Sistem Programlama

`gcc test1.c -o test1` komutu ile test1.c dosyasını test1 adlı dosya halinde derlemiş oluruz.

`./test1` ile de bu out dosyasını çalıştırabiliriz.

`gcc -S test1.c` komutu ile test1.c dosyasındaki c kodunu assembly diline çevirmiş oluruz ve çıktı olan dosyanın uzantısı .s yani test1.s olur.

Bir dosyayı derlemek o kodu makine diline yani 0-1'lere çevirmek demektir. Ortaya çıkan dosyaya da object dosyası denir.

Kütüphane dosyaları fonksiyonların nasıl çalışacağını belirtir. Örneğin makine printf'in ne yapacağını bilmiyor fakat stdio.h kütüphanesini include ettiğimizde stdio.h kütüphanesi printf'in nasıl çalışacağını, ne yapacağını söyler.

Out dosyası çalıştırılırken loader tarafından memory'e yazma işlemi yapılır ve bu işlemde her satıra bir adres ataması yapılır.

Neden header dosyalarına ihtiyacımız var? → printf de bir fonksiyon ve bunun ne yapacağı da header dosyalarında kodlu ve bunun için header dosyalarına modüler programlama için ihtiyacımız var.

Mainde string args dışarıdan parametre almak için kullanılır.

https://www.youtube.com/watch?v=PDzWHsBE4-w

C'de argv[0] cmd üzerinden programın adını yazarak başlattığımız için, programın adıdır.

C'de setlocale() double ve float değişkenlerde problem yaratmaktadır. Amerikan standartı ".", Türkiye "," olduğundan dolayı.

C'de class yoktur.

C'de String yoktur, charlarla ifade edilir. `char *a` şeklinde string alınabilir. (karakterler dizisi)

`printf("Merhaba Dunya");` şeklinde yazdırma yapılır.

`scanf("%d", &x);` x int veri tipindeki değere alınacak değeri atar.

%d → int

%f → float

%lf → long float yani double anlamına gelir

%c → char

%s → string (char*)

%x → adres yazdırma (yani hexadecimal)

Programın çalışıp kapanmaması için main'e `return 0;` öncesi `getchar();` eklenebilir.

C'de true false yoktur, 0 false, 0 hariç diğer sayılar true'dur. Örneğin if(100) c için true iken javada hata verir.

C'de sizeof, değişkenin bellekteki boyutunu gösterir yani içindeki değer ne olursa olsun default olarak kapladığı alanı söyler → `printf("%d", sizeof(int));` (sizeof int döndürür kaç byte %d)

Bir program içerisinde ya float ya double kullan her seferinde! Yani double ve float birlikte kullanılmamalıdır.

**Float ve double == ifadesinde aynı değere sahip olsalar da sonuç false çıkar. Çünkü double float'tan daha fazla yer kaplar, virgülden sonra daha fazla rakam barındırır.

C'de örneğin `double pi=3.14;` atanmış ve `int a=pi;` denilirse veri kaybı oluşur, yani C otomatik olarak tip dönüşümü yaparak int haline getirir ve pi 3 olur.

C'de const yani sabit tanımlaması yapılıyorsa tanımlandığı yerde muhakkak atama yapılmalıdır, sonradan atama yapılmasına izin vermez.

İzin vermez;

```
const double yercekim; yercekim=10;
```

İzin verir;

```
const double yercekim=10;
```

Aynı zamanda C'de main üzerine `#define degiskenAdi` şeklinde de sabit tanımlaması yapılabilir.

Diziler C/C++'da ve Java'da da ilk elemanın adresi olarak tutulur.

Stringler de C'de char dizileri olarak ifade edildiğinden aynıdır. Stringde *degiskenAdi(%c) ile dizinin ilk elemanına, degiskenAdi(%s) ile tamamına erişebiliriz.

C'de `int x[3];` şeklinde int tipinde 3 elemanlı x dizisi oluşturulur. `int x[]={10,20,30};` şeklinde de yine aynı şekilde stack bellek bölgesinde elemanları 10,20,30 olan 3 elemanlı x dizisi oluşturmuş oluruz. C'de dizinin boyutunun girilmesi zorunludur. Dizi oluşturulduktan sonra C ve C++ dizi elemanlarını otomatik olarak rastgele saçma sapan sayılar yapar. Yani bu dillerde dizi oluşturduktan sonra muhakkak tüm elemanları önce 0 yapılmalıdır.

C'de heap bellek bölgesinde dizi oluşturmak içinse

`int *dizi=malloc(sizeof(int)*diziBoyutu);` şeklinde dizi oluşturulabilir. Eğer tüm elemanlarının rastgele sayılar olması yerine 0 olmasını istiyorsak bunu calloc ile yapabiliriz fakat zaman açısından calloc daha çok vakitte yer ayırma işlemini tamamlar.

