SİSTEM ÇAĞRILARI

Bilgisayar açıldığında , bir program çalışmak istediğinde ilk olarak işletim sisteminin elinden geçer , işletim sistemi hemen hemen bilgisayardaki butun aktiviteleri kontrol eder. Bunların içersinde kimler giriş yaptı ne kadar diski kullandı nasıl kullandı gibi veriler ve ne kadar diğer bilgisayarlarla bağlantı kurdu gibi veriler vardır.bu kullandığımız işletim sistemine biz "unix" diyoruz.

programların işletim sistemiyle konuşlarının yoluna "sistem çağrıları" diyoruz.
Ve bu bir çeşit yordam çağrısı gibi (aşağıda göreceğiz) , fakat farklı bir yapısı var.
Bunun yapısı tam olarak bazı aktiviteleri kullanabilmek için işletim sisteminin kaynaklarını kullanmak gibi..

>>> sistem çağrıları çok maliyetlidir . bazı yordamlar birkaç makine işlemleri yapabilirken . Bir sistem çağrısı kendi durumunu kaydettirmek için bilgisayarın bütün kaynaklarını kullanabilir cpu dan bir kısım alabilir .çünkü bu gücünü işletim sisteminden almıştır . Kendi işi bittikten sonrada bu cpu yu işletim sistemi geri sunar.

>>>sistem çağrıları sisteme bağlı yapılardır . bunu bildiğimize göre , taşınabilir bi yapı yazmak istediğimiz zaman bunu kullanmak için çok dikkatli olmalıyız. Sistem çağrıları oldukça karmaşık bir yapıdır.en sık olarak 2 versiyonunu görürüz bunlar :TRAP ve RET sistem çağrılarını uygulamak için özel bir i/o registırına ihtiyaç vardır.
işlemlar sırayla kullanmalıdır bu registırı , bu durumda bir guvenlik sorunu olarak karşımıza çıkmaktadır bu yüzden çoklu kullanıma izin verilen bu registerda dikkatli olunmalıdır.

UNİX 'in giriş çıkış (i/o) işlemleri için 5 temel sistem çağrısı mevcut bunlar:

```
1.  int open(char *path, int flags [ , int mode ] );
      2.  int close(int fd);
      3.  int read(int fd, char *buf, int size);
      4.  int write(int fd, char *buf, int size);
      5.  off_t lseek(int fd, off_t offset, int whence);
```

Bunları not etmelisin çünkü bunlar çok regüler çağrılardır fazlasıyla kullanılıcak.gelelim bunları bir işlem gibi nasıl kullanacağına , bunların fakrını bilmelisin . sistem çağrıları işletim sistemine istekte bulunurlar . ama normalde bir sistem çağrısı olmasa bir yordam olsa bu sadece program ın kendi içinde bir yere atlaması gibidir .bir yordam çağrısı sistem çağrılarınıda çağırabilir ama . mesela (fopen() aslında open() komudunu çağırır) ama tabi bu biraz daha farklı bir olay.
Ortak kullanlar ve diğer nedenlerden dolayı işletim sistemi i / o lar önemli olduğu için bunların sistem çağrılarıyla yapılmasını istemiştir ,çünkü sistem çağrıları bişey yapmadan önce işletim sisteminden talepte bulunurlar .
Üstte gösterdiğim 5 çağrı tipini detaylı olarak görmek istiyorsanız manuel kullanımdan bulabilirsiniz . şu kodları yazarak kolayca erişiriz. (do '**man -s 2 open**', '**man -s 2 close**', etc).

Open

gelelim open'a open bir dosya kullanmak için işletim sistemine istekte bulunan bir komuttur. Bu komut için kullanacağınız dosyanın yolu bayrak bilgisi ve hangı tip kullanacağınızın bilgisi gerekmektedir . eğer işletim sistemi talebinize cevap verirse bu size dosya descriptor olarak dönecektir (descriptor = bi nevi proses id gibi bişeydir . yani dosya tanımlayıcı) bu değer eğerki size negatif dönerse isteğiniz reddedilmiştir , ne oldu ne bitti diye düşünürken ,perror () komudunu kullanırsanız aynı lecture notes for chapter 1 deki gibi errno yu öğrenerek çok kolay bir şekilde sebebini kontrol edebilirsiniz.

