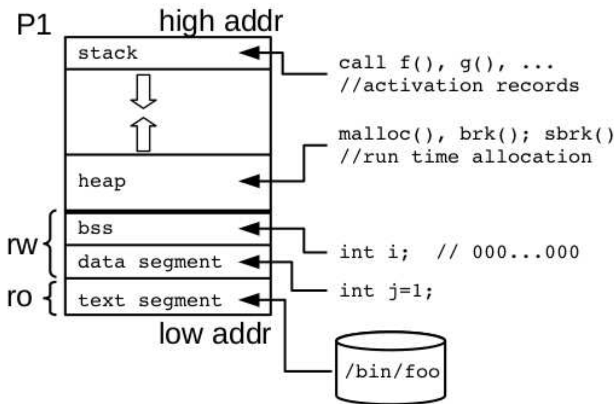
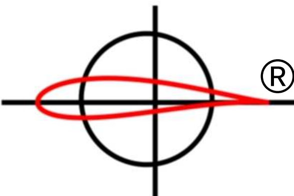
Linux altında Sistem Programlama Lab. Çalışması Notları

http://UCanLinux.Com

16 Mart 2018




```c
#include <stdio.h>
#include <stdlib.h>
#include <linux/netlink.h>
#include <sys/socket.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>

int main(int argc, char *argv[]){

   unsigned int        len;
   unsigned int        i;
   int                 res;
   int                 fd;
   char                buf[4096];
   struct sockaddr_nl nls;
   char                c;

   fd= socket(PF_NETLINK,  SOCK_RAW,NETLINK_KOBJECT_UEVENT);

   if (fd == -1){
      return EXIT_FAILURE;
      }

   memset(&nls, 0, sizeof(nls));

   nls.nl_family= AF_NETLINK;
   nls.nl_pid    = getpid();
   nls.nl_groups= 1;

   res = bind(fd, (struct sockaddr *)&nls, sizeof(nls));

   if (res == -1){
      return EXIT_FAILURE;
      }

   for(;;){
      len= recv(fd, buf, sizeof(buf), 0);

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

         c= buf[i];

         if (c == '\0') {
            fprintf(stdout, "\\0\n");
            continue;
            }

         if ( isprint(c) ){
            fprintf(stdout, "%c", c);
            continue;
            }

         fprintf(stdout, "0x%02X", c);

         } // for i

      fprintf(stdout, "\n");

      } // for ;;

   if ( close(fd) == -1 ){
      perror("close()");
      return EXIT_FAILURE;
      }

   return EXIT_SUCCESS;
}
```



Nâzım KOÇ

İçindekiler

1 Giriş **2**

2 Çalışma Ortamı ve İçerik **4**

3 Sistem çağrılarına giriş **8**

4 memory map **14**

5 shared memory **19**

6 Posix message queue **25**

7 Semaforlar **30**

8 Sinyaller **36**

9 Threads **43**

10 tcp/ip client/server **53**

11 tcp/ip server **60**

11.1 İteratif sunucu 61

11.2 Multithread sunucu 62

11.3 Proses bazlı sunucu 64
11.4 I/O multiplexing 65

12 UDP Soketleri **69**

13 Unix alanı soketleri **71**

14 En basit kernel modülü **72**

15 /proc file system örneği **75**

16 Statik kütüphaneler **79**

17 Paylaşımlı kütüphaneler **83**

18 plugins **96**

19 pkg-config **98**

20 pipe open **102**

21 daemon **104**

22 nice **110**

23 logger, sistem kayıtçısı **114**

24 udev **118**
24.1 Modüller'in otomatik yüklenmesi 131

25 Alarm sinyali ile zamanlama **134**

26 Posix Alarm **136**

27 fd destekli alarm sistemi **140**

28 pipes, unnamed **143**

29 pipe in shell **145**

30 Makefile **147**

31 -g ile derlemek **151**

32 Bellek Sızıntılarının Tespiti **153**

33 inode **155**

Şekil Listesi

3.1 Syscall and wrapper function 11

4.1 memory mapping 17

4.2 memory mapping, offset 18

5.1 mmap, genel görüntü 23

5.2 mmap, shmem ile kullanımı 24

6.1 Posix message queue 29

7.1 Üçlük semafor örneği 34

7.2 Mutex 35

7.3 Temel semafor işlemleri 35

8.1 Bir sinyalin işleme mantığı 38

8.2 Sinyallerin kümelenmesi 40

9.1 Threads 44

10.1 İstemci soketinin kurulması 57

10.2 Pasif soketin kurulması 58

10.3 Bağlantının kabul edilmesi 59

11.1 select ve makroları 68

20.1 pipe open 103

22.1 Recep ile Kibarlık Budalası 113

23.1 Sistem kayıtçısı 117

24.1 Hot plug sistemi 119

26.1 Posix alarm, timer_settime() 138
26.2 Posix alarm, timer_create() 139

28.1 pipe in C, unnamed 144

33.1 ext2 inode yapısı 157

Bölüm 1

Giriş

Bu belge 4 günlük Linux altında sistem programlama eğitiminin labratuvar belgesidir. Doğudan uygulama belgesi olduğu için çok az yazı içermektedir.

Sistem programlama konuları seçilirken, gerçek hayat problemleri ve çözümleri dikkate alınmış, mümkün olduğu kadar teorik veya akademik anlatımdan uzak durulmuş, daha çok kavramlar üzerinde çalışılmıştır.

Eğitimin esas ağırlığını sistem çağrıları teşkil etmektedir. Sistem çağrılarında da işletim sistemine bağımlılığı en aza indirmek için daha çok POSIX tabanlı sistem çağrılarına öncelik verilmiştir. Makefile yazımı, otomatik modül yükleme tekniği, udev gibi konulardan da bahsedilmektedir. Ayrıca kernel ve proc modülü yazımı gibi çekirdek tarafı ağır olan konulara çok kısaca değinilmiştir. tcp/ip tabanlı client ve server geliştirme konuları örnekleri ile genişçe açıklanmıştır. Ayrıca çıkışların renklendirilmesi ve dialog gibi bazı konular da araya serpiştriilmiştir.

Bu belge içinde bulunan bilgisayar programları, betikler ve Makefile dosyaları anlatımı kolaylaştırmak için programlama tekniklerine uygun olmayan şekilde yazılmıştır.

Örneğin pek çok sistem çağrısının dönüş değeri kontrol edilmemiş, pek çok yerde sabit kullanılmış, bolca global değişken kullanılmış, main() içine kod yazılmış, açıklamalar neredeyse hiç yazılmamıştır.

Anlatımı basitleştirmek için pek çok kodlama kuralı gözardı edilmiştir. Buradaki programlar kullanılacağı zaman mutlaka daha kaliteli yazılmalı ve buradaki yazım tekniği asla örnek alınmamalıdır.

Bu belgedeki çalışmaları yapabilmek için, development tool yüklü, standard bir Linux dağıtımı yeterlidir.

Telif Hakları

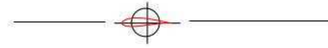
Bu belge[1] Linux altında sistem programlama eğitiminin labratuvar belgesidir.

Bu belgenin bütün telif hakları Nâzım KOÇ'a aittir. Bu belgenin tümü veya bir kısmı aşağıdaki şartlar sağlandığı takdirde hiç bir izne gerek kalmadan, her türlü ortamda çoğaltılabilir, dağıtılabilir, kullanılabilir.

1. `http://UCanLinux.Com` kaynak gösterilmelidir.
2. Yazı ve resimler üzerinde güncelleme yapılmamalıdır.

Bu belgenin en son sürümü `http://UCanLinux.Com` adresinden indirilebilir.

Bu belge ile ilgili her türlü bilgi, eleştiri ve yorum için `http://UCanLinux.Com` sitesindeki iletişim bilgileri kullanılabilir.



[1] file:/nk/workspace/projects/linux.egitimi/sysprog/latex, 16 Mart 2018

Bölüm 2

Çalışma Ortamı ve İçerik

Linux altında programlama veya Linux altında sistem programlama,

Süre ve Gereklilikler

- Eğitim 4 gün sürecektir.
- Bütün konularda uygulama yapılacaktır.
- Verimli bir eğitim için kara ekran ve C tecrübesi gereklidir.
- Ubuntu veya standard bir dağıtımın yüklü olması gereklidir. Dağıtım içinde C geliştirme ortamının yüklü olması gerekir.
- Bütün çalışmalar kara ekrandan yapılacaktır. Zehirli GUI'ler kullanılmayacaktır.

İçerik

- POSIX Sistem Çağrıları esas konumuzdur.
  Sistem çağrısı, genelde C ile, standard C kütüphanesi yardımı ile, Linux çekirdeği ile iletişim kurmaktır. Sistem çağrıları ile IPC, soket programlama, proses'ler, dosya sistemleri, vs üzerinde çalışılır. Genelde POSIX tabanlı anlatım yapılacaktır. Sonuçta bahsedilecek konular bütün POSIX tabanlı işletim sistemlerinde geçerli olacaktır.
- Kütüphaneler
  Kuruluş ve kullanım yöntemleriden bahsedilecektir.

- Kernel Modül Programlama
  Genel yapı ve proc dosya sisteminden bashsedilecektir.

- Makefile
  Temel kavramlar verilecektir.

- Man sayfaları,
  Her daim başvuru yapılacaktır.

- Kabuk altında programlama ve bazı Linux kavramları Yeri geldikçe bahsedilecektir.

Çalışma Ortamı

Belge ile birlikte gelen sysprog.tar.gz paketi, /opt/sysprog altına aşağıdaki gibi açılabilir. Farklı bir dizine açılmaması tavsiye edilir.

```
$ sudo -s
$ cd /
$ tar zxvf /tmp/opt_sysprog.tar.gz # tar.gz adı farklı olabilir.
$ cd /opt
$ id
$ sudo chown nazim:nazim -R sysprog  # herkes kendi id değerini girmelidir.
$ exit
```

POSIX Nedir?

POSIX Portable Operating System Interface for Unix

Programların,

- kaynak kod seviyesinde,
- platformlar arası taşınabilirliğini sağlamak için
- IEEE tarafından geliştirilen bir standarttır.

POSIX'in Amacı,

- Programlar arasında API birliği, read/write/stat/...
- Kabuktan girilen komutların yazım standardı, ls -l, tar xf foo.tar

- Dizin isimleri, /, /dev/console, /dev/null, /sys, /tmp
- Dosya isimleri ve yetkilendirmeler,
- Düzenli ifadeler,
- ...

POSIX kelimesinin sonundaki X, bazen Unix diye çevrilir ama genelde kozmetiktir. Unix ile ilgili sistemlerin veya programların sonuna genelde X eklenir. Linux, AIX, OSX, ...

Linux, Unix-Like işletim sistemidir ve Unix sertifikası(Single UNIX Specification) yoktur. Ama büyük oranda POSIX uyumludur.

Ayrıca POSIX sadece Unix benzeri işletim sistemleri için geçerli değildir. Daha geniş bir aileye hitap eder.

POSIX sertifikalı bazı sistemler

- Apple, OSX
- IBM, AIX
- HP, UX
- Oracle, Solaris (sonunda X yok :)
- Windows (sadece bazı sürümleri)

Linux dağıtımları, resmi olarak POSIX uyumlu veya UNIX değildir. Kimsenin de pek umurunda değildir. Linux kendi dağıtımları arasında Linux Standard Base (LSB) ile uyum sağlamaya çalışır.

man sayfalarında "CONFORMING TO" kısmına bak, `$ man 2 mkdir stat`

POSIX.1 Sistem çağrıları ve C kütüphanesi için API tanımlar.

POSIX.2 Kabuk özellikleri, çevre değişkenleri, PATH özellikleri ve isimleri, derleyici kullanımı ve özellikleri, ...

Linux her telden çalar! POSIX, Single Unix Specification(SUS), BSD, SVr4, ...

Bu belgeki konular hiyerarşik bir yapıda düzenlenmemiştir. Bir konu diğerinden bağımsızdır. Her bir konu bağımsız olarak çalışılabilir.

Yine de bahsedilecek konular aşağıdaki gibi gruplanabilir.

sistem çağrıları Tanım ve özellikleri.

ipc shared memory, queues, semaphors, signals

threads Kavram ve uygulamları.

tcp/ip client server yazımı, eş-zamanlı sunucular.

udp/ip udp tabanlı istemci ve sunucular.

unix domain unix tabanlı soketler.

kernel basit modül ve proc modülleri yazımı.

libs statik ve dinamik kütüphaneler, plugins, pkg-configs.

memory brk, mmap, realloc

shells kabuk yazımı, popen, daemon kavramı, notify, nice kullanımı.

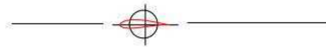
logs logger, logrotate.

udev Çalışma mantığı, kural yazımı, otomatik modül yükleme.

alarm Üç farklı zamanlayıcı kuruluşu.

piped C ve kabukda pipe kullanımı.

ortaya karışık exec-tar, makefile, yetkiler, memory layout, yeni sistem çağrısı yazmak, C ön işleyicisi, renklendirme, metin tabanlı dialog, bellek sızıntılarının bulunması, inode kavramı, session kavarmı.



Bölüm 3

Sistem çağrılarına giriş

Sistem çağrısı nedir? Çekirdekteki herhangi bir giriş noktasına sistem çağrısı denir. Giriş noktaları genelde fonksiyonlar ile tanımlanır.

Sistem çağrışları, user space tarafında çalışan programların, çekirdekten ihtiyaç duyduğu hizmeti almasını sağlar.

User space tarafında bir sistem çağrısı yapıldığında çekirdekteki belirli bir adres'e (entry point) gidilir. Bu adres hemen hemen her zaman bir fonksiyon adresidir.

Java gibi üst düzey platform kullanılıyorsa sistem çağrıları gerekli değildir. Platform gerekli soyutlamayı sağlar.

Sistem çağrılarını bilmek ve doğrudan kullanmak projenin verimini ve kalitesini artırır.

Sistem programlama sadece sistem çağrılarından oluşmaz ama sistem çağrıları sistem programlamanın temelini teşkil eder. Sistem progralamada uygulama ile işletim sistemi arasında hemen hemen hiç bir katman yoktur.

Linux sistem çağrıları 2 numaralı man sayfalarında anlatılır.

```
$ man 2 stat

$ man stat  # man 1'deki stat gelir, genelde wrapper fonksiyondur.

$ man -f stat # içinde stat geçen bütün man sayfalarını listeler.
```

Sistem çağrılarının pek çoğu aynı isimli C çağrıları ile yapılır. (C library

wrapper function) Böylece pek çok sistem çağrısı standard bir C çağrısı gibi yapılır.

Ekrana merhaba yazan program ve sistem çağrılarının izlenmesi.

```
$ make clean
$ make P=1

$ ./prog1

$ strace ./prog1

$ strace  -e trace=write ./prog1

# Seçenekler

# -t her satıra saniye bilgisini yaz
# -t -t her satıra mikro saniye bilgisini yaz
# -T çağrının harcadığı süre
# -e trace=network   # bütün network çağrıları
# -e trace=file   # bütün dosya çağrıları
# -e trace=open,close,write  # belirli dosya çağrıları
# -o fileName çıkışların kaydedilmesi

# istatistik almak
$ strace -c ./prog1
```

Daha önce, strace yazılmadan başlatılmış bir program -p ile izelenebilir.

Örnek kabuktaki sistem çağrılarının izlenmesi

```
# Terminal 1
$ echo $$
20240

# Terminal 2
$ sudo -s
$ strace -p 20240

# Terminal 1
$ pwd

Terminal 2'den sistem çağrılarını izle.
ctrl+c ile çık.
```

trace tehlikeli bir işlemdir, root yetkisi gerekir. Kullanım yetkileri /proc-

/sys/kernel/yama/ptrce_scope içinde mevcuttur. Bu değer 1'dir, 3 yapılırsa uygulama trace ile izlenemez.

```
$ sudo -s
$ echo 3 > /proc/sys/kernel/yama/ptrace_scope

$ strace ./prog1
strace: ptrace(PTRACE_TRACEME, ...): Operation not permitted
+++ exited with 1 +++

$ sysctl -w kernel.yama.ptrace_scope=0   # bug
```

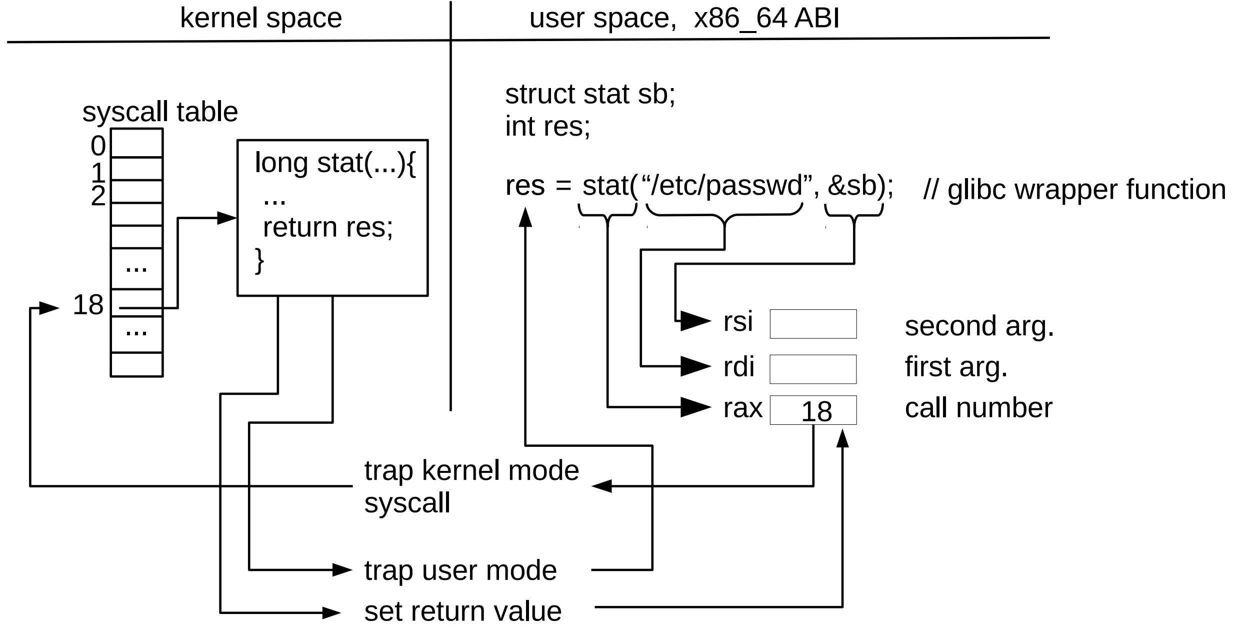
Açılışta /etc/sysctl.d/10-ptrace.conf içinde 1 olarak tanımlıdır. `$ strace prog` çalışır ama sonradan -p ile attach olmaya izin vermez.

stat sistem çağrısı

```
$ make P=4
$ ./prog4
$ strace ./prog4    # stat ve fstat gör.
```

Daha önceden başlamış bir programın sistem çağrıları -p seçeneği ile takip edilebilir.

```
$ ps -e  # pid bul.
$ sudo strace -p PID
```



Şekil 3.1: Syscall and wrapper function

Sistem çağrılarının işleme mekaznizması genelde aşağıdaki gibidir.

- argümanların ve çağrı numarası ilgili register'lara yüklenir. Her abi'de argüman transferi, syscall numarası ve dönüş değeri için farklı register'ler kullanılabilir. Her mimarinin ABI uygulamasında argümanların kernel'a nasıl geçirileceği verilmiştir. `$ man 2 syscall`
- Kernel moda interrupt yardımı ile geçilir.
- Kernel tarafında fonksiyon yürütülür. Hata durumunda genelde negatif sayı döner. Bu sayı wrapper tarafından pozitif yapılır ve ilgili dönüş registerine kaydedilir.
- user moda geçilir.
- wrapper fonkisyon, hata durumunda, sistem çağrısının dönüş değerini errno içine atar. User space tarafındaki dönüş değerini de genelde -1 yapar. Hata numaraları man 3 errno ile incelenebilir.
- sistem çağrısı tamamlanır.

Sistem çağrıları için yukarıda anlatına durum her zaman geçerli olmayabilir. Özel durumlar çok azdır ve man sayflarına bakılmalıdır.

Genelde 0 veya sıfırdan büyük dönüşler başarılı kabul edilir.

Her sistem çağrısının man sayfası MUTLAKA okunmalıdır.

Bütün sistem çağrılarının listesi aşağıdaki gibi elde edilebilir.

$ man 2 syscalls

Her sistem çağrısının wrapper fonksiyonu olmayabilir. Bu durumda ilgili sistem çağrısı doğrudan syscall() komutu ile numarası verilerek çağrılır.

C tarafında mapper fonksiyon yoksa syscall() kullanılmalıdır.

syscall() fonkisyonunun tanımı aşağıdaki verilmiştir.

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */
long syscall(long number, ...);

$ man 2 syscall
```

Örneğin gettid() sistem çağrısı thread id değerini verir ve wrapper fonkisyonu yoktur. `$ man 2 gettid`

multithread uygulamalarda bütün thread'ler aynı pid'e sahiptir. Fakat her biri farklı bir thread id değerine sahiptir. Thread yoksa pid ile tid aynı değere eşittir.

prog2.c örneğinde görüleceği gibi, bazı sistem çağrılarını kullanabilmek için "feature test macro" kullanmak gerekir.

Gerekli makrolar man sayfalarında SYNOPSIS başlığı altında verilmektedir.

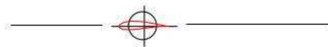
Bu makrolar include dosyalarının en üstünde tanımlanmalıdır.

```
$ man 3 strdup

$ man 7 feature_test_macros
```

Man sayfasındaki örnek kod için bakınızı prog3.c

```
$ make P=3
$ ./prog3
```

Kaynak: man 2 intro



Dizin syscall-intro/

Bölüm 4

memory map

Elimizde bir dosya mevcut. Bir veya daha fazla proses aynı dosyanın bir kısmını dinamik olarak işleyecek.

File operations (open, close, read, write, ...) ile dosyayı işlemek çok verimsiz olacaktır.

Dosyanın işlenecek kısmı belleğe (virtual address space) taşınır. Bu işleme memory map denir.

p= mmap(); çağrısında, dosyanın bir kısmı belleğe taşınır ve bellek başlangıç adresi p'dir. p bilinen yollarla işlenir.

msync() ile bellekte bulunan güncel veri geriye, dosyaya yazılır. Dosya fd ile temsil edilir.

Bellek adresi NULL ise, uygun adresi kernel seçer. Tavsiye edilir.

protocol argümanları EXEC, READ, WRITE, NONE ile open() argümanları uyumlu olmalıdır.

Bellek adresi verilmişse ve PAGESIZE katı değilse, kernel page başlangıcına en yakın adres seçilir.

Mapping ikiye ayrılır

file mapping fd ile ilişkilidir.

anonymous mapping fd değeri mevcut değildir. Yani açılmış bir dosya yoktur. Bu durumda ayrılan bellek alanı 0'lar ile doldurulur. fd olmadığı için msysnc() özelliği de olmayacaktır.

bash

```
$ getconf -a
$ getconf PAGESIZE
```

C

```
int sysconf (_SC_PAGESIZE);
```

PAGESIZE değerinin tam katı nasıl yapılır?

offset değeri kernel page size değerinin tam katı olmalıdır. offset'i PAGESIZE değerinin tam katı yapmak için pratik yol

`offset & ∼(PAGESIZE - 1)`

offset'i 4096'nın katından başlatmak için, örnek

```
  PAGESIZE   =       4096
  PAGESIZE   = 0x00001000
  PAGESIZE-1 = 0x00000FFF
~(PAGESIZE-1)= 0xFFFFF000
```

offset & 0xFFFFF000 ifadesi, offset adresinin son 12 bitini 0 yapar ve adresin tam 4096'nın katından başlamasını garantiler.

Bakınız $ man 2 mmap

NONE memory protection değeri NONE atanırsa, bu bölgeye her türlü erişim engellenir.

Özellikle interpreter uygulamalarında ya da stack gibi belirli bir sınır değeri olan uygulamalarda sınır adresine, protected map atanır. Böylece gözü kapalı bir biçimde push işlemi veya artım işlemi yapılabilir. Her işlemden önce

"acaba yer var mı?" diye kontrol etmeye gerek kalmaz. Eğer sınır aşılırsa bu alan korunduğu için kernel tarafından sinyal üretilir.

shared update'ler prosesler tarafından anında görülür. Fakat bu update'ler dosya üzerinde henüz yapılmamış olabilir. msync() veya munmap() yapılırsa, update'ler dosya üzerinde de yapılır.

private Aynı dosyayı map eden diğer prosesler update'leri göremezler.

anonymous Ayrılan belleğin fd ile yani bir dosya ile ilişkisi yoktur. Sonuçta bellekte yapılan güncellemeler dosyaya yazılmaz. Map alanı 0'lar ile doldurulur. fd ve offset değeleri anlamsızdır.

Sorun mmap() yapıldı, dosyanın bir kısmı bellekte işleniyor. Bu arada farklı bir biçimde dosyanın bu bölümü güncellendi. Bu durumda belirsizlik oluşur, asla yapılmamalıdır.

Tam katı alan yoksa?

Dosyalar her zaman kernel page size değerinin tam katında map edilir. Tam katı değilse, buçuklu kısım 0 ile doldurulur. Bu fazla kısım dosyaya geri yazılmaz.

Sorun map sırasında dosya boyu değişirse geriye yazım belirsizdir. Yapılmamlıdır.

unmap ve close proses çıkışında unmap otomatik olarak yapılır. close(fd) yapılmış olsa bile, unmap ayrıca yapılmalıdır.

MMAP_THRESHOLD glibc'nin malloc() fonksiyonu srbk() sistem çağrısı ile heap alanından yer alır. Fakat ayrılacak alan MMAP_THRESHOLD değerinden büyükse, anonymouse mmap ile yer ayrılır. Çünkü free list bu kadar büyük bir alanı tutacak yapıya sahip değildir.

MMAP_THRESHOLD için varsayılan değer 128K'dır. Bu değer mallopt(3) ile güncellenebilir.

mallopt - set memory allocation parameters

mmap() her zaman sbrk()'dan daha yavaştır. Çünkü bütün alanı 0 ile doldurur.

Test 1

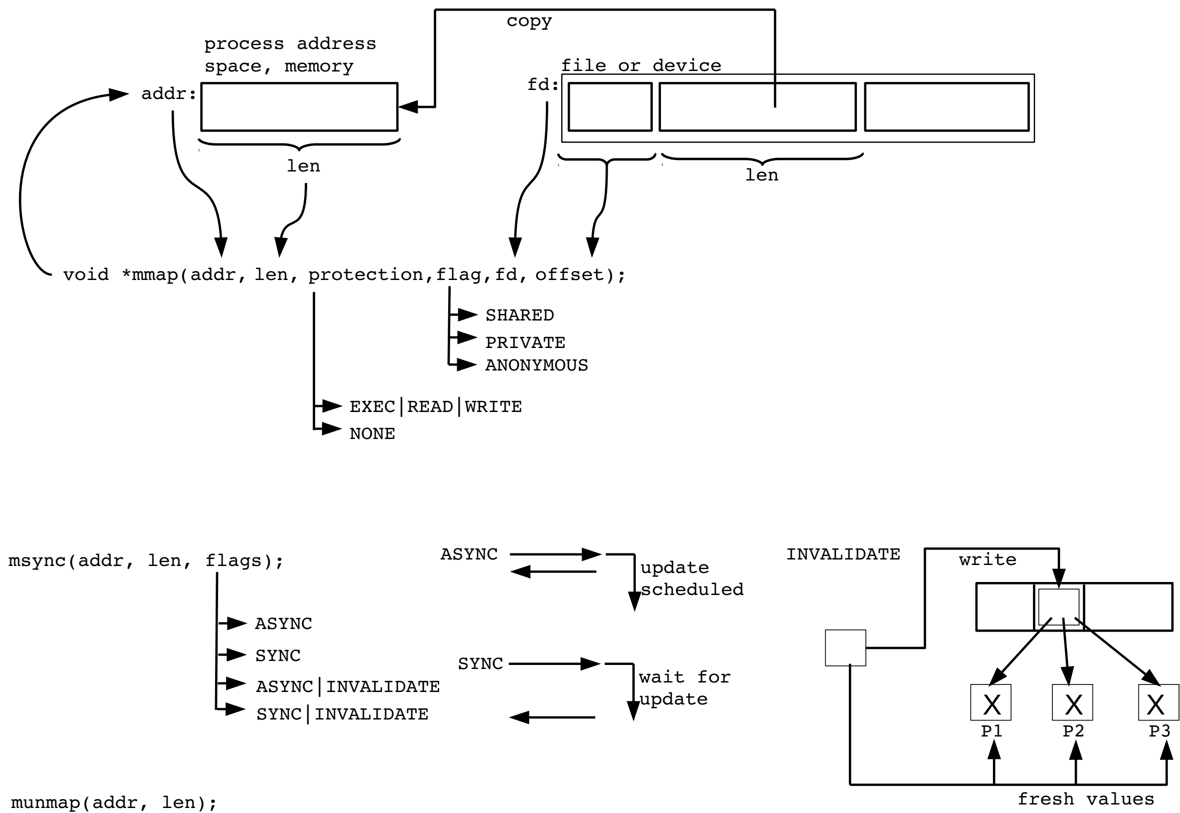
```
$ ./prog1
$ cat test.file # başka terminalden
```

Test 2