C'de [] şeklinde bir dizi oluşturmakla string için karakter dizisi oluşturmak aynı şey değildir. Görüleceği üzere 2. fonksiyondan boş çıktı veriyor çünkü fonk içi tanımlanan str değişkeni bir dizi ve return edilirken fonksiyon bittiği için bellekten silinmiş oluyor ve print etmek isteyince adresi boş olduğu için yazdıramıyor. 1. fonksiyonda ise char *str="Sakarya"'nın double x=10.5 gibi bir ifadeden farkı olmadığı için return edilebiliyor.

```
char* SehirDondur(){
  char* str="Sakarya";
  return str;
}
char* SehirDondur2(){
  char str[]="Ankara";
  return str;
}
printf("%s\n",SehirDondur());
printf("%s\n",SehirDondur2());
->Sakarya
->
```

C'de bir yeri gösterme olayı yani adresi tutma işi, * ile yapılır.

```
int x=100,y=50;
int *p=&x;
int *r=&y;
int *tmp=p;
p=r;
r=tmp;
```

```
printf("%d\n",*p);
printf("%d\n",*r);
->50
->100
```

Pointer yani * adres tutabilen demektir. *p x'in adresini tutuyor, *r y'nin adresini tutuyor yani p x'in, r de y'nin adresini tutuyor. Bunların refere ettikleri yerleri değiştirebilmek için *tmp tanımlayıp değiştiriyoruz ve en sonunda tam tersi duruma geliyorlar.

C'de static olarak tanımlanan değişken static bellek bölgesinde program çalıştığı anda oluşur ve program sonlanana kadar silinmez.

```
void f(){
  static int h=10;
}
int main(){
  ....
}
```

C'de heap bellekte nesne oluşturmak için malloc, heapten belleğe iade için free kullanılır. Malloc ile yer ayrıldıktan sonra ayrılan yerin boyutunda değişiklik yapmak içinse realloc kullanılır. **Free değişkenin kendisini silmez, gösterdiği yeri belleğe iade eder.

```
int *p=malloc(sizeof(int));
*p=100;
printf("%d\n",*p);
free(p);
->100
```

Heapte p int olduğu için int kadarlık yer açtırdık.

***Burada p stackte bir değişkendir, = dediğimiz yerde mallocla anonim heap bellekte int boyutunda bir yer açılır ve p değişkeni oranın adresini gösterir. Dolayısıyla `printf("%x",p);` dediğimizde tutuyor olduğu heap bellekteki adresi, `printf("%x",&p);` dediğimizde ise stack bölgedeki kendi adresini yazdırır.

`int *p=malloc(sizeof(int));` bu 1 elemanlı dizi oluşturmuş olur. Örneğin 9 elemanlı bir dizi heap bellek bölgesinde oluşturmak istersek,

`int *p=malloc(9*sizeof(int));`

Java ve C'de 2 int'i böleceksek ve bölündüğü değişkeni double yaparsak çıkan sonuç doğruyu vermez. Çünkü işleme giren değişkenler arasında o türden bir ifade gereklidir. Java için örn;

```
int a=7,b=2;
double c=a/b;
System.out.println(c);
->3.0
```

Fakat int'lerden birini double olarak değiştirirsek doğru sonucu alabiliriz. örn a'yı double yaparız.

C'de bool olmamasına rağmen aynı işlemi %d yani int yazdırırsak 0, 1 şeklinde ifade eder.

```
int a=4,b=3;
printf("%d",a>b);
->1
```

C'de bool bulunmamaktadır fakat typedef ve enumarate kullanımı ile benzetim yapılabilir.

`typedef enum BOOL{false, true}bool;`

enum tanımlaması "bool" adında bir değişken tanımlar ve bu bool değişken adının yalnızca false veya true olabileceğini ifade eder.

C'de for içerisinde "i" tanımlaması yapılması standart değildir, tavsiye edilmez! Bu şekilde yapılması doğru olandır.

```
int i;
for(i=0;....)
```

Bellek Bağlama

Bir değişkenin erişilebilir bir bellek hücresi ile ilişkilendirilmesi işlemine denir. Bellekte her hücrenin Adres, İsim ve Değer kısımları vardır. Bir bellek hücresinde bir değişkenin tutulma süresine ise **yaşam süresi** denir.

Ram Belleğin görünümü şu şekildedir; RTS → RunTime Stack

.exe dosyasına basıp çalıştırdığımızda derlenmiş kod olarak bellekte tutulur.

RTS (RunTime Stack)

Bütün olay burada dönüyor. Lokal değişkenler, fonksiyon parametrelerinin bulunduğu yerdir. Bir fonksiyon çağırıldığında bunun bir **aktivasyon kaydı** oluşur ve fonksiyonla ilgili değişken, parametreler ve geri dönüş adresi burada oluşur. Fonksiyon (bu for-while

da olabilir) kapandığı yani return yaptığı an o fonksiyonla ilgili her şey silinir. Dolayısıyla RTS'deki yaşam süresi o lokal bölgenin aktif olduğu zaman kadardır. Örn:

```
for (...){
    int a;
    ...}
```

şeklinde bir for döngüsü tanımlı ve a değişkeni içerde tanımlanmış. Dolayısıyla bu for döngüsü başlayıp bittiğinde o a değişkeni silinmiş olur.

Statik Bellek Bölgesi

Bazı programlama dillerinde desteklenmez bile. Bu bölgede global ve static local değişkenler bulunur. Global değişken emir esaslı dillerde bulunur ve kod içerisinde her yerden erişilebilir. Static bellek bölgesinin temel özelliği program başladığı an değişkenin yaşam süresinin başlayıp, sonlanmasıyla bitmesidir.