İşletim sisteminin çağrısını olan open ı kullanarak açmak istediğiniz dosyayının bütüün hareketlerini dosya descriptor ile yapmanız mümkündür artık .

Örnek o1.c de in1 isimli dosyayı okumak için açar (yani modu read'tir) ,ve size bir dosya descriptor u verir , eğer terslik varsa -1 değeri gelir ,in1 kapanmamasınından kaynaklı olabilir Eğer 3 gibi bişey geldiyse bu isteğiniz garanti altındadır gönül rahatlığıyla kullanabilirsiniz Peki gelelim neden 3 ? çünkü 3 tip standart değere sahiptir stdin yani 0 stdout yani 1 stderr yani 2

Bu da demek oluyorki bu işin bir limiti var bir program sınırlı sayıda dosya açabilir . doğru bir işlem için bu işlemde bütün herşey kaynaklara kaydedildiğinden şuan için diğerlerini kapatmadan bir şey yapmamalısın . bu cümlenin üstüne doğal olarak bir soru doğucaktır . Buda ne kadar dosya açabiliceğimdir. Bunun cevabıda sistem ve sistem kullanıcı ve kullanıcı olayına bağlıdır . bazı sistemlerde 2000 e yakın olan bu rakam , bazılarındada kullanıcı kaynaklarıyla limitlidir. Sun'ın makilerinde 256 dosya imkanın var .linux söylemesede 1024 tane açabiliyor sanırım ama hiç bi yerde okumadım .

Şimdi gelelim bayraklara . burayı not etmelisin veya açıp manuel sayfasından öğrenebilirsin. İlerde sana description vereyeceğiz bayrakların nasıl çalışacağına dair . ve sen fcntl.h a bakmalısın bunu da /usr/include veya /usr/include/sys/fcntl.h bulabilirsin orda sana o_rdonly nin ne anlama geleceğini anlatacağız.

Örnek: o2.c out1 i açmayı deniyor yazmak için , fakat hata ile karşılaşıcak çünkü daha çıkış yapılmamıştır .Bununla birlikte eğer yeni bir dosya açmak istiyorsanız ki buda yazmak için ise. Siz bunu şu bayrak argümenleriyle açmalısınız **(O_WRONLY | O_CREAT | O_TRUNC)** bakın o3.c deki örnek gibi . uyarı bu out2 gibi bişey oluşturur .eğer uzunluğu 0 sa program kapatılır . bu yüzden not edin o2.c ile o3.c deki flag hatalarını (perror () kullanarak tabiki .)

```
UNIX> o2
o2: No such file or directory
UNIX> o3
UNIX> ls -l out*
-rw-r--r--  1 plank           0 Sep 11 08:50 out2
UNIX>
```

Son olarak , bu modu yanlızca yeni dosya oluştururken kullanmalısınız ,bu özellikle yeni dosya için koruma modudur. 0644 bu en tipik değerdir size (ben okuyabilir ve yazabilirim der herkes okumak için kullanır)

Benzer dosyalar açabilirsin , farklı dosya descriptor leri elde etmiş olucaksındır.
Eğer benzer dosyaları yazma modunda açarsanız hayli garip sonuçlarla karşılabilirsiniz.