```
$ ./prog2
$ cat test.file
```

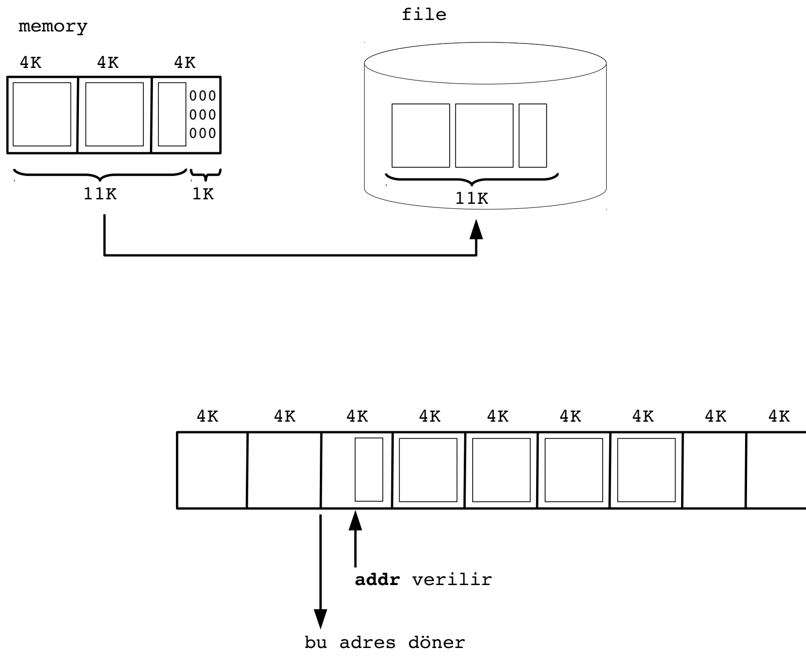
Test 3

```
$ ./prog2       # güncelle
$ ./prog3       # güncel içeriği göster
$ cat test.file # dosya sisteminden bak
```

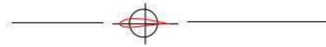


Şekil 4.1: memory mapping

Dizin alloc_mmap/



Şekil 4.2: memory mapping, offset



Bölüm 5

shared memory

Birden fazla prosesin kullanabileceği ortak alana shared memory denir. Bu alan shm_open() ile açılır, standard fd döner. Bu fd close() ile kapatılır.

shm_unlink() ile tutulan bütün kaynaklar ve ortak bellek serbest bırakılır.

Ortak bellek mmap() yardımı ile kullanılır.

rt kütüphanei içindedir, -lrt ile link edilmelidir.

Kuruluş

```
$ make P=1
$ ./prog1

$ ls -l /dev/shm/ortak_bellek
-rw------- 1 nazim nazim 1024 Ağu 21 20:46 /dev/shm/ortak_bellek
```

Paylaşılan bellek standard bir dosyadır. Dışarıdan müdahale ile güncellenebilir ama anlamsızdır, yapılmamalıdır.

mmap kavramı

I/O işlemlerinin read/write yerine doğrudan bellekte yapılmasını sağlayan mekanizmaya memory map denir.

mmap'in doğrudan shared memory ile ilgisi yoktur. mmap, üretilmiş bütün fd'ler üzerinde işler. Örneğin config dosyaları veya /etc/passwd gibi dosyların okunması veya değerlendirilmesi genelde mmap() ile yapılır. Böylece satır satır i/o yapmak yerine, veriler bir bütün olarak belleğe aktarılır.

mmap() sırasında, PRIVATE bayrağı açıksa, güncellemeleri diğer prosesler göremez.

SHARED bayrağı açıksa, güncellemeler bütün prosesler tarafında görülür.

mmap() ilk adresi her zaman NULL seçilmelidir. Bu durumda, işletim sistemi uygun bir adres döndürür.

NONE ile ilgili alana erişim tamamen kapatılabilir. Bu durumda ilgili adrese her türlü erişim hataya sebep olacaktır. Özellikle derleyici tasarımında belirli adres bloklarını sınırlandırmak için bu özellik çokça kullanılır.

mmap()'de yapılan değişiklikler her zaman dosyaya yansıtılmaz. Aslında munmap() kullanılanan kadar, verilerin dosyaya yazılacağının bir garantisi yoktur. msync() komutu, bellekte yapılan güncellemelerin dosyaya yazılmasını garantiler.

fd'yi kapatmak ilginç bir biçimde, unmap() sağlamaz.

Yazma

```
$ make P=2

$ ./prog2
Alfabe ortak belleğe yazıldı.
```

Örnek program alfabedeki büyük harfleri ortak belleğe yazar ve sonuna \n karakteri ekler.

Artık bu alan başka prosesler tarafından map edilerek kullanılabilir.

Okuma

Bir proses tarafından yazılan bilgiler okunmuş ve ufak harflere çevrilerek ekrana yazılmıştır.

```
$ make P=3

$ ./prog3
abcdefghijklmnopqrstuvwxyz
Alfabe ortak bellekten okundu.
```

Dışarıdan el yordamı ile okuma

```
$ cat /dev/shm/ortak_bellek
ABCDEFGHIJKLMNOPQRSTUVWXYZ

$ dd if=/dev/shm/ortak_bellek
ABCDEFGHIJKLMNOPQRSTUVWXYZ
2+0 records in
2+0 records out
1024 bytes (1,0 kB, 1,0 KiB) copied, 9,8543e-05 s, 10,4 MB/s

$ hexdump -C /dev/shm/ortak_bellek
00000000  41 42 43 44 45 46 47 48  49 4a 4b 4c 4d 4e 4f 50  |ABCDEFGHIJKLMNOP|
00000010  51 52 53 54 55 56 57 58  59 5a 0a 00 00 00 00 00  |QRSTUVWXYZ......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
* 00000400
*
```

mmap yapıldığı zaman alanlar otomatik olarak 0 ile doldurulur. hexdump ile bu durum açıkça görülebilir.

Büyük alanlar için ilk değerin 0 atanması büyük vakit kayıplarına sebep olabilir.

Yok etme Ortak bellek için tahsis edilen kaynaklar "kernel persistence" dir. Yani kernel ayakta olduğu sürece, ortak alanı kullanana bütün prosesler kapanmış olsa bile, kaynaklar serbest bırakılmaz.

Bundan dolayı her map işlemi mutlaka unmap ile kapatılmalıdır.

Bütün map'ler kapatıldıktan sonra unlink ile kernel kaynakları serbest bırakılmalıdır.

Eğer tek bir map dahi aktif ise yani unmap yapılmamışsa, unlink yapılsa dahi, kaynaklar serbest bırakılmaz.

```
$ make P=4
$ ./prog4
/ortak_bellek silindi.
```

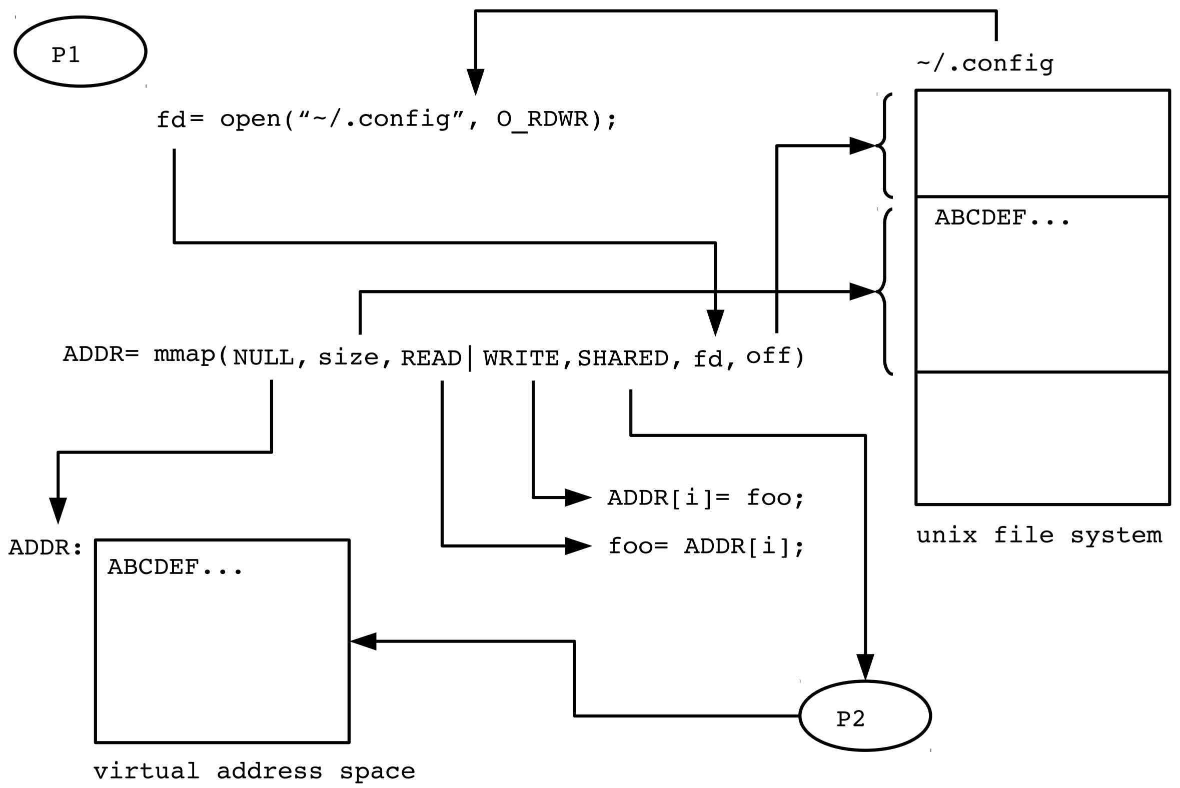
Kontrol et.

```
$ ls -l /dev/shm/ortak_bellek
ls: cannot access '/dev/shm/ortak_bellek': No such file or directory
```

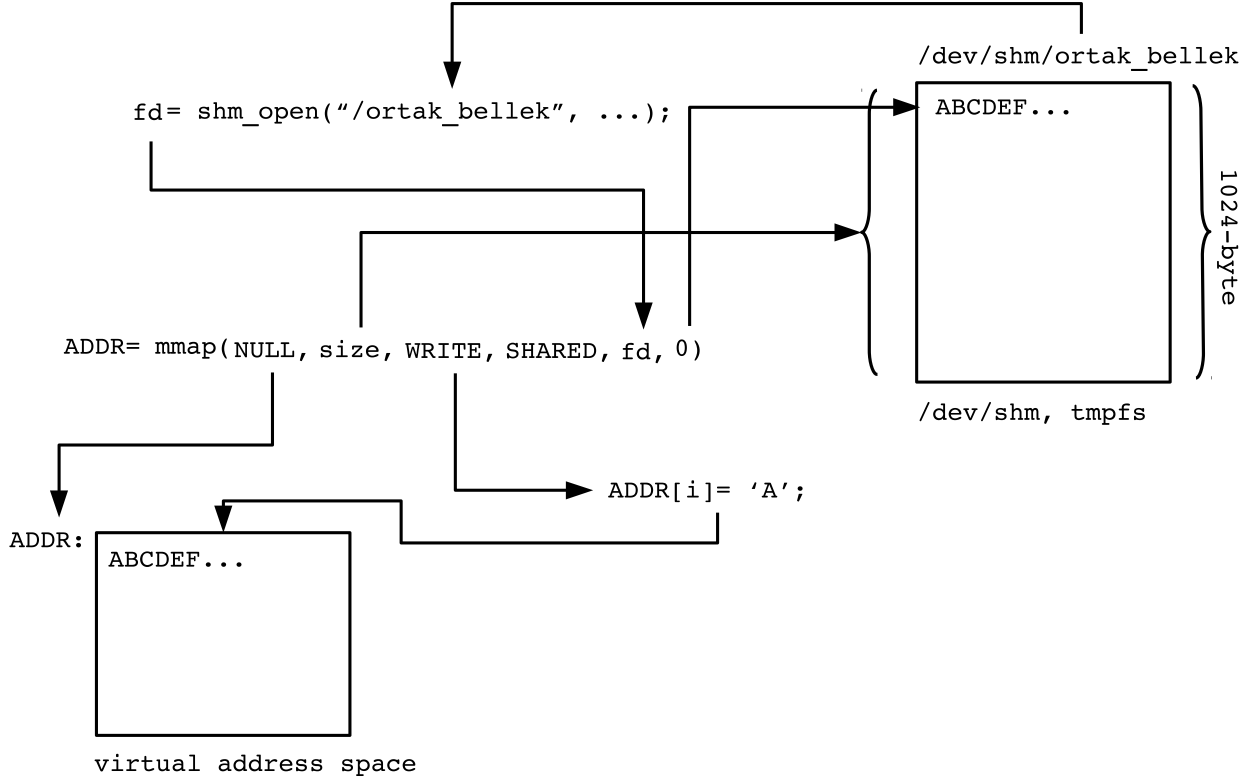
Örnek tmpfs dosya sistemi kuruluşu. Varsayılan kapasite fiziksel ram'in yarısıdır.

```
$ mkdir /tmp/disk
$ sudo mount -t tmpfs none /tmp/disk

nazim@nkoc:~$ df
Filesystem     1K-blocks      Used Available Use% Mounted on
udev             1946024         0   1946024   0% /dev
tmpfs             393260      6304    386956   2% /run
/dev/sda1       57541168  15276820  39318376  28% /
tmpfs            1966284     30220   1936064   2% /dev/shm
tmpfs               5120         4      5116   1% /run/lock
tmpfs            1966284         0   1966284   0% /sys/fs/cgroup
/dev/sda3      247891948 217266468  18010212  93% /nk
cgmfs                100         0       100   0% /run/cgmanager/fs
tmpfs             393260        80    393180   1% /run/user/1000
none             1966284         0   1966284   0% /tmp/disk

$ cd /tmp/disk

$ ls

$ df .
Filesystem     1K-blocks  Used Available Use% Mounted on
none             1966284     0   1966284   0% /tmp/disk

$ cd ..

$ sudo umount /tmp/disk
```

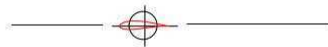
Dizin shmem/



Şekil 5.1: mmap, genel görüntü



Şekil 5.2: mmap, shmem ile kullanımı



Bölüm 6

Posix message queue

Prosesler arası mesaj aktarımı için kullanılır.

Örnek

Bütün öncelikler aynı

```
# server
$ make P=1
$ ./prog1

# client
$ make P=2
$ ./prog2
```

Ortalama aktarım süresi 50 mikrosaniye civarındadır.

Örnek

Keyfi öncelikler veriliyor

```
$ make P=3

$ ./prog1
pid 6389
/test üzerinden mesaj bekliyorum...
1504167434.755609: merhaba_000_164 <--
1504167434.755655: merhaba_007_253
1504167434.755663: merhaba_008_250
```

```
1504167434.755668: merhaba_006_242
1504167434.755672: merhaba_005_191
1504167434.755677: merhaba_002_163
1504167434.755681: merhaba_001_152
1504167434.755686: merhaba_009_122
1504167434.755690: merhaba_003_086
1504167434.755695: merhaba_004_084

$ ./prog3
1504167434.755402:  merhaba_000_164
1504167434.755560:  merhaba_001_152
1504167434.755569:  merhaba_002_163
1504167434.755575:  merhaba_003_086
1504167434.755581:  merhaba_004_084
1504167434.755586:  merhaba_005_191
1504167434.755591:  merhaba_006_242
1504167434.755597:  merhaba_007_253
1504167434.755603:  merhaba_008_250
1504167434.755608:  merhaba_009_122
```

Kuyruk ismi (queue name) /isim şeklinde bölü ile başlar. İsmin sonu
0 ile bitmeli ve boyu NAME_MAX değerinden ufak olmalıdır.

```
$ getconf -a | grep NAME_MAX
```

Birden fazla proses aynı kuyruk üzerinde çalışabilir. Proseslerin kuyruk ismini bilmesi yeterlidir.

Mesajlara öncelik (priority) verilebilir. En düşük öncelik 0'dır. En yüksek öncelik MQ_PRIO_MAX-1 olabilir.

```
$ getconf -a | grep MQ
```

Linux'da en yüksek öncelik 32768'dir. Posix, en az 32 adet öncelik olmasını söyler.

Kuyruklar çekirdek seviyesinde kalıcıdır (kernel persistence). Makine kapanmadığı sürece silinmezler.

Açıkça mq_unlink(3) ile silinebilirler.

Posix kuyruk uygulaması real time kütüphanesi içindedir. -lrt

Kuyruğa yeni nitelikler atanabilir.

```
$ man mq_getattr

mq_flags   : 0 veya O_NONBLOCK
mq_maxmsg  : eleman adedi üst sınırı
mq_msgsize : mesaj boyunun üst sınırı

mq_curmsgs : o anda kuyruktaki eleman sayısı, sadece get içindir.
```

mq_send() ile mesaj gönderilir.

mq_flag

```
Eğer mq_flags= 0 ise
   kuyuk dolu ile boşalana kadar proses bloke olur.

Eğer mq_flags= O_NONBLOCK ise
   hemen hata döner.
```

Kuyruk için /proc desteği mevcuttur. mq_open()'da attr değeri NULL gelirse, buradaki değeler kullanılır.

```
$ cd /proc/sys/fs/mqueue

$ ls -l
total 0
-rw-r--r-- 1 root root 0 Ağu 31 10:18 msg_default
-rw-r--r-- 1 root root 0 Ağu 31 10:20 msg_max
-rw-r--r-- 1 root root 0 Ağu 31 10:21 msgsize_default
-rw-r--r-- 1 root root 0 Ağu 31 10:21 msgsize_max
-rw-r--r-- 1 root root 0 Ağu 31 10:21 queues_max
```

- **msg_default**
  mesaj adedi, 10, attr= NULL ise kuyruktaki max. eleman sayısı bu alınır.
- **msg_max**
  kuyrukta bundan fazla eleman olamaz, 10
- **msgsize_default**
  mesaj boyu, 8192, attr=NULL ise bu değer kullanılır.
- **msgsize_max**
  8192, kullanıcı mq_msgsize değerini bu değerden fazla veremez.

- **queues_max**
  sistem çapında en fazla bu kadar kuyuk yaratılabilir, 256

/proc dosya sisteminden farklı olarak vfs üzerinden de kuyruklar incelenebilir.

```
$ ls -l /dev/mqueue
# Yoksa veya içi boş ise aşağıdaki gibi mount edilebilir.

$ mkdir /dev/mqueue
$ mount -t mqueue none /dev/mqueue

$ ls -l
total 0
-rw-r--r-- 1 nazim nazim 80 Ağu 31 09:50 test

$ cat test
QSIZE:0          NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

Bu dizindeki dosyalar rm ile kuyuk silinebilir, ls ve cat ile incelenebilir.

Mod değer 644 gözüküyor. `mq= mq_open("/test", ..., 0644, ...);` ile yaratılmıştı.

mq_notify ile async. uyarı alınablir.

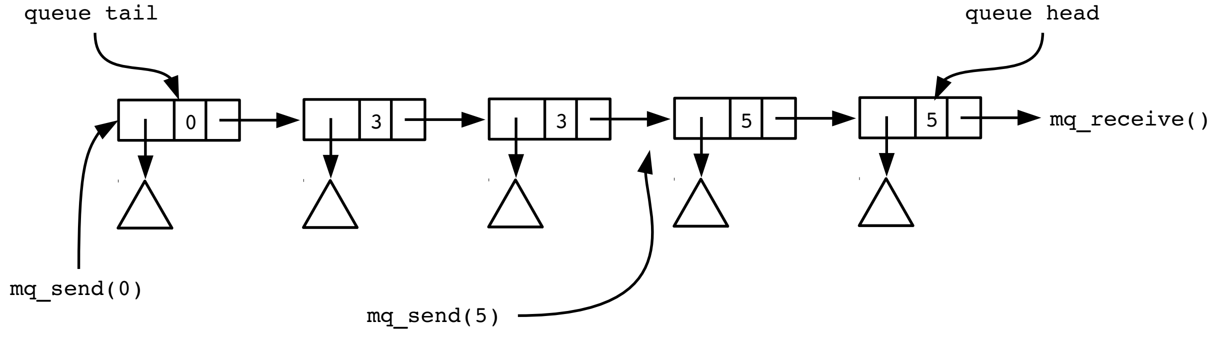
mq_timedreceive ile select()'deki gibi süre verilebilir.

```
QSIZE
Kuyruktaki toplam veri miktarı

NOTIFY_PID
0'dan farklı ise, PID numaralı proses mq_notify(3) ile async. mesaj bekliyor.
0 ise NOTIFY ve SIGNO alanları anlamsızdır.

   NOTIFY
   PID nasıl haberdar edilecek?
   SIGNAL, THREAD veya NONE

   SIGNO
   notify metodu signal ise, kullanılacak sinyalin numarası.
```

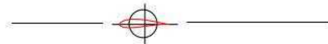
Linux tarafında mq_notify() kullanmaya hiç gerek yoktur. Çünkü mq_open() ile dönen değer standard bir fd'dir ve select() veya benzerleri senkron olarak, event loop içinde kullanılabilir. Ama POSIX değildir, kod taşınabilir olmaz.

mq_notify() örneği için bakınız `$ man 3 mq_notify`

Bakınız `$ man 7 mq_overview`



Şekil 6.1: Posix message queue



Dizin message_que/

Bölüm 7

Semaforlar

Semaforlar, gerçek hayatta işaret vermek için kullanılan bayraklardır. Yazılımda ise ortak kaynağa erişimi yöneten yapılardır.

Mutex'ler ikilik semaforlardır. Mutex'ler aynı proses ve thread'leri tarafından kullanılır. Fakat semaforlar farklı prosesler tarafından da kullanılabilir.

Semaforlar birden fazla kaynağı yönetebilirler. Eğer kritik bölge gibi tek bir kaynak yönetiliyorsa, bu semfor ikiliktir. Mutex'ler ikili semaforlardır.

Seamforlar, işaretsiz tamsayı kullanılarak uygulanır. Bütün semafor işlemleri atomiktir, bölünemez, işletim sistemi bunu garantiler.

Yordam

```
Bir proses veya thread, ortak kaynağa erişmek için başvuru yaptığında,

eğer semafor 0 değerinde ise,
   bloklu bekleme yapar.
   0'dan büyük ise kaynağa erişir ama tamsayı 1 azaltılır.
```

Mutex, semafor ve ilgili operasyonlar

Örnek senaryo, 3 adet yazıcının farklı 3 proses tarafından talep edilmesi.

```
s= open(3);
   P1:  wait(&s);   s= 2 olur ve kaynağı alır.
   P2:  wait(&s);   s= 1 olur ve kaynağı alır.
   P3:  wait(&s);   s= 0 olur ve kaynağı alır.

   P4:  wait(&s);   s= 0 olduğu için P4 bloklanır.

   P2:  post(&s);   s= 1 olur, artık kaynak serbesttir.

   P4:  wait(&s);   s= 0 olur ama kaynağı elde eder.
                    Yeni gelen talep s>0 olana kadar bekler.
```

Temel Semafor İşlemleri

open() Semafor kur ve aç veya kurulu ise sadece kullanıma aç.

post() s++

wait() s>0 ise s--; değilse blokla.

close() Semafora erişimi kapat.

unlink() Kullananan yoksa ismi sil.

open() çağrısında isim verilirerek açılan semaforlara isimli semafor "named semaphores" denir.

Verilecek isim / ile başayalabilir.
İsim boyu NAME_MAX-4 kadar olabilir.
İsimler /dev/shm/sem.* şeklinde saklanır.

İsimli semaforlar "kenrel persistence" özelliğine sahiptir. sem_unlink() yapmadıkça yaşarlar.

İsimsiz semaforlar da mevcuttur. Bunlara "memory based semaphores" da denir. İsimleri yoktur. Ortak bir bellek bölgesinde saklanırlar.

Örnek

İki farklı proses ekrana farklı katarları harf harf yazsın. Çıkışlar pipe ile tek bir yerden izlensin.

```
# Terminal 1

$ mkfifo pimas
$ cat pimas

# Terminal 2

$ make P=1
$ ./prog1 'ABCDE' > pimas &
$ ./prog1 '123' > pimas &

# Terminal 1'den karakter ve sayıların tamamen keyfi sırada gelişini izle.

ABCDEABCDEAB12CD3E1A2BC3D1EA23BC1DE2A31B23C12DE3A1B...

# Programları durdur.

$ killall prog1
$ jobs   # kontrol et
```

Örnek

Her proses kendi katarı bitene kadar diğeri beklesin.

```
# Terminal 1
$ cat pimas

# Terminal 2

# Semaforun başlangıç değerinin 1 olmasını garantile.
# ctrl+c ile çıkışlarda 0 da kalmış olabilir.
# Bu durumda ölümcül kilitlenmeye girilir.
#
$ ./prog2
sem value 1

$ ./prog2 ABCDE > pimas &

$ ./prog2 123    > pimas &

$ ls -l /dev/shm/sem.sayici
-rw-rw---- 1 nazim nazim 32 Ağu 21 14:20 /dev/shm/sem.sayici

# Terminal 1
# Sonuçları gözle
ABCDEABCDEABCDE123ABCDE123ABCDE123ABCDE123ABCDE123ABCDE123ABCDE123
```

```
# Terminal 2

$ killall prog2
[1]-  Terminated  ./prog2 ABCDE > pimas
[2]+  Terminated  ./prog2 123 > pimas

# Kontrol et, çalışan iş kalmasın.

$ jobs
```

Diğer işlemler

init() isimsiz semafor yaratır.

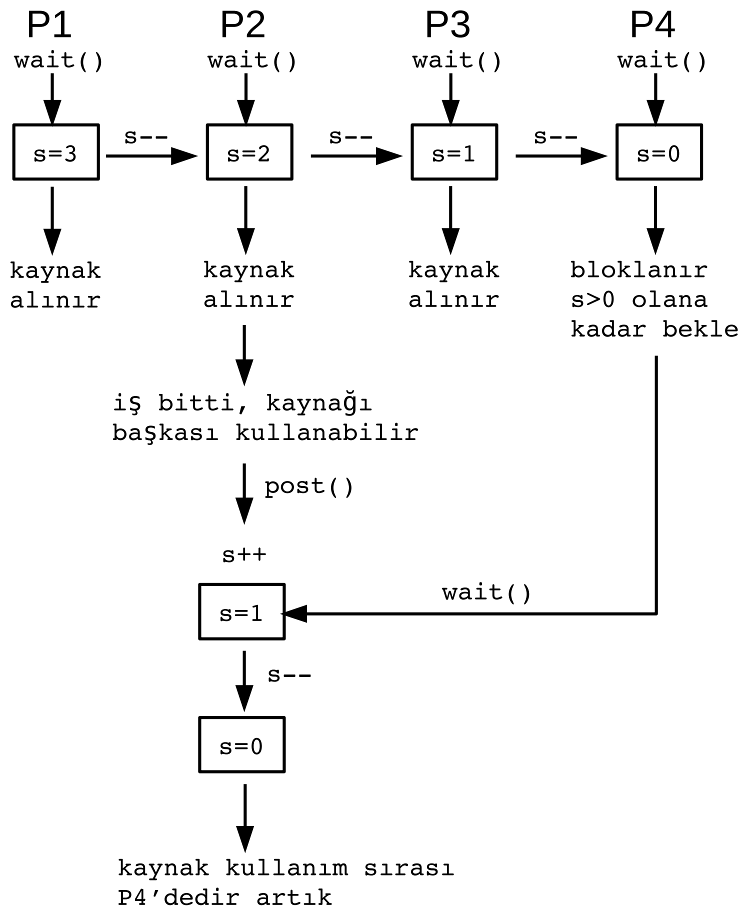
destroy() isimsiz semafor yok eder.

getvalue() semafor değerini elde eder. >= 0'dır.

trywait() wait() gibi işler, ama s=0 ise bloklama yapılmaz, hata döner.

