü hem *do*ğrudan hem de kütüphaneler aracılığıyla kullanabilmektedir. Program, shell (kabuk) ekranından herhangi bir komutu veya çalışan sürecin PID'sini argüman olarak alır. Kullanım prosedürü aşağıdaki *usage* fonksiyonu ile sağlanmaktadır.

```
void usage(void)
{
   printf("%s\n%s\n",
          "usage: Tracer –p <pid>",
          "       Tracer <command> [args]");
  exit(1);
}
```

Çalışmamızda Tracer'ın ikinci kullanımı olan komut girişi ile sistem çağrılarının izlenmesi gerçekleştirilmiştir.

initTraceFlags prosedürü gözetlenen süreç tarafından yapılan hangi sistem çağrılarının işletim sistemi çekirdeğine girişte ve çıkışta durdurulmasının istendiğini işletim sistemine bildirmek için kullanılır. Prosedür içinde sistemde mevcut bütün çağrıların çekirdeğe girişte ve çıkışta durdurulması istenmiştir. İzlenecek sistem çağrıları

syscalls dizisinden alınmaktadır. Aşağıda *initTraceFlags* prosedürünün icrası gösterilmektedir.

```
void initTraceFlags(int proc_fd) {
    int sysnum;
    sysset_t traced_calls;
    premptyset (&traced_calls);
    for (sysnum = 0; sysnum < 254; sysnum++)
    praddset (&traced_calls, sysnum); /* Tüm sistem çağrılarının izlenmesi */
    ioctl(proc_fd, PIOCSENTRY, &traced_calls);
    ioctl(proc_fd, PIOCSEXIT, &traced_calls);
}
```

Komut satırından girilen komut gözetlenen süreç tarafından icra edilmeye başlanır. İcranın başlangıcı gözetleyen sürecin gerekli bildirimleri işletim sistemine yapmasından hemen sonra olmalıdır. Girilen komutun icrası yine başka bir sistem çağrısı olan *execvp(...)* ile başlatılmaktadır.

```
 char **command;
void handler_cmd() {
  execvp (command[0], command);
}
```

Gözetleyen süreç (Tracer) icraya başlar ve komut satırından girilen veriyi inceleyerek icra edilecek komutu ya da çalışmakta olan sürecin PID'sini belirler. Tracer programına herhangi bir argüman geçilmemiş ise programın kullanımı ile ilgili bilgi gösterir.

```
main (int argc, char *argv[]) {
    char pid_s[15], sysname[50];
    prstatus_t status;
    struct syscall *sc;
    int c, i, fd, sysnum, pid=0, child_pid=0;
    while ((c = getopt(argc, argv, "p:o")) != -1) {
```

```
switch (c) {
case 'p': /* Specified pid */
pid = atoi(optarg); /* optarg = 1234 for "a.out -p 1234". */
break;
default:
usage();
}
}
argc -= optind; argv += optind;
if ((pid == 0 && argc == 0) || (pid != 0 && argc != 0))
usage();
```

Bir sonraki adımda gözetlenen süreç oluşturulur ve bu süreç içinde, gözetleyen süreç (parent process) sistem çağrısı bildirimlerini yapıncaya kadar beklenir. Bildirimlerin yapıldığı gözetleyen süreçten alınan SIGCHLD sinyali ile anlaşılır ve *handle_cmd* prosedürü ile komut icra edilir.

```
if (pid == 0) {
child_pid = 1;
if ((pid = fork()) == 0) {
command = argv;
signal(SIGCHLD, handler_cmd);
for(;;) ;
}
}
```

Komut satırından PID'si girilen ya da girilen bir komutu icra etmek için oluşturulan süreç ile ilgili olarak işletim sistemine bildirimlerde bulunulur. Bu bildirimlerden sonra, gözetlenecek süreç yeni oluşturulmuş ise ona SIGCHLD sinyali gönderilerek icrasını başlatması istenir.

```
sprintf (pid_s, "/proc/%d", pid);
fd = open (pid_s, O_RDWR | O_EXCL);
```

```
initTraceFlags(fd);
if (child_pid)
kill(pid, SIGCHLD);
```

Bir süreç durma eylemi ile karşılaşıldığı zaman yapılan sistem çağrısının adı, argümanları ya da geri dönüş değerleri gösterilir ve sürecin icrası durduğu noktadan itibaren yeniden başlatılır.

```
while (1) {
ioctl (fd, PIOCWSTOP, &status);
sysnum = status.pr_syscall;
sprintf(sysname, "%s", syscallnames[sysnum]);
sc = get_syscall(sysname);
if (sc) {
if (status.pr_why == PR_SYSENTRY) {
printf("%s(", sc->name);
for (i = 0; i < sc->nargs; i++)
printf("%s, ", get_arg(fd, sc->args_type[i], status.pr_sysarg[i]));
if (sc->nargs > 0)
printf("\b\b");
printf(")");
if (status.pr_what == SYS_exit)
break;
}
else {
if (sc->ret_type == Ptr)
printf(" = 0x%x\n", status.pr_reg[R_R0]);
else
printf(" = %d\n", status.pr_reg[R_R0]);
}
}
ioctl (fd, PIOCRUN, 0);
}
ioctl (fd, PIOCRUN, 0);
```

```
}
```

Bu programda kullanılan *syscallnames* dizisinin elemanları sistem çağrılarının isimlerinden oluşur. Bu dizi içerisinde her bir elemanın bulunduğu index numarası, Unix sistemi tarafından o sistem çağrısına tahsis edilen numaraya karşılık gelir. Örneğin, *open* sistem çağrısı 5 sayısı ile temsil edilir.

```
char *syscallnames[] = {
"syscall", "exit", "fork", "read", "write", "open", "close", "wait",
"creat", "link", "unlink", "exec", "chdir", "time", "mknod", "chmod",
"chown", "brk", "stat", "lseek", "getpid", "mount", ... }
```

