**SİSTEM ÇAĞRILARI**

Sistem çağrıları OS tarafından servisler için sağlanan arayüzdür. Doğrudan donanıma erişenler gibi düşük seviye görevler (tasks) assembly dilinde instructionlar içerdiği halde genellikle C, C++ dilleriyle yazılmışlardır. System call’lar sayesinde yazılımcı doğrudan donanıma müdahale etmez. Donanım üzerinde gerçekleştireceği işlemi system call kullanarak gerçekleştirir. Bu sayede olası sistem hatalarından kaçınılmış olur.

System call’ların da, sistemden sisteme değişiyor olması yazılımcının işini zorlaştırır. Bu değişiklik, bir OS için yazılan programın başka bir OS üzerinde çalışamamasına sebep olur. Bu yüzden programcılar, doğrudan sistem call’lar yerine application program interface(API) kullanmayı tercih ederler Her uygulama, mutlaka sistem çağrısı yapmak durumundadır. Yapılan sistem çağrıları disk üzerinden okuma, yazma gibi daha "fiziksel" ve donanıma yakın olabileceği gibi o anki sistem zamanını alma gettimeofday, çalışan uygulamanın process id değerini öğrenme getpid, uygulamaya öncelikler atama setpriority gibi doğrudan çekirdek içerisindeki belirli mekanizmalarla ilgili de olabilir. Bir uygulamanın hayata gelmesi ve çalışmaya başlayabilmesi için öncesinde bir başka uygulamanın fork() sistem çağrısıyla yeni bir process üretmesi de gereklidir. Linux çekirdeğinde X86 mimarisi için yaklaşık 380 civarında sistem çağrısı bulunur.

**Process Ve İplik Yönetim Çağrıları**

İş parçacığı, görevi gerçekleştirmek için işlem kaynaklarını kullanan bir program yürütmedir. Tek bir programdaki tüm iş parçacıkları mantıksal olarak bir işlem içinde bulunur. Çekirdek, her ipliğe bir yığın ve bir iplik kontrol bloğu (TCB) tahsis eder. İşletim sistemi, aynı işlemin iş parçacıkları arasında geçiş yaparken yalnızca yığın işaretçisi ve CPU durumunu kaydeder. İplikler üç farklı şekilde uygulanır; bunlar çekirdek seviyeli iplikler, kullanıcı seviyeli iplikler, hibrit ipliklerdir. İpler, çalışan, hazır ve engellenmiş üç durum içerebilir; sadece kaynak ekini değil, anahtarlama yükünü azaltan iletişim durumunu ve hesaplama durumunu içerir. Eşzamanlılığı (paralellik) arttırır, dolayısıyla hızı da artar.

**Yeni Proses Oluşturma Çağrıları**

Bir bilgisayar sisteminde çalışan tüm yazılımlar, proses olarak anılır. Genel olarak bir sistemde, kullanıcı ya  da  işletim  sisteminin  prosesleri  vardır.  Bir  proses,  sistem  çağrılarını  kullanarak  başka  bir  proses  oluşturabilir ya da diğer proseslerle etkileşim kurabilir.  Proses  daha  teknik  bir  tabirle,  çalışır  durumda  olan  programa  verilen  bir  isimdir.  CPU  tarafından  işletilmeyen  bir  program  pasif  bir  yapıya  sahip  olup  komutlar  dizisinden  ibarettir.  Bu  komutlar  işletime  alınmak suretiyle işlevlerini yerine getirebilirler.  Tüm modern bilgisayarlar aynı anda birkaç işi yapabilirler; kullanıcı programı çalışıyorken diskten okuma,  bir  text  dokümanı  ekranda  gösterme  veya  çıktı  işlemlerini  bir  arada  gerçekleştirebilir.  Çoklu  programlamayı destekleyen sistemlerde CPU, bir programdan diğer bir programa belli zaman dilimlerinde  anahtarlanır.  Aslında  CPU  aynı  anda  bir  programı  çalıştırabilmesine  rağmen,  prosesler  arasında  hızlı  anahtarlandığından dolayı bu prosesleri paralel çalıştırdığı izlenimini verir. Unix’te yeni process oluşturmak için fork() sistem çağrısı kullanılır ve address space olarak yeni process(child) parent process’in kopyasını kullanır. Genellikle fork() çağrısı exec() sistem çağrısı ile beraber kullanılır. exec() fonksiyonu çağrıldığı programa ait olan memory’i çalıştırılacak olan başka bir program ile değiştirir. Bu sayede başlangıçta parent processin kopyası olan child process, farklı işler yapabilir.

int main() {

  pid\_t pid;

  pid = fork();

  if(pid == 0)

    printf("Child process.\n");

  else if(pid > 0)

    printf("Parent process.\n");

İşletim sistemlerinde çalışan her process kendini diğerlerinden ayıran bir process id’ye sahiptir. Parent ve child process arasında ayırım yapmak için program içindeki pid değişkenini kullanıyoruz. Her iki process’de fork() çağrısından sonra aynı kodu çalıştırır, tek farkla. pid değeri child için 0 iken parent process için 0’dan büyük bir sayıdır. Bu sayede iki processi birbirinden ayırabildik.

