



# Sistem Programlama

---

DR. ÖĞR. ÜYESİ ABDULLAH SEVİN

# İçerik

---

a) Umask

b) Assembler

- <http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Umask-And-Others/>
- <http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Assembler1/lecture.html>

# Umask

---

- ❑ Umask, "dosya modu oluşturma maskesini" işleyen bir sistem çağrısıdır. Dosya oluşturma maskesi **dokuz bitlik** bir sayıdır.
- ❑ Umask'ta bir bit ayarlanmışsa, dosya oluşturan bir sistem çağrısı yaptığınızda, koruma modundaki o bit kapatılır.
- ❑ Normalde, bir dosyayı açtığınızda bir mod belirttiğinizde, gerçek koruma modu şöyle olacaktır:  $(\text{mod} \& \sim \text{mask})$
- ❑ "VE-DEĞİL"e alışana kadar kafa karıştırıcı olabilir. M maskeniz varsa, o zaman:
- ❑  $x = (a \& m)$ , x'i m'nin bir bitinin konumlarındaki a'daki tüm bitlere ayarlar. Başka bir deyişle, a'daki m'nin tüm sıfır bitlerini "kapatacaktır".
- ❑  $x = (a \& \sim m)$ , x'i m'nin sıfır bitlerinin konumlarındaki a'daki tüm bitlere ayarlayacaktır. Başka bir deyişle, a'daki m'nin bir bitinin hepsini "kapatacaktır".

# Umask

---

- ❑ Böylece, umask koruma bitlerini "kapatır".
- ❑ Umask'ın amacı, programların aşağıdaki koruma modlarına sahip dosyalar oluşturmaya izin vermektir:
- ❑ Normal metin ve veri dosyaları 0666 modu ile açılabilir. (f)
- ❑ Dizinler ve yürütülebilir dosyalar 0777 modu ile açılabilir. (d)
- ❑ **Kullanıcı**, koruma modlarını umask ile kendi beğenisine göre düzenleyebilir.
- ❑ read, write, execute =>      umask    User Access    Group Access    Other



# Umask

- ❑ Aşağıdaki örneklerde, sistem çağrısı yapmayacağız, sadece aynı şeyi yapan, ancak mevcut kabukta bulunan umask komutunu kullanacağız.
- ❑ Kabuğumuza umask yazarsak, o zaman bana şu anki umask'ı sekizli olarak söyleyecektir:

```
abduallah@abduallah-VirtualBox: ~/sist_prog/cs360-lecture-notes/Umask-And-Others
Dosya Düzenle Görünüm Ara Uçbirim Yardım
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ umask
0002
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ make
cc -Wall -Wextra -o bin/o1 src/o1.c
cc -Wall -Wextra -o bin/o2 src/o2.c
cc -Wall -Wextra -o bin/t1 src/t1.c
cc -Wall -Wextra -o bin/t2 src/t2.c
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ echo "Hi" > f1.txt
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ umask 0
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ echo "Hi" > f2.txt
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ umask 77
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ echo "Hi" > f3.txt
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ umask 777
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ echo "Hi" > f4.txt
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ ls -l f?.txt
-rw-rw-r-- 1 abduallah abduallah 3 May 21 21:36 f1.txt
-rw-rw-rw- 1 abduallah abduallah 3 May 21 21:36 f2.txt
-rw----- 1 abduallah abduallah 3 May 21 21:36 f3.txt
----- 1 abduallah abduallah 3 May 21 21:36 f4.txt
```

```
UNIX> umask
22
UNIX> echo "Hi" > f1.txt
UNIX> umask 0
UNIX> echo "Hi" > f2.txt
UNIX> umask 77
UNIX> echo "Hi" > f3.txt
UNIX> umask 777
UNIX> echo "Hi" > f4.txt
UNIX> ls -l f?.txt
-rw-r--r--. 1 plank loci 3 Feb 13 09:53 f1.txt
-rw-rw-rw-. 1 plank loci 3 Feb 13 09:53 f2.txt
-rw----- 1 plank loci 3 Feb 13 09:53 f3.txt
----- 1 plank loci 3 Feb 13 09:54 f4.txt
UNIX>
```

# Umask

---

- ❑ Kabuk, çıktı yeniden yönlendirmesi için bir dosya açtığı anda, 0666 modunu kullanır.
- ❑ Yukarıdan da görebileceğiniz gibi: Umask'ım 002 olduğunda, "group" ve " world " yazma bitleri kapatılır, bu nedenle dosyanın koruma modu 0664'tür.
- ❑ Umask'ım 0 olduğunda, hiçbir bit kapatılmaz ve dosyanın koruma modu 0666'dır.
- ❑ mask'ım 077 olduğunda, tüm «group» ve «world» bitleri kapatılır, bu nedenle dosyanın koruma modu 0600'dür.
- ❑ Umask'ım 0777 olduğunda, dokuz bitin tümü, dolayısıyla dosyanın koruma modu 0000'dir.
- ❑ Open() çağrısı bize yazmak için legal bir dosya tanımlayıcısı verdiği için, dosyaya "Merhaba" yazmama hala izin verildiğini fark edeceksiniz. Fakat artık başka hiçbir işlem dosyayı açamaz.

# Umask

- Aynı şeyler dizinler içinde geçerli; Umask komutunda ilk 0'ı eklememe gerek olmadığını fark edeceksiniz -- o, bağımsız değişkenini sekizli olarak yorumlar. Sistem çağrısında sekizlik belirtmelisiniz.