get_syscall ve *print_arg* kullanıcı tanımlı fonksiyonlardır. get_syscall, *sysname* ile belirtilen sistem çağrısının yapısını aşağıda birkaç elemanı verilen *syscalls* dizisinden alır. Sistem çağrılarının yapısı birbirinden farklı olduğundan dolayı, ilgilenilen her bir sistem çağrısı ile ilgili bilgi bu dizi içinde tutulur. *print_arg* fonksiyonu ise sistem çağrısının argüman değerlerini öğrenir.

```
struct syscall
syscalls[] = {
{ "mmap", Ptr, 6, { Hex, Int, Hex, Hex, Int, Quad }},
{ "open64", Int, 3, { String, Int, Octal }},
{ "open", Int, 3, { String, Int, Octal }},
{ "execve", Int, 3, { String, Hex, Hex }},
{ "ioctl", Int, 3, { Int, Ioctl, Hex }},
{ "lseek", Int, 3, { Int, Quad, Int }},
{ "link", Int, 2, { String, String }},
{ "rename", Int, 2, { String, String }},
{ "read", Int, 3, { Int, Ptr, Int }},
{ "write", Int, 3, { Int, Ptr, Int }},
{ "stat", Int, 2, { String, Ptr }},
{ "creat", Int, 2, { String, Hex }},
{ "chmod", Int, 2, { String, Hex }},
```

```
{ "unlink", Int, 1, { String }}
{ "chdir", Int, 1, { String }},
{ "close", Int, 1, { Int }},
{ "exit", None, 1, { Int }},
{ "fork", Int, 0, { }},
{ 0, 0, 0, { 0 }}
};
```

Bu dizinin her bir elemanı ilgili sistem çağrısının ismini, geri dönüş değerini, argümanlarının sayısını ve bu argümanların (types veri tipi tanımlanan) tiplerini içerir. Örneğin, *open* çağrısının geri döndürdüğü değer *Int* tipinden, argüman sayısı 3 ve argümanlarının tipleri sırasıyla String, Int, Octal'dır. Dolayısıyla yukarıda verilen dizinin elemanları aşağıdaki veri tipine sahiptir.

```
typedef enum {None=1, Hex, Octal, Int, String, Ptr, Stat, Ioctl, Quad} types;
struct syscall {
char *name; /* çağrının ismi */
types ret_type; /* geri dönüş değeri */
int nargs; /* argümanların sayısı */
types args_type[10]; /* herbir argümanın tipi */
};
```

Yukarıda geliştirilen program *truss*'dan biraz farklı çalışır. Dinamik olarak oluşturulan süreçler Tracer programı tarafından gözetlenmezler. Bu çeşit süreçlerin gözetlenmesi ya her biri için yeni bir gözetleyici süreç üreterek ya da poll sistem çağrısı kullanılarak tek bir gözetleyici süreç ile yapılabilir.

```
int waitForAProcessToStopOrDie(struct pollfd *pollfds, int n_fds, int *pindex)
{
int i, n;
while (1) {
n = poll(pollfds, n_fds, INFTIM);
if (n > 0) {
```

```
/* Check for an event of interest (STOPPED, DIED) */
for (i = 0; i < n_fds; i++) {
if (pollfds[i].revents & POLLPRI) {
*pindex = i;
return 0;
}
else if (pollfds[i].revents & POLLHUP) {
*pindex = i;
return 1;
} } } }
return -1;
}
```

Yukarıdaki prosedürde *poll* çağrısı bir süreç durdurulduğu zaman geri döner ve sonra hangi sürecin niçin durduğu belirlenir.

2.2.2. Dinamik Oluşturulan Süreçlerin Koştuğu Sistem Çağrılarını Truss Sistem Çağrısı ile İzleme

Bu kısımda çağrıların izlenmesi için sistemin bize sunmuş olduğu sistem çağrılarından faydalanılmaktadır. Farklı Unix türevlerinde çeşitli izleme çağrıları olmakla beraber, bizim tercih ettiğimiz çağrı *truss* komutudur. Program sunucu ve istemci olmak üzere iki kısımdan oluşmaktadır. Amacımız bir süreç tarafından koşulan sistem çağrılarını ve o sürecin oluşturduğu alt süreçlerin koştuğu sistem çağrılarını izlemek olduğu için aşağıdaki bölümde sadece sunucu program yani truss'ın kullanıldığı kısım izah edilmiştir.
İstemci program bir java applet olup sunucu ile haberleşmeyi soketler üzerinden yapmaktadır. Bu sebeple sunucu program istemciden gelecek bağlantıları dinleyeceği bir port (kapı) açarak haberleşmeyi bu şekilde sağlamaktadır. *create_socket(port_num)* fonksiyonu, gelen port numarasına ait bir soket oluşturur ve sırada bekleyebilecek bağlantı sayısı belirler.

```
int create_socket(u_short portnum)
{
   int s;
```

```
 struct sockaddr_in sa;
bzero(&sa, sizeof(struct sockaddr_in));

sa.sin_family = AF_INET;
sa.sin_addr.s_addr = inet_addr("10.0.0.225");
sa.sin_port=htons(portnum);
memset(&(sa.sin_zero), '\0', 8);
 if ((s = socket(AF_INET, SOCK_STREAM,0)) < 0)
 {
    printf("socket failed\n");
    return(-2); }
 if (bind(s, (struct sockaddr *)&sa, sizeof sa) < 0)
 {
    close(s);
    return(-3); }
 if (listen(s,5) < 0)
 {
    perror("listen");
    return(-1);   }
 return(s); }
```

Oluşturulan sokete ait bağlantıların beklendiği yer ise *baglanti_bekle(int s)* fonksiyonudur.

```
int baglanti_bekle(int s) {
   struct sockaddr_in isa ;
   int i;
   int t,true=1;
   int errno = 0;
   i = sizeof (struct sockaddr_in);
   do
   {
      if ((t = accept(s, (struct sockaddr *)&isa, &i)) >= 0)
         break;
```

```
  } while (true);
  return(t);
}
```

Ana süreç tarafından oluşturulan sürece ait olan çocuk süreçlerin haberleşmede kullanılacağı fonksiyon ise *kanal_ac(int bash_in[],int bash_out[],int truss_out[])*'dir.
Bu fonksiyonda süreç haberleşmesinde en etkin rolü oynayan *pipe* sistem çağrısı görülmektedir.

```
void kanal_ac(int bash_in[],int bash_out[],int truss_out[])
 {
     if (pipe(bash_in) < 0)
          {
               printf("Hata oluştu, pipe üretilemedi!\n");
               exit(1); }

     if (pipe(bash_out) < 0)
          {
               printf("Hata oluştu, pipe üretilemedi!\n");
               exit(1);   }

     if (pipe(truss_out) < 0)
          {
               printf("Hata oluştu, pipe üretilemedi!\n");
               exit(1); }
```