Heap Bellek Bölgesi

Yaşam süresini bizim belirleyebildiğimiz bölgedir. Bellek hücresinde bulunan Adres, İsim ve Değer'den ismi heap bellek bölgesinde kaybederiz. Bu yüzdendir ki heap bellek bölgesindeki elemanlarla iş yaparken isimle değil adreslerle uğraşırız. Gelişmiş dillerde bunu bir nebze kolaylaştırmaya yönelik referans olayı vardır. Örneğin Java'da `Kisi k=new Kisi();` dediğimizde k aslında RTS'de bir değişkendir fakat adres olarak gösterdiği yer Heap bellek bölgesindedir. Yani k new Kisi() ile heap bellek bölgesinde oluşturduğumuz bellek hücresinin bir referansıdır. k nesnenin kendisini temsil etmiyor, referansı. Bu işlem sonrası `Kisi a=k;` dersek a isimli yeni bir bölge oluşturmaz. Burada yaptığı iş yine a RTS bölgesinde bir değişkendir fakat gösterdiği yer k'nın gösterdiği yerdir.

new kelimesi Heap bellek bölgesinde yer oluşturmak anlamında gelir. Yani new olan yerde dönen şey her zaman adrestir.

C dilinde new, malloc'tur.

Heap bellekte açılan bu yeri boşalltmak için C'de free kodu bulunur.

static olarak bir int değişken tanımlanırsa ve buna atama yapılmazsa bu durumda default olarak onun değeri 0'dır. Örn `static int i;`

extern anahtar kelimesiyle test1.c source dosyasında tanımlanmış bir değişkeni test2.c source dosyasında okuyabiliriz. Örneğin test1.c dosyasında global scopeda tanımlanmış int i=3;'ü test2.c dosyasında global scope'da extern int i; dediğimizde mainde 1 arttırırsak bu durumda çıktı olarak 4 görürüz.

Pointerlerin değişken tipinin ne olduğu fark etmeksizin 32bitlik sistemde boyutu 4 byte, 64bitlik sistemlerde 8byte'dır.

Örneğin int 4byte yer kaplar ve int değişken oluşturduğumuzda bellekte 4byte ardışıl olarak ayrılır. İlk byte'ın adresi 0x7fff2efcdd9c ise 4.byte'ın adresi 0x7fff2efcdd9f'dir.

Pointer tanımlandığında null tanımlaması yapılmalıdır ki memoryde saçma sapan bir yeri göstermesin.

Struct'larda data için memory içerisindeki değişkenlerin boyutu kadardır. Fakat memoryde struct için ayrılan boyut int içeriyorsa 4'ün, double içeriyorsa 8'in, short içeriyorsa 2'nin katı olmalıdır. Örneğin;

```
typedef struct {
  char b;
  int i;
} Char_Int;
```

bu struct için char+int boyutu 5 byte'dır fakat bu struct'ın memoryde kapladığı alan 8byte'dır. Memoryde alanlara 0x..1,2,3,4,5,6,7,8 dersek char öncelikle 0x..1'e daha sonra ise int struct 8'in katları olacağı için 0x..5'e konumlandırılıyor ve arada kalan 0x..2,3,4 unused alan oluyor. Struct içerisinde bellek ayrımı değişkenlerin tanımlanma sırasına göre yapılır!!!! Fakat önce int sonra char tanımlanmış bile olsaydı yine struct

boyutu 8byte olacaktı çünkü int 4 byte olduğundan struct boyutu da bellekte erişebilmek için 4'ün katı olmak zorunda.

Bir hexde örneğin 2710 hex ise bu 2byte'dır.

Pointer aritmetiğinde sonuç, değişkenin tipinin byte'ına bölünür.

strdup() fonksiyonu içerisinde `strcpy(malloc(strlen(s)+1), s);` çalıştırır yani heap bellek bölgesinde stringin kopyasını oluşturur. Dolayısıyla kopyayla işimiz bittiğinde free ile heap bellek bölgesinden kopyayı boşaltmamız gerekir.

Struct gibi union da vardır. Struct'da tanımlanan değişkenlerin boyutu kadar boyutu oluyor fakat unionda union'ın boyutu içerisinde tanımlanan boyutu en büyük değişken kadardır. Fakat kullanım esnasında da bu tanımlanan değişkenlerden yalnızca bir tanesi kullanılabilir.

df → sistem diskinin kullanılabilir durumunu kullanımını gösterir

pwd → bulunulan klasörün yolunu gösterir

cd → klasörler arası geçiş

ls → klasörde bulunan dosyaları gösterir

ls -l → klasörde bulunan dosyaları detaylı gösterir, sonunda -x olanlar çalıştırılabilir dosyalardır

ps x → o anda aktif olan processleri listeler

ps 10 → 10 numaralı process e ait detayları görürüz

ps -o ppid 10 → 10 numaralı process'in parent pid'ini gösterir

ps x | grep x → x ile alakalı aktif olan processleri listeler

kill -l → default tanımlanmış sinyalleri listeler

İşletim sistemi sistemde hata oluştuğu zaman errno isimli bir global tamsayı değişkende hatayı tanımlayan numara saklar. Bu numaralar sistemde tanımlanan numaralardır ve herbirinin bir anlamı vardır. Bu hata kodlarının header dosyası /usr/include/asm-generic altında bulunur.

errno.h include edildikten sonra %d, errno şeklinde hata kodu bastırılarak hata saptanabilir.

assert.h kütüphanesi include edildiğinde assert fonksiyonu bir parametre bekliyor ve bu parametredeki koşul yanlışsa programın akışı durur. Örneğin program argv[1] dosya adı bekliyorsa ve beklediğimiz dosya adını strcmp ile karşılaştırırsak ve bu yanlışsa programın akışını durdurur. Veya assert(argc==2); ile de kontrol yapılabilir, 2 değilse program sonlanır. #include <assert.h> üzerine #define NDEBUG eklenirse tüm assert komutları iptal olur, işletilmez.

Bir process başka bir processden türetilir ve bu son üretilen process'i üreten process e parent process, üretilen process e ise child process denir.

getpid() ile process id, getppid() ile parent pid öğrenilebilir.