Şimdi exec() fonksiyonu ile beraber kullanalım. exec ile kastedilen bir fonksiyonlar grubudur. Hepsi temelde aynı işlevi yapsada farklı parametrelerle farklı kullanım şekilleri vardır.

 pid\_t pid;

  pid = fork();

**if**(pid == 0)

    execl("/bin/ls", "/bin/ls", (**char**\*) NULL);

**else** **if**(pid > 0)

    wait(NULL);

**printf**("Parent process.\n");

**exit**(0);

Çağırdığımız execl() fonksiyonu ilk argüman olarak çalıştırılabilir dosyanın tam yolunu(path) alıyor. Sonraki argüman ise çalıştırılacak komut oluyor. 2 ve sonrasındaki tüm argümanlar çalıştırılacak olan programa argüman olarak aktarılır. Örneğin ls komutuna ek olarak -a parametresi ekleyebilirdik. Verilen argüman listesinin bittiğini göstermek için ise NULL’u kullanıyoruz. Ayrıca parent processe yeni komutlar ekledik. wait() komutu parent processin child’ın işi bitene kadar beklemesini sağlıyor daha sonra exit() komutu ile kendini sonlandırıyor.

**Yeni İplik Oluşturma Çağrıları**

geleneksel işletim sistemlerinde her prosesin – özel adres uzayı ve – tek akış kontrolü var. aynı adres uzayında birden fazla akış kontrolü gerekebilir – aynı adres uzayında çalışan paralel prosesler gibi iplik = hafif proses aynı adres uzayını paylaşan paralel prosesler benzeri aynı proses ortamında birden fazla işlem yürütme imkanı iplikler tüm kaynakları paylaşır: – adres uzayı, bellek, açık dosyalar, .çoklu iplikli çalışma – iplikler sıra ile koşar

iplik kütüphane yordamları ile yeni iplikler yaratır – örn: thread\_create • parametresi: koşturacağı yordamın adı yaratılan iplik aynı adres uzayında koşar bazı sistemlerde iplikler arası anne – çocuk hiyerarşik yapısı var – çoğu sistemde tüm iplikler eşi

* Bir proses içinde birden fazla işlem olabilir
  + - Bazı işlemler bazen bloke olabilir; ipliklere bölmek performansı arttırır
* İpliklerin kendi kaynakları yok
  + - Baratılmaları / yok edilmeleri proseslere göre kolay – ipliklerin bazıları işlemciye yönelik bazıları giriş-çıkış işlemleri yapıyorsa performans artar
* hepsi işlemciye yönelikse olmaz
* çok işlemcili sistemlerde faydalı

İpliklerin Kullanıcı Uzayında Gerçeklenmesi ,pliklerin Kullanıcı Uzayında Gerçeklenmesinin avantajları – ipliklerin ayrı bir iş sıralama algoritması olabilir – çekirdekte iplik tablosu yeri gerekmiyor – tüm çağrılar yerel rutinler ⇒ çekirdeğe çağrı yapmaktan daha hızlı

Çekirdek uzayında modellemede çekirdek ipliklerden haberdardır. İplik tablosu çekirdekte olup yeni iplik oluşturmak için çekirdeğe sistem çağrısı yapılır. İpliği bloke edebilecek tüm çağrılarda yöneltilecek bir çağrı gönderir. İşletim sistemi hangi ipliğin çalışacağına karar verir. Kullanıcı uzayında ipliklerin modellenmesinde problem oluşturan çağrılarını yeniden yazılması gerekmez. Bu gerçekleme de sistem çağrısı yapmak maliyetlidir. Maliyet para değil zamandır. Çok sık iplik yaratma yok etme gibi işlemler varsa vakit kaybı çoktur.

**Bekleme Ve Sonlama Çağrıları**

Bu çağrıları kullanabilmek için derleyicimize,  
  
#include <sys/types.h>

#include <sys/wait.h>

kütüphanelerini eklememiz gerekiyor.

**Sleep() Çağrısı ;** Herhangi prosesi, parantezler içerisine yazacağımız saniye kadar bekletebiliriz. Hangi prosesi bekleteceğiniz ise tamamen size kalmış. Mesela çocuk proses aktif olacağı sırada ben program koduma sleep(1) yazdım diyelim. Bu durumda çocuk prosese sıra geldiği zaman çocuk proses çalışmayacak. 1 saniye bekleyecek. Bu sırada başka proseslerin işleri varsa onlar işlerini yaparlar ve 1 saniye sonra çocuk proses yeniden aktif olur. Yani sleep(1) yaparak istediğimiz herhangi bir prosesi 1 saniyeliğine uyutmuş oluruz. Eğer ki sleep(5) yazarasak, o anda aktif olan proses 5 saniyeliğine uyku moduna geçer ve bu süre dolana kadar da aktif olmaz. Süre dolduktan sonra tekrar kendiliğinden uyanır ve işlemlerine devam eder. Bilgisayardaki işlemler milisaliseler içerisinde gerçekleştiği için 1 bile saniye aslında önemli bir zaman aralığıdır.