Dinlenen soket üzerinden bash komutlarının (izlenmesi istenen sistem çağrısının) alınması *komut_al(char *komut,int msgSock)* fonksiyonu ile sağlanmaktadır.

```
void komut_al(char *komut,int msgSock) {
int sayac=0;
 char tbuf[1];
 for (;;) {
    if (read(msgSock,&tbuf,1) <= 0)
      continue;
    if ((int)tbuf[0] == '\r')
      break;
     komut[sayac++] = tbuf[0];
  }
```

```
++sayac;
 //komut[sayac-1] = '\r';
 komut[sayac]='\0'; }
```

İstemci tarafta truss sonuçları ve girilen komutun sonucu ayrı ekranlarda gösterilmiştir. Bash (Unix komut yorumlayıcı) ekran sonuçları ve truss sonuçları için iki ayrı süreç oluşturulmuştur. Bu iki süreç tüm izleme işlemini yapan ana sürecin çocuğu tarafından oluşturulur. Komutun icrası *execl()* sistem çağrısı ile yapılmaktadır. Burada dikkat edilmesi gereken nokta, truss'ın komut girerek değil de süreç pid'si verilerek çalıştırılmasıdır. İstenilen sürecin izlenmesi ve dolayısıyla sürece ait sistem çağrılarının süreç pid'leri ile listelenmesi sağlanmıştır. Aşağıdaki *execCommand* fonksiyonu ile truss sonuçları ve bash sonuçları liste olarak döndürülmektedir.

```
void execCommand(int bash_in[],int bash_out[],int truss_out[]) {
    char argTruss[6];
    int bashPid,trussPid;
    if(!(bashPid = fork())) {
       close(0);
       dup2(bash_in[0], 0);
       close(bash_in[0]);
       close(bash_in[1]);
       close(1);
       dup2(bash_out[1],1);
       close(bash_out[1]);
       close(bash_out[0]);
       close(2);
       dup2(1, 2);
       execl("/bin/bash", "bash", NULL);
       exit(1); }
    if(!(trussPid = fork()))
    {
      sprintf(argTruss,"%d",bashPid);
      close(1);
```

```
dup2(truss_out[1],1);
close(truss_out[1]);
close(truss_out[0]);
close(2);
dup2(1, 2);
execl("/bin/truss","truss","-pf",argTruss, 0);
exit(1);}}
```

Server programı içerisinde *Main(...)* fonksiyonu altındaki temel işlevleri bu şekilde özetlemiş olduk.

Farklı Unix versiyonlarında çeşitli çağrı izleme mekanizmaları mevcuttur. *Strace* ve *Ptrace* en sık kullanılan çağrı izleme komutları olarak karşımıza çıkmaktadır. Bu komutların truss komutuna göre en büyük dezavantajı derinine izleme yapamamalarıdır. Bunlarla sadece ana süreç altında koşulan sistem çağrıları izlenebilmektedir. Tablo 2.1'de farklı platformlar için kullanılan izleme çağrıları gösterilmiştir.

Tablo 2.1. Farklı Platformlar İçin Çağrı İzleme Komutları

İşletim Sistemi Versiyonu	**İzleme Çağrıları**

Sun Solaris 2.x, Unixware 7.0	Truss :
	Unixware 7.0
	$ truss -aefo <output file> <executable>
	Solaris
	$ truss -rall -wall -p <UNIX pid>

HP/UX 11	Tusc : $ tusc -afpo <output file> <pid> <executable>
IBM AIX 4.x	Sctrace : $ sctrace -Amo <output file> <executable>

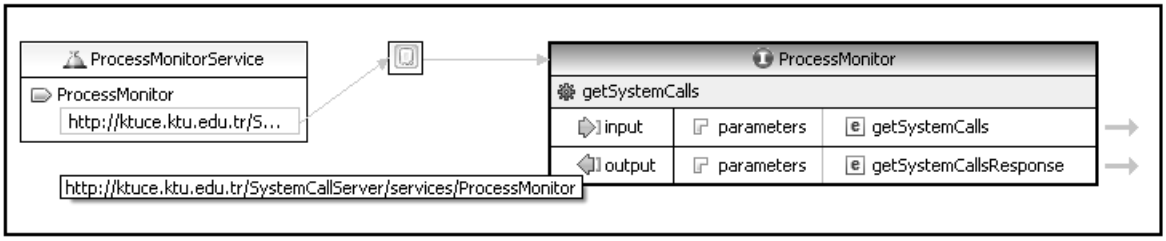
Linux	Strace :
	$ strace -fo <output file> <executable>
	$ strace -p <UNIX pid>

SGI IRIX 6.x	par, e.g.: $ par -siSSo <output file> <executable>
Compaq Tru64 Unix	Trace : $ trace -fo <output file> <executable>
Sequent Dynix/PTX	Truss : $ truss -aefo <output file> <executable>

2.2.3. Unix Sunucu Üzerinde Web Servisinin Yazılması

Çalışmamızda hedeflediğimiz önemli noktalardan birisi de çağrı izleme işleminin Unix ortamdan bağımsız hale getirilmesidir. Kullanıcılar izlenecek sistem çağrıları için illaki bir Unix makineye, shell (kabuk) ekranına ihtiyaç duymamalıdır. İşte bu noktada bizlere web servisi yardımcı olmaktadır.

XML Web Servisleri platform ve programlama dilinden bağımsız veri ve nesne paylaşabilmemizi sağlayan teknolojidir. Bunu yaparken XML dilini kullanır ve XML'i anlayan herhangi bir veritabanı, programlama dili veya platform ile çalışabilir. Bu çalışmada, Eclipse Ganymade J2EE editör, Web 2.5 dağıtım tanımlayıcısı (deployment descriptor) ve Apache 5.5 web sunucusu kullanılmıştır. Uygulamamızın web servisini nasıl çağıracağı, hangi protokolleri kullanacağı WSDL (Web Services Description Language) dili ile belirlenmiştir. Unix sunucu üzerindeki web servisini içeren uygulamamızın adı *SystemCallServer*'dır. Bu kapsamda yazdığımız wsdl servisinin içerdiği fonksiyonlar aşağıda belirtilmiş, fonksiyonlar arası ilişki Şekil2.1.'de gösterilmiştir.