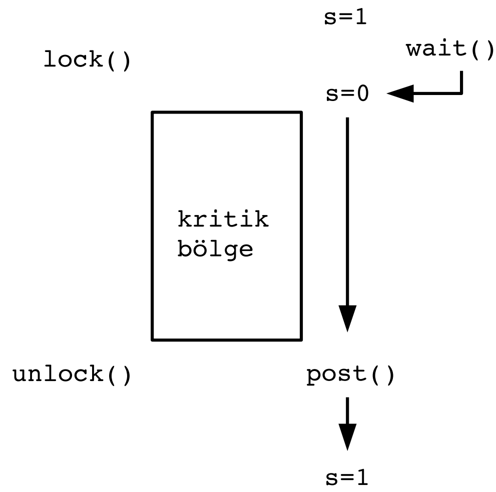
timedwait() wait() gibi işler, s=0 ise verilen süre kadar bekle, bu süre içinde hala s=0 ise timeout hatası döner.

Dizin semafor/

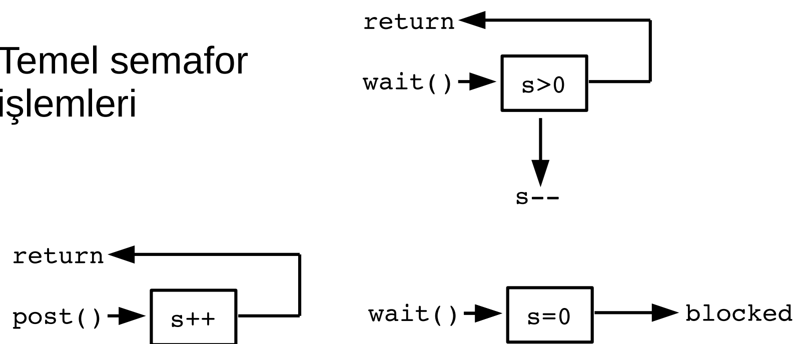


Şekil 7.1: Üçlük semafor örneği

İkilik Semafor, mutex



Şekil 7.2: Mutex



Şekil 7.3: Temel semafor işlemleri

Bölüm 8

Sinyaller

Sinyaller çalışan bir kodu, bir olay olduğunda farklı bir yöne aktaran, yazılım tabanlı kesmelerdir.

Sinyaller asenokrondur, ne zaman geleceği belli olmaz.

Kritik işlemler sinyal ile kesintiye uğrayabilir. Programcı, bloklama ile sinyalleri geciktirerek tedbir almalıdır.

Sinyal kaynakları

- Sıfıra bölüm, iadres dışında taşma, vs
- Açıkça kill sistem çağrısının kullanılması,
- kill komutunun kabuktan verilmesi,
- Uygulama içinde sinyal üreten çağrıların mevcut olması,
- Klavyeden basılan tuşlarla sinyal gönderilmesi,
- ...

Desteklenen sinyallerin listelenmesi

```
$ kill -l
```

Sinyal numaraları 1'den başlar. Sinyaller hem numara hem de isimle kullanılabilir. Numaralar standard olmayabilir. İsimler daha taşınabilirdir.

Örnek

```
$ make
$ say
ctrl+C
```

Örnek

```
$ say &
[1] 23705

$ kill -STOP 23705   # başka terminalden, veya
$ kill -19   23705   # başka terminalden

$ jobs
[1]+  Stopped  ./say

# Uyandır
$ kill -CONT 2375 # başka terminalden
$ kill -CONT %    # aynı terminalden

# Durdur, ctrl+c gibi,

$ kill 2375
$ kill -INT 2375
$ kill -2 2375
```

STOP ve KILL sinyalleri yakalanamaz, bloklanamaz ve gözardı (ignore) edilemez.

Sinyali alan prosesler her zaman aşağıdaki davranışlardan birini sergilerler.

$ man 7 signals

Term Default action is to terminate the process.

Ign Default action is to ignore the signal.

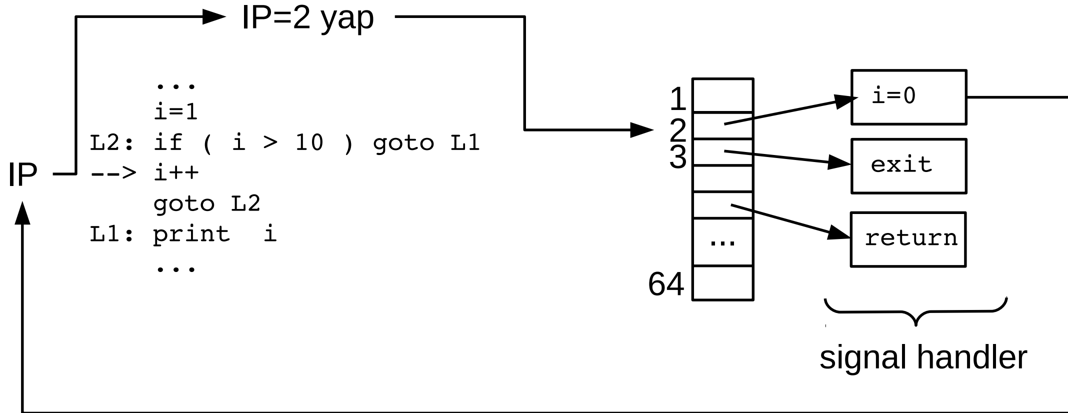
Core Default action is to terminate the process and dump core (see core(5)).

Stop Default action is to stop the process.

Cont Default action is to continue the process if it is currently stopped.

Ön plandaki prosesler için klavyeden sinyal gönderme

```
Ctrl+C   INT    bitir
Ctrl+Z   SUSP   durdur ve arka plana at.
```



Şekil 8.1: Bir sinyalin işleme mantığı

Sinyal ile, IP belirli bir adreste yürütmede iken, yürütmenin farklı bir IP adresine geçmesi istenir. Yani proses çalışır durumda olmalıdır. Bir prosese çalışma sırasını beklerken sinyal gönderilebilir ama bu sinyal ancak proses çalışmaya başlayınca aktif olur. Bundan dolayı sinyalin üretilmesi ve gönderilmesi arasında bir zaman farkı olabilir. Sinyal ancak proses çalışma durumda iken gönderilebilir. Üretilen sinyal, gönderilene kadar askıda (pending) bekletilir.

Örnek Eski sinyal yakalama tekniği, test1.c

```
$ ./test1
```

Başka terminalden

```
$ pidof test1
24465
$ kill -TERM 24465
```

Program çalışmaya devam eder. Sinyal davranışı değiştirilmiştir.

Yeni yöntemde sigaction yapısı kullanılır. Daha gelişmiştir. Aslında eski yöntem de sigaction ile uygulanmaktadır.

Pofesyonel bir program, bütün sinyallere karşı tedbir almalıdır. Böylece her sinyalde nasıl davranış gösterileceği önceden planlanabilir. Örneğin log basmak, açık dosyaları düzgün kapamak veya uyarı vermek gibi. Tedbir alınmayan bir sinyal genelde programın kontrolsüz biçimde sonlanmasına sebep olur.

sigaciton yapısı

```
$ man sigaction

 struct sigaction {
     void     (*sa_handler)(int);
     void     (*sa_sigaction)(int, siginfo_t *, void *);
     sigset_t   sa_mask;
     int        sa_flags;
     void     (*sa_restorer)(void);
     };
```

Örnek test2.c, sigaction ile sinyal yakalama. Sadece sa_handler kullanılacak.

```
$ ./test2

$ pidof test2
24685
$ kill -TERM 24685
$ kill -INT  24685
$ kill -KILL 24685
```

signal handler içinde, sadece signal-safe fonksiyonlar çağrılmalıdır.

Bu fonksiyonların listesi için,

`$ man 7 signal # bak, async signal safe`

printf() signal safety değildir. Handler içinde kullanılmamalıdır. Eğer çalışıyorsa, sadece tesadüftür. write(2) kullanılabilir.

C içinde sistem çağrısı

Her uygulama kendi kendine veya başka prosese kill ile sinyal gönderebilir.

```
kill(pid, SIGINT);
$ man 2 kill
```

Aynı zamanda raise ile de kendi kendine sinyal gönderebilir.

```
raise(SIGINT);
$ man 2 rise
```

abort ile kendini durdurabilir.

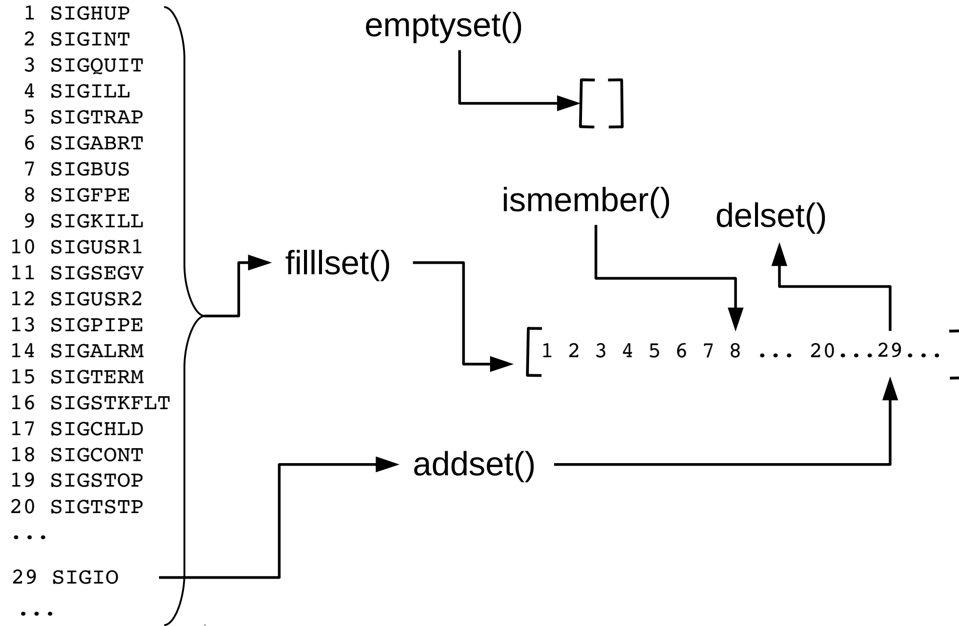
```
abort();
$ man 2 abort
```

```
pause ile kendine sinyal gelmesini bekleyebilir.
pause();
$ man 2 pause
```

Sinyalleri kümeleme

Bütün sinyaller veya berlirli sinyaller, bir amaç için kullanılmak üzere bir araya toplanabilir. Buna sinyal kümeleme denir.

Şekil'deki gibi birden fazla sinyal belirli bir küme içinde toplanabilir.



Şekil 8.2: Sinyallerin kümelenmesi

Maskeleme Programlar içinde signal-safe olmayan fonksiyonlar kullanılıyorsa, ki glibc fonksiyonlarının neredeyse hiç biri signal-safe değildir, sinyallere karşı maskeleme ile koruma yapılmalıdır. Bunun için sinyaller geçici süre bloklanır. Bloklama sonsuz süre için yapılamaz. Sadece ignore edilen sinyaller sonsuz süre bloklanır.

Genelde mutlaka bitmesi gereken i/o işlemleri veya global alan güncellemeleri sinyal maskeleme ile korunmalıdır.

Örnek Sinyal-safe olmayan kritik bölgenin sinyallerden korunması.

```
Kaynak: https://www.gnu.org/software/libc/manual/\
        html\_node/Testing-for-Delivery.html#Testing-for-Delivery
```

Örnekte, BLOCK ve UNBLOCK ile savunulan kritik bölge ALRM sinyalinden korunmaktadır. if()'den önce gelen ALARM sinyali
sigprocmask(SIG_UNBLOCK, &block_alarm, NULL);
ifadesine kadar bekletilir.

Bu ifadeden sonra ALARM sinyali signal handler fonksiyonuna gider. Bu sayede BLOCK-UNBLOCK arasındaki işlemler kesintisiz yürütülür. Bu sayede veri bütünlüğü sağlanmış olur.

```
/* This variable is set by the SIGALRM signal handler. */
volatile sig_atomic_t flag = 0;

int
main (void)
{
  sigset_t block_alarm;
  ...
  /* Initialize the signal mask. */
  sigemptyset (&block_alarm);
  sigaddset (&block_alarm, SIGALRM);

  while (1)
    {
      /* Check if a signal has arrived; if so, reset the flag. */
      sigprocmask (SIG_BLOCK, &block_alarm, NULL);
      if (flag)
        {
          actions-if-not-arrived
          flag = 0;
        }
      sigprocmask (SIG_UNBLOCK, &block_alarm, NULL);
      ...
```

```
    }
}
```

Garanti olması için bütün sinyaller sigset içine alınabilir. KILL ve STOP sinyalleri de sigset içine alınabilir. Bu sinyaller maskelenemez ama yürütme sırasında da hata vermez.

volatile tanımı volatile, "optimize etme" demektir. Genelde hardware destekli bir güncellemede kullanılır. Yani memory'e ulaşan interruptlar veya mmap erişimleri tarafından güncellenen değişkenler için gereklidir.

Kural Thread veya signal-safe uygulamalar için volatile sig_atomic_t her zaman beraber kullanılır. Signal handler içinde kullanılan global değişkenler bu şekilde tanıtılır. Böylece değişken okunur ve lock atılır. Diğer erişimlere iş bitene kadar kapatılır. Race cond. engellenir.

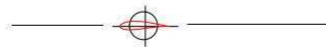
bash'da sinyal kullanımı

Örnek

```
$ ./test1.sh

# klavyeden ctrl+C

# veya başka terminalden

$ killall SIGINT  test1.sh   # veya
$ killall SIGTERM test1.sh
```

Örnek

```
$ ./test2.sh
adını gir ve bitir.

$ ./test2.sh
ctrl+c yap ve sonucu gör.
```



Dizin sinyaller/

Bölüm 9

Threads

Ortak veri alanı (data and heap segments) olan proseslere thread denir.

Verilen şekilde proses ve thread segmentlerinin karşılaştırılması yapılmıştır.

Şekilde 4 adet proses vardır. Her prosesin bütün segmentleri bağımsızdır.

P4 prosesi, T1 ve T2 ile gösterilen, 2 adet thread yaratmıştır. P4 ile T1 ve T2'nin data ve heap alanları paylaşımlıdır, ortaktır.

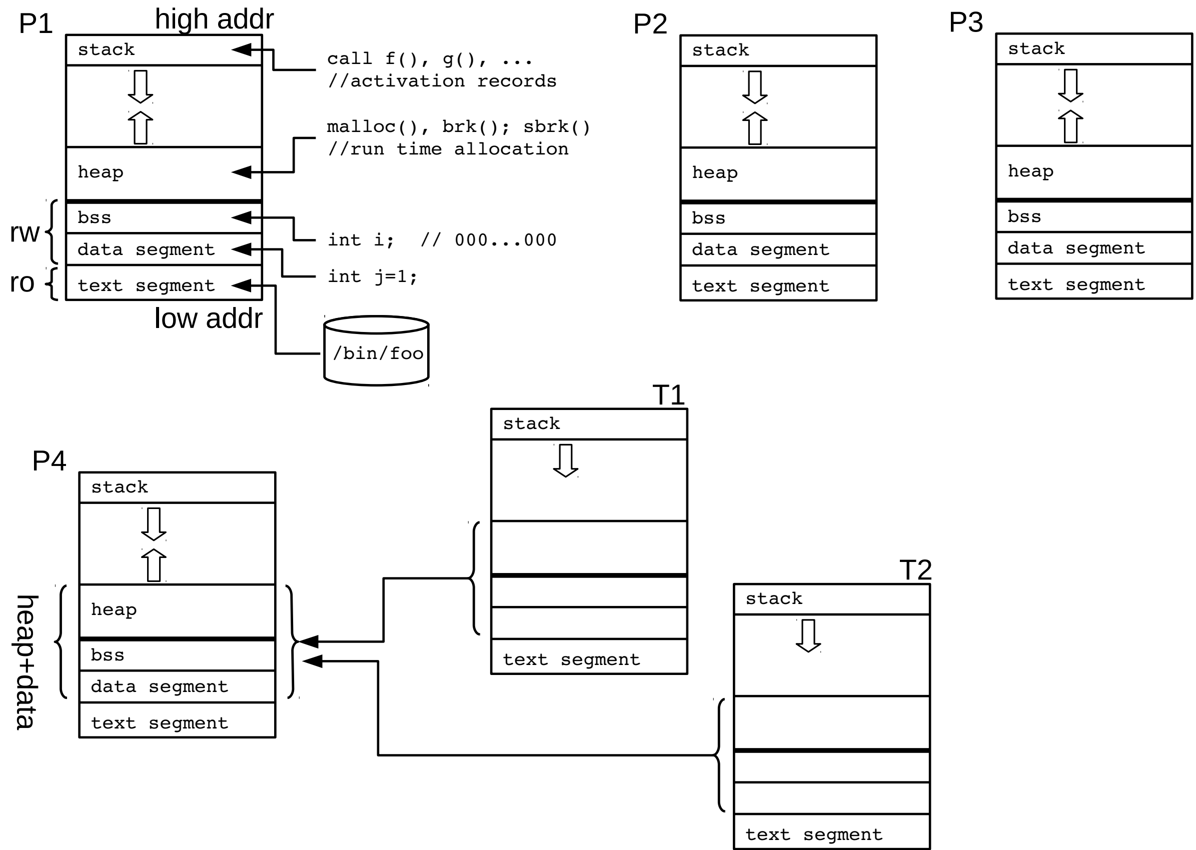
Linux'da thread kütüphanesinin adı pthread'dir. Derleme yaparken -lpthread ile kütüphane ismi açıkça verilmelidir.

`$ getconf -a` komut ile sistemdeki config değişkenleri sorgulanabilir. Thread'lerle ilgili config bilgileri aşağıdaki gibi elde edilebilir.

`$ getconf -a | grep THREAD`

Karşısı boş olan satırlar, undefined kabul edilir. Parametre ismi doğrudan da verilebilir.

Linux'da thread'lerle prosesler hemen hemen aynıdır. Thread'lere light weight process de denir.



Şekil 9.1: Threads

Prog 1 joinable thread örneği.

```
$ make P=1
$ ./prog1
```

Prog 2 Örnek program ancak sinyal ile dudurulabilir. Sistem komutları ile thread'leri incele.

```
$ make P=2
$ ./prog2 &
[1] 3914
tpid 3915
tpid 3916

# Arka planda çalışan işleri gör.
$ jobs
```

```
[1]+  Running                 ./prog2 &

# pstree ile proses ağacını gör
$ pstree -p
prog2---2*[{prog2}]

pstree PID ile proses ağacını gör
$ pstree -p 3914

prog2(3914)---{prog2}(3915)
            |
            ` {prog2}(3916)

# ps ile pid/ppid/tpid incele.
$ ps -eT | grep prog
 3914  3914 pts/8    00:00:00 prog2
 3914  3915 pts/8    00:00:00 prog2
 3914  3916 pts/8    00:00:00 prog2

# top ile incele
$ top -H -p 3914

top - 10:39:13 up  1:14,  1 user,  load average: 0,33, 0,25, 0,24
Threads:   3 total,   0 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s): 13,6 us,  2,3 sy,  0,2 ni, 77,0 id,  6,8 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Mem :  3932584 total,   468508 free,  1456856 used,  2007220 buff/cache
KiB Swap:  1999868 total,  1991736 free,     8132 used.  1842380 avail Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
3914 nazim     20   0   88448    716    632 S  0,0  0,0   0:00.00 prog2
3915 nazim     20   0   88448    716    632 S  0,0  0,0   0:00.00 prog2
3916 nazim     20   0   88448    716    632 S  0,0  0,0   0:00.00 prog2

# Not: -b -n 1 ile copy/paste yapıldı.

# /proc ile inceleme
$ cd /proc/3914/task/

$ ls -l
total 0
dr-xr-xr-x 7 nazim nazim 0 Ağu 20 10:26 3914
dr-xr-xr-x 7 nazim nazim 0 Ağu 20 10:26 3915
dr-xr-xr-x 7 nazim nazim 0 Ağu 20 10:26 3916

# Farklı bir dizinden jobs komutu verilirse wd gösterilir.

$ jobs
[1]+  Running  ./prog2 &  (wd: /nk/workspace/projects/linux.egitimi)
```

```
# Durdur
$ kill %   <ENTER> <ENTER>
[1]+  Terminated   ./prog2

# Kontrol et
$ pidof prog2
$ ps -eT | grep prog
```

joinable, detachable

Bütün thread'ler aksi söylenmedikçe joinable yaratılır. joinable thread yaratan proses, mutlaka join() ile dönüşü beklemelidir.

Joinable thread'ler çağıran yere veri döndürebilir.

Joinable thread'lerde dönüşten sonra kullanılan kaynaklar serbest bırakılmaz. Bellek kullanımı çok iyi analiz edilmelidir.

main() programı durursa, bütün thread'ler durur. Bundan dolayı main() durmadan önce veya daha evvelinde, joinable thread'ler mutlaka pthread_join() fonksiyonu ile beklenmelidir.

detachable thread'lerin dönüş zamanı belli değildir, join() ile beklenilmez.

Dönüş bitince kaynaklar serbest bırakılır. Çağıran yere veri döndürülemez.

Taşınabilirlik için bütün thread'ler joinable olmalıdır. Posix bunu zorlar. Her thread kütüphanesi detachable thread'leri desteklemeyebilir.

Joinable thread yaratılıp, join ile dönüş bilgisi elde edilmezse, bellek sızıntısı meydana gelebilir.

Prog 3 join olmazsa ne olur?

```
$ make P=3
$ ./prog3
4430: p[1]= 2   usleep(313208);
4429: p[1]= 1   usleep(670234);
4430: p[2]= 4   usleep(302413);
main bitti.
```

Thread'ler halen işlemekte iken, main() durunca, bütün thread'ler de durur. join zorunludur!

Prog 4 detachable thread örneği

```
$ ./prog4
main bitti.

# thread'ler yürümeye vakit bulamadı!

# Sona sleep(3) ekle, tekrar çalıştır.
$ ./prog4

  4895: p[1]= 1   usleep(599317);
  4896: p[1]= 2   usleep(680083);
  4895: p[2]= 2   usleep(42473);
  4895: p[3]= 3   usleep(245861);
  4896: p[2]= 4   usleep(855834);
  4895: bitti, toplam 6
  4896: p[3]= 6   usleep(174491);
  4896: p[4]= 8   usleep(586573);
  4896: bitti, toplam 20
main bitti.
```

Sorunsuz çalışır. Fakat main() program yeteri kadar beklemeli, erken bitmemelidir.

Kod sonuna `pthread_exit()` ekle, tekrar çalıştır.

```
$ ./prog4

$ ./prog4
  5391: p[1]= 2   usleep(688879);
  5390: p[1]= 1   usleep(639670);
  5390: p[2]= 2   usleep(829744);
  5391: p[2]= 4   usleep(38427);
  5391: p[3]= 6   usleep(85821);
  5391: p[4]= 8   usleep(170444);
  5391: bitti, toplam 20
  5390: p[3]= 3   usleep(132582);
  5390: bitti, toplam 6
```

main'in bitiş lafı gelmez. Fakat bütün thread'lerin bitmesi beklenir.

Ototmatik join gibi düşünülebilir. Ama thread'lerden dönüş bilgisi gelmez.

Eğer detach thread mevcutsa ya main() yeteri kadar beklemeli ya da main()'den `pthread_exit()` ile çıkılmalıdır.

mutex örneği

Kritik alan girişi ve çıkışında kullanılanılır. Birden fazla mutex varsa sıraya

dikkat edilmelidir. Kritik alandan mümkün olduğu kadar çabuk çıkılmalıdır. lock/unlock işlemleri maliyetlidir, context-swtich gerektirir. Aşırı kullanılmından kaçınılmalıdır. Çoklu işlemci varsa spin-lock ile context-switch engellenebilir.

Mutex için örnek kod parçası:

```
pthread_mutex_t count_mutex;
long count;

void increment_count(){
   pthread_mutex_lock(&count_mutex);
   count= count + 1;
   pthread_mutex_unlock(&count_mutex);
}

long get_count(){
   long c;

   pthread_mutex_lock(&count_mutex);
   c = count;
   pthread_mutex_unlock(&count_mutex);

   return (c);
}
```

Prog 5 Thread'e giden sinyal ana programa gelmiş kabul edilir.

```
$ make P=5

$ ./prog5 &
[1] 5935
tpid 5937
tpid 5936

$ pstree -p 5935

prog5(5935)---{prog5}(5936)
            |
            ` {prog5}(5937)

$ kill -TERM 5936
sinyal geldi Terminated
[1]+  Exit 1                 ./prog5

$ jobs
```

Öneriler

- thread'ler mümkünse kullanılmamalıdır. Proses ve IPC tabanlı çözümler yetersiz kalırsa thread kullanılmalıdır.
- Joinable olmalı ve mutlaka `main()` çıkışında `join()` ile bekleme yapılmalıdır.
- Joinable thread'lerde kullanılmayan kaynaklar açıkça serbest bırakılmalı, işletim sisteminin insafına kalınmamalıdır. Örneğin bir daha hiç kullanılmayacak bir dosya açıkça fclose() ile kapatılmalıdır. Eğer kapatılmazsa, main()'deki veya sistemdeki ilk exit()'den sonra kapatılma yapılacak ve sistem kaynağı boşuna kullanılacaktır.
- Thread'deki bir exit() veya aynı sınıftan bir işlem, main()'in olduğu ana proses dahil, bütün thread'leri kapatır.
- Thread'lerdeki sistem çağrıları mutlaka thread-safe olmalıdır. Birden fazla thread tarafından sorunsuzca kullanılabilen veya reentarnt olan bütün fonksiyonlar thread-safe'dir. man sayfalarından her çağrı tek tek incelenmelidir. man 7 pthreads sayfasında, thread kullanılırken nerlere dikkat edilmelidir bölümü mutlaka okunmalıdır. Yok yoktur.
- Sinyaller bütün thread'lere gönderilir. Sinyallere karşı tedbir alınmalıdır.
- Kritik bölgeler önceden tespit edilmeli ve mutex ile korumaya alınmalıdır.
- Kritik bölgeler aşırı derecede çağrılmamalıdır. mutex işlemi maliyetli bir işlemdir, çünkü her mutex context-switch gerektirir.
- Kritik bölgelerde oyalanmamak gerekir. Bu bölgelere giriş çıkış süresi çok az olmalıdır.
- Kritik bölgelerden çıkarken mutex'ler serbest bırakılmalıdır. Yoksa bütün program donabilir.
- Bir thread başka bir thread'i sonlandırmamalıdır. Bütün thread'leri ana prosesin yönetmesi daha mantıklıdır.
- `pthread_cancel()` fonksiyonu mümkünse hiç kullanılmamalıdır. Thread'ler her çalışma adımında güvenle duramazlar. Thread'ler her zaman `return` veya `pthread_exit()` ile bitmelidir.
- Thread'ler prosesler gibi kabuktan hemen durdurulup başlatılamaz.
- Multithread programların testleri son derece zordur.
- Bellek sızıntılarının tespiti çok sorunludur.

Ne zaman kullanılmalı?

- Thread'lerin her zaman proseslerden çok daha hızlı başlarlar.
- Daha az sistem kaynağı tüketirler, çünkü heap ve data alanları paylaşımlıdır.
- Kullanımları göreceli kolaydır ve IPC gerektirmezler.
- Data alanları ortak olduğu için veri paylaşımı ve ortak veri üzerinde çalışmak aşırı kolaydır.
- Şahsi fikrimiz, baldıran zehiridir.