**Wait() ve Waitpid() Çağrıları ;** Bu ifadeyi aktif olan proses içine yazdığımız zaman o proses, herhangi bir alt prosesin işlemini bitirmesi için bekler. Sonlanmadığı için programı da sonlandırmaz. Yani proses vardır ama işlem yapmıyordur diyebiliriz. Diğer prosesin işini yapmasını bekler. İpliklerde pthread\_join() ifadesi bu görevi görüyordu. Proseslerde ise wait() ve waitpid() çağrıları bu görevi yapar.  
  
**wait(***int \*durum***)** şeklinde tek bir parametre alır içine. Bu parametre adres tutar. Beklenen proses sonlandığında, sonlanan bu prosesin çıkış değerini tutar. wait() çağrısı, waitpid() çağrısının eski versiyonudur.  
  
**Waitpid(***pid\_t pid, int \*durum, int secenekler***);** şeklinde 3 parametre alır.  
Birinci parametreye bekletmek istediğimiz prosesin ID numarasını yazıyoruz.  
 **1. parametre :** Hangi prosesi bekleyeceksek onun ID'sini yazıyoruz. Herhangi bir çocuk proses beklenecekse ilk parametreye "-1" değeri girilir.  
**2. parametre :** Bu parametre adres tutar. Beklenen proses sonlandığında int \*durum olarak belirttiğimiz ikinci parametre ifadesi, sonlanan bu prosesin çıkış değerini tutar.  
**3. paramtere :** Özellikler parametresidir. "0" veya bazı özel değerler alır. Sıfır yazmadığımız durumlarda alabileceği bu bazı özel değerlerin 3 tanesine bakalım :  
  
**WNOHANG     :** Hiçbir çocuk proses sonlanmıyorsa hemen geri dönmeyi belirtir.  
**WUNTRACED :** Çocuk proses durdurulduğunda geri dönülecektir.  
**WCONTINUED:**Durdurulan çocuk proses SIGCONT sinyali aldığında geri döner.

**Yürütme Çağrıları**

Bir kullanıcı programı bir sistem çağrısını çağırdığında, bir sistem çağrısı yönergesi yürütülür. Bu işlem, işlemcinin çekirdek koruma etki alanında sistem arama işleyicisini yürütmeye başlamasını sağlar. Bu sistem arama işleyicisi aşağıdaki eylemleri gerçekleştirir:

Bu değeri belirler ut\_erroruthread yapısındaki alan: 0Çağıran iş parçacığıyla ilişkili bir çekirdek yığınına geçer İstenen sistem çağrısını gerçekleştiren işlevi çağırır. Sistem yükleyicisi, her sistem çağrısı için kullanılan işlevlerin bir tablosunu tutar. Sistem çağrısı, çağıran iş parçacığından çalışır, ancak çekirdek koruma etki alanında çalışan sistem çağrıları nedeniyle daha fazla ayrıcalığa sahip olur. Sistem çağrısı gerçekleştiren işlev, istenen işlemi gerçekleştirdikten sonra, denetim sistem çağrısı işleyicisine döner. Eğerut\_erroruthread yapısındaki alanın sıfır dışında bir değeri vardır; değer, uygulamanın iş parçacığa özgü errno değişkenine kopyalanır. Bekleyen bir sinyal varsa, sinyal işleme gerçekleşebilir; bu durum, uygulamanın sinyal işleyicinin çağrılma sonucunu verebilir. Beklemekte olan bir sinyal yoksa, sistem çağrı işleyicisi, kullanıcı koruması etki alanında sürdürülen çağıran iş parçacığın durumunu geri yükler.