Şekil 2.1. WSDL servisinin yapısı

İstemci tarafta izlenilmesi istenen sistem çağrısı, Unix sunucu programa *http://ktuce.ktu.edu.tr/SystemCallServer/services/ProcessMonitor* adresinden web servisi aracılığıyla iletilir. Gelen istekler *ProcessMonitorService* tarafından yönlendirilir. *SystemCallServer* içinde tanımlanan *edu.ktu.ktuce.ProcessMonitor* sınıfında kullanılacak metodlar tanımlanmıştır. Uygun sınıf altındaki metodlara yönlendirme işlemlerinin tümü *axis.jar* library dosyası içinde gerçekleştirilir. Şekil 2.1.'den de görüldüğü gibi çağrının icrası ve sonuçlarının alınması *ProcessMonitor* sınıfı altında *getSystemCalls* metodu ile sağlanmaktadır. Fonksiyon giriş parametresi olarak istemci taraftan girilen komutu, dönüş parametresi olarak *SystemCall* dizisi alacak şekilde tasarlanmıştır.

Aşağıda tanımlanan sınıf ve kullanılan metodlar gösterilmiştir.

```
public class ProcessMonitor  implements Serializable{...}
public SystemCall[] getSystemCalls(String komut)  throws Exception {...}
```

SystemCall sınıfından üretilen ve *getSystemCalls* fonksiyonun dönüş parametresi olan *SystemCall* nesnesi, çağrıyı icra eden sürecin pid'sini, sistem çağrısını, çağrının çalışma parametrelerini içeren değişkenleri ve bu değişken değerlerini döndüren fonksiyonları içermektedir.

```
public class SystemCall implements Serializable
{   public SystemCall() {}
   public String getParameter()
   {
      return parameter; }
   public void setParameter(String parameter)
   {
      this.parameter = parameter; }
   public String getProcessId()
   {
      return processId;  }
   public void setProcessId(String processId)
   {
      this.processId = processId;  }
   public String getKomut()
   {
      return komut; }
   public void setKomut(String komut)
   {
      this.komut = komut;  }
   public Long getId()
   {
      return id; }
```

```
public void setId(Long id)
{  this.id = id;  }
private static final long serialVersionUID = 1L;
private Long id;
private String processId;
private String komut;
private String parameter; }
```

Sistem çağrılarının izlenmesine yönelik iki programdan önceden bahsetmiştik. Java ortamı derlenmiş C programlarını çalıştırabilme özelliğine sahip olduğu için, kullanımda Tracer programı tercih edilmiştir. Çağrıların izlenmesi Unix sunucu üzerinde Tracer programıyla yapılmış olup *getSystemCalls* metodu içerisinde nasıl kullanıldığı aşağıda gösterilmiştir.

```
process = Runtime.getRuntime().exec(argv);
BufferedReaderbufferedReader=newBufferedReader(new
InputStreamReader(process.getErrorStream()));
List systemCalls = new ArrayList();
String str;
    while((str = bufferedReader.readLine()) != null)
{
String list[] = str.split(":");
SystemCall systemCall = new SystemCall();
systemCall.setProcessId(list[0]);
String processData = list[1];
int startIndex = processData.indexOf("(");
int endIndex = processData.indexOf(")");
systemCall.setKomut(processData.substring(0, startIndex));
String parameter = processData.substring(startIndex + 1, endIndex);
systemCall.setParameter(parameter);
systemCalls.add(systemCall); }
```

Yukarıdaki kod kısmında *arg* parametresi ile girilen komutun icrası *Runtime.getRuntime().exec(argv)* fonksiyonu tarafından gerçekleştirilmektedir.
exec() fonksiyonu ile shell ekranından girilen komutların koşulması, derlenmiş herhangi bir programın çalıştırılması java ortamından mümkün olmaktadır.

2.2.4. Web Ara Yüzünün Hazırlanması

İstemci tarafında geliştirdiğimiz web uygulamasının temel amacı, izlenmesi istenen sistem çağrısının davranışını modelleyerek kullanıcıya sistem çağrılarının kullanımı hakkında genel bir bilgi vermek ve süreçlerin çağrıları nasıl kullandığını öğretmektir. Kullanıcının, çağrıların ne işe yaradığını daha basitçe anlayabilmesi amacıyla, çağrıları Tablo 2.2.'de gösterildiği gibi belirli kategorilere ayırdık.

Tablo 2.2. Sistem çağrılarının sınıflandırılması

Süreç Yönetimi	Bellek Yönetimi	Dosya Sistemi	Diğer
exec, fork, vfork, pipe, dup, dup2, exit, signal, ptrace, setuid, kill, poll	mmap, brk, swaporı, swapoff, mlock, sync, mprotect,	open, creat, read, write, link, symlink, unlink, chdir, rmdir, chmod, rename, chown, mknod, mount, mkdir, truncatc, flock, close, lseek, fcntl, readdir, stat	ioctl,time,sync, plock

Başlangıç olarak basit sistem çağrılarının koşulmasını inceleyelim. Aşağıdaki örnekte *date* sistem çağrısı programla izlendiği zaman kullanıcı ekranına dönen sistem çağrılarını, bu çağrıların sahibi süreçleri ve çağrı sınıflandırmaları gösterilmiştir. Tablo 2.3.'de gösterildiği gibi tüm izleme işlemini pid'leri 10450 ve 10451 olan iki süreç üstlenmiştir. 10451 pid'li süreç10450 sürecinin bir alt sürecidir. Çocuk süreç olarak nitelendirilen bu alt süreçler dinamik oluşturulduğu için tüm izleme mekanizmaları tarafından analiz edilemezler. Çalışmamızda çağrı izleme işlemi boyunca üretilen tüm süreçlerin izlenmesi *Tracer* programıyla mümkün olmaktadır. Burada şunu vurgulamamız gerekir ki, ister tek bir sistem çağrısını ister birden fazla çağrıyı pipe (boru)'larla paralel koşalım, en az bir süreç izleme işlemini devralacaktır. Çocuk süreçlerin koşulan çağrının icrası esnasında ana