Thread safe fonksiyonlar

Birden fazla thread tarafından veri bütünlüğü bozulmadan kullanılan fonksiyonlara thread-safe fonksiyon denir. Bu fonksiyonların listesi posix'de verilmiştir.

`$ man 7 pthreads`

Cancel

thread cancel nasıl yapılır? Bakınız `$ man 3 pthread_cancel`

size

size komutu ile bir programın segment boylarının incelenmesi.

```
$ vi prog6.c

#include <stdio.h>
int main(){
    return 0;
}

$ gcc -Wall prog6.c -o prog6

$ size prog6
   text    data     bss     dec     hex filename
   1099     544       8    1651     673 prog6
```

bss block started by symbol or better save space!

3 adet int sayı ekleyelim.

```
$ vi prog6.c

#include <stdio.h>

int i=1; // 0 ise bss'ye gider!
int j=2;
int k=3;

int main(){
    return 0;
}

$ gcc -Wall prog6.c -o prog6

$ size prog6
   text    data     bss     dec     hex filename
   1099     556       4    1659     67b prog6
```

data alanı 3*sizeof(int)=12 bayt büyüdü. İlklendirme yaparken dikkat.

```
#include <stdio.h>

int p[4096];

int main(){
    return 0;
}

$ gcc prog6.c -o prog6
$ ls -l prog6
-rwxrwxr-x 1 nazim nazim 8576 Eyl  2 22:42 prog6

$ size prog6
   text    data     bss     dec     hex filename
   1099     544   16416   18059    468b prog6
```

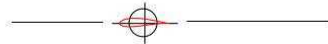
Şimdi p'ye ilk değer verelim.

```
#include <stdio.h>

int p[4096]= { 8 };

int main(){
    return 0;
}
```

```
$ gcc prog6.c -o prog6

$ ls -l prog6
-rwxrwxr-x 1 nazim nazim 24976 Eyl  2 22:44 prog6

$ size prog6
   text    data     bss     dec     hex filename
   1099   16944       8   18051    4683 prog6
```

data ve bss alanları yer değiştirmiştir. Fakat en önemlisi global atamalarda kod boyu çok büyür.

Global ilkleme yapılınca, kod boyu 8,576 bayttan, 24,976 bayta çıkmıştır. Diskte yer sıkınıtısı varsa, önlem olarak, global değeler statik olarak atanmayabilir.



Dizin thread/

Bölüm 10

tcp/ip client/server

Bir bağlantının her bir ucundaki yapıya soket denir. Ya da bir bağlantıdaki her bir ucun mantıksal ifadesidir.

İki yönlü iletişim sağlar.

IPC için de kullanılır.

Diğer IPC yöntemlerinde prosesler aynı makinede olmak zorundadır. Soket yönteminde prosesler aynı veya farklı makineler üzerinde koşabilir.

pipe kavramının ağ üzerine genişletilmiş halidir.

Addr info bilgisi

Belirli kriterlere göre adres çözümlemesi yapar. Kriterler hint yapısı içine girilir ve sonuçlar bağlı liste şeklinde sunulur.

Genel yapı aşağıdaki gibidir.

```
*flags     AI_V4MAPPED | AI_ADDRCONFIG
*family    AF_INET, AF_INET6, AF_UNSPEC
*socktype  SOCK_STREAM, SOCK_DGRAM, 0
*protocol  0
 addrlen   sizeof(addr)
 addr      <ip, port>
 cannoname UCanLinux.Com
 next      sonraki düğüm.
```

* ile gösterilenler hint ile verilir. Nihai bağlı listede hepsi mevcuttur.

Örnek için prog1.c'ye bak.

Şekiller, sadece yorumdur.

Örnek tcp client

prog1 ve nc ile tcp client testi.

```
# Terminal 1

$  nc -l 2306  # prog1'i de başlat.
merhaba
MERHABA
nasılsın?
NASıLSıN?

# Terminal 2
$ make P=1
$ ./prog1  localhost 2306
bağlandı, localhost:2306
merhaba
nasılsın?
sunucu kapandı.
tcp client bitti.

2306 yerine doğrudan bilinen port isimleri de girilebilir.

# Sunucu dinliyor mu?
$ netstat --listen --tcp

Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:2306                  *:*                     LISTEN


# Kapanışın incelenmesi
$ netstat --tcp -c komutunu gir.

# Sonra ctrl+d ile sunucuyu kapat.

$ netstat --tcp  -c

Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:36492         localhost:2306          ESTABLISHED
tcp        0      0 localhost:2306          localhost:36492         ESTABLISHED
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:36492         localhost:2306          ESTABLISHED
```

```
tcp        0      0 localhost:2306          localhost:36492         ESTABLISHED
...
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:2306          localhost:36492         TIME_WAIT
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:2306          localhost:36492         TIME_WAIT
...
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
```

Örnek tcp server

Çok basit tcp server.

Aynı anda tek bağlantı kabul eder. İteratif bir sunucudur. Bir istemci diğerini engeller.

Engeli kaldırmak ve çoklu bağlantıya destek vermek için çözümler,

- thread'ler kullanılabilir, çok hızlı ve verimlidir, yönetimi çok zordur.
- fork ile proses tabanlı yapılabilir, yavaştır, çok sistem kaynağı yer, yönetimi çok kolaydır.
- select ile blocked/senkron ile tek proses yapılabilir. Bütün fd'ler aynı anda monitör edilir. Senkron i/o multiplexing de denir. Biraz programcılık yeteneği gerektirir. Çok basit ve hızlı bir yöntemdir. Yönetimi en basit olandır.

Server Test

```
$ make P=2
$ ./prog2 2306

# bash ile test:

$ echo merhaba    > /dev/tcp/127.0.0.1/2306
$ cat /etc/passwd > /dev/tcp/127.0.0.1/2306

nc ile test:

$ nc 127.0.0.1 2306
```

```
merhaba
1
2
3
ctrl+d
```

Client Test

```
$ nc -l 2306
$ ./prog1
merhaba
MERHABA
```

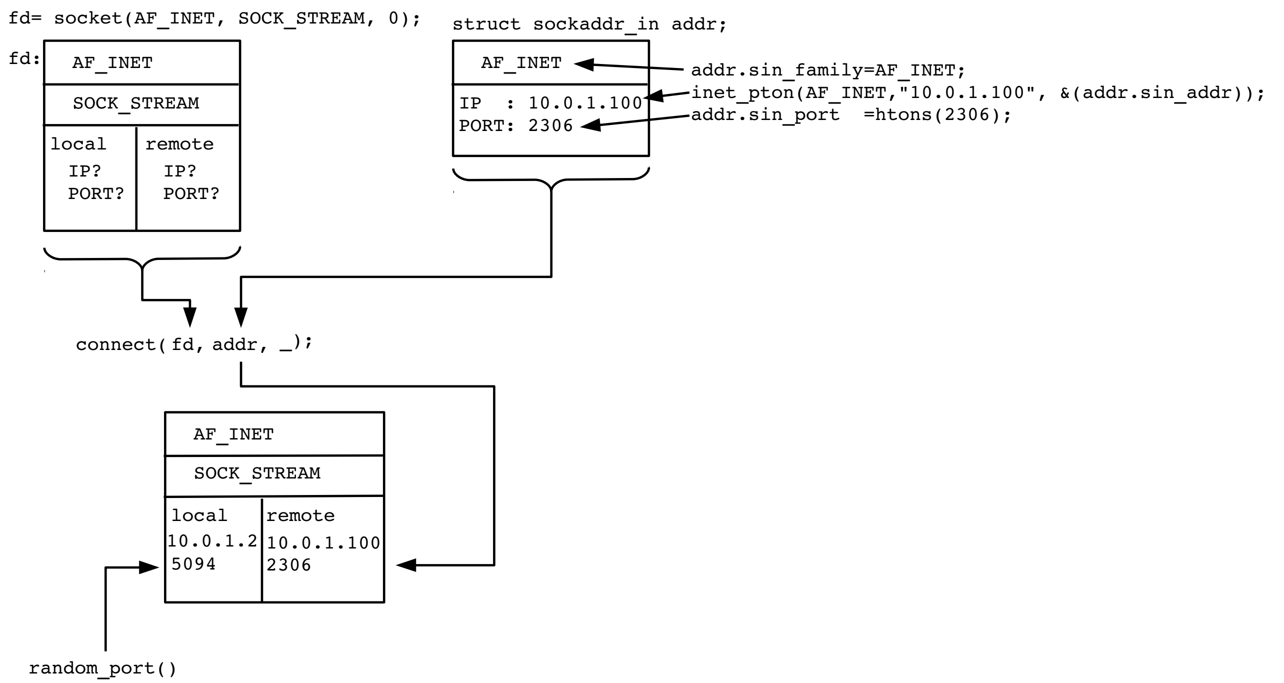
Eski yöntem

```
struct sockaddr_in addr;

fd= socket(AF_INET, SOCK_STREAM, 0);

addr.sin_family=AF_INET;
addr.sin_port  =htons(22000);
inet_pton(AF_INET,"127.0.0.1", &(addr.sin_addr));

connect(fd, (struct sockaddr *)&addr, sizeof(addr));
```

endian kavramı çalışan sistem ile network arasındaki sayıların bayt sırası uyumu için hton ve ntoh fonksiyonları kullanılır.

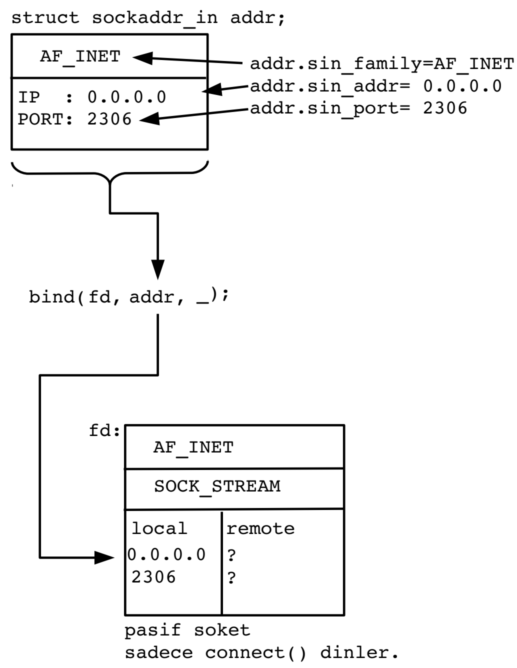
port numaraları için 16 bit ayrılmıştır, en fazla 65535 olabilir. İlk 1024 protu sadece root, bind edebilir. Bu portlara secure port denir.

Dizin tcp-ip-client/



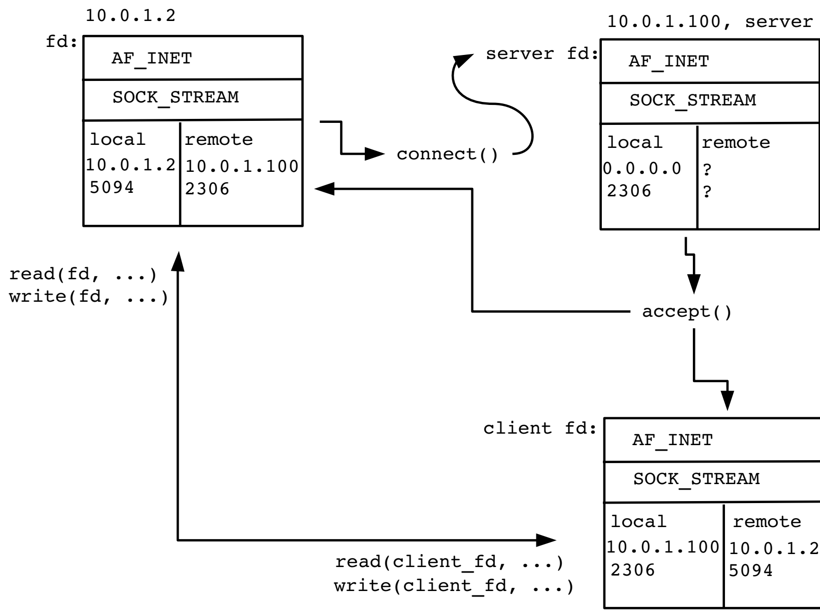
Şekil 10.1: İstemci soketinin kurulması

Pasif soketin kurulması ve adres ataması



Şekil 10.2: Pasif soketin kurulması

Bağlantının kabul edilmesi



Şekil 10.3: Bağlantının kabul edilmesi

Bölüm 11

tcp/ip server

İteratif sunucular Aynı anda sadece tek bir istemciye destek verirler. Diğer istemciler sıralarını beklerler

Eş-zamanlı sunucular Aynı anda keyfi sayıda istemciye hizmet verirler. İstemciler birbirlerini engellemezler.

Eş zamanlı sunucular her bir bağlantı için bir thread veya bir proses kullanabilir. Ya da tek bir proses içinde io multiplexing yapabilir. Bunu da select() sistem çağrısı ile gerçekler.

11.1 İteratif sunucu

iteratif/ içine bak.

istemci, satır okur ve sunucuya gönderir. Sunucu satırı büyütüp geri döndürür.

Aynı anda tek bir bağlantı kabul eder. İkinci bağlantıyı bekletir.

```
$ ./prog2 2306

$ ./prog1 locahost 2306
merhaba
dünya
ctrl+d

$ ./prog2 2306
cat /etc/passwd | ./prog1 127.0.0.1 2306
```

Örnek prog2.c, iteratif sunucu örneği

11.2 Multithread sunucu

MT sunucu için thread_based/ içine bak.

```
$ ./prog2 2306

bağlantı bekliyorum, port:2306  pid:7817

bağlantı kabul edildi, 127.0.0.1:42064

bağlantı bekliyorum, port:2306  pid:7817

fd: 4
4 --> merhaba
4 --> nasilsin

bağlantı kabul edildi, 127.0.0.1:42066

bağlantı bekliyorum, port:2306  pid:7817
fd: 5

5 --> bu ikincisidir.
5 --> engel yok.

# İlk bağlantı

$ ./prog1 localhost 2306

bağlandı, localhost:2306
merhaba
MERHABA
nasilsin
NASILSIN


# İkinci bağlantı

$ ./prog1 127.0.0.1 2306

bağlandı, 127.0.0.1:2306
bu ikincisidir.
BU IKINCISIDIR.
engel yok.
ENGEL YOK.

# netstat ile inceleme
ESTABLISHED olan her bir satır tek bir sokete karşı gelir.
```

```
$ netstat --tcp -a | grep 2306
tcp        0      0 *:2306                  *:*                     LISTEN
tcp        0      0 localhost:2306          localhost:42064         ESTABLISHED
tcp        0      0 localhost:2306          localhost:42066         ESTABLISHED
tcp        0      0 localhost:42064         localhost:2306          ESTABLISHED
tcp        0      0 localhost:42066         localhost:2306          ESTABLISHED


# top ile inceleme

$ top -p 7817 -H

top - 18:53:26 up 10:11,  1 user,  load average: 0,09, 0,16, 0,17
Threads:   3 total,   0 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,7 us,  0,3 sy,  0,0 ni, 98,7 id,  0,3 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem :  3932584 total,   536736 free,  1329128 used,  2066720 buff/cache
KiB Swap:  1999868 total,  1983384 free,    16484 used.  1902736 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 7817 nazim     20   0   22916    816    732 S  0,0  0,0   0:00.00 prog2
 7819 nazim     20   0   22916    816    732 S  0,0  0,0   0:00.00 prog2
 7822 nazim     20   0   22916    816    732 S  0,0  0,0   0:00.00 prog2

# pstree ile inceleme

$ pstree  -p 7817
prog2(7817)---{prog2}(7819)
           `-{prog2}(7822)
```

Örnek prog2.c, multithread sunucu örneği

11.3 Proses bazlı sunucu

process_based/ altına bak.

Her bağlantı için ana proses kendini yeni bir proses olarak kopyalar.

Öyleki ana proses ölse bile, yani bağlantıları bekleyen sunucu ölse bile istemciler çalışmaya devam eder, fakat yeni bağlantı kabul edilmez.

Aşağıdaki örneklerde fd=4 sayısı tekrar eder, yeni fd atanmaz. Çünkü fd'ler proses bazında bağımsızdır. Fakat thread'li uygulamada her bağlantıda fd değeri 1 artmaktadır.

Örnek

```
$ ./prog2 2306

bağlantı bekliyorum, port:2306  pid:8786

bağlantı kabul edildi, 127.0.0.1:42116

bağlantı bekliyorum, port:2306  pid:8786

çocuk başladı fd:4 pid:8788
4 --> bu ilk istemci

bağlantı kabul edildi, 127.0.0.1:42118

bağlantı bekliyorum, port:2306  pid:8786
çocuk başladı fd:4 pid:8791
4 --> bu ikincisi
```

Örnek prog2.c, proses tabanlı sunucu örneği.

11.4 I/O multiplexing

select_based/ altına bak. Yeni thread veya proses oluşturulmaz.

Bütün hizmet ana proses üzerinden verilir.

Senkron ve bloklu bir mekanizmadır. Fakat bloksuz da kullanılabilir.

Şekil 1 de bit dizisi işlemleri verilmiştir. Bütün mantık bu bitler üzerinde dönmektedir.

İşlemler select() sistem çağrısı ve makroları ile yapılır.

Bu çağrı ve ilgili makroları fd veya ilgili buffer'lar üzerinde hiç bir güncelleme yapmaz. Sadece okunmaya/yazmaya hazır veri olup olmadığını, busy waiting yapmadan haber verir.

Önceki testlerle tamamen aynı şekilde test edilir.

pselect()

select() yerine pselect() de kullanılabilir. Farkları?

- pselect() nanosaniye seviyesine kadar süre aşımı değeri alabilir.
- select() sadece mikrosaniye hassasiyete sahiptir.
- select() dönüşte kalan süreyi günceller, pselect() güncellemez.
- pselect() sinyal maskesini argüman olarak alır, select()'te bu özellik yoktur.

pselect() örneği

```
#include <errno.h>

/* Flag that tells the daemon to exit. */
static volatile int exit_request = 0;

/* Signal handler. */
static void hdl (int sig){
        exit_request = 1;
}
```

```
int main(){

   sigset_t mask;
   sigset_t orig_mask;
   struct sigaction act;

   memset (&act, 0, sizeof(act));
   act.sa_handler = hdl;

   if (sigaction(SIGTERM, &act, 0)) {
      perror ("sigaction");
      return -1;
      }

   ...

   sigemptyset(&mask);
   sigaddset (&mask, SIGTERM);
   sigprocmask(SIG_BLOCK, &mask, NULL);

   ...

   while( !exit_request ){

        ...

        res= pselect(max_fd+ 1, &fds, NULL, NULL, NULL, &mask);

        if (res < 0 && errno != EINTR) {
           perror ("select");
           break;
           }

        if (exit_request){
           break;
           }

        ...

        } // while

}
```

pselect() ile select() arasındaki ana fark, man 2 select'den.

```
ready = pselect(nfds, &readfds, &writefds, &exceptfds, timeout, &sigmask);
```

```
is equivalent to atomically executing the following calls:

sigset_t origmask;

pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

Sinyali geciktirme

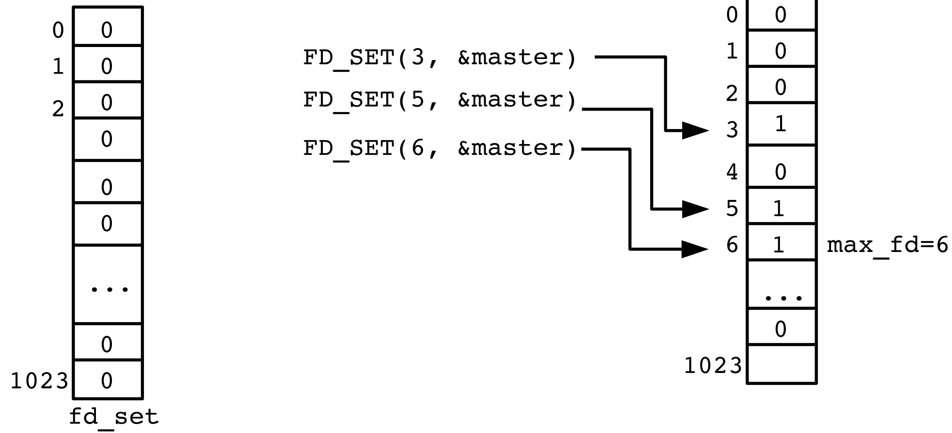
```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

sigprocmask()  is  used  to  fetch and/or change the signal mask of the calling thread.
The signal mask is the set of signals whose delivery is currently blocked for the caller

The behavior of the call is dependent on the value of how, as follows.

SIG_BLOCK
       The set of blocked signals is the union of the current set and the set argument.

SIG_UNBLOCK
       The  signals  in set are removed from the current set of blocked signals.
       It is permissible to attempt to unblock a signal which is not blocked.

SIG_SETMASK
       The set of blocked signals is set to the argument set.


# Hangi sinyaller gecikecek?

int sigprocmask(how, set, _);

BLOCK    yeni sinyal kümesi= current + set
UNBLOCK  yeni sinyal kümesi= current - set
MASK     yeni sinyal kümesi= set
```

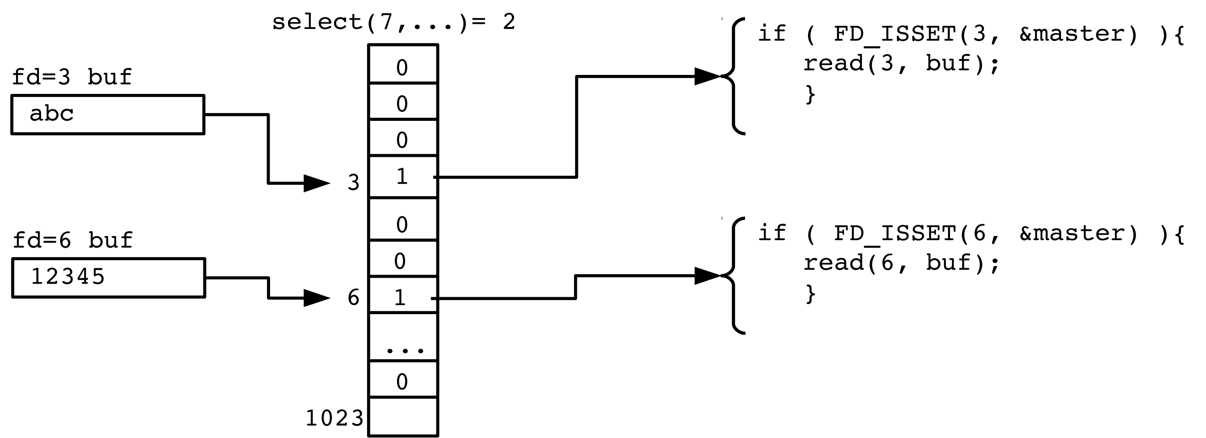
Kaynak `$ man 3 select_tut`

Örnek prog2.c, senkron i/o multiplexing tabanlı sunucu örneği.

prog1.c Ortak istemci programı.

Dizin tcp-ip-server/





Şekil 11.1: select ve makroları

Bölüm 12

UDP Soketleri

Bağlantı gerektirmeyen bir iletişim protokolüdür.

Veri boyu kısıtlıdır.

Güvenilir (reliable) değildir.

Paket bazlıdır, stream bazlı değildir.

Paketler kaybolabilir, tekrar edebilir veya sıraları bozulabilir, bunun bir denetimi yoktur.

Client veya server tarafı birbirlerinin kapandığını anlayamaz. Hatta client, server'ın varlığı veya ayakta olup olmadığı konusunda bir bilgiye dahip olmasa da olur.

Genelde tek paket alışverişi gerektiren çok basit iletişimlerde kullanılır.

udp server

Sunucu, kendine gelen paketleri hem ekrana yazar, hem de büyüterek geri gönderir.

```
$ make P=2
$ ./prog2
```

udp client

Klavyeden girilen bilgileri sunucuya gönderir. Sunucudan gelen bilgileri ekrana yazar.

```
$ make P=1
$ ./prog1
cümle gir?
Merhaba
MERHABA

cümle gir?
ne haber?
NE HABER?
```

Başka bir terminalden prog1 tekrar başlatılabilir. Bağlantısız olduğu için, aynı program hepsine hizmet verebilir.

bash udp/ip client

```
echo merhaba > /dev/udp/127.0.0.1/2306
Geri gelen veri gözükmez, bash'ın bu özelliği tek yönlüdür.
Veri gönderilir ama okunamaz.
```

nc ile

```
$ nc -u 127.0.0.1 2306
 merhaba
 MERHABA
 nasılsın
 NASıLSıN
 ctrl+c

netstat ile kontrol,
$ netstat -a | grep 2306
udp        0      0 localhost:2306          *:*
```

Dizin udp-soketler/

Bölüm 13

Unix alanı soketleri

tcp veya udp soketleri gibidir. Farklı olarak sadece lokal prosesler arası kullanılabilr.

Veri alışveriş hızı ip tabanlı çözümlerden daha hızlıdır. Fakat dağıtık ortam için işlemez.

Adres olarak IP/Port yerine dosya ismi mevcuttur. Mevcut bir dosya sistemi üzerinde s tipinde bir dosya kurulur ve kullanılır.

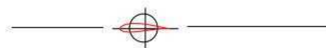
Örnek Prog 2, echo server

Gelen mesajı ekrana yazar, aynı zamanda büyüterek geri döndürür.

```
$ make P=2
$ ./prog2

$ ls -l echo_socket
srwxrwxr-x 1 nazim nazim 0 Ağu 23 09:53 echo_socket
```

Örnek Prog 1, echo client. Klavyeden girilen mesajı sunucuya gönderir. Yanıtı ekrana yazar.

prog*.c kaynağı: http://beej.us/guide/bgipc/output/html/multipage/unixsock.html



Dizin unix-soketler/

Bölüm 14

En basit kernel modülü

En basit modül, "merhaba dünya" örneği

header'lar yüklü olmalıdır.

```
$ apt-get install build-essential linux-headers-$(uname -r)

$ vi hello.c

$ vi Makefile

$ make

$ file hello.ko

$ ls -l
total 32
-rw-rw-r-- 1 nazim nazim  596 Tem 20 16:14 hello.c
-rw-rw-r-- 1 nazim nazim 3976 Tem 20 16:16 hello.ko
-rw-rw-r-- 1 nazim nazim  713 Tem 20 16:16 hello.mod.c
-rw-rw-r-- 1 nazim nazim 2816 Tem 20 16:16 hello.mod.o
-rw-rw-r-- 1 nazim nazim 2040 Tem 20 16:16 hello.o
-rw-rw-r-- 1 nazim nazim  150 Tem 20 16:15 Makefile
-rw-rw-r-- 1 nazim nazim   78 Tem 20 16:16 modules.order
-rw-rw-r-- 1 nazim nazim    0 Tem 20 16:16 Module.symvers
-rw-rw-r-- 1 nazim nazim  791 Tem 20 16:27 oku

$ make clean
$ ls -l
$ make

$ insmod hello.ko
```

```
$ insmod hello.ko  # hata verir.

$ modinfo hello.ko

$ lsmod
$ modinfo herhangi.ko

$ lsmod | head
$ dmesg|tail

$ tail /var/log/syslog

$ rmmod hello     # veya
$ rmmod hello.ko

$ lsmod | grep hello
```

Parametre aktarımı

```
$ insmod hello.ko my_city="ankara"
$ tail /var/log/syslog
$ tail /var/log/kern.log
```

Hatalı parametre aktarımı

```
$ rmmod hello.ko
$ insmod hello.ko city="ankara"
$ tail /var/log/syslog   # hata mesajını gör.
```

Bağımlılık çok fazlaysa modprobe kullanılır.

İsimler aşağıdaki gibi olursa, module_init() ve module_exit() tanımlarına gerek yoktur.

```
int  init_module(void);
void cleanup_module(void);
```

printk

```
KERN_EMERG
   Used for emergency messages, usually those that precede a crash.

KERN_ALERT
   A situation requiring immediate action.
```

```
KERN_CRIT
   Critical conditions, often related to serious hardware or software failures.

KERN_ERR
   Used to report error conditions; device drivers often use KERN_ERR
   to report hardware difficulties.