Sinyaller programa gönderilen bir çeşit kesmedir(interrupt). Sinyal oluşması durumunda programın nasıl davranacağı belirlenebilir. Belirlenmezse default olarak tanımlanmış davranışlar gerçekleşir. Örneğin Windows'da Ctrl+C default bir sinyal davranışıdır. Bu sinyalin ismi ise SIGINT'tir. Segmentation ihlali için ise SIGSEGV sinyali default tanımlanmıştır.

Bir process'i sonlandırmak için terminalden kill -9 pid komutu yazılabilir. Buradaki -9 SIGKILL sinyalidir. Bir process'i durdurmak için -20(SIGSTOP), yeniden devam ettirmek içinse -18(SIGCONT) kullanılabilir.

Bir sinyal oluştuğunda yapılacak işlemleri tanımlamak için signal(sinyalTipi, sinyalFonk) fonksiyonu kullanılabilir. (void sinyalFonk(int signum){})

sinyalFonk yerine SIG_IGN yazarsak bu durumda bu sinyali ihmal etmiş oluyoruz. Örneğin signal(SIGINT, SIG_IGN); dersek bu durumda interrupt durumunda herhangi bir şey olmayacaktır. Belli bir bölgede örneğin interrupt'ı işlevsiz kılmak ve sonrasında tekrar default haline döndürmek istiyorsak bu durumda ise signal(SIGINT, SIG_DFL); diyebiliriz.

Bir sinyal icra edilirken sinyal içerisinde bir başka sinyalin icra edilmesi durumunda yeni icra edilen sinyalin işi yapılır ve sonrasında ilk icra edilen sinyalin işi biter. (Sinyal içerisinde sinyal)

signal(SIGALRM, alarmHandler) ile sonrasında alarm(1) yazılırsa 1 saniye sonra SIGALRM sinyali oluşturulur.

open() bir sistem çağrısı iken fopen() bir fonksiyon çağrısıdır. Fonksiyon çağrısının sistem çağrısından farkı sistem çağrılara donanıma erişerek ve bilgileri recover ederek işlem yapar. Zorunda kalınmadıkça fonksiyon çağrısı kullanılır. Sistem çağrıları aynı zamanda çok maliyetlidir, süre vs.

open fonksiyonu bir fd int değer (dosya numarası) döndürür ve bundan sonra işlemler bu int değer üzerinden yürütülür;

Dosya yoksa R ve W için dosya yoksa -1, dosya varsa 3 döndürür. Sonraki açılan dosyalar için 4, 5 .. olarak bu değer os tarafından verilmeye devam eder. Örneğin 3 değerine sahip dosya close ile kapatılırsa sonraki açılacak dosyaya 4, 5 dahil olsa dahi 3 boşa çıktığı için 3 verilir. 3'ten başlamasının sebebi 0 standart input, 1 standart output, 2 ise standart error için kullanılıyor. Örneğin read(0, ..) dediğimizde standart inputtan oku demiş oluruz.

```
int fd=open("txt/in1.txt", O_RDONLY);
```

O_RDONLY → sadece okuma

O_WRONLY → sadece yazma

O_RDWR → okuma ve yazma

O_APPEND → dosyanın sonuna ekleme

O_CREAT → dosya mevcut değilse oluşturma

O_TRUNC → dosya varsa içeriğini silme

Dosyalar için mod da bulunur. rwx gibi ifadeler izinleri belirtir r read, w write, x execute anlamına gelir ve 3 blok halindedir. rwx-rwx-rwx ilk ifade user, ikincisi group, üçüncüsü ise other kullanıcıların izinlerini ifade eder. ve bunların total 8 ihtimali olduğu için 3 bit binary ile ifade edilirler. 001 execute, 010 write, 011 wx, 100 read .. anlamına gelir.

Örneğin open şu şekilde de kullanılabilir;

```
int fd=open(”txt/out2.txt”, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

Burada da sadece yazma izni veya yoksa oluşturu veya varsa da içeriğini sıfırlamasını söyleyerek sonda da izinleri belirtmiş oluyoruz.

0644 → rw-r--r--

close kullanımı;

```
close(fd);
```

Açık olmayan dosya kapatılmaya çalışırsa program hata verir.

read kullanımı;

`int size=read(fd, c, 10);` → fd dosyasına daha önce yer ayırtılmış olan string c’ye ilk 10 karakteri yaz demek. Read fonksiyonu kaç karakter aldığını da int olarak döndürür. Örneğin 100 karakteri okumak isteseydik fakat dosyada 50 karakter olsaydı size 50 dönerdi. Bu aldığımız size değeri ile c’nin size’ıncı indexine \0 ekleyerek string’in sonunu belirlemiş oluyoruz.