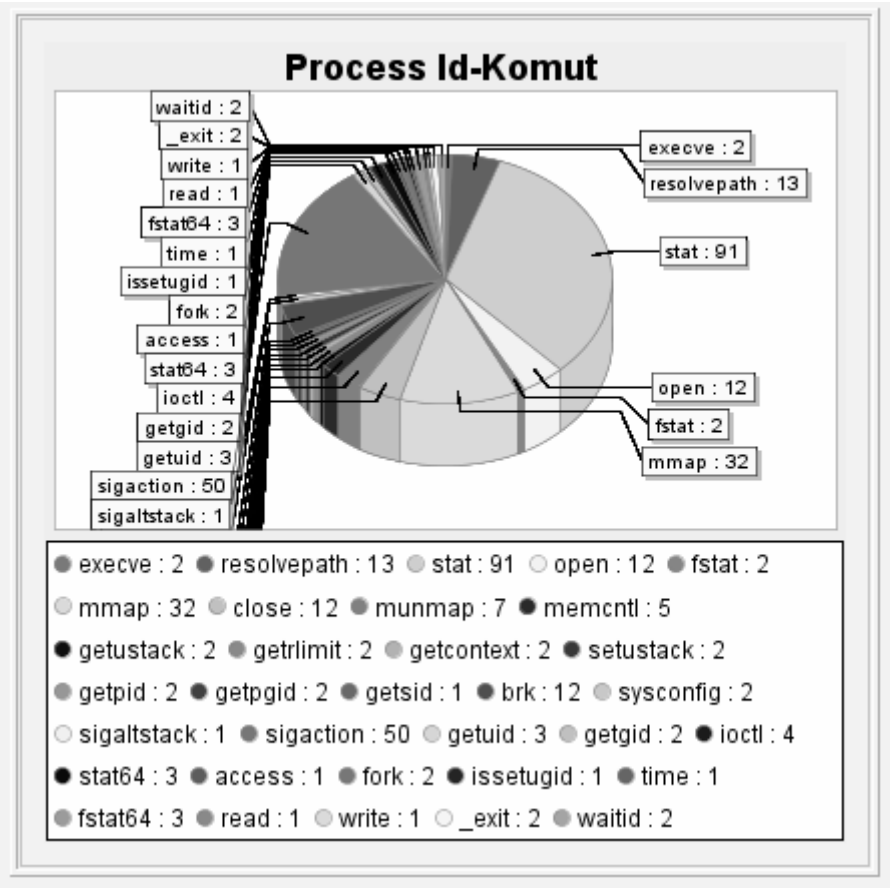
süreç tarafından üretilmesi beklenir ve sayısı hakkında net bir bilgi verilemez. Pipe ile paralel koşulan çağrılarda ise en az çağrı sayısı kadar süreç olması gerekir.

Tablo 2.3. Date sistem çağrısının koşulması sırasında koşan bazı sistem çağrıları

PID	Sistem Çağrısı	Çağrı Kategorisi	Çağrı Sonucu
10450	Execve("/usr/bin/sh", 0xFFBFFB8C, 0xFFBFFB9C)	Süreç Yönetimi	argc = 1
10450	Resolvepath("/usr/lib/ld.so.1","/usr/lib/ld.so.1", 1023)	Dosya Sistemi	16
10450	stat("/usr/bin/date", 0xFFBFF858)	Bellek Yönetimi	0
10450	mmap(0x00000000, 112, PROT_READ, MAP_SHARED, 3, 0)	Bellek Yönetimi	0xFF3A0000
10450	close(3)	Dosya Sistemi	0
10450	brk(0x00024508)	Bellek Yönetimi	0
10450	access("/usr/bin/date", 9)	Dosya Sistemi	0
10450	Fork()	Süreç Yönetimi	10451
10451	Getpid()	Süreç Yönetimi	10451[10450]
10451	execve("/usr/bin/date", 0x0003A5FC, 0x0003A758)	Süreç Yönetimi	argc = 1
10451	open("/usr/share/lib/zoneinfo/Turkey", O_RDONLY)	Dosya Yönetimi	3
10451	read(3, " T Z i f\0\0\0\0\0\0\0".., 993)	Dosya Yönetimi	993
10451	write(1, " S u n F e b 8 1".., 29)	Dosya Yönetimi	29
10451	exit(0)	Süreç Yönetimi	
10450	waitid(P_PID, 10473, 0xFFBFF7B0)	Süreç Yönetimi	0

10451 pid numaralı süreci oluşturan sistem çağrısı, süreç yönetiminin en temel çağrısı olan *fork*'tur. Diğer çağrılar çok farklı özelliklere sahip olmakla birlikte, çok çeşitli parametreler almaktadır. Tablodan da görüldüğü üzere bir çağrının icrası boyunca genel

olarak süreç, bellek, dosya yönetim çağrıları icra edilmektedir. Fakat farklı çağrıların koşulmasıyla değişik sistem çağrıları da kategorilere dahil edilebilir, kategori veritabanı genişletilebilir. Program ekranına genel bilgi vermek amacıyla, çağrıların kullanım istatistikleri de eklenmiştir. Şekil 2.2.'de örnek *date* sistem çağrısının izlenmesi sonucu iki süreç tarafında toplam icra edilen sistem çağrıları gösterilmiştir.



Şekil 2.2. Date çağrısının icrası esnasında koşulan sistem çağrılarının grafiği

Programda farklı komutlar izlendiği zaman web sayfasının tamamı güncellenmemekte, sadece tabloların ve grafiklerin gösterildiği alanlar yenilenmektedir. Bunun bize sağladığı en büyük avantaj ise performanstır. Sayfada belli bölgelerin yenilenmesi işlemi AJAX teknolojisi ile sağlanmıştır. *Richfaces* kütüphanesini kullanarak geliştirdiğimiz sayfada, herhangi bir bileşenin (butonlar, paneller, v.s.) tetiklenmesinden sonra, bileşenin *reRender* özelliğine atanan diğer bileşenler güncellenir. Örneğin ekranımızda *Çalıştır* butonuna basıldıktan sonra panellerin, grafiklerin, tabloların

güncellenmesi tanımlanmıştır. Benzer tanımlamalar diğer bileşenler içinde kodlara yazılmıştır. Aşağıdaki kod bloğunda buton aktivitesinin nasıl icra edildiği gösterilmiştir.