KERN_WARNING
   Warnings about problematic situations that do not, in themselves,
   create serious problems with the system.

KERN_NOTICE
   Situations that are normal, but still worthy of note.
   A number of security-related conditions are reported at this level.

KERN_INFO
   Informational messages. Many drivers print information about the hardware
   they find at startup time at this level.

KERN_DEBUG
   Used for debugging messages.
```

Bakınız Documentation/kbuild/modules.txt, How to Build External Modules

Makefile

```
obj-m += hello.o

all:
		make -C /lib/modules/`uname -r`/build M=$(PWD) modules

clean:
		make -C /lib/modules/`uname -r`/build M=$(PWD) clean
```

Dizin module_hello/

Bölüm 15

/proc file system örneği

Amaç kernel ve user space arasında, tek veya iki yönlü bilgi alışverişidir.

Genel yapı

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>

static ssize_t test_read(struct file *filp, char *buffer, size_t length, loff_t *offset){
   ...
   return n;
}

static ssize_t test_write(struct file *file, const char *buffer, size_t len, loff_t * off){
   ...
   return n;
}

static const struct file_operations test_fops = {
.owner = THIS_MODULE,
.read = test_read,
.write = test_write
};

static int __init example_init(void){
if (proc_create(PROC_FILE, 0666, NULL, &test_fops) == NULL) {
   return -ENOMEM;
   }
```

```
        ...
return 0;
}

static void __exit example_exit(void){
remove_proc_entry(PROC_FILE, NULL);
}

module_init(example_init);
module_exit(example_exit);
```

Derleme Tekniği

Makefile

```
obj-m += hede.o

all:
		make -C /lib/modules/`uname -r`/build M=$(PWD) modules

clean:
		make -C /lib/modules/`uname -r`/build M=$(PWD) clean
```

İncele

$ uname -r

Sistemde birden fazla kernel ve bunlarla ilgili farklı modüller olabilir. Bundan dolayı uname -r ile ayaktaki kernel'in numarası ile uygun modül seçilir.

```
$ ls -l /lib/modules/‘uname -r‘/build
lrwxrwxrwx 1 root root 39 Ağu 12 02:49 /lib/modules/4.4.0-93-generic/build -> /usr/src/linux-h

$ cd /lib/modules/‘uname -r‘/build
$ ls -l
```

Derleme sırasında gerekli header dosyaları /lib/modules/‘uname -r‘/build ile gösterilen yerden elde edilir. Çekirdek bare metal'dir. Kütüphaneye ihtiyaç duymaz, header'lar yeterlidir.

Derle Derleme için header dosyaları yoksa, indirilmelidir.

```
$ apt-get install linux-headers-$(uname -r)
```

```
$ make clean
$ make
```

Üretilen dosyalar

```
test.c            modülün kaynak kodu

test.ko           yüklenebilir modül

test.mod.c        yardımcı C dosyası, otomatik üretilir

test.mod.o        test.mod.c'nin object kodu

test.o            test.c'nin object kodu

Makefile          modülün Makefile dosyası

Module.symvers    export edilen değişkenler, bizde yok.

modules.order     modüllerin hangi sırada derlendiğini tutar,
          çoklu modülleri uygun sırada yüklemek için kullanılır.
```

Modülü yükle

```
$ sudo insmod test.ko
$ lsmod | head
$ ls -l /proc/aselsan

$ tail -f /var/log/syslog
```

Test

```
$ tail -f /var/log/syslog

# Başka terminalden
$ echo merhaba > /proc/aselsan

$ cat /proc/aselsan
$ cat /proc/aselsan
```

Modülü kaldır

```
$ sudo rmmod test
$ lsmod | head

$ tail -f /var/log/syslog
```

Soru finished değişkeni niye vardır? Olmasaydı nasıl test_read() nasıl işlerdi?

Yanıt Kernel, read işlemini eof bulana kadar devam ettirir. İlk read çağrısında buffer[]'a yazılan miktar döner. Sonraki read'de ise 0 dönmelidir ki kernel eof geldiğini anlasın.

```
ilk read okuması aşağıdaki gibi işler
   finished=0
   copy data to user space
   finished= 1
   kopyalanan bayt sayısı döner.

Sonraki read okuması aşağıdaki gibi işler
   finished artık 1'dir.
   finished= 0; yapılır ki sonraki okumada tekrar aynı veri gönderilsin.
   0 sayısı döner, bu da çekirdeğe eof geldiğini söyler.

Eğer finished kullanılmamış olsaydı, sonsuz döngü oluşurdu.

    finished = 0;

    if ( finished ){
       finished = 0;
       return end_of_file;   // yani 0
       }

    finished= 1;

    copy_to_user(buffer, buf, ptr);

    return ptr;    // okunan eleman sayısı döner.
```

Dizin proc_file_system/

Bölüm 16

Statik kütüphaneler

Statik kütüphaneler, aynı amaç için biraraya getirilmiş, arşivlenmiş object kodların bütünüdür. Program içinde kullanılan kütüphane fonksiyonları, link aşamasında, kütüphaneden program içine alınır. Her kod içinde statik kütüphaneler tekrar eder. Bundan dolayı, mümkünse her zaman paylaşımlı kütüphane kullanılmalıdır.

Kuruluş

```
$ gcc foo.c -c
```

foo.o oluşur. -c Derle ama link etme demektir. Yani sadece *.o oluşturur. İçinde main() olmamlıdır.

ar ile bütün *.o'ları biraraya toplanır.

```
$ ar -rcs fileName.a *.o

-r insert object file
-c create
-s write an object-file index into the archive
```

Alışkanlık gereği statik kütüphanelere .a uzantısı verilir.

Kütühane ismi sonda verilmelidir. Önce semboller saklanır sonra aranır. test.a'dan sonra, test.a'yı kullanan object olmamalıdır.

Sembollerin incelenmesi

Arşiv içindeki sembolleri listele

T: text code section

```
$ nm libtest.a

add.o:
0000000000000000 T add

sub.o:
0000000000000000 T sub

mult.o:
0000000000000000 T mult

div.o:
0000000000000000 T div
```

```
$ gcc -O2 -Wall -o prog1 prog1.c libtest.a

Sembollerin incelenmesi.

$ nm prog1
...
00000000004005f0 T add
...
0000000000400600 T mult
...
```

Bütün kütüphane değil, sadece kullanılan fonksiyonlar çalışabilir koda eklenir.

Prensip olarak kütüphaneler hem statik hem dinamik derlenir.

Dinamik derleme daha geneldir. Her dinamik kütüphanenin genelde bir de statik kütüphanesi bulunur ama her zaman olmayabilir.

Örnek

```
$ cd /usr/lib/x86_64-linux-gnu
$ ls  *reso*
libresolv.a  libresolv.so
```

Örnek Kuruluş

```
$ make  # veya el ile,
```

```
$ gcc -O2 -Wall add.c -o add.o  -c
$ gcc -O2 -Wall sub.c -o sub.o  -c
$ gcc -O2 -Wall mult.c -o mult.o  -c
$ gcc -O2 -Wall div.c -o div.o  -c

$ ar -rcs libtest.a add.o sub.o mult.o div.o

$ file libtest.a
libtest.a: current ar archive
```

Kullanım

```
$ gcc -o prog1 prog1.c

/tmp/ccBhsjaF.o: In function ‘main’:
prog1.c:(.text+0x1a): undefined reference to ‘add’
prog1.c:(.text+0x44): undefined reference to ‘mult’
collect2: error: ld returned 1 exit status

$ gcc -o prog1 prog1.c -L. -ltest

# İçinde libtest.so mevcut değil.
#
$ ldd prog1
    linux-vdso.so.1 =>  (0x00007fff04d14000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4478f37000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f4479301000)

# veya

$ gcc -Wall -o prog1 prog1.c ./libtest.a

# veya aynı anda libtest.so da varsa, statik olanı kullanmaya zorlamak için,
#
$ gcc -Wall -o prog1 prog1.c -Wl,-Bstatic -L. -ltest  -Wl,-Bdynamic
                                                      ^
                                                      Program içinde dinamik
                                                      kütüphane olacağı için
                                                      zorunludur.

$ ./prog1
   add(2,3) = 5
   mult(2,3)= 6
```

Makefile

```
# Makefile

OBJS=add.o sub.o mult.o div.o
PROG=libtest.a

all: $(OBJS) $(PROG)

%.o: %.c test.h
	gcc -O2 -Wall $< -o $@  -c
	ls -l $@
	@echo

$(PROG): $(OBJS)
	ar -rcs $(PROG) $(OBJS)
	ls -l $(PROG)
	file $(PROG)

clean:
	rm -f *.o $(PROG) prog1
```

Dizin static_libs/

Bölüm 17

Paylaşımlı kütüphaneler

Dynamic Sahred Objects, *.so

Proses adres alanına sonradan yüklenen kütüphanelerdir.

Çalışan kodun dosyasından ayrı bulunurlar.

Pek çok program tek bir kütüphaneyi ortak kullanabilir. Bundan dolayı paylaşımlı denir.

Geriye uyumluluk varsa, kütüphane güncellendiğinde uygulamanın tekrar derlenmesine gerek yoktur.

dlopen API ile yürütme zamanında yüklenip işi bitince tekrar serbest bırakılabilirler. plugin yöntemi

Çalışabilir bir kodun ihtiyaç duyduğu kütüphaneler ldd veya objdump ile tespit edilebilir.

Uygulama programlarında aşırı tekrar eden yerler paylaşılan kütüphane yapılabilir.

RAM'de fiziksel olarak tek bir yerde bulunurlar. Ama her programın mantıksal adres uzayı faklıdır. Yani kodların pozisyondan bağımsız olması gerekir. Bundan dolayı paylaşılan kütüphaneler PIC seçeneği ile derlenirler.

Çalışan kod ortak olsa bile, data alanı, yani global değişkenler veya ortak kullanılan alanlar proses bazında birbirinden bağımsızdır. Güncelleme yapılacağı zaman, o proses için ortak alanın bir kopyası çıkarılır. Proses kopya üzerinden devam edeceği için diğer prosesler yapılacak güncellemeden haberdar olamazlar. Buna copy-on-write tekniği denir.

Kuruluş

```
$ gcc -fPIC -O2 -Wall add.c -o  add.o  -c
$ gcc -fPIC -O2 -Wall sub.c -o  sub.o  -c
$ gcc -fPIC -O2 -Wall mult.c -o mult.o -c
$ gcc -fPIC -O2 -Wall div.c -o  div.o  -c

$ gcc -shared -o libtest.so   add.o sub.o mult.o div.o

# veya

$ make
```

Kullanım

```
$ gcc -O2 -Wall -o prog1 prog1.c
/tmp/cc0Bd9f0.o: In function ‘main’:
prog1.c:(.text.startup+0xf): undefined reference to ‘add’
prog1.c:(.text.startup+0x38): undefined reference to ‘mult’
collect2: error: ld returned 1 exit status
```

Kütüphane araması hangi dizinlerde yapıldı?

```
$ gcc -print-search-dirs | grep libraries
```

Kütüphane isimleri libXXX.so şeklindedir ve -lXXX şeklinde derleyiciye verilir.

```
$ gcc -O2 -Wall -o prog1 prog1.c -ltest
/usr/bin/ld: cannot find -ltest
collect2: error: ld returned 1 exit status
```

-l ile kütüphane ismi verilir. Öndeki lib kelimesi ayrıca belirtilmez. Ayrıca varsa sürüm numarası ve .so uzantısı verilmez.

-L ile library path verilebilir.

Eğer dinamik kütüphane hiç bir yerde kullanılmıyor ise -l ile verilse bile, varsayılan davranış olarak, linker tarafından gözardı edilir.

```
$ gcc -O2 -Wall -o prog1 prog1.c -ltest -L.
```

-L birden fazla kullanılabilir.

Kütüphane ismi, absolute veya relative path şeklinde, doğrudan verilebilir.

```
$ gcc -O2 -Wall -o prog1 prog1.c /opt/sysprog/lib/libtest.so

$ gcc -O2 -Wall -o prog1 prog1.c ./libtest.so
```

Statik linker kütüphane aramalarını aşağıdaki sırada yapar. Derleme sırasında verilen -L, en öncekliklidir.

```
$ ld --verbose | grep DIR
```

Derleme sorunsuz ama çalışma zamanında bu kütüphane bulunamaz.

```
$ ldd prog1
linux-vdso.so.1 =>  (0x00007ffd4d2ec000)
libtest.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f408c6cd000)
/lib64/ld-linux-x86-64.so.2 (0x000056132168d000)

$ ./prog1
# ./prog1: error while loading shared libraries:
           libtest.so:
           cannot open shared object file:
           No such file or directory
```

Çalışma zamanında kütüphaneyi bulabilmesi için LD_LIBRARY_PATH tanıtılıp, export edilebilir. export edilmeden kullanılamaz.

```
$ export LD_LIBRARY_PATH=.   # veya
$ export LD_LIBRARY_PATH=/opt/sysprog/lib

$ ldd prog1
linux-vdso.so.1 =>  (0x00007ffe9c3af000)
libtest.so => /nk/workspace/projects/linux.egitimi/sysprog/konular/shared_libs/libtest.so (0x0
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8d6248d000)
/lib64/ld-linux-x86-64.so.2 (0x00005647ed075000)

$ ./prog1
add(2,3) = 5
mult(2,3)= 6

# Nereden yüklendiğini açıkça gör.
$ strace ./prog1

# Ya da doğrudan verilir,
$ LD_LIBRARY_PATH=/opt/sysprog/lib ./prog1
```

Çalışma zamanında arama sırası

Çalışma zamanında kütüphaneler aşağıdaki sırada aranır

- export LD_LIBRARY_PATH ile kabukda verilen dizinler,
- /lib, /usr/lib standard kütüphaneler
- /etc/ld.so.conf dosyasında, ldconfig ile üretilmiştir.

Çok kullanışsız olduğu için RPATH ve RUNPATH gözardı edilmiştir.

Genelleştirme

Kütüphane bilinen bir yere kopyalanır.

```
$ make install
cp libtest.so /opt/sysprog/lib
```

Kütüphanenin yeri ld.so.conf.d/ içine yazılır.

```
$ cd /etc/ld.so.conf.d

$ sudo vi siemens.conf
/opt/sysprog/lib

ldconfig ile cache dosyası oluşturulur.
libtest.so satırı görülmelidir.

$ sudo ldconfig -v
...
/opt/sysprog/lib:
        libtest.so -> libtest.so

# veya

$ sudo -s
$ ldconfig -v | grep libtest.so
```

`ldconfig` programı dinamik kütüphanelerin isimleri ve bulunduğu yerleri ld.so.cache dosyasına yazar.

```
$ ls -l /etc/ld.so.cache
-rw-r--r-- 1 root root 138007 Ağu 27 08:57 /etc/ld.so.cache
```

```
$ ldconfig -p    # print cache

$ ldconfig -p | grep libtest    # print cache
libtest.so (libc6,x86-64) => /opt/sysprog/lib/libtest.so
```

Kütüphane cinsi ve ait olduğu mimari de gösterilir.

Daha sonra dinamik linker (ld.so veya ld-linux.so* programları veya interpreter) bu cache dosyasını okur ve hızlı bir biçimde ilgili dinamik kütüphaneye erişir.

Test

```
prog1'i HOME dizinine taşı.

$ mv prog1 ~

$ ldd prog1
linux-vdso.so.1 =>  (0x00007ffe926dd000)
libtest.so => /opt/sysprog/lib/libtest.so (0x00007fa15ecf8000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa15e92e000)
/lib64/ld-linux-x86-64.so.2 (0x000055aca15fa000)

$ ./prog1
add(2,3) = 5
mult(2,3)= 6
```

Artık bütün programlar libtest.so'yu LD_LIBRARY_PATH tanıtmadan yürütebilir. Fakat derleme sırasında -L ile kütüphanenin bulunduğu dizin verilmelidir.

ld.so

ldd ile bakıldığında dinamik bağımlılığı olan bütün programlarda ld-linux-x86-64.so.2 kütüphanesi görülebilir.

Ya da file ile bakıldığında,

```
$ file prog1
prog1: ELF 64-bit LSB executable,
       x86-64,
       version 1 (SYSV),
       dynamically linked,
```

```
       interpreter /lib64/ld-linux-x86-64.so.2,
       for GNU/Linux 2.6.32,
       BuildID[sha1]=46145578eb4c5a77eab2330ee02262bcba89b3da,
       not stripped

# Ya da readelf ile,

$ readelf -l prog1 | grep interpreter
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Bu çok özel kütüphaneye run-time linker veya interpreter(yorumlayıcı) denir. Bu program yürütme zamanında çalışır.

Bir de compile-time linker vardır. Bu da ld komut ile gerçeklenir ve derleme zamanında, genelde gcc içinde kapalı bir biçimde kullanılır. Compile-time linker elf kodu içine kullanılacak kütüphanelerin isimlerini yazar, aynı zamanda link aşamasında kütüphanelerin tablolarını kullanır.

Dinamik bağımlı bir program çalışacağı zaman önce program sonra yorumlayıcı yüklenir. Yorumlayıcı ld.so.cache'i kullanarak, link aşamasında program içinde isimleri gömülü olan dinamik kütüphaneleri bulur ve gerekli adres çözümlemelerini yapar ve programı başlatır.

64 bit x86 için yorumlayıcı ismi ld-linux-x86-64.so.2 ile verilmiştir. Farklı mimarilerde yorumlayıcı isimleri farklı olabilir.

Bakınız `$ man 8 ld.so`

ldd

Kütüphaneler ldd ile tespit edilebilir. ldd özyineli bağımlılıkları da bulur.

Programın doğrudan bağlı olduğu kütüphanaler readelf ile de bulunabilir.

```
$ readelf -d prog1 |grep NEEDED
0x0000000000000001 (NEEDED)             Shared library: [libtest.so]
0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]

# objdump'da kullanılabilir.

$ objdump -p prog1 | grep NEEDED
  NEEDED               /opt/sysprog/lib/libtest.so
  NEEDED               libc.so.6
```

readelf aynı zamanda kütüphaneler için de kullanılabilir.

```
$ readelf -d libtest.so  | grep NEEDED
0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Böylece kütüphanelerin bağlı olduğu kütüphaneler de tespit edilebilir. ldd bu işi otomatik olarak yapar.

Çalışma zamanında tespit

```
$ strace -e trace=open ./prog1
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/opt/sysprog/lib/libtest.so", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
add(2,3) = 5
mult(2,3)= 6
+++ exited with 0 +++

$ sleep 100 &
[1] 11521

$ lsof -p 11521 | grep ".so"
sleep   11521 nazim  mem    REG    8,1  1868984  654540 /lib/x86_64-linux-gnu/libc-2.23.so
sleep   11521 nazim  mem    REG    8,1   162632  654402 /lib/x86_64-linux-gnu/ld-2.23.so
```

Yorumlayıcılar, yani 64 bit x86 için ld-linux-x86-64.so.2 programı, toolchain içinde bulunur. ve "başka kütüphanelere" bağımlı değildir.

```
$ readelf -d /lib64/ld-linux-x86-64.so.2 | grep NEEDED
$
```

Yorumlayıcı başka kütüphanelere bağımlı olmadığı gibi çok yetenekli bir programdır. Örneğin kütüphanaleri yükelerken, birbirlerine olan bağımlıklarını da tespit edebilir.

`$ man ld.so`

Örnek ldd gibi kullanım

argc, ve arv[] gibi argüman geçişi olmadığından, gerekli argümanlar çevre değişkeni olarak atanırlar.

```
$ LD_TRACE_LOADED_OBJECTS=1  /lib64/ld-linux-x86-64.so.2 ./prog1
```

```
linux-vdso.so.1 =>  (0x00007ffd13cad000)
libtest.so => /opt/sysprog/lib/libtest.so (0x00007fa6f79a1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa6f75d7000)
/lib64/ld-linux-x86-64.so.2 (0x00005639502f6000)
```

LD_TRACE_LOADED_OBJECTS=1 ataması, sadece ELF dosyalar içindir ve programı çalıştırmak yerine, ldd de olduğu gibi, bağımlılıkları tespit eder.

Bir .so programı, standard bir ELF dosyası gibi işletilebilir. Bu özellik sadece interpreter için geçerli değildir.

Örnek

```
$ /lib/x86_64-linux-gnu/libc.so.6
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu9) stable release version 2.23, by Roland McGrath et a
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
        crypt add-on version 2.1 by Michael Glad and others
        GNU Libidn by Simon Josefsson
        Native POSIX Threads Library by Ulrich Drepper et al
        BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
```

Yorumlayıcı hariç, bu özellik çok da kullanışlı değildir. Genelde katalog bilgilerini veya usage bilgilerini içerir.

Her paylaşımlı kütüphanede bu özellik olmayabilir. Eğer bu özellik yani bir "entry point" yoksa, program "core dump" ile son bulur.

entry point'e sahip kütüphaneler x moduna sahiptir.

Örnek

```
$ cd /lib/x86_64-linux-gnu

$ ls -l # x'e sahip olanlara bak.

Ya da
```

```
$ find . -executable -type f

./libc-2.23.so
./libpthread-2.23.so
./ld-2.23.so

$ ./libc-2.23.so
$ ./libpthread-2.23.so
$ ./ld-2.23.so          # ayrıca export ile argüman ister.

$ ./libutil-2.23.so
bash: ./libutil-2.23.so: Permission denied
```

Kütüphanelere entry point eklenmesi

```
# Makefile
#
OBJS=add.o sub.o mult.o div.o info.o
PROG=libtest.so

all: $(OBJS) $(PROG)

%.o: %.c test.h
	gcc -fPIC -O2 -Wall $< -o $@  -c
	ls -l $@
	@echo

$(PROG): $(OBJS)
	gcc -shared $(OBJS) -o $@ -Wl,-e,info
	ls -l $(PROG)
	file $(PROG)

install:
	cp $(PROG) /opt/sysprog/lib

clean:
	rm -f *.o $(PROG)

# Makefile2
#

OBJS=add.o sub.o mult.o div.o
PROG=libtest.so.3.0.1
SO_NAME=libtest.so.3

all: $(OBJS) $(PROG)

%.o: %.c test.h
```

```
	gcc -fPIC -O2 -Wall $< -o $@  -c
	@echo

$(PROG): $(OBJS)
	gcc -shared $(OBJS) -o $@ -Wl,-soname,$(SO_NAME) -Wl,-e,info
	ls -l $(PROG)
	file $(PROG)

install:
	cp $(PROG) /opt/sysprog/lib
	cd /opt/sysprog/lib && ln -s $(PROG) $(SO_NAME) && ls -l

clean:
	rm -f *.o $(PROG)
```

info.c

```
/* info.c */

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

const char service_interp[] __attribute__((section(".interp"))) =
                  "/lib64/ld-linux-x86-64.so.2";

void info(void){

   fprintf(stdout, "\n");
   fprintf(stdout, "Bu test kütüphanesini biz yazdık.\n");
   fprintf(stdout, "Hata olursa bana@yaz.com adresine yaz.\n");
   fprintf(stdout, "Version 0.1\n\n");

   _exit(0);
}
```

Örnek

`$ make clean $ make -f Makefile2`

Örnek

`$ make clean $ make -f Makefile2`

`$ ./libtest.so`

```
Bu test kütüphanesini biz yazdık.
Hata olursa bana@yaz.com aresine yaz.
Version 0.1
```

Kütüphane sürümleri

```
$ cd /lib/x86_64-linux-gnu

$ ls -l libz*
lrwxrwxrwx 1 root root     13 Mar  3 20:52 libz.so.1 -> libz.so.1.2.8
-rw-r--r-- 1 root root 104864 Mar  3 20:52 libz.so.1.2.8

# 3 isim de bulunabilir,

libz.so      -> libz.so.1.2.8    # linker name, sadece kütüphane istenirse
libz.so.1    -> libz.so.1.2.8    # shared object name, majör uyumluluk
libz.so.1.2.8                    # real name, tam uyumluluk

# Genelde
# libz.so.Majör.Minör.Patch
#   Majör geriye uyumluluk yok,   libz.so.3 ile libz.so.2 uyumsuzdur.
#   Minör geriye uyumluluk mevcut libz.so.2.x ile libz.so.2.y uyumludur.
#   Patch her zaman uyumlu        bütün libz.2.5.x'ler uyumludur.
```

Örnek Kütüphanede tam bağımlılığın istenmesi,

```
$ make  # /opt/sysprog/lib/libtest.so'yu kur.

# Test amacı ile sürüm numaraları ekle.

$ mv libtest.so libtest.so.3.0.1

# Doğrudan bu isim ile derle

$ gcc -O2 -Wall -o prog1 prog1.c libtest.so.3.0.1

# Kontrol et

$ readelf -d prog1 | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libtest.so.3.0.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Tavsiye edilen bir yol değildir. Kütüphane sürümünün ilk değiştiği durumda sorun çıkacaktır.

Kütüphanede majör bağımlılığın istenmesi

En genel durumdur.

libtest.so.3 gibi bağımlılık istenirse, statik link sırasında libtest.so.3.0.1 dosyasının aslında libtest.so.3 olarak işleneceği soname seçeneği ile belirtilir.

```
$ make -f Makefile2
...
gcc -shared add.o sub.o mult.o div.o -o libtest.so.3.0.1 -Wl,-soname,libtest.so.3

$ readelf -d libtest.so.3.0.1  | grep SONAME
 0x000000000000000e (SONAME)             Library soname: [libtest.so.3]
```

Dosya ismi libtest.so.3.0.1 olmasına rağmen yorumlayıcı libtest.so.3 olarak arama yapacaktır. Bu durumda libtest.so.3 isimli bir dosya mevcut değildir. Hatayı engellemek için libtest.so.3 sembolik ismi oluştururlur.

```
$ ln -s libtest.so.3.0.1 libtest.so.3

$ ls -l libtest*
lrwxrwxrwx 1 nazim nazim   16 Ağu 29 11:36 libtest.so.3 -> libtest.so.3.0.1
-rwxrwxr-x 1 nazim nazim 8048 Ağu 29 11:33 libtest.so.3.0.1

# Tekrar derle
$ gcc -O2 -Wall -o prog1 prog1.c libtest.so.3.0.1

$ readelf -d prog1 | grep NEEDED
  0x0000000000000001 (NEEDED)             Shared library: [libtest.so.3]
  0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Kütüphane ismi olarak libtest.so.3.0.1 yazılmasına rağmen çalışma zamanında libtest.so.3 aranacaktır. Sembolik link ile hata olması engellenir.

Çalışma zamanında kütüphanelerin sorunsuz kullanılabilmesi için ldconfig ile cache içine eklenebilir.

```
$ cp -a libtest.so.3* /opt/sysprog/lib/

$ sudo ldconfig -v | more

opt/sysprog/lib:
        libtest.so.3 -> libtest.so.3.0.1
```

```
$ ldconfig -p | grep libtest
   libtest.so.3 (libc6,x86-64) => /opt/sysprog/lib/libtest.so.3
```

Sadece kütüphane adının verilmesi

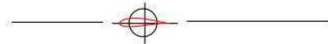
Sembolik link genelde major numaralı kütüphane ismini gösterir.

```
$ cd /opt/sysprog/lib

$ ln -s libtest.so.3 libtest.so

$ ls -l
total 8
lrwxrwxrwx 1 nazim nazim   12 Ağu 29 11:56 libtest.so -> libtest.so.3
lrwxrwxrwx 1 nazim nazim   16 Ağu 29 11:36 libtest.so.3 -> libtest.so.3.0.1
-rwxrwxr-x 1 nazim nazim 8048 Ağu 29 11:33 libtest.so.3.0.1
```

Bu durumda derleme sorasında -ltest seçeneği kullanılabilir. O yüzden bu isme ”linker name” denir.

```
$ gcc -O2 -Wall -o prog1 prog1.c -L/opt/sysprog/lib -ltest

$ ldd prog1
linux-vdso.so.1 =>  (0x00007fff725fe000)
libtest.so.3 => /opt/sysprog/lib/libtest.so.3 (0x00007f5b2e479000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5b2e0af000)
/lib64/ld-linux-x86-64.so.2 (0x000055ec58757000)
```

Dikkat edilirse, her durumda majör uyumlu kütüphane kullanılır.



Dizin shared_libs/

Bölüm 18

plugins

Kütüphaneler yürütme zamanında yüklenebilir. Bu tür kütüphanelere plugin denir. Linux'da plugin'ler, dopen(), dlclose() ve dlsym() komutları ile yönetilirler.