Close

close () derki işletim sistemi bana dosya descriptorunu ver bende onu kapatayım .bu komutla ilgili çeşitli örnekleri c1.c göreceksiniz çoklu kullanımlarda dikkat edilmelidir çünkü close komutu tamamen legal bir komuttur unix için

Read

Read() derki ; işletim sistemi bana dosya descriptorunu (yani fd) okunacak dosyanın buf bilgisini ve boyutunu ver . ben de sana okuyup nerede kaldığım bilgisini döndüreyim

```
UNIX> cat in1
Jim Plank
Claxton 221
UNIX> r1
called read(3, c, 10).  returned that 10 bytes  were read.
Those bytes are as follows: Jim Plank

called read(3, c, 99).  returned that 12 bytes  were read.
Those bytes are as follows: Claxton 221

UNIX>
```

Burada not edilmesi gereken bir iki şey vardır .İlk olarak buf bilenen bir belleği işaret edilmeli Buda calloc() tarafından c de arşivlenmiştir eğer bunu daha önce ben görmedim diyorsanız size statik olark declare edilmiş bir Char c[100] gibi bir örnek vereceğim

İkinci olarak , bunu anlamanız için printf () ile ilgili bir çağrı sağlayacağım sonrada read() i kullancağım

Üçüncü olarak, read () 0 döndüğünde dosyanın bittiğine işarettir bunu r1.c de görmeniz mümkün.

Son olarak, bunu not almanızda fayda var , ilk 10 karakter read ile çağırıldı diğer 12 karakterde 2 . ci satırda çıktı . bunun nedeni printf() den gelen bir olay , 1.ci c in içinde olan bir şey diğeride printf() alanından gelen bir şey

Write

Kullanımı aynı read' e benzeyen ,farklılık olarak yazdıklarının numarasını döndürüyor.
W1.c de "cc360\n" yazdırılmıştır dosya out3 tür

```
UNIX> w1
called write(3, "cs360\n", 6).  it returned 6
UNIX> cat out3
cs360
UNIX>
```

Lseek

Her açılan dosya bir dosya işaretcisi (file pointer) ile açılır ,dosya açıldığındada bu işaretci dosyanın başlangıcını işaret eder ,dosya okunsada yazılsada aynı muhabbet geçerlidir .yazma ve okumada dosya işaretcisi hareket eder kaldığı yeri tutmak için . örnek r1.c ye bakın in1 de 11.ci baytı tutuyor . tabi manuel kullanımı açıp bunu istediğiniz yerden başlatmayı öğrenebilirsiniz nasıl mı yapıcaksınız tabiki açıcaksınız sys/types.h ı ve unistd.h nerden mi biliyorum . e tabiki "man –s 2 Iseek" yazıyorum çıkıyor.

Standard Input, Standard Output, and Standard Error

Şimdi , gel gelelim unixde her proses 3 dosya descriptor içerir

- dosya descriptor 0 için standard input(giriş)
- dosya descriptor 1 için standard output. (çıkış)
- dosya descriptor 2 için standard error. (error)

tabiki, program yazarken , s.i ile okursun bunun için read(0,……) kullanırsın aynı şekilde yazarkende tabiki s.o yani write (1,……..) kullanırsın .

böylece , basit bir cat programı yazdık buna simpcat.c den ulaşmanız mümkün.

```
main()
{
  char c;

  while (read(0, &c, 1) == 1) write(1, &c, 1);
}
```

```
UNIX> simpcat < in1
Jim Plank
Claxton 221
UNIX>
```

Bize aynı dosyayı işaret eden birkaç dosya tanımlayıcısıda olabilir.

```
UNIX> cat tfile
hope not
```

takip eden kod da 2 kere tfile açılmıştır 2 değişik dosya tanımlıyacıyla write ve read olarak

```
#include < string.h >
#include < unistd.h >
#include < fcntl.h >

int main (void)
{
    int fd[2];
    char buf1[12] = "just a test";
    char buf2[12];

    fd[0] = open("tfile",O_RDWR);
    fd[1] = open("tfile",O_RDWR);

    write(fd[0],buf1,strlen(buf1));
    write(1, buf2, read(fd[1],buf2,12));

    close(fd[0]);
    close(fd[1]);

    return 0;
}
```

Çıktısı şu şekilde:

```
UNIX> a.out
just a testUNIX>
```

bu durumda, t file içeriği tutarlıdır.bağımsız olarak fd[0],fd[1]

şimdi , biraz aşağıdaki koda bir göz atın bakalım.