```
c[size]=’\0’;
```

write kullanımı;

```
int size=write(fd, c, strlen(c));
```

fd dosyasının başına c stringini yazar. size’a da kaç karakter yazdırdığımızı döner.

lseek kullanımı;

SEEK_SET → cursor’ı dosyanın başına götürür

SEEK_CUR → cursor’ın mevcut konumunu int döndürür

SEEK_END → cursor’ı dosyanın sonuna götürür

```
int i=lseek(fd, 0, SEEK_CUR);
```

```
lseek(fd, 0, SEEK_SET);
```

0 yerine 6 kullanılsaydı dosyanın başından 6 karakter ileri giderdi. SEEK_END içinse -6 kullanılsa sondan 6 karakter geri gider.

cmd time komutu uygulama çalıştırıldığında icra edeceği süreyi gösterir. Örn;

```
time ./simpcat1 < large.txt > /dev/null
```

Burada simpcat1 programını çalıştırıyo ve ona standart input olarak large.txt'yi veriyoruz. > ile de bunun çıktısının nereye basılacağını belirtiyoruz. Buraya herhangi bir txt dosyası da verebilirdik. /dev/null'un anlamı sanki cmd ekranına yazıyormuş gibi simüle etsin fakat ekranda çıktısı görünmesin demek.

buffering → bir seferde birden fazla byte okuma ve yazma

Çok fazla sistem çağrısını engellemek adına buffering kullanılır. fonksiyon çağrıları bu buffering'i arkaplanda kendisi yapar. Bundan dolayı sistem çağrılarına göre maliyetleri düşüktür.

df -i → inode sayılarını verir

Inode klasör ve dosya bilgisi verir. Bulunulan dizin, bir üst dizin ve bulunulan dizindeki dosyaların inode numaraları vardır.

ls -li → hangi inode'un neye ait olduğu, burada görülen her satıra metadata denir. Yani inode dosya hakkında bilgilere.

İşletim sistemi inode aracılığı ile dosyayı bulur.

Hard Link → ln komutu ile oluşturulur. Örneğin klasörde f1 dosyası var fakat f2 dosyası yoksa, ln f1 f2 dediğimizde f2 dosyası f1 ile hard linked olmuş olur ve aynı inode numarasına sahip olurlar. Dolayısıyla f1 ve f2 aynı dosyayı gösterir. Dosyaların içeriği de aynıdır. Klasörler için hard link oluşturulamaz!

Hard Link bağlantı yapıldığında f1 veya f2 içeriğinde bir değişiklik yapıldığında inode numaraları aynı olduğundan her bu değişiklik her iki dosyada da gerçekleşir.

chmod → dosya izinlerini değiştirmeyi sağlar. Örn;

`chmod 0644 f1` → f1'in user, group ve others izinlerini günceller. Hard link bağlılarsa bu değişiklik f2'ye de uygulanır.

rm komutu ile link silinebilir. Örn f1'i rm f1 ile silsek dahi f2 silinmez.

C compiler ile bir derleme yapıp programın adını f2 yaparsak bu durumda f2 ile f1 artık aynı inode numarasına sahip olmaz.

c programında `chmod("f1.txt", 0644)` şeklinde veya `chmod(fd, 0644)` şeklinde dosyanın izni düzenlenebilir.

Yeni oluşturulan bir dosya 2 adet link içerir. Çünkü her oluşturulan klasör . dizini ve .. dizini içerir. Bu yeni oluşturulan klasörün içinde bir klasör daha oluşturulsa bu durumda link sayısı 3'e çıkar.

Soft Link → ln-s komutu ile oluşturulur. Soft linklerin erişim izinleri 777'dir. Oluşturulan soft link dosya aynı inode numarasına sahip olmaz fakat oluşturulan soft link dosyaya açılmaya çalışıldığında aynı inode numaralarına sahip olmamalarına rağmen soft linkinin oluşturulduğu dosyaya gider. Dolayısıyla f2 dosyasında oluşturulan değişiklik f1'e de, f1 üzerinde oluşturulan değişiklik f2'ye etki eder.

Yani soft linkler kısayoldur.

chmod ile f2 üzerinde yapılacak izin değişikliği f2'ye etki etmez.

rm ile f1'in silinmesi durumunda sonrasında f2'ye erişmeye çalışırsak f2'ye eriştiğimiz zaman f2 üzerinden f1'e gittiği için hata alırız.

Klasörün soft linki oluşturulamaz!

Inode numarası ile bir dosyayı silmek için;

```
find -inum 121212 -delete
```

Shell Yönlendirme;

> d → standart output'u d dosyasına (yoksa oluşturur) yazar, içeriği silerek yazar.

>> d → standart output'u d dosyasına yazar, içeriği silmeden yazar. (bir alt satıra)

< d → dosya içeriğini standart input'a yönlendirir.

`cat dosya1 dosya2 > dosya3` ile dosya1 ve dosya2'nin içeriklerini dosya3'e yazabiliriz. dosya1 veya dosya2 mevcut değilse dosyanın olmadığı hatasını ekrana basar fakat

dosya3'ü cat ile baktığımızda sadece mevcut olan dosyanın içeriğini dosya3'e eklediğini görürüz.

`cat d1 d2 2>h1` dediğimizde errorları h1 dosyasına yazdırmış oluruz. Örneğin d2 mevcut değilse mevcut olmadığının hata mesajını h1 dosyasına yazdırmış oluruz. Yani normalde terminalde bu hatayı görecekken bu hatayı h1 dosyasına kaydetmiş olduk.

`cat d1 d2 >& d3` ile hataları ve dosya içeriklerini d3'e yazdırabiliriz.

Eğer dosyayı fopen ile açmadan 3 fd numarası ile dosyaya bir yazma işlemi gerçekleştirmek istersek bu durumda bad file descriptor hatası alırız. Şayet `./dosya 3>sonuc.txt` komutunu yazarsak bu durumda 3 fd numaralı dosya açılır, sonuc.txt dosyasına yazma işlemi yapılır ve hata almayız.

stat sistem çağrısı bir dosya ile ilgili bilgileri verir.

`stat x.txt` şeklinde komut satırında bilgileri verir.

`struct stat buf; int exists=stat("./x.txt", &buf);` şeklinde çalışır. Bu şekilde dosya bilgileri alınmış olur. exists -1 ise dosya bulunamamıştır.

Sonrasında buf.structVariable ile dosyanın bilgilerine erişilebilir.

stat struct'ında dosya adı bilgisi bulunmaz. Dosyanın boyut bilgisi, hard link sayısı ve inode numarası gibi bilgiler bulunur.

stat lseek sistem çağrısı yerine kullanılır, lseek gibi satır satır ilerlemez ve daha az maliyetlidir.

Bulunulan dizindeki tüm dosyalar için bir işlem yapmak istersek de dirent struct'ını kullanırız. Bunu kullanırken de readdir, writedir, opendir, closedir kavramları kullanılır.

dirent struct'ı üzerinde dosya tipi, inode numarası, dosya adı bilgileri bulunur.