Örnek plugin'i oluşturan programlar için bakınız, add.c, sub.c, mult.c, div.c ve test.h

```
$ make clean
$ make
$ ls -l libtest.so

$ gcc -O2 -Wall -o prog1 prog1.c -ldl

prog1 içinde libtest.so yoktur.
$ ldd prog1
   linux-vdso.so.1 =>  (0x00007fffa4f2d000)
   libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9a4d44e000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9a4d084000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f9a4d652000)


$ ./prog1
evrene dair sorunun cevabı 42
```

Hatalı durum oluşturmak için kütüphane adını değiş.

```
$ mv libtest.so _libtest.so.
$ ./prog1
./libtest.so: cannot open shared object file: No such file or directory
```

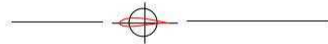
RTLD_NOW dlopen() dönmeden önce shared lib içindeki bütün değişkenler çözülür, kullanıma hazır hale gelir.

RTLD_LAZY Çağrıldığında çözümleme yap.

dlopen(NULL, _) komutunda ilk argüman NULL ise, dlsym() ile arama işlemi, main içinde ve o ana kadar yüklenmiş bütün shared lib'ler içinde yapılır.

Aynı shared object 2. kez yüklenirse bir hata olmaz ve aynı handle döner.

Kütüphane araması, ld.so'daki mantığa göre yapılır.



Dizin dlopen/

Bölüm 19

pkg-config

Yüklü kütüphaneler hakkında bilgi temin eder.

Bir kütühane kullanacaksınız ama

1. include dosyaları nerede?
2. Kütühaneler hangi dizinde yüklü?
3. Kütüphanenin adı ne?
4. Farklı bir dağıtımda bütün bu bilgiler nasıl sabit kalacak?

Bütün bu sorunları pkg-config çok şık bir biçimde çözer.

Esas amaç standard dışı dizinlerde bulunan kütüphaneler hakkında bilgi edinmektir.

Daha çok derleme aşamasında kullanılan -I bilgisi ve link aşamasında kullanılan -L ve -l bilgisi elde eder.

Bunun dışında kütüphane ismi ve tanımı, kütüphanenin URL şeklindeki kaynağı, sürüm numarası gibi pek çok meta bilgi pkg-config tarafından sunulur.

Bu bilgilerin bütünü .pc uzantılı bir dosyadan temin edilir.

Örnek

libxml2 kütüphanesi'nini include dosyası bir dağıtımda /usr/local/xml bir dağıtımda /opt/include içinde bulunsun.

Bu durumda Makefile yazarken bütün bu dizin bilgilerine sahip olmak gerekir. Ama pkg-config bu bilgileri kuruluş sırasında yaratılan .pc dosyasıdan elde eder.

```
$ gcc -O2 -Wall test.c -o test -I/usr/local/xml

# yerine aşağıdaki ifade daha geneldir.

$ gcc -O2 -Wall test.c -o test `pkg-config --Cflags xml`
```

Tabii ki libxml için her dağıtımda .pc dosyasının varlığı kabul edilmektedir.

Örnek

siemens_iot.c

```
# siemens_iot.pc
#
# Bu dosyayı /usr/share/pkgconfig altına kopyala.
#
prefix=/opt/sysprog
exec_prefix=${prefix}/bin
libdir=${prefix}/lib
includedir=${prefix}/include

Name: siemens_iot
Description: iot uygulaması
Version: 3.0.2
Cflags: -I${includedir}
Libs:  -L${libdir} -lsiemens
```

.pc dosyaları genelde aşağıdaki dizinlerde aranır.

```
/usr/lib/pkgconfig,
/usr/share/pkgconfig
/usr/local/lib/pkgconfig
/usr/local/share/pkgconfig

# Ubuntu da
/usr/lib/x86_64-linux-gnu/pkgconfig
```

Bu dizinler dışındaki .pc dosyalarının yerleri PKG_CONFIG_PATH çevre değişkeni ile verilebilir. .pc dosyaları standard yerler dışında ayrıca burada da ararır.

Örnek dosyayı standard bir yere kopyala

```
$ sudo cp *.pc /usr/share/pkgconfig
```

Bu iş aslında kütüphane ilgili makineye yüklenirken otomatik olarak yapılacaktır. Her kütüphanenin .pc dosyası olmayabilir.

```
$ pkg-config --list-all

$ pkg-config --cflags python
-I/usr/include/python2.7 -I/usr/include/x86_64-linux-gnu/python2.7

$ pkg-config --libs python
-lpython2.7

$ pkg-config --cflags --libs python
-I/usr/include/python2.7 -I/usr/include/x86_64-linux-gnu/python2.7 -lpython2.7
```

Nasıl kullanılır?

```
CFLAGS  = -g -Wall
CFLAGS += `pkg-config --cflags siemens_iot`

LDFLAGS += $(pkg-config --libs siemens_iot)   # veya
LDFLAGS += `pkg-config --libs siemens_iot`

# Doğrudan kullanım
$ gcc `pkg-config --cflags --libs siemens_iot` -Wall -O2 main.c -o main.o
```

Esas amaç dağıtımlardan bağımsız Makefile kurmaktır.

Örnekler

Kütüphane isminin sadece bir kısmı biliniyorsa grep ile arama yapılabilir.

```
$ pkg-config --list-all | grep iot

$ pkg-config --cflags siemens_iot
-I/opt/sysprog/include
```

```
$ pkg-config --cflags --libs siemens_iot
-I/opt/sysprog/include -L/opt/sysprog/lib -lsiemens

$ pkg-config --debug --cflags siemens_iot
```

Dizin pkg-config/

Bölüm 20

pipe open

C içinde iken bir komut çalıştırıp sonucunu analiz etmek istiyorum? Örneğin top komutunun başlık bilgileri gerekli oldu?

popen(), dışarıda çalışan bir prosesin çıktı bilgilerini toplar. Prosesin stdin girişine bilgi de gönderebilir.

Bunun için aşağıdakileri yapar

```
create pipe
fork
exec shell
```

pipe tek yönlüdür. Kabuk `/bin/sh -c` ile başlatılır.

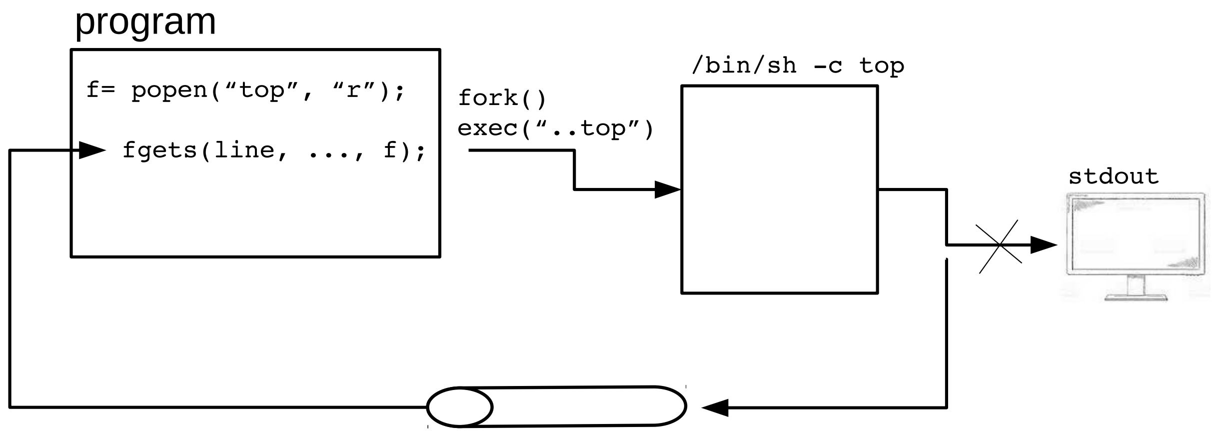
popen() ve pclose() dışında bütün i/o işlemleri yüksek düzeyli i/o ile yapılabilir.

Örnek

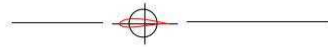
```
$ make
$ ./prog1
```

Programda buf[] içine okunan satırlar, string işleme yöntemleri ile değerlendirilir.

En önemli sorun /bin/sh veya proses de mi hata var, ayrıd edilemez.



Şekil 20.1: pipe open



Dizin popen/

Bölüm 21

daemon

User input'u olmayan proseslere arka planda çalışan proses veya deamon denir. Aslında stdout da kısıtlıdır. Session kapandığı zaman stdout da yok olur.

Başka bir tanım

Herhangi bir terminal session'a bağlı olmayan proseslere daemon denir.

Çalışan bir prosesi daemon yapmak

```
$ ./prog
ctlr+z
$ bg
```

Terminalden daemon yapmak

```
$ nohup prog 1>foo.out 2>foo.err &
```

stdin varsa iptal edilir. stdout ve stderr eğer terminal ile nohup dosyasına otomatik olarak yönlendirilir. Aslında bu durumda program tam da daemon özellikleri göstermez. Çünkü halen bir oturum ile bağlantılır. Halen mevcut dizinde çalışır, bu da tercih edilen bir durum değildir, /'a geçilmelidir. Oturum kapatıldığında tam bir daemon olur.

Örnek

```
$ make P=1
gcc -Wall prog1.c -o prog1

$ tty
/dev/pts/5

$ nohup ./prog1 1>/tmp/foo.out 2>/tmp/foo.err &
[1] 6081

$ cat /tmp/foo.err
nohup: ignoring input

$ cat /tmp/foo.out
i:   0
i:   1
i:   2

$ jobs
[1]+  Running                 nohup ./prog1 > /tmp/foo.out 2> /tmp/foo.err &

$ ls -l /proc/6081/fd
total 0
l-wx------ 1 nazim nazim 64 Ağu 26 09:06 0 -> /dev/null
l-wx------ 1 nazim nazim 64 Ağu 26 09:06 1 -> /tmp/foo.out
l-wx------ 1 nazim nazim 64 Ağu 26 09:06 2 -> /tmp/foo.err

$ ps -e |grep prog1
 6081 pts/5    00:00:00 prog1
```

Terminalden çık,

```
$ ps -e |grep prog1
6157 ?        00:00:00 prog1

$ cat /tmp/foo.out
```

Halen çalışmaya devam eder. Bu türden bütün proseslerin PID değerleri init prosesini gösterir.

C içinden daemon yapmak

`daemon(nochdir, noclose);`

nochdir 0 ise cd /

noclose 0 ise stdin/stdout/stderr=/dev/null yapar.

Genelde (0, 0) kullanılır. POSIX değildir.

Örnek

prog2.c içinde daemon() kapalı iken,

```
$ make P=2

$ tty
/dev/pts/19

$ ./prog2
```

Başka terminalden

```
$ cat /tmp/foo.out
0 1 2 3 4

$ pidof prog2
7012

$ ps -e |grep prog2
 7012 pts/19   00:00:00 prog2
Prosesin bağlı olduğu terminal pts/19

$ ls -l /proc/7012/fd
total 0
lrwx------ 1 nazim nazim 64 Ağu 26 10:02 0 -> /dev/pts/19
lrwx------ 1 nazim nazim 64 Ağu 26 10:02 1 -> /dev/pts/19
lrwx------ 1 nazim nazim 64 Ağu 26 10:02 2 -> /dev/pts/19
l-wx------ 1 nazim nazim 64 Ağu 26 10:02 3 -> /tmp/foo.out
```

prog2.c içinde daemon() açık iken,

```
$ make P=2

$ ./prog2
```

& yapmadığımız halde prompt geri gelir.

```
$ tail -f /tmp/foo.out
0 1 2 3 4 5 6 7 8 9
ctrl+c

$ ps -ef |grep prog
nazim     7066  2371  0 10:04 ?        00:00:00 ./prog2
```

Artık ilişkili terminal yok, ? gözüküyor. PPID olarak init prosesi gözükür. Ubuntu için init prosesi upstart programıdır!

```
$ ls -l /proc/7066/fd
total 0
lrwx------ 1 nazim nazim 64 Ağu 26 10:09 0 -> /dev/null
lrwx------ 1 nazim nazim 64 Ağu 26 10:09 1 -> /dev/null
lrwx------ 1 nazim nazim 64 Ağu 26 10:09 2 -> /dev/null
l-wx------ 1 nazim nazim 64 Ağu 26 10:09 3 -> /tmp/foo.out

$ killall prog2

$ ps -e|grep prog
```

Kontrol et, ölmüş olmalı.

Glibc, tekli fork ile daemon yapar. Bu da çok özel bazı durumlarda hatalara sebep olabilir. Bakınız, $ man daemon, BUG paragrafı.

Yordam

1. fork() ile kopya çıkar
2. exit ile anne'yi kapat
3. setsid() çocuğu yeni session'a bağla, ki bu inittir
4. cd / yap
5. açık dosyaları kapat
6. 0,1 ve 2'yi NULL yap

Örnek daemon() kod parçası, single fork()

```
pid= fork();

if ( pid > 0 ) exit(EXIT_SUCCESS); // parent terminate

setsid(); // init/upstart'a bağlanır, yeni session/group id atanır.

chdir("/");

for(fd= sysconf(_SC_OPEN_MAX); fd>=0; fd--){
   close(fd);
   }

open("/dev/null", O_RDWR);
dup(0);
dup(0);
```

Zombi Kavramı

Anne (parent) proses ölürse, bu prosesin bütün çocoukları, kernel tarafından 1 numaralı prosese evlatlık verilir. Yani proseslerin PPID değerleri 1 olur.

Çocuk ölürse, annesine {pid, termination status,...} gibi bilgilerin bulunduğu ufak bir bilgi kırıntısı gönderir. Bu bilgi anne tarafından alınana kadar çocuk proses zombi durumunda kalır. Bu durumda çocuk proses kernel proses tablosunda yer işgal eder ama gönderdiği bilgi kırıntısı dışında sistem kaynağı kullanmaz.

Anne proses wait() sistem çağrısı ile bu bilgi kırıntısını proses tablosundan çekmelidir. Bilgi kırıntısı wait() ile çekildiği an zombi durumu son bulur.

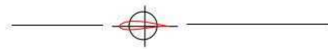
Her durumda, anne proses wait() ile çocukların bitişini takip etmelidir. Bunun için en pratik yollardan birisi, annenin SIGCHLD sinyalini kurmasıdır. Ölen çocuk annesine aynı zamanda SIGCHLD sinyali de gönderir. Anne proses, signal handler içinde wait() ile, çocuktan gelen bilgi kırıntısını, proses tablosundan anında çeker ve hiç zombi proses gözükmez.

SIGCHLD sinyali anne tarafından kurulmaz, yani bu sinyal için bir signal handler yazılmazsa, bu sinyal ignore edilir. SIGCHLD sinyalinin varsayılan eylemi Ignore'dur. Yani gözardı edilir. Bakınız $ man 7 signal

Anne proses ölürse, zombi olan bütün prosesler de otomatik olarak yok olurlar. Yani annenin zombi proseslerine ait bilgi kırıntıları proses tablosundan hemen silinir.

Örnek prog3 ile zombi üretilmesi ve yok edilmesi.

```
$ ./prog3
parent pid 7464
zombiyi öldürmek için ENTER'a basın.
child pid 7465
# top ile kontrol et
zombi öldü, dağılın, tekrar ENTER'a basın

Başka terminalden
$ top -p 7464,7465

zombiyi gör, sonra diğer terminalden ENTER'a bas.
Sonra da zombinin yok oluşunu gör.
```



Dizin daemon/

Bölüm 22

nice

Proseslerin mevcut işlem önceliğini güncellemek için kullanılır.

```
$ man 2 nice  # sistem çağrısı
$ man nice    # komut
```

Hemen her sistem çağrısının glibc'de wrapper karışılığı, benzer veya yakın bir adla mevcuttur. Ayrıca aynı sistem çağrısının komut olarak karşılığı da olabilir.

Örneğin nice, stat, mkdir, chdir/cd, kill, ...

nice, nicer value'den gelir. Kibarlık değeridir.

Ne kadar kibarsan o kadar az hizmet alırsın. Kibarca, öncelikten feragat edilir.

`higer nice -> low priority`

root kullanıcısı negatif değer veya "higher nice" tanıtabilir.

Linux'da nice değeri -20..+19 arasında değişir. Varsayılan değer 0'dır.

Düşük öncelik ile daha büyük CPU zamanı alınır.

`$ nice`

Mevcut nice değerini yazar. Varsayılan değer 0'dır.

`$ nice cmd`

cmd için nice değeri verilir. Çalışan kodun nice değeri renice ile değiştirilir.

```
$ man 2 renice
```

```
$ top
```

Mevcut proseslerin nice değerleri incelenir. -20 olanlar çok fazla CPU zamanı kullanabilirler.

```
$ ps ax -o pid,ni,cmd
```

Mevcut proseslerin nice değerleri daha özet bir çıkışa incelenebilir.

```
/**
 * prog1.c
 *
 * CPU yiyicisi.
 * nice/renice ile CPU zamanı talep edilir.
 *
 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

   for(;;){
      }

   return EXIT_SUCCESS;
}
```

Aynı anda mevcut CPU sayısının 2 kadar proses başlat ve mevcut nice değelerini ve CPU'dan aldıkları zamanı incele.

```
$ make

$ ./prog1 &
$ ./prog2 &
$ ./prog3 &
$ ./prog4 &

$ jobs
$ top -p 5690,5691,5692,5693
```

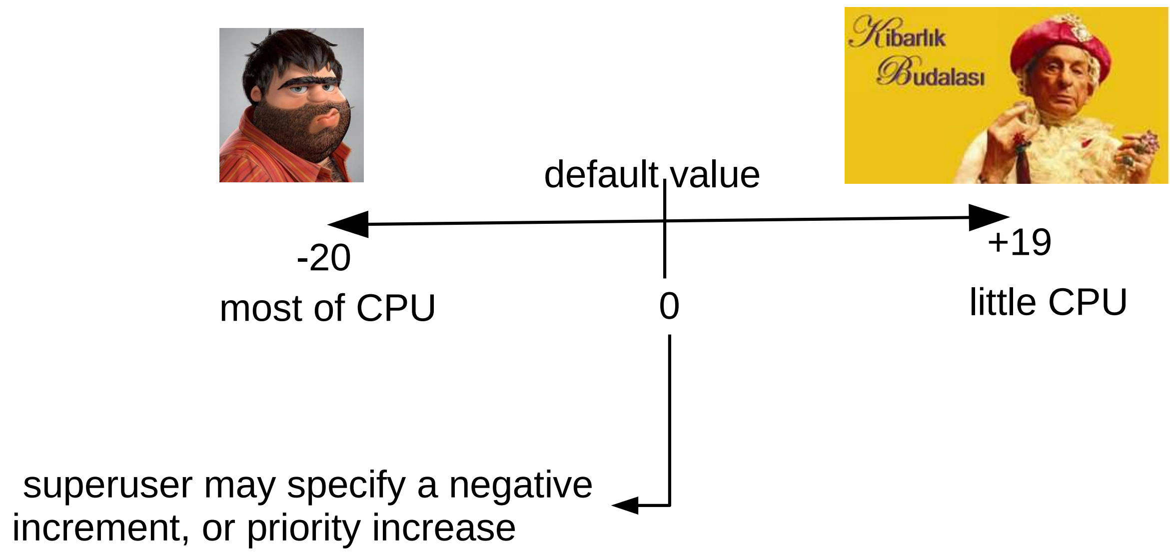
Proseslerin kullandığı CPU zamanlarını kısıtla.

```
$ renice -n 19 -p 5692 -p 5693
5692 (process ID) old priority 0, new priority 19
5693 (process ID) old priority 0, new priority 19
$ top -p 5690,5691,5692,5693

$ renice -n 10 -p 5690
5690 (process ID) old priority 0, new priority 10
$ top -p 5690,5691,5692,5693

$ renice -n -20 -p 5691
renice: failed to set priority for 5691 (process ID): Permission denied

$ sudo renice -n -20 -p 5691
5691 (process ID) old priority 0, new priority -20
$ top -p 5690,5691,5692,5693

$ killall prog1
$ jobs
```

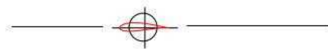
nice değeleri ilk çalıştırıken de verilebilir.

```
$ nice -n 10 ./prog1 &
$ nice -n 10 ./prog1 &

$ sudo -s

$ nice -n -10 ./prog1 &   # doğrudan sudo kullanma!
$ nice -n -10 ./prog1 &

$ top

$ killall prog1
$ sudo killall prog1
```

Dizin nice/



Şekil 22.1: Recep ile Kibarlık Budalası



Bölüm 23

logger, sistem kayıtçısı

Çekirdekten veya proseslerden gelen bilgileri bir dosyada toplayan sistemin bütünüdür.

Örnek C ile log üretimi, test.c

```
$ make

$ ./test
pid 9259

$ tail /var/log/syslog
 ...
Sep  2 15:46:30 nkoc siemens iot[9259]: merhaba, burası dünya.
```

log'lar genelde /var/log/syslog veya /var/log/messages isimli bir dosyaya yazılır.

Bu dosya logrotate ile yönetilir. Bu dosya belirli bir büyüklüğü veya boya erişince yeni dosyaya geçilir. Eskisi sıkıştırılır. Genelde 7 adet dosya tutulur.

```
$ ls -l /var/log/syslog*
-rw-r----- 1 syslog adm  35373 Eyl  2 15:35 /var/log/syslog
-rw-r----- 1 syslog adm 207134 Eyl  2 08:48 /var/log/syslog.1
-rw-r----- 1 syslog adm  67335 Ağu 31 08:32 /var/log/syslog.2.gz
-rw-r----- 1 syslog adm  33960 Ağu 30 06:58 /var/log/syslog.3.gz
-rw-r----- 1 syslog adm  39284 Ağu 29 07:59 /var/log/syslog.4.gz
-rw-r----- 1 syslog adm  33297 Ağu 27 05:56 /var/log/syslog.5.gz
```

```
-rw-r----- 1 syslog adm  31124 Ağu 26 05:59 /var/log/syslog.6.gz
-rw-r----- 1 syslog adm  34751 Ağu 24 08:42 /var/log/syslog.7.gz
```

logrotate crond ile düzenli arlıklarla işletilir. syslog için örnek conf dosyası, Ubuntu'dan,

```
$ cat /etc/logrotate.d/rsyslog

/var/log/syslog
{
rotate 7
daily
missingok
notifempty    # Dosya boşsa, değiştirme

delaycompress # compress varsa anlamlıdır.
compress

postrotate
invoke-rc.d rsyslog rotate > /dev/null
endscript
}
```

syslogd, "system logger deamon" demektir. klogd, yani "kernel logger deamon", syslogd benzeri bir programdır.

Kernel, /proc/kmsg ile mesajlarını user space tarafına gönderir. Eğer /proc yoksa, user space programı system çağrısı ile log talep eder.

/dev/log, unix soketidir.

klogd çalışmıyorsa, kernel mesajları syslog dosyasında gözükmez.

klogd çalışsın veya çalışmasın, kernel mesajları dmesg komutu ile görülebilir. dmesg komutu doğrudan /dev/kmesg karakter cihazını okur.

```
$ ls -l /dev/kmsg
crw-r--r-- 1 root root 1, 11 Eyl  2 08:43 /dev/kmsg

$ strace -e open dmesg > /dev/null
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libtinfo.so.5", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/librt.so.1", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

```
open("/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open("/dev/kmsg", O_RDONLY|O_NONBLOCK)  = 3
open("/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache", O_RDONLY) = 4
+++ exited with 0 +++
```

klogd ve bütün user space prosesler bilgilerini /dev/log soketine yazarlar.

syslogd programının görevi /dev/log soketine yazılan bilgileri /var/log/syslog dosyasına yazmaktır. Bu dosyanın ismi dağıtımıdan dağıtıma farklılık gösterebilir. Bir diğer isim de /var/log/messages'dir.

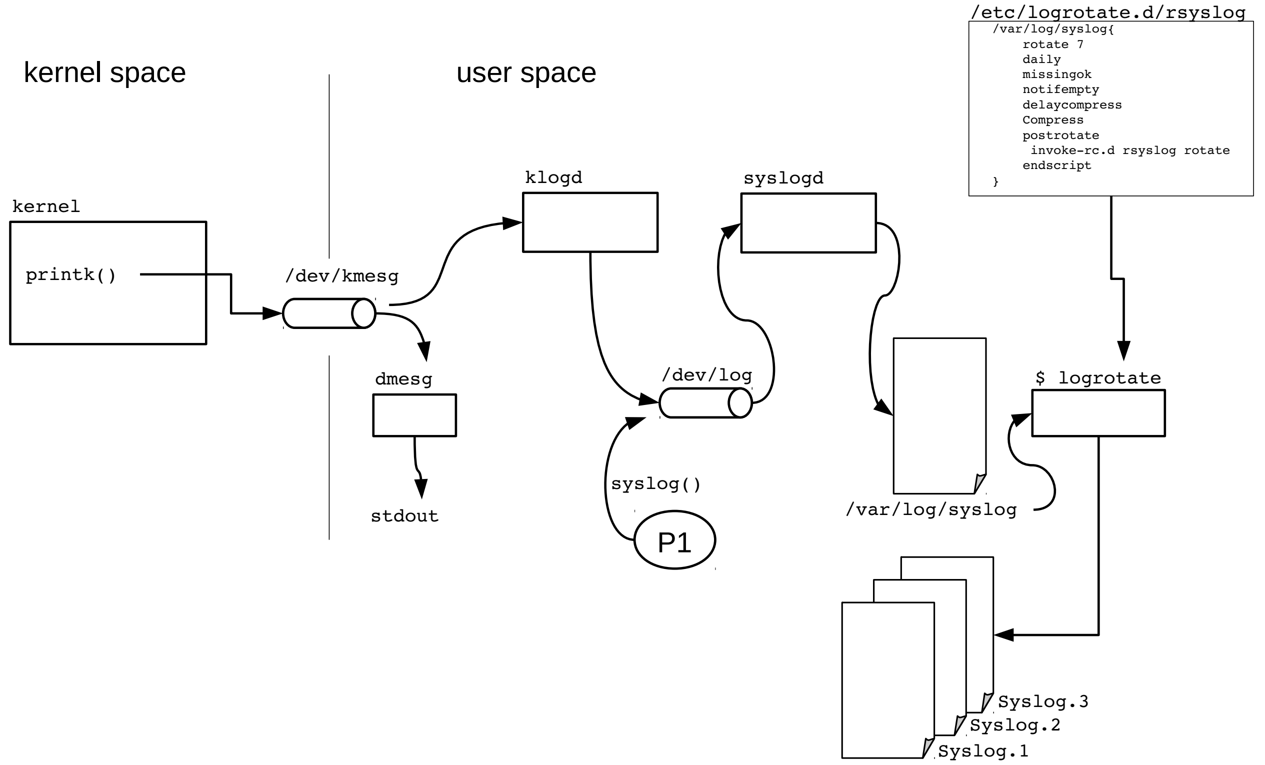
syslog dosyası zamanla çok büyüyebilir. Genelde cron destekli logrotate programı ile log dosyaları yönetilir.

Ubuntu bu standard komutlar yerine rsyslogd kullanmaktadır. Fakat bu program da syslogd projesinden alınmıştır.

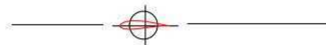
Kabuktan da benzer biçimde log gönderilebilir.

```
$ logger -i  merhaba, burası dünya.