**Kontrol Grubu Sistem Çağrıları**

Süreç kontrolü, süreçleri yönlendirmek için kullanılan sistem çağrısıdır. Bazı süreç kontrol örnekleri arasında oluşturma, yükleme, iptal etme, sonlandırma, yürütme, işleme, işlemi sonlandırma vb. bulunur**.** Kontrol Grupları, görev kümelerini ve bunların gelecekteki tüm çocuklarını özel davranışa sahip hiyerarşik gruplar halinde toplamak/bölümlemek için bir mekanizma sağlar. Bir grup , bir veya daha fazla alt sistem için bir dizi parametre ile bir dizi görevi ilişkilendirir. Bir alt sistem , görev gruplarını belirli şekillerde ele almak için gruplar tarafından sağlanan görev gruplama olanaklarını kullanan bir modüldür. Bir alt sistem, tipik olarak, bir kaynağı planlayan veya grup başına limitler uygulayan bir "kaynak kontrolörüdür", ancak bir sanallaştırma alt sistemi gibi, bir grup süreç üzerinde hareket etmek isteyen herhangi bir şey olabilir. Bir hiyerarşi , bir ağaçta düzenlenmiş bir grup gruptur, öyle ki sistemdeki her görev tam olarak hiyerarşideki gruplardan birinde ve bir dizi alt sistemde bulunur; her alt sistem, hiyerarşideki her bir gruba bağlı sisteme özgü duruma sahiptir. Her hiyerarşi, kendisiyle ilişkilendirilmiş cgroup sanal dosya sisteminin bir örneğine sahiptir. Herhangi bir zamanda birden fazla aktif görev grubu hiyerarşisi olabilir. Her hiyerarşi, sistemdeki tüm görevlerin bir bölümüdür. Kullanıcı düzeyinde kod, grup sanal dosya sisteminin bir örneğinde grup grupları oluşturabilir ve yok edebilir, bir görevin hangi gruba atandığını belirleyebilir ve sorgulayabilir ve bir gruba atanan görev PID'lerini listeleyebilir. Bu yaratımlar ve atamalar, yalnızca grup dosya sisteminin bu örneğiyle ilişkili hiyerarşiyi etkiler. Gruplar için tek kullanım, basit iş takibi içindir. Amaç, diğer alt sistemlerin, bir gruptaki süreçlerin erişebileceği kaynakları hesaplama/sınırlama gibi gruplar için yeni nitelikler sağlamak üzere genel grup desteğine bağlanmasıdır. Örneğin, cpusets bir dizi CPU'yu ve bir dizi bellek düğümünü her bir gruptaki görevlerle ilişkilendirmenize izin verir Kontrol Grupları, çekirdeği şu şekilde genişletir:

* Sistemdeki her görevin bir css\_set için referans sayılan bir işaretçisi vardır.
* Bir css\_set, sistemde kayıtlı her bir cgroup alt sistemi için bir tane olmak üzere, cgroup\_subsys\_state nesnelerine referansla sayılan bir işaretçi kümesi içerir. Her hiyerarşide üyesi olduğu grupla bir görevden doğrudan bağlantı yoktur, ancak bu, cgroup\_subsys\_state nesneleri aracılığıyla işaretçiler izlenerek belirlenebilir. Bunun nedeni, alt sistem durumuna erişmenin, performans açısından kritik kodda sık sık gerçekleşmesi beklenen bir şey olmasına karşın, bir görevin gerçek grup atamalarını (özellikle, gruplar arasında hareket etme) gerektiren işlemler daha az yaygın olmasıdır. Bağlantılı bir liste, css\_set->tasks'ta sabitlenmiş css\_set'i kullanarak her task\_struct'ın cg\_list alanı boyunca çalışır.
* Kullanıcı alanından tarama ve manipülasyon için bir grup hiyerarşisi dosya sistemi monte edilebilir.
* Herhangi bir gruba bağlı tüm görevleri (PID'ye göre) listeleyebilirsiniz.

**Bilgi Alma Çağrıları**

Pek çok sistem çağrısı, aynı isimdeki glibc wrapper fonksiyonları üzerinden çağrılmaktadır.Örnek olarak sistem çağrılarını takip etmede sık kullanacağımız strace aracının çıktısındaki open() çağrısına bakalım:

open("/tmp/index.jpeg", O\_RDONLY)= 3

Burada kastedilen glibc içerisindeki open() fonksiyonu değil, open sistem çağrısıdır. Strace üzerinden sistem çağrısına geçirilen argümanları ve geri dönüş değerini (3) görmekteyiz. Normalde bu yöntemle sistem çağrılarının ismi değil numarası izlenebilir. Strace uygulaması elde ettiği sistem çağrısı numarasını kendi veritabanında arayıp bizler için daha okunabilir bir formda gösterir. Strace uygulamasının bu işlemi hangi yöntemle gerçekleştirdiğine dair detaylı bilgileri ilerleyen bölümlerimizde bulabilirsiniz. Sistem çağrıları normal fonksiyon çağrılarına oranla oldukça yüksek maliyetli işlemlerdir. Her sistem çağrısında uygulamanın o anki durumunun saklanması, çekirdeğin işlemcinin kontrolünü ele alması ve ilgili sistem çağrısı ile çekirdek kipinde talep edilen işlemleri gerçekleştirmesi, sonra ilgili uygulamanın tekrar çalışma sırası geldiğinde, uygulamanın saklanan durumunun yeniden üretilip işlemlerin kaldığı yerden devamının sağlanması gereklidir Sistem çağrılarını glibc üzerinden kullandığımız için, glibc tarafında sistem çağrısından EINTR hatası geldiğinde, uygulamaya geri dönüş değeri olarak -1 dönülür fakat errno global değişkeni EINTR şeklinde ayarlanır.Bu aslında hata olmayan istisnai durum, zaman zaman pek çok uygulama kodunda gözardı edilmektedir Bazı kullanım senaryolarında yukarıdaki senaryo istisnai olmaktan çıkıp, ilgili yazılımın doğası gereği sürekli veya sıklıkla da (read, write, open, connect vb.) oluşabilir.