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

int main (void)
{
    int i, r, w;
    char msg[12];
    char buf[2] = {0, 0};

    for (i = 0; i < 3; i ++) {
      if ((r = read(i,buf,1))<0) {
         sprintf(msg,"read  f%d:%s",i,buf);
         perror(msg);
      }
      if ((w = write(i,buf,1))<0) {
         sprintf(msg,"write f%d:%s",i,buf);
         perror(msg);
      }
      fprintf(stderr,"%d, r = %d, w = %d, char =
%d\n",i,r,w,(int)(buf[0]));
    }

    return 0;
}
```

Bu kod parçası biraz ilginçtir çünkü çalışma muhabbeti farklıdır. Çok daha değişik sonuçlar alırız. Aşadakinde bold mekturları keybord girişleridir karekterler çıkışa göre planlandığı zaman. Dosya in1 in text dosyası:

```
UNIX> cat in1
ABCDEFGHIJK
UNIX> a.out                    |UNIX> a.out < in1              |UNIX> a.out
< in1 > out1
A                              |write f0: Bad file descriptor  |write f0:
Bad file descriptor
A0, r = 1, w = 1, char = 65    |0, r = 1, w = -1, char = 65    |0, r = 1, w
= -1, char = 65
                               |B                              |read  f1:
Bad file descriptor
1, r = 1, w = 1, char = 10     |B1, r = 1, w = 1, char = 66    |1, r = -1,
w = 1, char = 65
C                              |                               |B
C2, r = 1, w = 1, char = 67    |2, r = 1, w = 1, char = 10     |B2, r = 1,
w = 1, char = 66
```

Biri bana söyleyebilirmi burda ne dönüyor arkadaşım !!!

Son olarak , veri güdümlü programlama denilen kavrama haydi bir göz atalım . yukardaki programda ,çıktılar f1,f2,f3,etc bize kolay bir şey sunuyor .tutarlı bir model olmadığı için ancak, stdin gibi isimleri çıktılamak olmaz .fakat stdout ve stderr açık net. Fakat sonrasında döngü için bir daha düzgün bir kaldıraç olamaz . birçok durumda , bir çok uygulamada numara mümkün olduğunda 3 döner aşağıdaki kod size büyük karşama gibi gelsede:

```
....

for (i = 0; i < 3; i++) {

  if (read(...) < 0) {

    switch i {
       case 0: sprintf(msg, "read stdin ...
       case 1:
       case 2:
      ....
    }

    perror....
  }

  if (write(...) < 0) {

    switch i {
       case 0: sprintf(msg, "write stdin ...
       case 1:
       case 2:
      ....
    }

    perror....
  }
```

```
  ....
  }
```

In light of this, take a look at the following instead:

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

char msg[6][15] = {"read  stdin",
                   "write stdin",
                   "read  stdout",
                   "write stdout",
                   "read  stderr",
                   "write stderr"};

int main (void)
{
    int i, r, w;
    char buf[2] = {0, 0};

    for (i = 0; i < 3; i ++) {
      if ((r = read(i,buf,1))<0)
         perror(msg[i*2]);

      if ((w = write(i,buf,1))<0)
         perror(msg[i*2+1]);

      fprintf(stderr,"%d, r = %d, w = %d, char =
%d\n",i,r,w,(int)(buf[0]));
    }

    return 0;
}
```

Çıktısı ise:

```
UNIX> a.out < in1 > out1
write stdin: Bad file descriptor
0, r = 1, w = -1, char = 65
read  stdout: Bad file descriptor
1, r = -1, w = 1, char = 65
B
B2, r = 1, w = 1, char = 66
UNIX>
```

Şimdiden mekanızmaları ve politasıyla ilgili bir konuşma yapmış olduk bu programla ilgili ,bakarsanız çıktıya çok net bi şekilde herşeyi görebilmeniz mümkün.fakat gerçek programda karışık gördüğünüz bu kod sade olmalı . şimdi umarım birşeyler anlamışsınızdır cs folks adına