```
struct stat buf;
int exists;
DIR *dir;
struct dirent *de;

dir=opendir("."); //bulunulan dizin açıldı
if(dir == NULL) {
```

```
  fprintf(stderr, "Couldn't open \n");
  exit(1);
}

for(de == readdir(dir); de != NULL; de = readdir(dir)){ //dosyaları okuma
  exists = stat(de->d_name, &buf);
  if(exists < 0){
    fprintf(stderr, "%s not found\n", de->d_name);
  } else {
    printf("%s %ld\n", de->d_name, buf.st_size);
  }
}
closedir(d); //açılan dizin kapatıldı
```

`printf("%*s\n"), -20, c);` şeklinde max 20 karakter olacak şekilde c stringini hizalı bir şekilde yazdırabiliriz. *s yazılacak olan karakter sayısının belirtileceği anlamına gelir.

lstat ile sembolik linklerin de tanınması sağlanabilir. stat ile sembolik linkler tanınmaz.

S_ISDIR(buf.st_mode), S_ISLNK(buf.st_mode) gibi komutlarla tip öğrenimi yapılabilir.

umask bir sistem çağrısıdır. umask ile bir protection oluşturulabilir. Örneğin standart olarak oluşturulan bir dosyanın izinleri 0666'dır fakat biz umask değerini 0022 yaparsak bu durumda bunun değili ~umask alınır ve bu 0755 olur, ardından chmode'un yani 0666 ile and işlemine ~umask tabi tutulunca dosyanın gerçek izni ortaya çıkar. Yani umask'ı istediğimiz gibi belirledikten sonra örneğin hiçbir zaman full izinli bir dosya oluşturulmasına engel olmuş olabiliriz. → mode& ~umask

terminalde `umask 0022` şeklinde umask belirlenebilir.

`remove("f1.txt")` ile dosya silme işlemi gerçekleştirilebilir.

terminalde `mv file1 file2` komutu ile file1 dosyasının ismi file2 olarak değiştirilebilir.

yalnızca boş olan klasörleri silmek için terminalde rmdir komutu, yeni bir klasör oluşturmak için mkdir kullanılır.

Assembler;

8 tane genel kullanım için CPU'da register bulunur ve bunların tümü 4 byte'dır.

Bunlardan ilk beşi r0, r1, r2, r3, r4'dür. 6.sı sp(stack pointer), 7.si fp(frame pointer), 8.si ise pc(program counter)'dır. En önemlileri sp, fp ve pc'dir. Bunlar memory üzerinde işlem yaparken adres alanlarının kayıtlı olduğu bölgelerdir.

3 tane de hep aynı değeri taşıyan registerlar vardır.

g0 = 0, g1 = 1, gm1 = -1 'dir.

Kullanıcının direkt olarak erişemediği ise 2 adet özel register vardır.

IR → o anda çalışan komutun alanını tutan register

CSR → şu anki ve önceki IR'yi tutan register

Örneğin pc 0x2040 değerine sahipse IR 0x2040'dan başlayarak 4 bytelık değeri okur.

3 tip instruction(çağrı) vardır. İlki bellek ile register arasındaki geçişi sağlayan instructionlar; ld memory'den register'a yüklemek için kullanılır. registerler CPU'ya daha yakındır, dolayısıyla artık çalışacak olan şeyi yüklemiş oluyoruz. st ise registerdan memorye yüklemek için kullanılır.

```
ld mem -> %reg
```

```
st %reg -> mem
```

Örn; `st %r0 -> i` bu ifade r0 register'ındaki değeri global variable olan i'ye store ediyor.

`st %r0 -> [r1]` ifadesi ise r0 registerında tutulan değeri r1 register'ında tutulan adres değerine yükle demek. yani r1 pointer gibi davranıyor.

Stack bölgede yeni bir değer geldiğinde ilk konulacak yer sp register'ının gösterdiği yerdir. fp'nin gösterdiği yer en baştır, sp'nin ise en üst. Yani fp stack bölgenin adres başlangıcı, sp ise stack bölgenin sonu, yani yeni eklemede eklenmeye başlanacak yerdir.

2. tip instruction ise register değerinin register'a set edildiğidir. bu da mov ile yapılır → `mov %reg -> %reg` ile yapılır veya sabit bir değer de register'a set edilebilir.

→ `mov #val -> %reg`

Registerlar arasında aritmetik işlemler ise;

x %reg1, %reg2 -> %reg3 şeklinde yapılır. 1. ve 2. işleme tabi tutularak reg3 e yazılır.

toplama işlemi için add, çıkarma için sub(1'den 2 çıkarma), çarpma için mul, bölme için idiv(int bölme) (2 1'e bölünüyor), mod için imod(1'in 2'ye modu).

push %reg, push #val ile değer alınır, itilir. pop %reg, pop #val ile çekilir, çıkartılır.

3. tip instruction ise kontrol instructionlarıdır. `jsr a` komutu ile a instructionu veya fonksiyonu çağırılır, o fonksiyona atlanır. ret komutu ile de a'da ret 'i gördüğümüzde programa kaldığımız yerden devam ediyoruz.

Bunlardan ayrıca compare ve branch instructionları da bulunur. Örneğin if-then kullanımı gibi.

Bellekte heap alanı aşağıya doğru, stack alanı yukarıya doğru genişler. Örneğin stack alanı 0x800000001'den başlıyorsa bir sonraki değer 0x800000000'dir.

Örn şu C kodu için;