**Hata Ayıklama Çağrıları**

**GNU Debugger**

GNU sistemlerinde standart olan, GDB (GNU Debugger) uygulamasını inceleyeceğiz. gdb bir komut satırı uygulaması olarak çalışmakta

ve C, C++, Objective-C, assembly ve Java olmak üzere bir çok dili desteklemektedir. gdb ayrıca, Eclipse, Qt Creator, NetBeans gibi bir çok IDE ve GNU DDD (Data Display Debugger) grafik önyüz uygulaması üzerinden de kullanılabilir. gdb ile bir programın içsel durumunu inceleyebilir, işleyişine müdahale edebiliriz. Tipik olarak, kodu adım adım işletebilir,

değişken ve yazmaçların değerlerini gözleyip değiştirebilir, kod üzerinde kesme noktaları belirleyebilir, akışa müdahale edebilir ve fonksiyonların çağrılma sırasını takip edebiliriz. gdb çok sayıda komut ve seçeneği barındırmasına karşın çoğu durumda göreli olarak daha az bir komut setiyle bir çok işi yapabilmekteyiz.

**Strace**

Unix tabanlı sistemlerde strace gibi bir uygulamanın varolabilmesi için gereken ptrace sistem çağrısı uzun yıllardır (SVr4 ve 4.3BSD) bulunmaktadır. Sistem çağrısı ismini process trace kavramından alır. ptrace sistem çağrısı üzerinden bir uygulama başka bir uygulamanın durumunu takip edebildiği gibi değişiklikler yapma imkanına da sahip olmaktadır. ptrace sistem çağrısı temel olarak gdb gibi debug uygulamalarında, strace, ltrace gibi sistem veya kütüphane çağrılarını takip uygulamalarında, code coverage araçlarında, çalışan yazılım koduna dokunmadan bazı hataların giderilmesinde veya güvenlik kontrollerinden geçirilmesinde kullanılır. ptrace çağrısıyla bir uygulamanın kontrolü tamamen başka bir uygulama verilmektedir. Buradaki kontrolden kastımız, uygulamanın kullandığı tüm bellek alanına erişim, sinyallerin alınması, değiştirilmesi, file descriptor'ların yönetimi hatta uygulamanın kod segmentinin değiştirilerek yamalar yapılması dahil aklımıza gelebilecek hemen her türden tehlikeli değişikliklere izin veriliyor olmasıdır. Bahsettiğimiz bu özelliklerinden ötürü, bir uygulamanın başka bir uygulamayı ptrace ile kontrol edebilmesi için, ilgili uygulamaya sinyal gönderme yetkisinin bulunması gerekir. Dolayısıyla özel durumlar haricinde her kullanıcının kendi sahibi olduğu diğer uygulamaları ptrace ile kontrol edebileceğini, root kullanıcısının da sistemdeki tüm uygulamaları kontrol edebileceğini söyleyebiliriz. Linux Capabilities API sisteminin geliştirilmesinden sonra yukarıda koşullardan bağımsız olarak, CAP\_SYS\_PTRACE özelliği sayesinde de ptrace izni verilebilmektedir.

**Proseslerde Sinyal Çağrıları**

Sinyal çağrıları içi kullanılması gereken kütüphane;

#include <signal.h> Sinyal çağrılarından sadece kill() ifadesine değineceğim için en başından şunu belirtmeliyim ki bu çağrı;

kill(pid\_t pid, int signal); olmak üzere 2 parametre alır. İlk parametre proses ait bilgileri tutar. İkinci parametre ise sinyal bilgisini tutar. O proses ne yaptırmak istiyorsak, signal kısmına o sinyalin adını yazmamız gerekiyor. Alttaki ifadelerden de ne demek istediğimi daha iyi anlayacaksınız.

pid\_t prosesID; //olsun.

prosesID = fork(); //çocuk prosesin bilgileri artık "prosesID" ifadesi içinde saklanıyor.Artık "prosesID" ifadesini kullanarak, çocuk prosese kill() aracılığıyla istediğimiz sinyali uygulayabiliriz. kill(prosesID,SIGSTOP) ve kill(prosesID,SIGCONT) Çağrıları

