Kayıt Dosyaları ve I/O Hakkında

Kayıt Dosyası Tutmak

Aşağıdaki kodu bilgisayarınızda deneyin;

```
UNIX> last | head -60
```

Sonuç olarak bilgisayarınıza yapılan son 60 girişin listelendiğini göreceksiniz. Bu "kayıt dosyası" kullanımıyla ilgili güzel bir örnektir. Kullanıcıların bilgisayara giriş yapabilmelerini sağlayan sistem programları giriş ve çıkış yapılan tarihleri "/var/adm/wtmp" adlı dosyaya kaydederler. "**last**" programı da bu dosyadan gerekli satırları okur ve ekrana yazdırır.

Kayıt tutması gereken bir program yazmak istediğimizi varsayalım. Daha önce hazırlamış olduğumuz "**fill**" adlı programı kullananların kayıtlarını tutmak istiyoruz. Yapmamız gereken aşağıdaki gibi bir kod yazıp bunu "**fill**" programından çağırmaktır.

```
#define LOG_FILE "/blugreen/homes/plank/cs360/notes/Logfile/fill_log_file"

write_to_fill_log()
{
  char *username;
  long t;
  FILE *f;

  username = getenv("USER");
  t = time(0);

  f = fopen(LOG_FILE, "a");
  /* hata kontrolu */

  fprintf(f, "%s %ld\n", username, t);
  fclose(f);
}
```

Dosyaları istediğiniz herhangi bir klasöre kopyalayıp çalıştırabilirsiniz. "**logfill1**" dosyasını çalıştırın ve "*/blugreen/homes/plank/cs360/notes/Logfile/fill_log_file*" dosyasında kaydınızın tutulup tutulmadığını kontrol edin.

Şu ana kadar bir sorun yok. Fakat eğer 2 kişi farklı proseslerde "**write_to_fill_log()**" metodunu aynı anda çağırırsa ne olacak? Sorun şu ki her iki proses de dosya sonunu aynı görecek ve sonuç olarak kayıt dosyasına aynı yolu girecekler. Bu durumun gerçekleşme olasılığının çok düşük olduğunu düşünebilirsiniz. Eğer böyle düşünüyorsanız problemi yoksayabilirsiniz. Fakat bu duruma bir çözüm getirmeniz gerekecek.

Çözüm yollarından birisi "**open()**" sistem çağrısının **O_SYNC** bayrağıdır. Detaylı bilgi için **man** sayfasını okuyun. Şimdi de aşağıdaki kodu deneyin.

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

#define LOG_FILE "/blugreen/homes/plank/cs360/notes/Logfile/fill_log_file2"

write_to_fill_log()
{
  char *username;
  long t;
  int fd;
  char s[100];

  username = getenv("USER");
  t = time(0);

  fd = open(LOG_FILE, O_APPEND | O_SYNC | O_CREAT | O_WRONLY, 0666);

  if (fd < 0) {
    fprintf(stderr, "Can't write log file %s\n", LOG_FILE);
    return;
  }

  sprintf(s, "%s %ld\n", username, t);
  write(fd, s, strlen(s));
  close(fd);
}
```

Atomik İşlemler

İşlemci tarafından çalıştırılırken bölünme ihtimali olmayan en küçük işleme *atomik işlem* denir. Sistem programlama alanında çok büyük önem arzederler.

Atomik işlemlere 2 örnek;

1. Dosya açıldığı zaman "**write()**" çağrısı (**O_APPEND | O_SYNC**)

Bir dosyaya yazmak istediğinizde 2 şey gerçekleşir. **Lseek**() dosya sonuna taşınır ve yazma işlemi gerçekleşir. Eğer bu iki işlem gerçekleşmemişse dosya sonu başka bir proses tarafından değişmiş olabilir. O yüzden işletim sistemi tarafından bu iki işlemin atomik olması garanti edilmelidir.

2. Bir dosyayı sadece mevcut değilse açmak.

F1 adlı dosyayı sadece eğer öyle bir dosya yoksa açmayı istediğinizi varsayalım. Yazmanız gereken kod aşağıdaki gibi olacaktır.

```
int fd;

    fd = open("f1", O_WRONLY);
    if (fd < 0) {
      fd = open("f1", O_WRONLY | OCREAT, 0644);
    } else {
      close(fd);
      fprintf(stderr, "Hata: f1 mevcut\n");
      exit(1);
    }
```

Bu kod **f1** dosyasının sadece mevcut olmadığı durumlarda açılmasını garanti eder mi? Cevap, kod segmentinin atomik olduğu durumlar için evettir. Aksi takdirde işlemci "**fd = open()**" ve "**if**" arasında bir kesmeye uğrarsa bu esnada başka bir proses **f1** dosyasının durumunu değiştirebilir. Bunu C dilinde garanti etmenin bir yolu olmadığından dolayı Unix bize bu iş için özel bir yol sunmuştur;