```
int i;
int j;

int main()
{
  i = 3;
  j = 2;
  j = i + j;
}
```

Assembly kodu;

```
/i ve j global değişkenlerini tanım
    .globl i
    .globl j
main:
/önce r0 register'ını 3'ü atıyoruz, ardından r0 register'ını global i değişkenine atıyoru
z.
    mov #3 -> %r0 /i=3
    st  %r0 -> i
    mov #2  -> %r0 /j=2
    st  %r0 -> j
/registerlarla toplama yapıldığı için i'yi r0'a j'yi r1'e alıyoruz
    ld  i   -> %r0
    ld  j   -> %r1
    add %r0,%r1  -> %r1
```

```
    st  %r1 -> j
    ret
```

;

```
int main()
{
  int i, j;

  i = 3;
  j = 2;
  j = i + j;
}
```

```
main:
/stackte yer ayırmak için push 2 değer olduğu için #8, daha sonra ilk değer fp-4 sonrasınd
a fp olduğu için i'yi fp-4'ün gösterdiği adrese yazıyoruz
        push #8 / This allocates i and j
        mov #3   -> %r0
        st  %r0 -> [fp-4] / Set i to 3
        mov #2  -> %r0
        st  %r0 -> [fp] / Set j to 2
        ld  [fp-4]  -> %r0
        ld  [fp]    -> %r1
        add %r0,%r1 -> %r1 / Add i and j and put the result
        st  %r1 -> [fp] / back into j
        ret
```

;

```
int a(){
  return 1;
}
int main(){
  int i;
  i = a();
}
```

```
a:
  mov #1 -> %r0
  ret
main:
  push #4
```

```
  jsr a
  st %r0 -> [fp]
  ret
```

;

```
int a(int i, int j)
{
  int k;

  i++;
  j -= 2;
  k = i * j;
  return k;
}
int main()
{
  int i, j, k;

  i = 3;
  j = 4;
  k = a(j+1, i);
  return 0;
}
```

```
a:
    push #4 / Allocate k, which will be [fp]

    ld [fp+12] -> %r0 / i++
    add %r0, %g1 -> %r0
    st %r0 -> [fp+12]

    ld [fp+16] -> %r0 / j -= 2
    mov #2 -> %r1
    sub %r0, %r1 -> %r0
    st %r0 -> [fp+16]

    ld [fp+12] -> %r0 / k = i * j
    ld [fp+16] -> %r1
    mul %r0, %r1 -> %r0
    st %r0 -> [fp]

    ld [fp] -> %r0 / return k
    ret

main:
    push #12 / Allocate i, j, k. / i is [fp-8], j is [fp-4], k is [fp]
```

```
    mov #3 -> %r0 / i = 3
    st %r0 -> [fp-8]
    mov #4 -> %r0 / j = 4
    st %r0 -> [fp-4]

    ld [fp-8] -> %r0 / Push i onto the stack
    st %r0 -> [sp]--

    ld [fp-4] -> %r0 / Push j+1 onto the stack
    add %r0, %g1 -> %r0
    st %r0 -> [sp]--

    jsr a / Call a(), then pop the arguments
    pop #8

    st %r0 -> [fp] /Put the return value into k

    mov #0 -> %r0 / Return 0
    ret
```

;

```
int a(int i, int j)
{
  int k;

  k = (i+2)*(j-5);
  return k;
}

int main()
{
  int i;

  i = a(44, 22);
}
```

```
a:
    push #4/ Allocate k
    st %r2 -> [sp]--/ Spill r2

    ld [fp+12] -> %r0
    mov #2 -> %r1
    add %r0, %r1 -> %r0/ Calculate (i+2) and put the result in r0

    ld [fp+16] -> %r1
    mov #5 -> %r2
    sub %r1, %r2 -> %r1/ Calculate (j-5) and put the result in r1
```

```
    mul %r0, %r1 -> %r0
    st %r0 -> [fp]/ Do k = r0 * r1

    ld [fp] -> %r0
    ld ++[sp] -> %r2/ Unspill r2
    ret

main:

    push #4/ Allocate i

    mov #22 -> %r0/ Push arguments onto the stack in reverse order
    st %r0 -> [sp]--
    mov #44 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8/ Always pop the arguments off the stack after jsr

    st %r0 -> [fp]
    ret
```

;