Bu ifadeler sayesinde, ID numarasını bildiğimiz bir prosesin işlemini istediğimiz zaman sonlandırıp, istediğimiz zaman yeniden aktif edebiliyoruz. Mesela şöyle bir örnek vereyim. Çocuk prosesin ID numarasının "prosesID" olduğunu varsayalım. Ben kill(prosesID,SIGSTOP) ifadesini kullanırsam eğer, çocuk proses duracaktır ve hiçbir iş yapamayacaktır. Taa ki biz yeniden kendisini aktif edene kadar. kill(f,SIGCONT) ifadesi ise işte burada devreye giriyor. SIGSTOP komutu ile işlem yapmasını durdurduğumuz bir prosesi SIGCONT komutu ile istediğimiz zaman yeniden aktif hale yani iş yapabilir hale getirebiliyoruz. kill(prosesID,SIGKILL) Çağrısı Bu ifade ile ID numarasını bildiğimiz bir prosesi istediğimiz zaman öldürebiliyoruz. Öldürdüğümüz proses bir daha işlem yapamaz ve eğer ana proses ile programın sonlanmasına sebep olur. Öldürmek istediğimiz ana prosesin ID numarasının "prosesID" olduğunu varsayalım. Bu durumda kill(f,SIGKILL) olarak kullanılır bu ifade. Derlediğiniz programda ID numarası yazılması gereken kısım olan "prosesID" kısmında "getpid()" ifadesini kullanabilirsiniz.

kill(getpid(),SIGSTOP)

kill(getpid(),SIGCONT)

kill(getpid(),SIGKILL

**İplik Sinyal Çağrıları**

plikler kolay oluşturulup kolay yok edilirler. Bir program yazarken, programın mevcut prosesi içinde bir iplik yaratıp birden çok işlem yapmak istiyorsanız eğer, derleyicinize;

#include <pthread.h>

kütüphanesini eklemeniz gerekiyor.

Threadler yani iplikler ile işlemler yaparken bazı ifadeleri bilmeniz gerekiyor. İplikler ile işlem yaparken bu ifadeler olmazsa olmaz ifadelerdir. Programı yazmaya başladığınız zaman main kısmında öncelikli olarak yapacağınız işlemlerdir. Kısacası bu ifadeler iplik işlemlerinin demirbaş işlemleridir. pthread\_t() İpliklere ait bilgileri tutar. Bir ipliğin sahip olması gereken tüm özellikler bu ifade ile tanımlanmış olan ifadede saklanır. pthread\_create() :

Bu ifade ile yeni bir iplik oluşturulur. 4 farklı parametre alır.

1. parametresi : İpliğe ait bilgileri tutan değişkenin adresini tutar.

2. parametresi : İpliğin özelliklerini tutar. Yapacağımız örneklerde NULL olarak kalacak.

3. parametresi : İpliğe ait olan fonksiyonun adını tutar.

4. parametresi : İpliğimize ait olan fonksiyona değer döndereceksek eğer, 4. parametre kısmı kullanılır. Bu parametre için unutmamanız gereken en önemli özellik ise, bu parametre sadece void değerler alır. Eğer ki farklı türdeki değerler yollayacaksanız fonksiyona, ilk önce (void \*)x işlemi yaparak değerinizi void türüne dönüştürmeniz gerekiyor. Videolarda bunu daha iyi anlayacaksınız zaten.

pthread\_join()

"İpliğin işlemini bitirmesini bekle" anlamına gelir. Eğer iplik kullanarak bir program yazacaksanız bu ifade olmazsa olmaz bir ifadedir. Bu ifade sayesinde iplik işini bitirmeden proses sonlanmaz. Bu ifade 2 parametre alır.

1. paremetresi : İpliğe ait bilgileri tutan değişkenin kendisini tutar.

2. parametresi : Geri dönüş değerlerini tutar. Eğer ipliğinizin geri dönüş değeri var ise, ikinci parametre bu veriyi tutar. Yarattığınız thread geriye değer döndermiyor ise eğer, bu parametreye NULL yazılır.

pthread\_exit()

İçine void değerler alır. İpliğin geri dönüş değerini tutarlar. Geri dönüş değeri yok ise içine NULL yazılır.

**Dosya Dizin Yönetimi Çağrıları**

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.

**Dosya Temel İşlem Çağrıları**

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 232 -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.. /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. /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. /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 . Herhangi bir kelime işlemci burada yer alan bir dosyayı açmaya kalkıştığında, dosya çekirdek bilgilerinden otomatik olarak oluşturulmaktadır

**Dosya Erişim Çağrıları**

Erişim hakları, Linux dosya sistemi güvenliğinin belkemiğini oluşturur. Her dosyaya ayrı verilebilen erişim izinleri sayesinde çok daha rahat bir sistem yönetimi gerçekleştirilebilir. Bu, konuya sadece sistem görevlisi tarafından yaklaşıldığı zaman çıkartılabilecek bir sonuçtur. Kullanıcı bazında erişim hakları bazen daha da anlamlı olabilir. Yanlış kullanıldığında hoş olmayan süprizlere yol açabilir. Linux altında üç çeşit erişim hakkı vardır :

* **Okuma izni** : Dosyanın okuma izni varsa içeriği görülebilir, dizinin okuma izni varsa içerdiği dosyaların listesi alınabilir.
* **Yazma izni** : Dosyanın yazma izni varsa dosyayı değiştirebilir veya silebilirsiniz. Dizine yazma izni verildiğinde dizin altındaki dosyalar yazılabilir veya silinebilir.
* **Çalıştırma izni** : Dosyayı çalıştırma hakkını verir.