```
fd = open("f1", O_WRONLY | O_CREAT | O_EXCL, 0644);
```

Bu kod dosyanın var olup olmadığını kontrol eder ve eğer dosya yoksa 0'dan büyük bir değer döndürür.

Bu ve bilgisayar mühendisliğinin başka alanlarında "**X otomatik olarak çalıştırılıyor**" gibi cümleler duyabilirsiniz. Bu x'in hiç bir kesmeye uğramadan çalışması gerektiğini ifade eder. Kendi isteğinize

göre atomik işlem oluşturamayacağınızı unutmayın. Başka bir deyişle işletim sistemine bir kod bloğunun kesintiye uğramadan çalışmasının gerektiğini söyleyemiyoruz. Onun yerine yukarıda da örnek olarak verdiğimiz gibi işletim sisteminin bize sunduğu bazı atomik işlemler vardır. İleriki konularda kendi atomik işlemlerimizi tanımlayacak duruma gelebiliriz.

Kitabın 3.11 bölümü de atomik işlemlerden bahsetmektedir.

Umask

Kitabın 4.8. bölümünu ve umask ile ilgili man sayfasını okuyun.

umask() işlemin dosya oluşturma maskesini ayarlar ve

maskenin önceki değerini döndürür. Düşük sıralı 9

maske biti bir dosya oluşturulduğunda kullanılır, dosya giriş izinlerinde bunlarla ilgili bitleri temizlenir. (Bkz.

durum(2V)). Bu bit silme işlemi varsayılan erişimi kısıtlar.

Maske alt işlemler tarafından kalıtım alır..

Bir programdan umask ı çağırdınızda, veya bir kabuktan, "Dosya oluşturma maskesi" ni değiştirir. Bu maske 9 bit içermektedir. Bir dosya oluştuğunda, örnek olarak open() ile, creat(), veya mkdir(), ve

m in belirtilen bir moduna göre, daha sonra dosya o mod ile oluşturulur.:

(m & ~umask)

Umask sistem çağrıları umask ın eski değerlerini döndürür.

Örneğin, aşağıdaki programa bakalım (um1.c):

```
main()
{
  int i;
  int old_mask;
```

```
  old_mask = umask(0);
  i = open("f1", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  close(i);
  printf("olusturuldu f1: 0666\n");
  i = open("f2", O_WRONLY | O_CREAT | O_TRUNC, 0200);
  close(i);
  printf("olusturuldu f2: 0200\n");

  umask(022);
  i = open("f3", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  close(i);
  printf("olusturuldu f3: %o\n", 0666 & ~022 & 0777);
  i = open("f4", O_WRONLY | O_CREAT | O_TRUNC, 0777);
  close(i);
  printf("olusturuldu f4: %o\n", 0777 & ~022 & 0777);
  i = open("f5", O_WRONLY | O_CREAT | O_TRUNC, 0200);
  close(i);
  printf("olusturuldu f5: %o\n", 0200 & ~022 & 0777);
}


UNIX> um1
olusturuldu f1: 0666
olusturuldu f2: 0200
olusturuldu f3: 644
olusturuldu f4: 755
```

```
olusturuldu f5: 200
UNIX> ls -l f?
-rw-rw-rw-  1 plank          0 Eyl 28 15:05 f1
--w-------  1 plank          0 Eyl 28 15:05 f2
-rw-r--r--  1 plank          0 Eyl 28 15:05 f3
-rwxr-xr-x  1 plank          0 Eyl 28 15:05 f4
--w-------  1 plank          0 Eyl 28 15:05 f5
UNIX>
```

Umask değeri her işlemle ayarlanır, her kullanıcıyla değil. Kabuğunuzun umask ı 022 ise, ve 0 olarak umaskı ayarlanmış bir programınız varsa , bu kabuğunuzu etkilemeyecektir:

```
UNIX> cat um2.c
main()
{
  umask(0);
}
UNIX> umask
22
UNIX> um2
UNIX> umask
22
UNIX>
```

Rastgele Dosya/Inode Sistem Çağrıları.

- chmod(char *path, mode_t mode) -- Kabuktan uyguladığınızda sadece chmod gibi çalışır. Örn. chmod("f1", 0600) f1 dosyasının rw- korumasını ayarlayacaktır senin için, ve --- herkes için.

Daha detaylı ve içerikli bir açıklama için 4.9. bölümü okuyun

Ayar bitlerini S_IUSR gibi kullanma, Örn.

- chown() -- ignore.

- link, unlink, remove, rename: Kitabın 4.15. bölümü.

straightforward: link(char *f1, char *f2) works just like

UNIX> ln f1 f2

f2 has to be a filename though, and cannot be a directory.

unlink(char *f1) aşağıdaki kod gibi çalışır

UNIX> rm f1

remove(char *f1) unlink gibi çalışır, fakat hem de boş dizinlerde de çalışır. unlink ise boş dizinlerde hata verir.

rename(char *f1, char *f2) aşağıdaki komut gibi çalışır.

UNIX> mv f1 f2

- symlink ve readlink: 4.17. bölümü okuyn. Bu yordamlar sembolik bağlarla uğraşmaz.

- utime: 4.19. bölümü okuyun. Bu yordamlar size bir inode dosyasının zaman alanını değiştirmenize izin verir. Bu sistem çağrıları yasal değiş gibi gözüküyor

(örnek, Bir program yazılarak bir ödevi zamanında bitmiş gibi gösterilebilir...), ancak çok kullanışlı,

özellikle tar yazmak için (ve jtar).

- mkdir ve rmdir: straightforward ve benzerleri aşağıdaki gibidir.

UNIX> mkdir ...

UNIX> rmdir ...

4.20. bölümü okuyun.

- chdir, getcwd: Benzeri aşağıdaki gibidir.

UNIX> cd ..

UNIX> pwd

4.22. bölümü okuyun. jtar için buna ihtiyaç duymazsınız, fakat istediğiniz takdirde kullanabilirsiniz.