```
int a(int i, int j)
{
  int k;

  k = (i+2)*(j-5);
  return k;
}

int main()
{
  int i;

  i = (a(10, 20) + a(30, 40));
}
```

```
a:
    push #4              / Allocate k
    st %r2 -> [sp]--     / Spill r2

    ld [fp+12] -> %r0
    mov #2 -> %r1
    add %r0, %r1 -> %r0  / Calculate (i+2) and put the result in r0

    ld [fp+16] -> %r1
```

```
    mov #5 -> %r2
    sub %r1, %r2 -> %r1  / Calculate (j-5) and put the result in r1

    mul %r0, %r1 -> %r0
    st %r0 -> [fp]       / Do k = r0 * r1

    ld [fp] -> %r0
    ld ++[sp] -> %r2     / Unspill r2
    ret

main:

    push #4              / Allocate i
    st %r2 -> [sp]--     / Spill r2

    mov #20 -> %r0       / Call a(10, 20) and store the result in r2
    st %r0 -> [sp]--
    mov #10 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8
    mov %r0 -> %r2

    mov #40 -> %r0       / Call a(30, 40) and add the result to r2
    st %r0 -> [sp]--
    mov #30 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8
    add %r0, %r2 -> %r0
    st %r0 -> [fp]

    ld ++[sp] -> %r2     / Unspill r2
    ret
```

;

```
int main()
{
  int i, j, *jp;

  jp = &j;
  j = 15;
  i = *jp;
}
```

```
main:
    push #12               / Allocate the three locals
```

```
    mov #-4 -> %r0         / jp = &j.
    add %fp, %r0 -> %r0
    st %r0 -> [fp]

    mov #15 -> %r0         / j = 15
    st %r0 -> [fp-4]

    ld [fp] -> %r0         / i = *jp
    ld [r0] -> %r0
    st %r0 -> [fp-8]

    ret
```

;

```
int a(int *p)
{
  return *p;
}

int main()
{
  int i, j;

  j = 15;
  i = a(&j);
}
```

```
a:
    ld [fp+12] -> %r0      / get p's value
    ld [r0] -> %r0         / dereference it
    ret

main:
    push #8

    mov #15 -> %r0         / j = 15
    st %r0 -> [fp]

    st %fp -> [sp]--       / push &j on the stack
    jsr a                  / and call a()
    pop #4
    st %r0 -> [fp-4]

    ret
```

;

```
void a(int *p)
{
  int i;

  i = p[0];
  i = p[3];
  i = p[i];
}

int main()
{
  int array[5];

  array[0] = 10;
  array[1] = 11;
  array[2] = 12;
  array[3] = 2;
  array[4] = 15;

  a(array);
}
```

```
a:
   push #4

   ld [fp+12] -> %r0     / i = p[0]
   ld [r0] -> %r0
   st %r0 -> [fp]

   ld [fp+12] -> %r0     / i = p[3]
   mov #12 -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   st %r0 -> [fp]

   ld [fp] -> %r0        / i = p[i]
   mov #4 -> %r1
   mul %r0, %r1 -> %r0
   ld [fp+12] -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   st %r0 -> [fp]

   ret

main:
   push #20
```

```
   mov #10 -> %r0        / Store the values of array
   st %r0 -> [fp-16]
   mov #11 -> %r0
   st %r0 -> [fp-12]
   mov #12 -> %r0
   st %r0 -> [fp-8]
   mov #2 -> %r0
   st %r0 -> [fp-4]
   mov #15 -> %r0
   st %r0 -> [fp]

   mov #-16 -> %r0        / Push array onto the stack
   add %fp, %r0 -> %r0
   st %r0 -> [sp]--       / call a
   jsr a
   pop #4
   ret
```

Assembly'de if için cmp komutu kullanılır. `cmp %r0, %r1` olarak kullanılır.

Bu işlemin sonucu CSR denilen control status register'a kaydedilir.

CSR sonucuna göre beq, ble, blt, bge, bgt, bne komutları bulunur.

If için cmp sonrası csr değerinin tersi kontrol edilerek else'e gidip gitmeyeceğine bakılır.

```
int a(int i, int j)
{
  int k;

  if (i < j) {
    k = i;
  } else {
    k = j;
  }
  return k;
}

int main()
{
  return a(3, 4);
}
```

```
a:
  push #4/ Allocate k
```

```
  ld [fp+12] -> %r0/ Compare i & j
  ld [fp+16] -> %r1/ Branch on negation of less-than
  cmp %r0, %r1
  bge l1 / küçüktürün tersi büyük eşit, else'e gidecekse l1'e gider aksi halde k=i işlemin
i yaparak l2'ye yani if else sonrasına gider

  ld [fp+12] -> %r0/ k = i
  st %r0 -> [fp]
  b l2

l1:
  ld [fp+16] -> %r0/ k = j
  st %r0 -> [fp]

l2:
  ld [fp] -> %r0/ return k
  ret

main:
  mov #4 -> %r0
  st %r0 -> [sp]--
  mov #3 -> %r0
  st %r0 -> [sp]--
  jsr a
  ret
```

;

```
int a(int k)
{
  int i, j;

  j = 0;

  for (i = 1; i <= k; i++) j += i;

  return j;
}

int main()
{
  int i;

  i = a(4);
}
```

```
a:
  push #8/ Allocate i and j on the stack

  st %g0 -> [fp-4]/ Set j to zero

  st %g1 -> [fp]/ Initialize the for loop  (S1)
  b l2

l1:

  ld [fp] -> %r0/ Do i++ (S2)
  add %r0, %g1 -> %r0
  st %r0 -> [fp]

l2:
  ld [fp] -> %r0/ Perform the test, and
  ld [fp+12] -> %r1/ branch on the negation
  cmp %r0, %r1
  bgt l3

  ld [fp-4] -> %r0/ Do j += i  (S3)
  ld [fp] -> %r1
  add %r0, %r1 -> %r0
  st %r0 -> [fp-4]
  b l1

l3:
  ld [fp-4] -> %r0/ return j (S4)
  ret

main:
  push #4

  mov #4 -> %r0
  st %r0 -> [sp]--
  jsr a
  pop #4
  st %r0 -> [fp]
  ret
```