$ tail -f /var/log/syslog
...
Sep  2 17:17:52 nkoc nazim[10737]: merhaba, burası dünya.
```

Dizin logger/



Şekil 23.1: Sistem kayıtçısı



Bölüm 24

udev

devfs dosya sistemi sabittir ve sadece sistemde olan veya olabilecek cihaz isimleri ile ilgilenir.

udev, /dev dizinini yöneten bir uygulamadır ve sabit cihaz isimlendirmesinden gelen sorunları çözer.

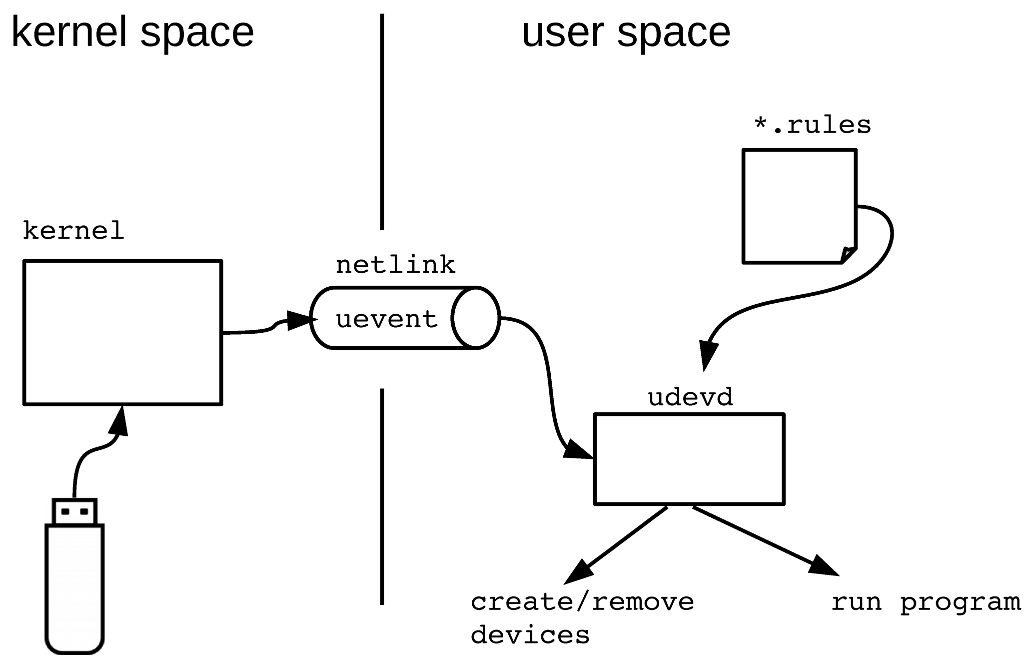
udev aşağıdakileri yapabilir

1. Var olan bir cihaz ismini değiştirebilir.
2. Var olan bir cihaz ismine yeni bir isim verebilir.
3. Olmayan bir ismi, cihaz takıldığı an yaratabilir.
4. Bir programı çalıştırırp sonucuna göre isimlendirme yapabilir.
5. Cihaz ismine chmod ve chown uygulayabilir.
6. Cihaz takılıp çıkarıldığı zaman bir program başlatabilir.
7. Network arabirimlerinin isimlerini değiştirebilir.

Kernel tarafından netlink soketleri üzerinden gönderilen "A=B\0\n" biçimindeki ASCII mesajlarına uevent denir.

Ubuntu'da systemd-udevd.service programı uevent'leri dinler.

Bakınız `$ man 7 udev`



Şekil 24.1: Hot plug sistemi

Dağıtımla gelen kurallar /lib/udev/rules.d/ altında *.rules isimleri ile bulunurlar.

```
$ cd /lib/udev/rules.d
$ ls -l
```

Başta bulunan sayılar sayesinde kurallar ls ile gösterilen sırada çalışırlar. Yani küçükten büyük sıraya doğru yürütülürler.

Yerel kurallar, kullanıcı kuralları ise /etc/udev/rules.d/ altında bulunur.

50-udev-default.rules kuralı içinde varsayılan kurallar (default rules) vardır.
`$ more 50-udev-default.rules`

Örnek satır

```
SUBSYSTEM=="tty", KERNEL=="tty[0-9]*", GROUP="tty", MODE="0620"
#   if                  if                 set           set
```

Buraya ayrıca kural eklenmez. Son derece sabit bir kural dosyasıdır. Fakat 50-'den daha büyük veya 50-'den daha küçük isimle başlayan kural dosyası yazılarak, bu sabit dosyadan önce veya sonra kural yazılabilir. Ama bu kural dosyası doğrudan güncellenmemelidir.

Her kural tam bir satırdır, bölünemez. Kural, key-value ikilileriden oluşur ve "," ile birbirlerinden ayrılırlar.

Soyutlama

```
A=="a", B=="2", C="c", D="d"

Kuralını dengi aşağıdaki gibi düşünülebilir.

if ( A=="a" && B=="2" ){
   C="c";
   D="d";
   }
```

Kurallar önce belleğe okunur sonra işlenir.

Birden fazla kural eşleşirse hepsi işlenir. İlk eşleşen kuralda durulmaz, bütün kurallar baştan sona işlenir.

Her kuralda en az 1 adet "==" ve en az 1 adet "=" ifadesi olmalıdır.

```
== match key
=  assignment key.
```

Örnek, en basit kural örneği

Debug aç.

```
$ vi /etc/udev/udev.conf
udev_log="info"
```

root ile

```
$ cd /etc/udev/rules.d
```

```
$ vi 90-egitim.rules
```

```
KERNEL=="sdb", NAME="egitim_diski"
```

Güncellemeyi haber ver.

```
$ udevadm control --reload   # veya
$ service udev restart
```

Sisteme yeni bir usb takıldığında adının egitim_diski olmasını bekleriz.

```
$ ls -l /dev/egitim_diski
brw-rw---- 1 root disk 8, 16 Eyl  4 09:44 /dev/egitim_diski
```

Ama kural çalışmaz!

```
$ tail -f /var/log/syslog
```

```
Sep  4 10:00:12 nkoc systemd-udevd[4339]: NAME="egitim_diski" ignored,
     kernel device nodes can not be renamed;
     please fix it in /etc/udev/rules.d/90-egitim.rules:1
```

```
$ cd /etc/udev/rules.d
```

```
$ vi 92-egitim-rules
```

```
KERNEL=="sdb", SYMLINK+="egitim_diski"
```

Eski kuralı sil

```
$ rm 90-egitim-rules
```

Güncellemeyi haber ver.

```
$ service service udev restart
```

USB tak

```
root@nkoc:/dev# ls -l /dev/egitim*
lrwxrwxrwx 1 root root 3 Eyl  4 10:07 /dev/egitim_diski -> sdb
```

sdb ismi, varsayılan cihaz ismidir ve sembolik link her zaman varsayılan cihaz ismine yapılır. Bu isim ayrıca belirtilmez.

Operatörler

Kurallarda % ile başlayan operatörler kullanılabilir. En yaygın olanları

```
%k  kernel name   sda
%n  kernel number 3
%%  % işareti
$$  $ işareti
```

Event üretmek için mount/umount komutu kullanılacak.

```
$ cd /tmp
$ dd if=/dev/zero of=test.img bs=1M count=1
$ mkfs.ext2 test.img
$ mkdir disk
```

Kurallarda kullanılan değişkenler nereden elde edilecek?

Örnek, kolay okunması için alt alta yazıldı.

```
99-systemd.rules:
SUBSYSTEM=="block",
ENV{DEVTYPE}=="disk",
KERNEL=="md*",
ATTR{md/array_state}=="|clear|inactive",
ENV{SYSTEMD_READY}="0"
```

Aşağıdaki komutla elde edilen değişkenler, kurallarda ENV{...} şeklinde kullanılanılabilir.

```
$ udevadm monitor --env

$ sudo mount /tmp/test.img /tmp/disk

udevadm terminalindeki çıkış

monitor will print the received events for:
UDEV - the event which udev sends out after rule processing
KERNEL - the kernel uevent

KERNEL[8958.128078] change   /devices/virtual/block/loop0 (block)
ACTION=change
DEVNAME=/dev/loop0
DEVPATH=/devices/virtual/block/loop0
DEVTYPE=disk
MAJOR=7
MINOR=0
SEQNUM=2731
SUBSYSTEM=block
...
```

ATTR değelerinin elde edilmesi

Aşağıdaki komutla elde edilen değişkenler, kurallarda attr{...} şeklinde kullanılanılabilir. attr değeleri, udev tarafından /sys dosya sisteminden elde edilir.

```
$ alias udevinfo="udevadm info "

$ udevinfo -q path -n /dev/sdb1
/devices/pci0000:00/0000:00:14.0/usb4/4-2/4-2:1.0/host8/target8:0:0/8:0:0:0/block/sdb/sdb1
```

-q path query path -n name device name

```
$ udevinfo -a -p $(udevinfo -q path -n /dev/sdb1)

Udevadm info starts with the device specified by the devpath and then
walks up the chain of parent devices. It prints for every device
found, all possible attributes in the udev rules key format.
A rule to match, can be composed by the attributes of the device
and the attributes from one single parent device.

  looking at device '/devices/pci0000:00/0000:00:14.0/usb4/4-2/4-2:1.0/host8/target8:0:0/8:0:0
    KERNEL=="sdb1"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{alignment_offset}=="0"
    ATTR{discard_alignment}=="0"
    ATTR{inflight}=="       0        0"
    ATTR{partition}=="1"
    ATTR{ro}=="0"
    ATTR{size}=="31205264"
    ATTR{start}=="112"
    ATTR{stat}=="      67        0     4072       64        0        0        0        0


$ udevadm info -a -p /sys/class/net/eth0/
  looking at device '/devices/pci0000:00/0000:00:1c.3/0000:03:00.2/net/eth0':
      KERNEL=="eth0"
      SUBSYSTEM=="net"
      DRIVER==""
      ATTR{addr_assign_type}=="0"
      ATTR{addr_len}=="6"
      ATTR{address}=="00:90:f5:dc:51:d2"
      ATTR{broadcast}=="ff:ff:ff:ff:ff:ff"
      ATTR{carrier}=="1"
      ATTR{carrier_changes}=="4"
      ATTR{dev_id}=="0x0"
      ...
```

Kullanıcı A==B olan veya ATTR{...} şeklinde bütün ifadeleri kurallarda kullanabilir.

ENV{A}==B if() gibidir.

ENV{A}=B atamadır. Yani env değişkenleri güncellenebilir.

```
Kurallar sadece tek bir parent için yazılabilir.
Aynı anda 2 parent bilgisi kullanılamaz.
```

Örnek

/dev/lp0 adını /dev/epson_680 haline getir.

Kaynak http://www.reactivated.net/writing_udev_rules.html

```
# udevinfo -a -p $(udevinfo -q path -n /dev/lp0)
   looking at device '/class/usb/lp0':
     KERNEL=="lp0"
     SUBSYSTEM=="usb"
     DRIVER==""
     ATTR{dev}=="180:0"

   looking at parent device '/devices/pci0000:00/0000:00:1d.0/usb1/1-1':
     SUBSYSTEMS=="usb"
     ATTRS{manufacturer}=="EPSON"
     ATTRS{product}=="USB Printer"
     ATTRS{serial}=="L72010011070626380"

My rule becomes:

    SUBSYSTEM=="usb", ATTRS{serial}=="L72010011070626380", SYMLINK+="epson_680"
```

Garip eternet ismini eth0 olarak değiş.

```
$ udevadm info -a -p /sys/class/net/eth0

Udevadm info starts with the device specified by the devpath and then
walks up the chain of parent devices. It prints for every device
found, all possible attributes in the udev rules key format.
A rule to match, can be composed by the attributes of the device
and the attributes from one single parent device.
```

```
  looking at device '/devices/pci0000:00/0000:00:1c.3/0000:03:00.2/net/eth0':
    KERNEL=="eth0"
    SUBSYSTEM=="net"
    DRIVER==""
    ATTR{addr_assign_type}=="0"
    ATTR{addr_len}=="6"
    ATTR{address}=="00:90:f5:dc:51:d2"
    ATTR{broadcast}=="ff:ff:ff:ff:ff:ff"
    ...
    ATTR{dev_id}=="0x0"
    ...
    ATTR{type}=="1"

$ cat /etc/udev/rules.d/70-persistent-net.rules
SUBSYSTEM=="net",
ACTION=="add",
ATTR{address}=="00:90:f5:dc:51:d2",
ATTR{dev_id}== "0x0",
ATTR{type}=="1",
   NAME="eth0"
```

Açık anlamı

```
if ( SUBSYSTEM=="net" && ACTION=="add" && ATTR{address}=="00:90:f5:dc:51:d2" &&
     ATTR{dev_id}== "0x0" && ATTR{type}=="1"
   ){
   NAME="eth0"
   }
```

---

file sistem üzerindeki değişiklikler inotify araçları ile monitör edilebilir.

Örnek

Bir USB takılınca bir kural çalışsın, çıkarılınca başka bir kural çalışsın.

Önce admin programı ile ilgili USB'nin özelliklerini öğren.

```
$ udevadm monitor --env
```

Örnekte gelen değişkenler değil, udev'in kendi içinde atadığı ACTION ve KERNEL değişkenleri kullanılacaktır. monitor sonucu gelen değişkenler kullanıldığında daha hassas USB seçimi yapılabilir.

Yerel kurallar her zaman /etc/udev/rules.d altına yazılır.

```
$ cd /etc/udev/rules.d

$ cat 93-egitim.rules
ACTION=="add",    KERNEL=="sdb", RUN+="/opt/sysprog/etc/usb_add"
ACTION=="remove", KERNEL=="sdb", RUN+="/opt/sysprog/etc/usb_remove"
```

RUN da çalışacak program "/" ile başlamıyorsa, yani absolute path değilse, program /lib/udev altında aranır.

Çalışacak programlar stdin/stdout/stderr ile ilişkili olmamalıdır. Eğer betik varsa mutlaka uygun shebang olmalıdır. Ayrıca her durumda +x modunda olmalıdır.

Örnek betik

```
$ cd /opt/sysprog/etc
$ cat usb_add

#!/bin/bash

F="/opt/sysprog/log/"

if [ "$ACTION" == "add" ]
   then
   F="$F/usb.add"
   logger "USB takıldı.";
   echo "add işlemleri yapıldı" > $F
   else

   F="$F/usb.remove"
   logger "USB çıkarıldı.";
   echo "remove işlemleri yapıldı" > $F
fi

env                 >> $F
echo "PATH $PATH"   >> $F
echo "DATE `date`"  >> $F
```

reload ile kural tababının güncelle.

```
$ udevadm control  --reload

$ tail -f /var/log/syslog
```

usb tak ve çıkart. usd_add ve usb_remove betikleri çalışır.

syslog çıkışı,

```
Sep  4 15:53:23 nkoc root: USB takıldı.
...
Sep  4 15:53:38 nkoc root: USB çıkarıldı.
```

Betikler sistem loguna bilgi yazar ve aynı zamanda /opt/syslog/log dizininde de dosya yaratır. usb.add ve usr.remove isimli dosyalar kurulur.

USB takılınca üretilen dosya aşağıdadır.

```
$ cd /opt/sysprog/log
$ cat usb.add
```

cat add işlemleri yapıldı!!!

```
------ ENV ------
ID_USB_DRIVER=usb-storage
ID_MODEL=USB_DISK
ID_PATH_TAG=pci-0000_00_14_0-usb-0_2_1_0-scsi-0_0_0_0
ID_MODEL_ENC=USB\x20DISK\x20\x20\x20\x20\x20\x20\x20\x20
ID_REVISION=1100
DEVTYPE=disk
.MM_USBIFNUM=00
ID_BUS=usb
SUBSYSTEM=block
ID_SERIAL=SMI_USB_DISK_AA00000000013509-0:0
DEVPATH=/devices/pci0000:00/0000:00:14.0/usb4/4-2/4-2:1.0/host31/target31:0:0/31:0:0:0/block/s
TAGS=:systemd:
ID_MODEL_ID=1000
ID_VENDOR_ENC=SMI\x20\x20\x20\x20\x20
MINOR=16
ACTION=add
PWD=/
ID_PART_TABLE_UUID=c3072e18
USEC_INITIALIZED=25898227449
MAJOR=8
DEVLINKS=/dev/egitim_diski /dev/baska_disk /dev/disk/by-path/pci-0000:00:14.0-usb-0:2:1.0-scsi
ID_VENDOR_ID=090c
DEVNAME=/dev/sdb
SHLVL=1
ID_TYPE=disk
```

```
ID_PART_TABLE_TYPE=dos
ID_INSTANCE=0:0
ID_VENDOR=SMI
ID_USB_INTERFACE_NUM=00
ID_SERIAL_SHORT=AA00000000013509
ID_PATH=pci-0000:00:14.0-usb-0:2:1.0-scsi-0:0:0:0
ID_USB_INTERFACES=:080650:
SEQNUM=3333
_=/usr/bin/env

PATH /usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin:.

DATE Mon Sep  4 15:53:45 +03 2017
```

Buradaki bütün incelik şudur: udev programı env tablosuna pek çok değişken atar. Böylece betikler bu değişkenlere standard yollarla erişir.

PATH ve DATE bilgisi ayrıca yazılmıştır.

Hata arama

test programına /sys dosya sistemindeki cihaz adı verilirse işlem akışı görülebilir. test sistemi kuralları işletir ama action kısmındaki işleri yapmaz.

Eğitim amacı ile yazılan kuralları sil ve reload ile kuralları eski haline getir.

Özet

Özel bir cihaz için kural yazılacaksa

`$ udevadm monitor --env` ve `$ udevadm info` komutu ile değişken, ENV ve ATTR bilgilerini elde et.

/etc/udev/rules.d/NN-falan.rules isimli bir kural yaz.

`$ udevadm control --reload` komutu ile kuralların güncellendiğini udevd'ye haber ver ki kuralları tekrar belleğe okusun.

Test amacı ile cihazı tak, çıkart ve sonuçları izle.

Hata analizi için /etc/udev/udev.conf dosyasına `udev_log=info` satırını ekle.

`$ tail -f /var/log/syslog` ile çekirdek mesajlarını izle.

Cihazın /sys içindeki adını tespit et ve `$ udevadm test /sys/class/block/sdb`

komut ile işleyişi izle. test programı sadece eylemleri gösterir, hiç bir kuralı işletmez.

Veriler kernel tarafından nasıl toplanıyor?

Örnek prog1.c kernel uevent'leri dinler ve ekrana yazar. Diğer bir deyişle `$ udevadm monitor --env`'nin ürettiği sonucu üretir.

```
$ make P=1
$ ./prog1

Başka terminalden aşağıdaki komutları gir ekran çıktılarını incele.

$ sudo -s
$ mount test.img /tmp/disk
$ umount /tmp/disk
# USB tak
# USB çıkart
```

Netlink soketleri kernel ve user space arasında mesaj alışverişi yapabilen, iki yönlü IPC tekniğidir.

`$ man 7 netlink`

İnternet alanı için soket tanımı

`fd= socket(AF_INET , SOCK_STREAM , 0);`

netlink alanı için soket tanımı

```
fd= socket(PF_NETLINK, SOCK_RAW, NETLINK_KOBJECT_UEVENT);
           domain,     type,     protocol

PF_NETLINK communication between kernel and user space
NETLINK_KOBJECT_UEVENT Kernel messages to userspace

struct sockaddr_nl {
     sa_family_t     nl_family;  /* AF_NETLINK */
     unsigned short  nl_pad;     /* Zero. */
     pid_t           nl_pid;     /* Port ID. */
     __u32           nl_groups;  /* Multicast groups mask. */
};
```

Diğer bütün işlemler standard soket işlemleri gibidir. SOCK_RAW veya SOCK_DGRAM aynı anlamdadır.

24.1 Modüller'in otomatik yüklenmesi

```
$ cd /lib/modules/`uname -r`
$ more modules.alias
```

"usb:" geçeni rastgele bir cihaz seç ve modinfo ile driver adını incele

```
$ grep "usb:" modules.alias
...
alias usb:v050Dp0002d*dc*dsc*dp*ic*isc*ip*in* uss720
alias usb:v1293p0002d*dc*dsc*dp*ic*isc*ip*in* uss720
alias usb:v0729p1284d*dc*dsc*dp*ic*isc*ip*in* uss720
...

$ modinfo uss720
 filename:       /lib/modules/4.4.0-93-generic/kernel/drivers/usb/misc/uss720.ko
 license:        GPL
 description:    USB Parport Cable driver for Cables using  \
                 the Lucent Technologies USS720 Chip
 author:         Thomas M. Sailer, t.sailer@alumni.ethz.ch
 srcversion:     2123E82B72AE6CAE821BC3F
 alias:          usb:v050Dp0002d*dc*dsc*dp*ic*isc*ip*in*
 alias:          usb:v1293p0002d*dc*dsc*dp*ic*isc*ip*in*
 alias:          usb:v0729p1284d*dc*dsc*dp*ic*isc*ip*in*
 alias:          usb:v0557p2001d*dc*dsc*dp*ic*isc*ip*in*
 alias:          usb:v047Ep1001d*dc*dsc*dp*ic*isc*ip*in*
 depends:        parport
 intree:         Y
 vermagic:       4.4.0-93-generic SMP mod_unload modversions
```

modules.alias

Çekirdek derlemesi sırasında otomatik olarak kurulur. Modüllerin otomatik olarak yüklenmesi için kullanılır.

Her cihaz sürücüsü kendi cihazını tanıtan bir id yapısına sahiptir. Derleme sırasında bu yapılar modules.alias dosyasına yazılır.

Bu yapı aşağıdaki gibidir.

`kernel 4.12.10, /include/linux/mod_devicetable.h`

```
struct pci_device_id {
    __u32 vendor, device;        /* Vendor and device ID or PCI_ANY_ID*/
    __u32 subvendor, subdevice;  /* Subsystem ID's or PCI_ANY_ID */
    __u32 class, class_mask;     /* (class,subclass,prog-if) triplet */
    kernel_ulong_t driver_data;  /* Data private to the driver */
};
```

Örnek bir pci cihazı için bu yapı, driver yazan kişi tarafından aşağıdaki gibi doldurulur.

```
static struct pci_device_id xircom_pci_table[] =
{
  {0x115D, 0x0003, PCI_ANY_ID, PCI_ANY_ID,},
  {0,},
};
MODULE_DEVICE_TABLE(pci, xircom_pci_table);
```

Çekirdek derlemesi sırasında bu yapıdaki bilgileri, aşağıdaki gibi modules.alias dosyasına yazılır.

```
alias pci:v0000115Dd00000003sv*sd*bc*sc*i* xircom_cb

v  0000115D   vendor id
d  00000003   device id veya product id
sv *          sub vendor
sd *          sub device
bc *          base class
sc *          sub class
i  *          programming interface
```

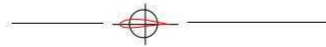
Bu cihaz takıldığı zaman çekirdek bir uevent üretir ve user space tarafına bir cihazın takıldığı duyurusunu yapar. udevadm ile bu duyuru aşağıdaki gibi incelenebilir.

```
$ udevadm monitor --env
...
MODALIAS=pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00
..
```

```
Kernel tarafından üretilen uevent, netlink üzerinden udevd programına ulaşır.
Yazılan kurala göre, genelde aşağıdaki gibi bir komut ile modül yüklenir.

modprobe pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00

Genel udev kuralı her zaman aşağıdaki gibidir.
RUN içinde ENV{...} yazmanın anlamı yoktur, $env{...} ile bu değer elde edilir.

ACTION=="add", ENV{MODALIAS}=="?*", RUN+="/sbin/modprobe $env{MODALIAS}"
```

Özetle

- Cihaz takıldığı zaman, kernel (core driver), pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00 uevent bilgisini üretir ve netlink üzerinden udev'e gönderir.
- udev programındaki kural ateşlenir ve RUN ile verilen modprobe pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00 komutu işletilir.
- modprobe programı modules.alias dosyasını dolaşır ve buradaki pattern ile uyan modülün ismini tespit eder ve yükler. Yükleme yaparken modules.dep dosyasını kullanarak bağımlı modülleri en önce yükler.
- Kullanıcı modül ismini biliyorsa doğrudan insmod veya modprobe ile yükleyebilir. insmod bağımlılıkları yüklemez.
- Ayrıca alias'daki driver isim kullanılarak /etc/modprobe.d/isim.conf adı ile bir dosya kurulabilir. Bazı modül parametreleri buradan verilebilir.



Dizin udev/

Bölüm 25

Alarm sinyali ile zamanlama

Saniye bazında hassasiyetle alarm kurmamızı sağlar.

Kolsaati zamanı (elapsed time) kullanılır.

Süre dolunca, çekirdek prosese ALRM sinyali gönderilir. ALRM sinyaline karşı gelen callback çalışır.

$ kill -l

Linux'da 14 numaralı sinyaldir.

```
alarm(N) ile N saniye beklenir.
alarm(0) ile kurulu alarm iptal edilir.
```

Kalan zaman geri döner. Bu durumda daha evvelden atanmış alarmlardan biri işlemiştir. Dönüş değeri 0 ise atanmış başka alarm yok demektir. Kalan alarm süresi öğrenilemez.

alarm() fonksiyonu, alarmı 1 kez kurar. Periyodik zamanlama için alarm() her seferinde yeniden kurulmalıdır.

Aynı proses'de alarm ve sleep kullanılmaz. Çünkü sleep()'de ALRM ile uygulanmış olabilir.

Sinyal callback fonksiyonları içindeki değişkenler volatile sig_atomic olmalıdır. Ayrıca bütün fonksiyonlar signal-safe olmalıdır. Ayrıca mümkün olduğunda basit ve kısa olmalıdır. Prensipte sadece bir global değişken atanmalıdır.

Örnek

```
volatile sig_atomic_t flag=1;

void handle(int sig){
   flag--;
}
```

fprintf() yerine her zaman write() sistem çağrısı kullanılmalıdır.

Bakınız, `$ man 7 signal` içinde `Async-signal-safe functions`

Örnek

```
$ ./prog1
1503555961 şimdi
alarm kuruldu.
1503555963
1503555965
1503555967
ENTER
$
```

Örnek

Başta alarm(6); ile alarm kur. Yeni gelen alarm eskisini iptal eder, ama iptal ederken de eski alarmdan kalan süreyi döndürür.

```
$ ./prog1
1503556603 şimdi
Birkaç saniye bekle ve ENTER'a bas
17 saniye sonrası çalışacak bir alarm var.
1503556608
1503556610
1503556612
1503556614
ENTER
```

Dizin timer_alarm/

Bölüm 26

Posix Alarm

Standard ALRM sinyali ile zaman tutmak çok da pratik değildir.

ALRM ile

- Aynı anda bir adet zaman tutulabilir.
- Kalan zaman öğrenilemez.
- Periyodik zaman tanımı yoktur.
- Saat olarak sadece kolsaati zamanı kullanılabilir.
- Hassasiyet saniye mertebesindedir.
- Döngü içinde (event loop) kullanılamaz.
- sleep gibi sistem çağrıları da aynı ALRM sinyalini kullandığı için karışıklık meydana gelir.

Bütün bu olumsuz özelliklerin tamamı POSIX zamanlayıcı ile çözülmüştür. Burada keyfi sayıda zamanlayıcı tanıtılabilir ve 4 farklı saat cinsi kullanılabilir.

Saat Türleri

REALTIME Kolsaati zamanıdır, sistem saatinden etkilenir, kullanılması tavsiye edilmez.

MONOTONIC Açılışta sıfırlanan zamandan saymaya başlar, güvenilirdir ve sistem saatinden etkilenmez. Fakat sistemin suspend olduğu zamanları sayamaz.

PROCESS_CPU_TIME_ID Zaman, proses ve thread'lerinin harcadığı user+sys süresi ile ölçülür.

THREAD_CPU_TIME_ID Zaman, sadece çağıran prosesin user+sys süresi ile ölçülür.

Eylemler

ALRM sinyalinde zaman dolunca sadece bir callback fonksiyonu çağrılırdı. POSIX zamanlayıcı da ise 3 farklı eylemde bulunulabilir.

NONE Hiç bir iş yapılmaz. Bu durumda genelde kalan zaman ölçülür.

SIGNAL ALRM de olduğu gibi bir callback çağrılır. Ayrıca callback fonksiyonuna bir adet parametre geçirilebilir.

THREAD Otomatik olarak bir thread yaratılır ve önceden tanımlanan bir fonksiyon çağrılır. Bu fonksiyonu parametre aktarılabilir. thread'in özellikleri de ayrıca tanımlanabilir.

Duyarlılık

```
struct timespec {
       time_t tv_sec;     /* Seconds */
       long   tv_nsec;    /* Nanoseconds */
       };

struct itimerspec {
       struct timespec it_interval;  /* Timer interval */
       struct timespec it_value;     /* Initial expiration */
       };
```

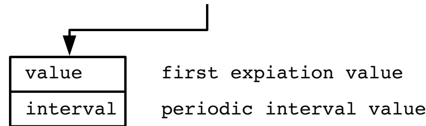
Örnek

`$ man 2 timer_create`

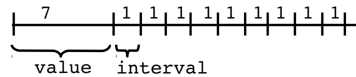
2 adet zamanlayıcı kurulur. Biri saniyede 2 kez, diğeri 4 kez zamanlama yapar. ctrl+c ile çıkılır.