Bir dosya veya dizin ilk yaratıldığı anda Linux tarafından öntanımlı bazı izinler verilir. Genellikle bu izin çalıştırma ve okumadır. Dosyanın oluşma anında verilen izini değiştirmek için erişim yetki kalıbı olarak da bilinen umask komutu kullanılır. Daha fazla bilgi için komuta ait man dosyasına bakın.

Erişim haklarının dışında bir dosyanın üç izin düzeyi daha vardır. Bunlar, dosyanın sahibi, dosyanın grubu ve diğer kullanıcılardır. Dosyanın sahibi, o dosyayı oluşturan kişidir. Her kullanıcının bir grubu da olduğu için, dosya oluşturulurken kullanıcı hangi grupta ise dosya da o gruba ait olacaktır. Dosyanın sahibi olmayan ve grubu da dosyanın grubuyla uyuşmayan sistemdeki kullanıcılar ``diğer'' sınıfına girerler. Gruplar hakkında daha fazla bilgi için *Linux Sistem Yönetimi* konusuna bakın.

Dosya İzinlerinin Değiştirilmesi;

Dosya izin bilgilerini görebilmek için ls komutu, -l parametresiyle kullanılır. Aşağıdaki dosya üzerindeki erişim haklarına göz atalım :

-rwxr-xr-x 2 gorkem users 182 Feb 12 03:58 deneme

Dosya bilgisinin en sol kısmında izin hakları yeralır. En baştaki ``-'' işareti bunun düz dosya olduğunu belirtir. ``gorkem'', dosyanın sahibi; ``users'' ise grubudur. Ardından sırayla uzunluk, son değiştirilme tarihi ve dosya ismi gibi bilgiler gelir. Son harf kümesine dosyanın modu da denir.

Solda yeralan r,w ve x karakterleri sırayla okuma, çalıştırma ve yazma haklarını belirtir. En soldaki ``-'' işaretini ayırdığımız zaman geri kalan harfleri üç adet üçlü grup haline getirelim:

rwx r-x r-x

Sırayla birinci harf kümesi dosya sahibinin izinlerini, ikinci harf kümesi grup izinleri ve son küme de diğer kullanıcıların izinlerini belirtir. Buna göre yukarıdaki dosyada

* rwx : kullanıcı okuyabilir, yazabilir, çalıştırabilir
* r-x : grup okuyabilir, çalıştırabilir fakat yazamaz.
* r-x : diğerleri okuyabilir, çalıştırabilir fakat yazamaz.

Sol baştaki karakter ``d'' olsaydı, bir dizini inceliyor olacaktık.

Dosyanın izinlerini değiştirmek için chmod komutu kullanılır. Bu komutu kullanırken hangi izin düzeyine (kullanıcı, grup veya diğerleri) hangi izinlerin verileceği veya kaldırılacağı yazılır.

Örnek olarak yukarıdaki dosyanın okuma iznini kaldırmak için

$ chmod -r deneme

kullanılabilir. "-" işareti, iznin kaldırılacağını belirtir. İzin vermek için ise "+" yazın. Dosyanın yeni görüntüsü şöyle olur :

--wx--x--x 2 gorkem users 182 Feb 12 03:58 deneme

Sadece grubun, sadece kullanıcının veya sadece diğerlerinin erişim hakkını değiştirebilmek için her izin düzeyi için bir harf tanımlanmıştır. Kullanıcı için "u", grup için "g" ve diğerleri için "o" yazarak belirli bir izin düzeyi için erişim hakkını değiştirmek mümkün olur. Bu üç harf, izinden hemen önce yazılır.

$ chmod u+x deneme (dosyanin sahibi calistirabilir)

$ chmod o+r t2sac (digerleri okuyabilir)

$ chmod g-w deneme (dosyanin grubu yazamaz)

Her izin için aynı zamanda sayılar da tanımlanmıştır.

**Özel Dosya Ve Dosya Bilgilerine Erişim Çağrıları**

Bazen MS-DOS altından oluşturulamayan dosya sistemlerini (ext2 gibi) oluşturmak gerekebilir. Linux altında bir dosya sistemini oluştururken aynı zamanda bunu kontrol eden programlar vardır. Bunların en çok kullanılanı, Slackware sürümünde gelen mkfs/'tir.

mkfs ile Linux native (ext2) dosya sistemi oluşturmak için mkfs.ext2 programı kullanılır.

mkfs.ext2 <aygit> <blok-sayisi>

"aygıt" yerine üzerine dosya sistemi kurulacak olan cihazın /dev dizini altındaki düğüm dosyasının ismi yazılır. Blok sayısı yerine, fdisk yazılımından o bölümün kaç blok olduğunu öğrenip bu sayıyı girmelisiniz. Bir blok 1024 bayttan oluşur. Örneğin

# mke2fs /dev/hda3 163829

komutu, /dev/hda3 disk bölümünde ext2 dosya sistemi kurar. Bu komutu kullanmadan önce iyi düşünün, zira burada yeralan tüm dosyalar silinir ve bu hatanın geri dönüşü olmaz. Aşağıdaki komut ile bir disket üzerinde ext2 dosya sistemi kuruluyor.