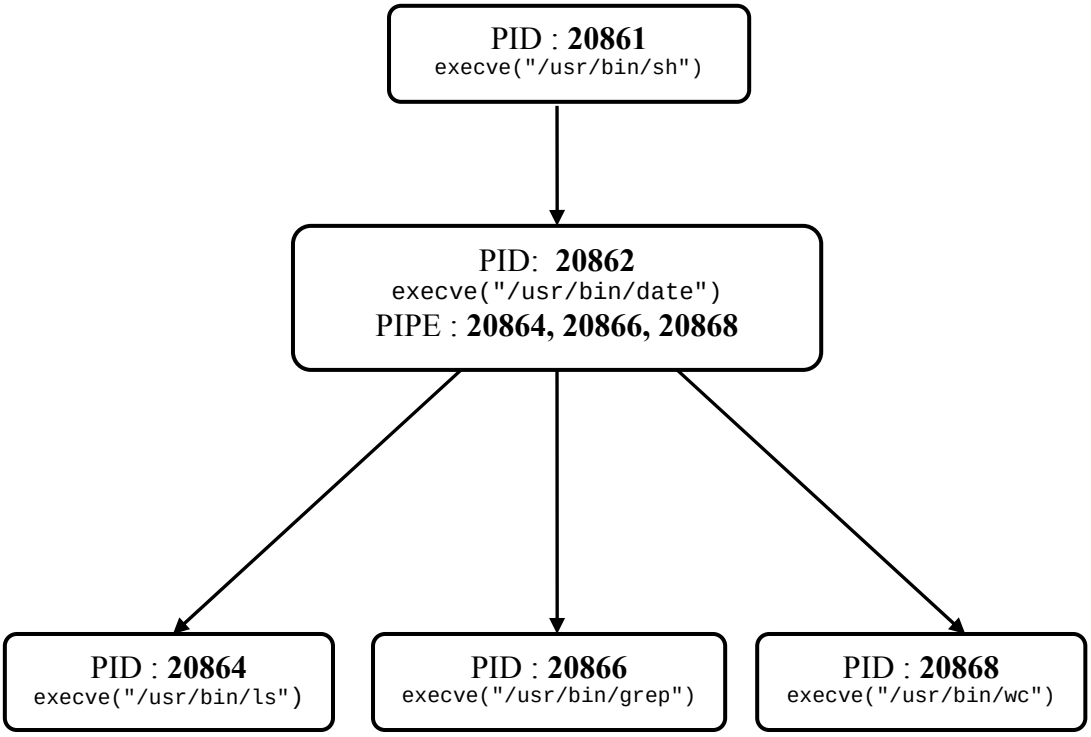
```
<rich:panel style="width=400">
<f:facet name="header">
   Komut
</f:facet>
<h:outputText value="Komut"/>
<h:inputText value="#{proxyBean.systemCall}" required="true">

</h:inputText>
      <a4j:commandButton
            action="#{proxyBean.getSystemCallsInformation}"
            styleClass="btnH1W1" value="Calistir"
            reRender="graphicPanel,table, ,chartList,tree,treePanel" />
      </rich:panel>
```

Yukarıdaki kod bloğunda görüldüğü gibi butonun yapacağı işlem *action* özelliğiyle tanımlanmıştır. İlk olarak *proxyBean* sınıfı içindeki *getSystemCallsInformation* metodu ile çağrı hakkında tüm bilgiler okunur. Daha sonra *reRender* özelliğinde belirtilen *graphicPanel,table, ,chartList,tree,treePanel* gibi bileşenler güncellenir. Bu tarz güncelleme özellikleri bize *richfaces* içerisinde gelen *A4J* kütüphanesi ile sunulmuştur.

Çalışmamızın temel amacı unix sistem çağrılarının, süreçlerin nasıl çalıştığını grafiksel olarak modelleyebilmek ve kullanıcılara öğretmek olduğu için, programda mümkün olduğu kadar farklı şekillerle bir ağaç yapısı çizdirilmiştir. Bir çağrının yaşam döngüsü ağaç yapısı şeklinde, çağrıyı ilk izleyen süreçten son sürece kadar tanımlanmaya ve ekranda resmedilmeye çalışılmıştır.

2.2.5. Süreç İşleyişinin Grafiksel Aktarılması

Projemizin temel amaçlarından birisi kullanıcılara sistem çağrılarının işleyişini süreçler bazında grafiksel olarak göstermektir. Şekil 2.3.'de "ls –l | grep myfork | wc | date" komutunun izlenmesi gösterilmiştir.



Şekil 2.3. "ls –l | grep myfork | wc | date" komutunun izlenmesi

Yukarıdaki komut sonuçlarına göre sistemde 5 adet süreç oluşmuştur. PID : 20861 numaralı süreç kabuk yorumlayıcı (shell)'ya ait olup temel izleme işlemini başlatan süreçtir. Komuta bakılırsa farklı sistem çağrılarının pipe yapılarak koşulduğu gözükmektedir. Bu da girilen her pipe komutu kadar sistemin pipe() çağrısını üretmesi anlamına gelir. "ls- l | grep myfork" komutunda pipe'ın sol tarafının ürettiği çıktılar, pipe'ın sağ tarafına girdi olarak aktarılır, benzer mantık diğer pipe'lar için de geçerlidir. Bu şekilde tüm komut sonuçları son komut olan date çağrısına pipe'lar ile aktarılır. 20862 numaralı süreç, oluşturduğu çocuk süreçlerle haberleşmek için 3 ayrı pipe() komutu koşmuştur. Koşulan her bir sistem çağrısı için yeni bir süreç oluşturulmuştur. Kullanılan pipe kadar da pipe() komutu icra edilmiştir.