```
$ make
$ ./prog1

signal 34 caught on 1503595861.163  name: birinci zamanlayıcı
signal 34 caught on 1503595861.164  name: ikinci zamanlayıcı
signal 34 caught on 1503595861.413  name: ikinci zamanlayıcı
signal 34 caught on 1503595861.663  name: birinci zamanlayıcı
signal 34 caught on 1503595861.664  name: ikinci zamanlayıcı
signal 34 caught on 1503595861.913  name: ikinci zamanlayıcı
signal 34 caught on 1503595862.163  name: birinci zamanlayıcı
signal 34 caught on 1503595862.164  name: ikinci zamanlayıcı
signal 34 caught on 1503595862.413  name: ikinci zamanlayıcı
signal 34 caught on 1503595862.663  name: birinci zamanlayıcı
signal 34 caught on 1503595862.664  name: ikinci zamanlayıcı
signal 34 caught on 1503595862.913  name: ikinci zamanlayıcı
signal 34 caught on 1503595863.163  name: birinci zamanlayıcı
signal 34 caught on 1503595863.164  name: ikinci zamanlayıcı
signal 34 caught on 1503595863.413  name: ikinci zamanlayıcı
signal 34 caught on 1503595863.663  name: birinci zamanlayıcı
signal 34 caught on 1503595863.664  name: ikinci zamanlayıcı
^Csinyal yedim.
```

timer_setime(timer_id, *val , *old_val);

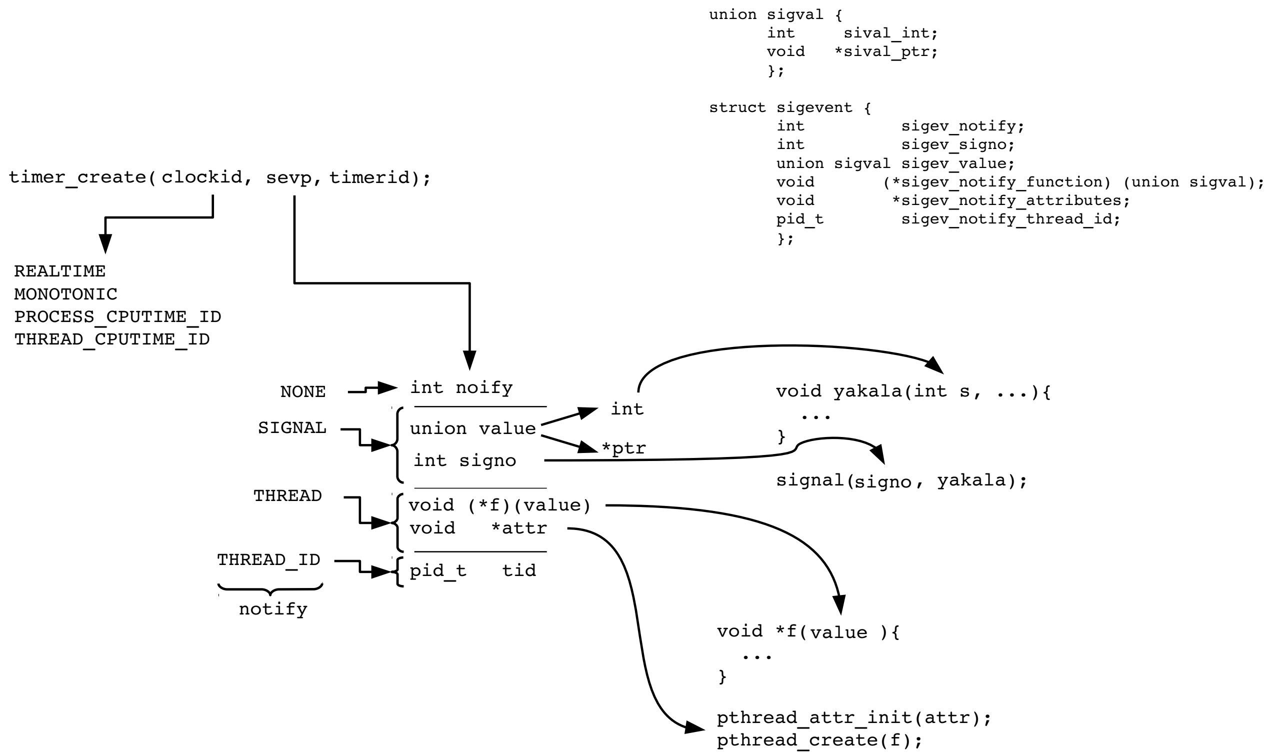



```
struct timespec {
        time_t tv_sec;     /* Seconds */
        long   tv_nsec;    /* Nanoseconds */
        };

 struct itimerspec {
        struct timespec it_interval;  /* Timer interval */
        struct timespec it_value;     /* Initial expiration */
        };
```



7 saniye sonra 1'er saniyelik
periyotlarla timer başlat.

Şekil 26.1: Posix alarm, timer_settime()



Şekil 26.2: Posix alarm, timer_create()

Dizin posix_alarm/

Bölüm 27

fd destekli alarm sistemi

Hem ALRM sinyali hem de POSIX timer işlemleri asenokron'dur. Sinyalin ne zaman geleceği belli değildir.

Asenkron olduğu için signal handler fonksiyonu veya thread kullanılmalıdır. Fakat signal handler içinde kullanılan fonksiyonların async-signal-safe olması gerekir. Bu da büyük bir kısıtlamadır.

Thread kullanım durumunda ise timer sayısı arttıkça prosesi yönetmek zorlaşacaktır.

Bir döngü içinde, sinyal ve thread bağımlılığı olmadan da select() sistem çağrısı ile zamanlayıcı başlatılabilir. select() yerine, benzer senkron i/o multiplexing sistem çağrıları da, pselect(), poll(), ppoll() kullanılabilir. Tek timer varsa, doğrudan bloklu read()'de kullanılabilir.

Bu yöntem çok basit olmasına rağmen Linux'a özgüdür. Taşınabilir kodlar için uygun değildir.

Bu yöntemin genel çerçevesi aşağıda verilmiştir.

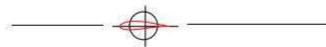
```
// standard bir fd yarat.
fd= timerfd_create(CLOCK_MONOTONIC, TFD_CLOEXEC);

// Başlama ve periyot sürelerini ata.
value.it_value.tv_sec = 2;
value.it_value.tv_nsec= 0;

value.it_interval.tv_sec  = 0;
value.it_interval.tv_nsec = 500 * 1000 * 1000;
```

```
// Süreyi fd'ye ata.
timerfd_settime(fd, 0, &value, NULL);

// readfs hazırla.
FD_ZERO(&readfs);
FD_SET(fd, &readfs);

// max_fd tespit et.
max_fd= fd;

// Ana döngüye gir.
// Artık burada async-signal-safe sıkınıtısı olmayacaktır.
//
for(;;){

   tmpfs= readfs;

   // Bir timer doldu mu select() döner.
   // Ayrıca sinyal durumunda da dönebilir, dikkat.
   //
   res= select(max_fd+ 1, &tmpfs, NULL, NULL, NULL);

   // Sinyal yedik.
   if (res < 0) {
      perror("select()");
      break;
      }

   // Son argümanı NULL olmasaydı,
   // select() 0 dönebilirdi.
   //
   // Burada res== 0 olamaz, çünkü son argüman NULL'dur.

   // fd'ye bağlı timer dolmuşsa, ilgili bit 1 kalır.
   //
   if ( FD_ISSET(fd, &tmpfs) ){
      read(fd, &s, sizeof(uint64_t)); // dönüş kontrol edilmelidir.
      f();              // İstenilen işi oyalanmadan yap ve hemen geri dön.
      res--;            // res== 0 olana kadar, varsa diğer fd'leri incele.
      }

   ...

   } // for;;

// Herhangi bir anda kalan süre elde edilebilir.
//
timerfd_gettime(fd, &value);
```

```
   // fd'ler bilinen yolla kapatılır.
   //
   close(fd);
```

Örnek

```
$ make
$ ./prog

fd1:3  fd2:4  max_fd:4

timer1 1503600330.412
  timer2 1503600330.412
  timer2 1503600330.662
timer1 1503600330.912
  timer2 1503600330.912
  timer2 1503600331.162
timer1 1503600331.412
  timer2 1503600331.412
  timer2 1503600331.662
timer1 1503600331.912
  timer2 1503600331.912
  timer2 1503600332.162
timer1 1503600332.412
  timer2 1503600332.412
  timer2 1503600332.662
timer1 1503600332.912
  timer2 1503600332.912
  timer2 1503600333.162
timer1 1503600333.412
  timer2 1503600333.412
^C
```



Dizin fd_alarm/

Bölüm 28

pipes, unnamed

Prosesler arası ilerişim için kullanılan tek yönlü veri aktarım yöntemidir. Pipe için dosya sisteminde isim verilmediğinden unnamed pipe olarak adlandırılır.

fd değerleri doğrudan kernel tarafından üretildiği için ayrıca bir open() komutu kullanmaya gerek yoktur.

`pipe(fd[]);` komutu ile bir pipe kurulur.

`read(fd[0], ...);` ve `write(fd[1], ...);` yapılır.

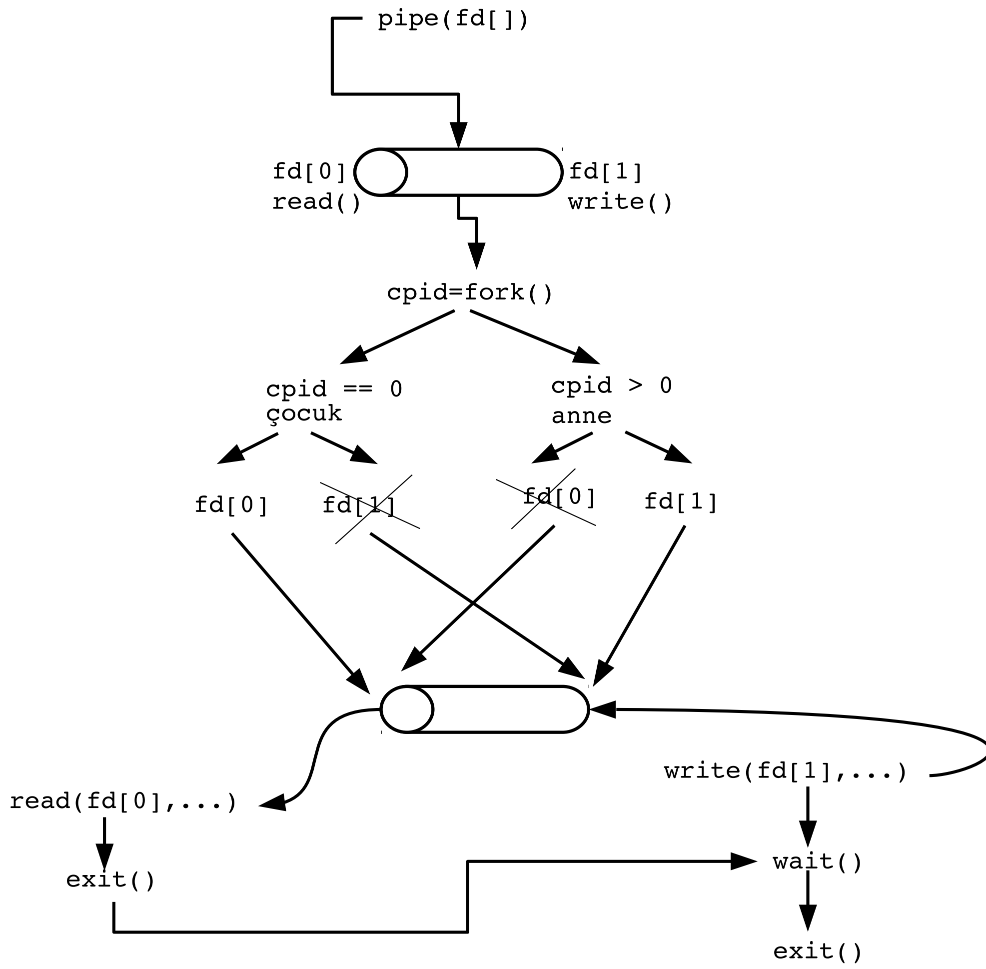
Prosesler arası kullanmak için fork() kullanmak gerekir. fork() yaptıktan sonra kullanılmayan uçlar yani fd'ler kapatılmalıdır. Kapatılmadığı takdirde 2 yönlü kullanımları pratik değildir. İki yönlü iletişim için 2 adet farklı pipe açılmalıdır.

Bağımsız başlatılan prosesler unnamed pipe kullanamazlar.

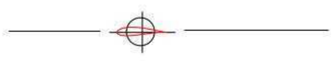
Örnek

```
$ man 2 pipe

$ make
$ ./prog1 merhaba
MERHABA
```

Dizin pipe-c/



Şekil 28.1: pipe in C, unnamed



Bölüm 29

pipe in shell

Farklı proseslerin haberleşmesi için dosya sistemi üzerinde kurulan yapılardır. Tek yönlüdür. İsme sahip oldukları için named pipe veya FIFO denir.

Veriler asla pipe üzerinden geçmez. Dosya ismi sadece ilk açılışta mod ve sahiplik bilgileri için kullanılır. Veriler her zaman kernel seviyesine gider, gelir. Veriler fiziksel olarak pipe dosyasına yazılmaz veya okunmaz.

Daha çok farklı tipte yazılmış proseslerin haberleşmesi için kullanılır.

Örneğin bir proses php diğer proses C ile yazılmış olsun. FIFO üzerinden haberleşebilirler.

pipe, mkfifo ile kurulur rm ile silinir.

C tarafında doğrudan, open(2) komut ile sıradan bir dosya gibi açılabilirler. fd değeri select() ile dinlenemez, her zaman geri döner.

Birden fazla proses aynı FIFO'yu aynı anda okuma ve yazma yaparak kullanabilir.

FIFO aynı anda iki uçtan da açılmalıdır. Bir uç read, diğer uç write açılmalıdır.

Bir uçtaki open(), diğer uçtaki open() çağrısı yapılana kadar prosesi bloklar. İstenirse unblocking modda da açılabilir. Varsayılan açılış modu blokludur.

pipe sadece yerel makinede çalışır, dağıtık değildir.

Örnek Bash uygulaması.

```
$ mkfifo pimas

$ ls -l pimas
prw-rw-r-- 1 nazim nazim 0 Ağu 31 18:19 pimas

$ cat pimas

# Başka terminalden

$ echo merhaba > pimas
$ cat /etc/passwd > pimas
$ echo quit > pimas
```

Sunucu durur.

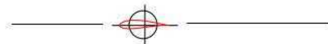
Örnek C uygulaması.

Standard bir dosya gibi işlenir. Fakat tek yönlü olduğu için bir taraf read diğer taraf write açmalıdır.

```
FILE *f;
f= fopen("pimas", "w");
fprintf(f, "%s\n", foo);
close(f);
```

Karşı taraftaki programın ne olduğu hiç önemli değildir.

Düşük seviye I/O da kullanabilir. open(), read(), write() ve close() gibi.



Dizin pipe-shell/

Bölüm 30

Makefile

Amaç programları hızlı ve verimli bir biçimde derlemektir.

Genelde program geliştirme içine kullanılır ama birbirine bağımlılığı olan bütün hiyerarşik işlerde kullanılabilir.

`$ make` komut girildiğinde sıra ile GNUmakefile, makefile ve Makefile dosyaları aranır. GNUmakefile'ın kullanılması tavsiye edilmez. Makefile ismi çok yaygındır. make'den sonra `-` denirse, stdin giriş kabul edilir.

-f ile açıkça dosya ismi verilebilir. Aynı dizinde birden fazla makefile mevcut ise -f faydalı olacaktır.

Kural tanımı

```
hedef: bağımlıklar
        komut
        @komut
        -komut
        @-komut
        ...
```

Veya

```
target : prerequisites     # önkoşullar
        recipe             # action, eylemler
```

Veya

```
target : dependent
        command
```

komut satırının başında mutlaka en az 1 tab ile boşluk bırakılmalıdır.

`@komut` ifadesi varsa, komut ekrana yazılmadan yürütülür.

`-komut` ifadesinde ise hata olsa bile yürütme devam eder.

Makefile

```
test1:
        @X=3
        @echo $$X

test2:
        @X=3 && echo $$X

test3:
        ls foo
        echo bitti

test4:
        -ls foo
        echo bitti

clean:
        rm -f *.o prog1
```

Örnek

Hiç bir hedef yazılmazsa `all` kabul edilir. all yoksa ilk hedefe gidilir. İlk hedef clean gibi bir giriş olmamalıdır.

```
$ make    # veya
$ make test1

test1:
        @X=3
        @echo $$X
```

Üstteki X değeri altta gözükmez. Çünkü her bir satır için yeni bir shell açılır. Doğru yazılışı test2'de verilmiştir.

Örnek

```
$ make test2

test2:
        @X=3 && echo $$X
```

Örnek

Hatalı durumda make hemen yürütmeyi bitirir.

```
$ make test3

test3:
        ls foo       # Yürütme burada biter.
        echo bitti
```

Örnek

Hatalı durum olsa bile devam etmesi için komut önüne `-` konur. `@-` de kullanılabilir.

```
test4:
        -ls foo     # Hatalı durum,
        echo bitti  # Ama yürütme buraya gelir.
```

Sonuç

Her kural için bir kabuk açılır. Birbirlerine bağlı işlemler varsa alt alta değil yan yana && ile yazılmalıdır.

```
test.o: test.c test.h
        komut1 && falan1 && falan2
        komut2
```

hedefin tarihi bağımlıklar kısmında verilen dosyaların hepsinin tarihinden daha büyükse yürütme komut kısmına düşmez, kural es geçilir.

Örnek

```
test.o: test.c test.h
        komut1
        komut2
```

```
if ( date(test.o) > date(test.c) && date(test.o) > date(test.h) ){
   kuralı es geç
   }
   else{
   yeni bir shell aç ve komut1'i işlet.
   yeni bir shell aç ve komut2'yi işlet.
   }
```

Eğer test.o yoksa, tarihi 0 gibi işlem görür. Bu durumda sağında bulunan herhangi bir dosya ile yürütme alta düşer.

Hiç bağımlılık yoksa, yürütme hemen alta düşer.

Yazılan ifade bir dosya değil de bir kural ise açıkça `.PHONY` ile belirtilebilir.

`--debug=b` ile işlemler izelenebilir.

Bütün işleyiş sistem saatine bağımlıdır. Bundan dolayı sistem saati değişirse mutlaka make işlemi sıfırdan yapılmalıdır.

```
clean:
        rm -f *.o
```

Eğer clean diye bir dosya varsa, bu kural çalışmaz. `.PHONY: clean` satırı eklenmeli.

-C seçeneği

```
$ make -C /tmp/foo
cd /tmp/foo && make satırı ile denktir.

# Makefile içinde make kullanımı
foo:
        cd /tmp/foo && make     # veya
        cd /tmp/foo && $(MAKE)
        $(MAKE) -C /tmp/foo
```

Dizin makefile/

Bölüm 31

-g ile derlemek

Müşteriye -g seçeneği ile derlenmiş program verilmişse, kaynak koduda beraber gitmiş demektir. Kodu kaptırmayalım!

Örnek

```
$ make P=1  veya

$ gcc -Wall prog1.c -o prog1 -g

$ ls -l prog1
-rwxrwxr-x 1 nazim nazim 9776 Eyl  2 22:54 prog1

$ gdb prog1
(gdb) run
Starting program: /nk/workspace/projects/linux.egitimi/sysprog/konular/debug/prog1
File :prog1.c
Date :Sep  2 2017
Time :22:54:06
Line :9
ANSI :1
[Inferior 1 (process 17198) exited normally]

(gdb) list 1,100
1        #include <stdio.h>
2        #include <stdlib.h>
3
4        int main(int argc, char *argv[]){
5
6           printf("File :%s\n", __FILE__ );
7           printf("Date :%s\n", __DATE__ );
8           printf("Time :%s\n", __TIME__ );
```

```
9           printf("Line :%d\n", __LINE__ );
10          printf("ANSI :%d\n", __STDC__ );
11
12          return EXIT_SUCCESS;
13       }
14
(gdb) q

$ file prog1
prog1: ELF 64-bit LSB executable, ..., not stripped
                                       ^
                                       ???
```

Müşteriye mutlaka strip edilmiş kod verilmelidir.

```
$ objcopy --strip-all prog1
```

veya derlerken -g kullanma. Ama her durumda kodu küçültmek için strip yapılmalıdır.

```
$ ls -l prog1
-rwxrwxr-x 1 nazim nazim 6312 Eyl  2 22:59 prog1

$ file prog1
prog1: ELF 64-bit LSB executable, ..., stripped

$ gdb prog1

(gdb) run
Starting program: /nk/workspace/projects/linux.egitimi/sysprog/konular/debug/prog1
File :prog1.c
Date :Sep  2 2017
Time :22:57:44
Line :9
ANSI :1
[Inferior 1 (process 17266) exited normally]

(gdb) list 1,100
1        <built-in>: No such file or directory.
(gdb)
```

Dizin debug/

Bölüm 32

Bellek Sızıntılarının Tespiti

mcheck/mtrace sistemi, bellek sızıntılarını tespit etmek için çok basit ve etkin bir yöntemdir.

Bu yöntemde basitçe malloc(), realloc() ve free() fonksiyonları sayılır.

ASLA üretim sistemlerinde kullanılmamalıdır. Hem kodu büyütür hem de inanılmaz derecede yavaşlatır, hem de çok fazla disk alanı kullanır.

Thread'e sahip uygulamalarda düzgün çalışmaz.

Sadece malloc/free ifadelerini değil, fclose() gibi bırakılmayan kaynakları da raporlar.

Eğer Caller olarak isim değil de adres mevcutsa, ya program -g ile derlenmemiş ya da strip edilmiş bir kütüphane çağrısı mevcuttur.

Örnek free(p); yokken.

```
$ export MALLOC_TRACE=mtrace.out
$ gcc -O0 -g -DMALLOC_TRACE=mtrace.out -o prog1 prog1.c

$ ./prog1

$ mtrace ./prog1 mtrace.out

Memory not freed:
-----------------
Address            Size     Caller
0x0000000001023450 0x20  at /opt/sysprog/src/mtrace/prog1.c:17
```

Örnek Kod içinde free(p); mevcut iken. -O0 ile derle ki derleyicinin hışmına uğramayalım.

```
$ gcc -O0 -g -DMALLOC_TRACE=mtrace.out -o prog1 prog1.c

$ ./prog1

$ cat mtrace.out  # malloc/free adresleri ve boyları kaydedilir.

$ mtrace ./prog1 mtrace.out
No memory leaks.
```

Dizin mtrace/

Bölüm 33

inode

inode, dosya sistemindeki dosya/dizin nesnesinin veri yapısıdır. inode, index node'un kısaltması olabilir.

Şekil'de ext2 için inode yapısı verilmiştir.

inode'un incelenme yöntemleri

```
$ stat foo
  File: 'foo'
    Size: 7          Blocks: 8          IO Block: 4096   regular file
    Device: 801h/2049d Inode: 2117778     Links: 1
    Access: (0664/-rw-rw-r--)  Uid: ( 1000/   nazim)   Gid: ( 1000/   nazim)
    Access: 2017-09-02 23:40:54.673854734 +0300
    Modify: 2017-09-02 23:40:58.209816720 +0300
    Change: 2017-09-02 23:40:58.233816462 +0300
     Birth: -
```

```
Access - the last time the file was read
Modify - the last time the file was modified (content has been modified)
Change - the last time meta data of the file was changed (e.g. permissions)
```

Gömülü sistemlerde, r/w bağlanmış disklerin ömrünü uzatmak için, dosya sistemi noatime seçeneği ile mount edilir. Böylece her erişimde inode'un güncellenmesi engellenir.

```
$ ls -li /tmp/foo
$ 2117778 -rw-rw-r-- 1 nazim nazim 7 Eyl  2 23:40 /tmp/foo
```

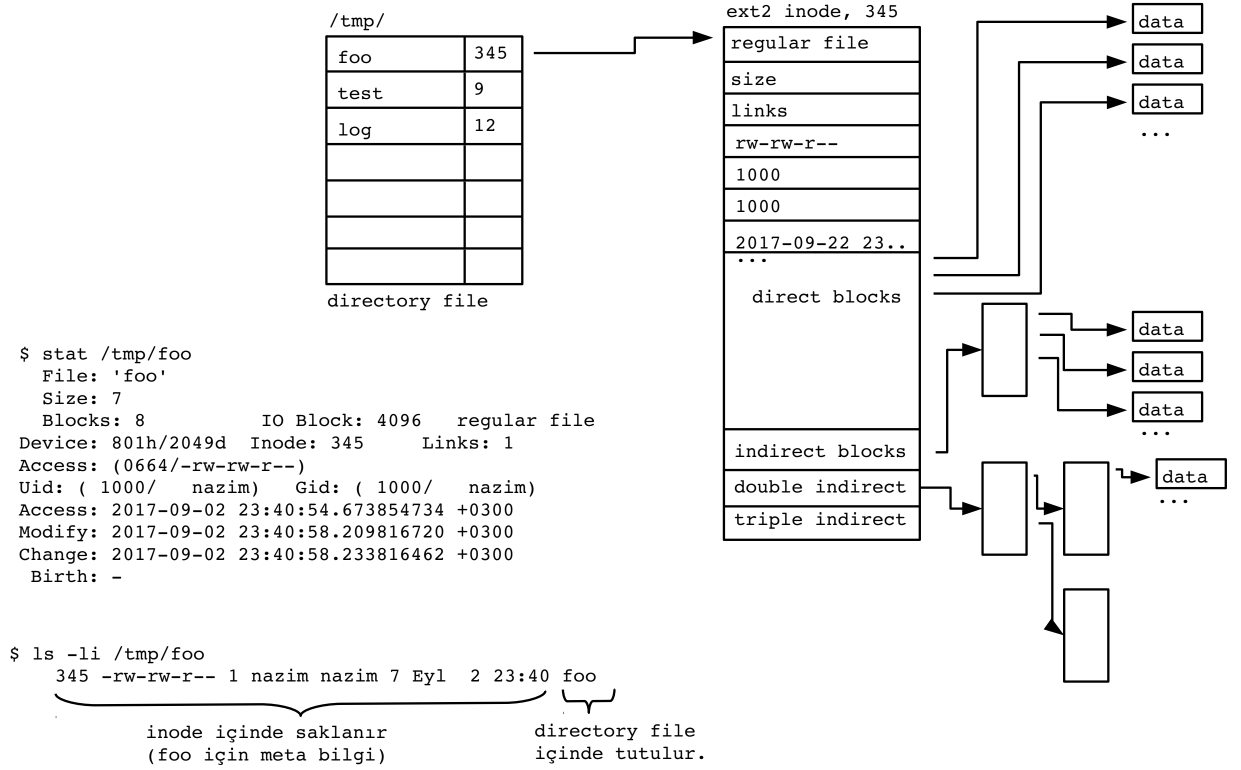
Dosya sisteminde kaç inode var?

```
$ df -i
Filesystem        Inodes   IUsed    IFree IUse% Mounted on
udev              486506     500   486006    1% /dev
tmpfs             491571     767   490804    1% /run
/dev/sda1        3662848  653182  3009666   18% /
tmpfs             491571      48   491523    1% /dev/shm
tmpfs             491571       6   491565    1% /run/lock
tmpfs             491571      18   491553    1% /sys/fs/cgroup
/dev/sda3       15753216 4060835 11692381   26% /nk
cgmfs             491571      14   491557    1% /run/cgmanager/fs
tmpfs             491571      47   491524    1% /run/user/1000

$ df -i .
Filesystem      Inodes  IUsed   IFree IUse% Mounted on
/dev/sda1      3662848 653182 3009666   18% /
```

```
$ sudo tune2fs -l /dev/sda1
```

First inode ve inode size incele. inode bilgisi her zaman sabit ve sürekli bir alanda tutulur, yani dizidir.

Dizin inode/



Şekil 33.1: ext2 inode yapısı