# mke2fs /dev/fd0 1440

mke2fs ile mkfs.ext2 komutları aynıdır. Benzer şekilde, MS-DOS dosya sistemi yaratmak için mkfs.msdos, minix dosya sistemi yaratmak için mkfs.minix kullanılabilir. Blok sayısı girilmez ise mke2fs bunu otomatik olarak bulacaktır.

Herhangi bir sebepten dolayı diskte bozulma ve veri kaybını en aza indirmek için de yazılımlar vardır. fsck ile dosya sistemi kontrolü yapılabilir. fsck ile ext2, dosfsck (veya fsck.msdos) ile MS-DOS formatlı alanlar kontrol edilir.

linux:~# fsck /dev/hda2

Parallelizing fsck version 0.5b (14-Feb-95)

e2fsck 0.5b, 14-Feb-95 for EXT2 FS 0.5a, 95/03/19

/dev/hda2 is mounted. Do you really want to continue (y/n)? yes

Pass 1: Checking inodes, blocks, and sizes

Pass 2: Checking directory structure

Pass 3: Checking directory connectivity

Pass 4: Checking reference counts

Pass 5: Checking group summary information

Fix summary information<y>? yes

Block bitmap differences: -46486 -46487 -46488 -46489 -46490. FIXED

Free blocks count wrong for group 5 (2803, counted=2808). FIXED

Free blocks count wrong (16326, counted=16331). FIXED

/dev/hda2: \*\*\*\*\* FILE SYSTEM WAS MODIFIED \*\*\*\*\*

/dev/hda2: \*\*\*\*\* REBOOT LINUX \*\*\*\*\*

/dev/hda2: 3494/29632 files, 42904/59235 blocks

Bir dosya sistemini kontrol etmeden önce okunan bölümü root dosya sisteminden ayırın ve kontrol işleminden sonra bilgisayarı kapatıp tekrar açın.

debugfs programı, dosya sistemi parametrelerini incelemek için kullanılır. Disk üzerine doğrudan erişım yaptığı için dikkatli kullanmalısınız. ext2fs yardımıyla kurtarılamayan bazı dosyalar (özellikle silinmiş dosyalar) debugfs yardımıyla kurtarılabilir.

**Dizin İşlemleri**

Linux işletim sistemini ilk kurduğunuz anda, yüzlerce dizin altında binlerce dosyanız olacaktır. find komutu, bu dosyaların arasında tarama yapabilmek için geliştirilmiştir. Bu komutla sadece dosya isimlerine değil, dosyanın sahibi, erişim hakları, son erişim tarihi gibi verilere de ulaşmak mümkündür. find komutunun en sık kullanış şekli şudur :

find <dizin-ismi> -name <ne-aratiliyor>

dizin-ismi yerine taranacak olan dosyanın bulunduğu dizin ismi yazılır. -name parametresinden sonra ise aranan dosyanın ismini girin. Burada "\*" ve "?" gibi joker karakterlerini de kullanabilirsiniz. (Bu gibi özel karakterler hakkında daha geniş bilgi Bash kabuğu konusu altında bulunabilir)

Aşağıda, find komutunun kullanımına ilişkin birkaç örnek yeralıyor.

$ find . -name "\*.html"

(bulundugunuz dizinden itibaren sonu .html ile biten dosyalari arar)

$ find /home/halil -name "screen"

(/home/halil dizininden itibaren screen isimli dosyalari arar)

Bu komut, -type d parametresi yardımıyla sadece dizin isimleri arasında arama yapar.

$ find /usr/ -name "linux" -type d

find komutuna bir alternatif olan locate komutu ile dosyalar çok kısa bir sürede taranabiliyor. update komutu, her sabah belirli bir saatte sistemdeki tüm dosya ve dizin isimlerini bir veri tabanına yükler. Daha sonra bu dosyada yeralan belirli bir dosya veya dizin ismini locate ile arayabilirsiniz:

# locate pico

/usr/bin/pico

/usr/man/man1/pico.1.gz

Yukarıdaki komut ile içinde pico kelimesi geçen dosyaları bulduk.

**Güvenlik İçin Sistem Çağrıları**

**Zaman Yönetimi Çağrıları**

**Wındows API**

**Proses Yönetim API’leri**

**Dosya Yönetim API’leri**

**Dizin Yönetim API’leri**

**Güvenlik API’leri**

**Zaman Yönetim API’leri**