3. SONUÇLAR

Bu çalışmamızda UNIX işletim sisteminde kullanılan sistem çağrılarının, web ortamından süreçler bazında izlenmesi, çağrıların çalışma mantığının grafiksel olarak gösterilmesi ve kullanıcılara öğretilmesi amacıyla bir web projesi oluşturulmuştur. Bunun için çeşitli web teknolojilerinden yararlanılmıştır. Özellikle UNIX işletim sistemiyle ilgilenen, sistem programlamayla uğraşan, çekirdek ve işletim sisteminin davranışlarını araştıran kullanıcılar için yaralı olacağını düşündüğümüz bir proje hazırlanmıştır. Çalışmamız istemci-sunucu mimarisi üzerine kurulmuş bir yapıdan oluşmuştur. İstemci tarafta kullanıcının izlemek istediği komutların girişini yapacağı ve sonuçları göreceği ekran, sunucu tarafta ise çağrı izleme işlemini gerçekleştiren program ve servisler yer almıştır.

Tasarlanan arayüzlerle sistem çağrıları anlamsal olarak disk,bellek,süreç, diğer çağrılar şeklinde şematize olarak gösterilmiştir. Bir çağrının yaşam döngüsü kullanıcıya komutlardan ziyade daha rahat anlayabileceği şekilde aktarılmış ve çağrıların öğretilmesi amacı gerçekleştirilmiştir.

4. ÖNERİLER

Web ortamından sistem çağrılarının izlenmesi ve grafiksel yorumlatılması performans olarak çok iyi netice vermemektedir. Çünkü koşulan çağrısının ürettiği süreç sayısına ve süreçler altındaki çağrı sayısına göre ekrana bastırılan grafik çeşidi farklılık göstermektedir. Performans açısından lokasyon bazlı ekran güncellemesi yapılsa da çağrı sonuçlarının yenilenmesi belirli bir zaman almaktadır. Çalışmamızda, günümüzde en yaygın kullanılan java kütüphanelerinden birisi olan *Richfaces* kullanılsa da daha hızlı arayüzler sağlayan diğer kütüphaneler tercih edilebilir.

Çağrıların izlenmesi işlemi, unix sunucu tarafında geliştirdiğimiz C programıyla yapılmaktadır. İzleme işlemi daha hızlı çalışan *Truss* gibi standart sistem çağrılarıyla da gerçeklenebilir.

Süreçler bazında gruplanan sistem çağrılarının yaptığı işlevler, veritabanında tuttuğumuz belirli kategorilere göre grafiklerle yorumlanmıştır. Süreç davranışlarının yorumlanması için daha detaylı şemalar gösterilebilir, görsel öğeler arttırılabilir.

Amacımız öncelikli olarak web ortamından izleme işleminin sağlanması olduğu için ekran tasarımını çok zengin yapamadık. Bu noktada daha farklı tasarımlar tercih edilebilir. Özellikle kullanıcılara öğretme işlevinin, daha efektif aktarılabilmesi için çalışmalar yapılabilir.

5. KAYNAKLAR

1. Bach, M.J., The Design of the UNIX Operating System, Prentice-Hall, New Jersey, 1986.

2. Çelik, S., UNIX Sistemleri için Dosya Sistemi Erişimlerinin Güvenilir Hale Getirilmesi, Yüksek Lisans Tezi, KTU, Fen Bilimleri Enstitüsü, Trabzon, 2004.

3. Marshall, A. D., Programming in C UNIX System Calls and Subroutines using C, Wiley Publishing, New Jersey, 1994.

4. Rochkind, M.J. , Advanced UNIX Programming, Prentice-Hall, New Jersey, 1985.

5. Stallman, R., The GNU source-level debugger. Distributed by the Free Software Foundation, Boston, 1989.

6. Faulkner, R. ve Gomes R., The Process file system and process model in UNIX system V., 1991Winter USENIX Conference, 1991, Dallas, Bildiriler Kitabı, 243-252.

7. Spinellis, D., Trace: A Tool for Logging Operating System Call Transactions, Operating Systems Review, 28, 4 (1994) 56-63.

8. Bernaschi, M., Gabrielli, E. ve Mancini, L., Operating System Enhancements to Prevent the Misuse of System Calls, Proc. ACM Conf. Computer and Comm. Security, 2000, Athens, Bildiriler Kitabı, 174-183.

9. Garfinkel, T., Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools, Proc. Network and Distributed Systems Security Symp., 2003, Bildiriler Kitabı, 163-176.

10. Eskin, E., Lee, W. ve 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, 2001, Arizona, Bildiriler Kitabı, 165 -17.

11. Kruegel, C., Mutz, D., Valeur, F., ve Vigna, G., On the Detection of Anomalous System Call Arguments, Proc. 8. European Symp. Research in Computer Security (ESORICS ’03), 2003, Norveç, 326-343,.

12. Sekar, R., Bendre, M., Dhurjati, D., ve Bollineni, P., A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors, Proc. IEEE Symp. Security and Privacy, 2001, California, Bildiriler Kitabı, 144-155.

13. Spinellis, D., A C execution profiler for MS-DOS, http://usenet.group.comp.source, 8, 8. 1989.

14. Holyer, I. ve Pehlivan, H., A recovery mechanism for shells, The Computer Journal, 3, 4 (2000) 1-9.

15. Graham, S. L., Kessler, P. B. and Marshall K. McKusick. An execution profiler for modular programs, Software Practice and Experience, Wiley Publishing, California, 1983.

16. Ignatin, G. R., Let the hackers hack: Allowing the reverse engineering of copyrighted computer programs to achieve compatibility, University of Pennsylvania Law Review, 140 (1992) 1999–2050.

17. Pehlivan, H. ve Tenekeci, M. E., Unix İşletim Sistemleri İçin Akıllı Geri Dönüşüm Kutusu Tasarımı ve Gerçekleştirilmesi, Eleco Ulusal Konferansı, Kasım, 2006, Bursa, Bildiriler Kitabı, 178-182.

18. Zimmermann, O., Tomlinson, M. ve Peuser, S., Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects, Springer, 2005.

19. Newcomer, E., Understanding Web Services: XML, WSDL, SOAP, and UDDI, Independent Technology Guides, 2008.

20. Allen, K., WSDL 100 Success Secrets Essentials of Understanding and Applying Web Services Description Language - THE XML based protocol for information exchange in decentralized and distributed environments, 2002.

21. Holdener, A., Ajax: The Definitive Guide, O'Reilly, 2008.

22. Riordan, R., Head First Ajax, O'Reilly, 2008.

23. Schalk, C., Burns, E. ve Holmes, J., JavaServer Faces: The Complete Reference, Complete Reference Series, 2006.

24. Katz, M., Practical RichFaces, Apress, 2008.

25. Bates, B., Head First Java, 2nd Edition, Oreilly, 2008.

26. Bloch, J., Effective Java (2nd Edition), Sun Java Series, 2008.

27. Flanagan, D., Java In A Nutshell, 5th Edition, Oreilly, 2005.

6. EKLER

Ek 1. ProcessMonitor Sınıfı Kodu

```
public class ProcessMonitor
   implements Serializable
{

   public ProcessMonitor()
   {
   }

   public SystemCall[] getSystemCalls(String komut)
      throws Exception
   {
      String argv[] = new String[2];
      Process process = null;
      argv[0] = “Tracer";
      argv[1] = komut;
      SystemCall resultList[] = (SystemCall[])null;
      try
      {
         process = Runtime.getRuntime().exec(argv);
         BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(process.getErrorStream()));
         List systemCalls = new ArrayList();
         String str;
         while((str = bufferedReader.readLine()) != null)
         {
            String list[] = str.split(":");
            SystemCall systemCall = new SystemCall();
            systemCall.setProcessId(list[0]);
            String processData = list[1];
            int startIndex = processData.indexOf("(");
            int endIndex = processData.indexOf(")");
            systemCall.setKomut(processData.substring(0, startIndex));
            String parameter = processData.substring(startIndex + 1, endIndex);
            systemCall.setParameter(parameter);
            systemCalls.add(systemCall);
         }
         resultList = new SystemCall[systemCalls.size()];
         for(int i = 0; i < systemCalls.size(); i++)
            resultList[i] = (SystemCall)systemCalls.get(i);

      }
      catch(Exception e)
      {
         SystemCall call = new SystemCall();
         call.setId(Long.valueOf(0xf4241L));
```

```
            call.setParameter(e.toString());
            resultList = new SystemCall[1];
            resultList[0] = call;
            e.getStackTrace();
        }
        return resultList;
    }

    private static final long serialVersionUID = 1L;
}
```

Ek 2. SystemCall Sınıfı Kodu

```
public class SystemCall
    implements Serializable
{

    public SystemCall()
    {
    }

    public String getParameter()
    {
        return parameter;
    }

    public void setParameter(String parameter)
    {
        this.parameter = parameter;
    }

    public String getProcessId()
    {
        return processId;
    }

    public void setProcessId(String processId)
    {
        this.processId = processId;
    }

    public String getKomut()
    {
        return komut;
    }

    public void setKomut(String komut)
    {
        this.komut = komut; }
```

```
public Long getId()
{
   return id;
}

public void setId(Long id)
{
   this.id = id;
}

private static final long serialVersionUID = 1L;
private Long id;
private String processId;
private String komut;
private String parameter;
```

Ek 3. Tracer Programından Kodlar

```
main (int argc, char *argv[]) {
  char pid_s[15], sysname[50];
  prstatus_t status;
  struct syscall *sc;
  int c, i, fd, sysnum, pid=0, child_pid=0;

  while ((c = getopt(argc, argv, "p:o")) != -1) {
    switch (c) {
    case 'p':
      pid = atoi(optarg);
      break;
    default:
      usage();
    }
  }

  argc -= optind; argv += optind;
  if ((pid == 0 && argc == 0) || (pid != 0 && argc != 0))
    usage();

  if (pid == 0) {
    child_pid = 1;
    pid = fork();
    if (pid == -1) {
      printf ("fork failed\n");
      exit (1);
    }

    if (pid == 0) {
      command = argv;
```

```
//      signal(SIGCHLD, handler_cmd);
//      for(;;) ;
sleep(1);
handler_cmd();
    }
  }

  sprintf (pid_s, "/proc/%d", pid);
  fd = open (pid_s, O_RDWR | O_EXCL);
  initTraceFlags(fd);
//  if (child_pid)
//    kill(pid, SIGCHLD);

  while (1) {
    ioctl (fd, PIOCWSTOP, &status);
    sysnum = status.pr_syscall;
    sprintf(sysname, "%s", syscallnames[sysnum]);
    sc = get_syscall(sysname);
    if (sc) {
      if (status.pr_why == PR_SYSENTRY) {
        printf("%s(", sc->name);
//        printf("%ld: %s(", status.pr_pid, sc->name);
//        for (i = 0; i < status.pr_nsysarg; i++)
        for (i = 0; i < sc->nargs; i++)
          printf("%s, ", get_arg(fd, sc->args_type[i], status.pr_sysarg[i]));
        if (sc->nargs > 0)
          printf("\b\b");
        printf(")");

        if (status.pr_what == SYS_exit) {
          printf("\n");
          break;
        }
      }
      else {
//      if (status.pr_why == PR_SYSEXIT) {
        if (sc->ret_type == Ptr)
          printf(" = 0x%x\n", status.pr_reg[R_R0]);
        else
          printf(" = %d\n", status.pr_reg[R_R0]);
      }
    }

    ioctl (fd, PIOCRUN, 0);
  }

  ioctl (fd, PIOCRUN, 0);
}
```

ÖZGEÇMİŞ

1982 yılında Trabzon' da doğdu. İlk öğrenimini Yomra Merkez İlkokulunda, orta öğrenimini Zehra Kitapçıoğlu ortaokulunda, lise öğrenimini Fatih Lisesi'nde tamamladı. 2000 yılında Karadeniz Teknik Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü'nde lisans programına başladı ve 2006 yılında bu bölümden mezun oldu. Aynı yıl ENO Bilişim ve Danışmanlık şirketinde SAP-ABAP danışmanı olarak çalışmaya başladı. Eylül 2007 de SAP danışmanı ve ERP Sistem uzmanı olarak Bilim İlaç San. ve Tic. A.Ş. 'de çalışmaya başladı. Halen bu görevini sürdürmektedir. Yabancı dil olarak orta düzeyde İngilizce bilmektedir.