Octal value : Permission

0 : read, write and execute

1 : read and write

2 : read and execute

3 : read only

4 : write and execute

5 : write only

6 : execute only

7 : no permissions

```
UNIX> rm -rf f?.txt
UNIX> umask 22
UNIX> mkdir d1
UNIX> umask 0
UNIX> mkdir d2
UNIX> umask 077
UNIX> mkdir d3
UNIX> umask 0777
UNIX> mkdir d4
UNIX> ls -l | grep 'd.$'
drwxr-xr-x. 2 plank loci      6 Feb 13 09:59 d1
drwxrwxrwx. 2 plank loci      6 Feb 13 09:59 d2
drwx-----. 2 plank loci      6 Feb 13 09:59 d3
d------. 2 plank loci      6 Feb 13 10:00 d4
UNIX> rm -rf d?
UNIX> umask 22
UNIX>
```

# chmod

---

## ❑ Rastgele Dosya/Inode Sistem çağrıları;

❑ `chmod(char *path, mode_t mode)` -- Kabuktan çalıştırıldığında tıpkı `chmod` gibi çalışır. Örneğin. `chmod("f1", 0600)`, `f1` dosyasının korumasını sizin için `"rw-"` ve diğer herkes için `"---"` olarak ayarlayacaktır.

❑ `chmod()` -- "`man -s 2 chmod`" man sayfası, moddan tek tek bitlere erişmek için yararlı olan `<sys/stat.h>`'den bir grup `#define` gösterir.

❑ `src/o1.c`'yi derleyin ve çalıştırın:



# chmod

---

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

int main()
{
    int fd;

    printf("Opening the file:\n");
    fd = open("f1.txt", O_WRONLY | O_CREAT | O_TRUNC);
    sleep(1);

    printf("Doing chmod\n");
    chmod("f1.txt", 0000);
    sleep(1);

    printf("Doing write\n");
    write(fd, "Hi\n", 3);

    return 0;
}
```

UNIX> **bin/o1**

Opening the file:

Doing chmod

Doing write

UNIX> **ls -l f1.txt**

What will this show as the protection mode and the size of the file?

UNIX> **cat f1.txt**

What will this do?

UNIX>

# chmod

---

- ❑ Dosya tanımlayıcı, yazmak için geçerli bir dosya tanımlayıcıdır.
- ❑ chmod() komutu açık dosyaya hiçbir şey yapmadı, bu nedenle proses dosyaya başarılı bir şekilde yazabilir. Bu nedenle dosyanın boyutu sıfır değil, üçtür.
- ❑ Koruma modu elbette chmod komutuyla değiştirildi.
- ❑ Koruma modu "-----" olduğundan, cat programı dosyayı açmaya çalıştığında bir hata aldı (büyük ihtimalle open()'ı çağıran fopen() ile).

```
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ bin/o1
Opening the file:
Doing chmod
Doing write
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ ls -l f1.txt
----- 1 abduallah abduallah 3 May 21 22:08 f1.txt
abduallah@abduallah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ cat f1.txt
cat: f1.txt: Erişim engellendi
```

# chmod

---

- ❑ Yeniden o1 programını çalıştırırsak;
- ❑ f1.txt dosyasının değiştirilme zamanının değişmediğini fark edeceksiniz. Bunun nedeni, open() çağrısının başarısız olması ve -1 döndürmesidir.
- ❑ Dosya herhangi bir şekilde kesilmemiş veya değiştirilmemiştir. Bu nedenle değişiklik süresi değişmez.
- ❑ chmod() komutu başarılı oldu, ancak write() sistem çağrısına -1 dosya tanıtıcısı verildi, dolayısıyla o da başarısız oldu.
- ❑ Dosyayı silebiliriz;

UNIX> **rm -f f1.txt**

# link

---

- ❑ link, unlink, remove, rename: Bunlar basittir: link(char \*f1, char \*f2) aynen şu şekilde çalışır:
- ❑ UNIX> ln f1 f2
- ❑ f2 bir dosya olmalıdır - bir dizin olamaz.
- ❑ unlink(char \*f1) şu şekilde çalışır:
- ❑ UNIX> rm f1
- ❑ remove(char \*f1), unlink() gibi çalışır, ancak (boş) dizinler için de çalışır.
- ❑ Unlink() dizinlerde başarısız olur.

# Open/remove/read

---

src/o2.c'ye bir göz atın:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

int main()
{
    int fd;
    char s[11];
    int i;

    printf("Opening f1.txt and putting \"Fun Fun\" into s.\n");
    strcpy(s, "Fun Fun\n");
    fd = open("f1.txt", O_RDONLY);
    sleep(1);

    printf("Removing f1.txt\n");
    remove("f1.txt");
    sleep(1);

    printf("Listing f1.txt, and reading 10 bytes from the open file descriptor.\n");
    system("ls -l f1.txt");
    i = read(fd, s, 10);
    s[i] = '\0';
    printf("Read returned %d: %d %s\n", i, fd, s);
    return 0;
}
```

# Open/remove/read

---

- ❑ Bu program f1.txt dosyasını okumak için açar, bir saniye uyur ve ardından f1.txt dosyasını kaldırır. Tekrar uyur, uzun bir listeleme yapar ve ardından açık dosyadan 10 bayt okumaya çalışır.
- ❑ Soru şu: f1.txt dosyasını kaldırdığımızda ne olur? Okuma çağrısı başarılı mı yoksa dosya gittiği için başarısız mı olacak?

```
UNIX> rm -f f1.txt
UNIX> echo "Jim Plank" > f1.txt
UNIX> bin/o2
Opening f1.txt and putting "Fun Fun" into s.
Removing f1.txt
Listing f1.txt, and reading 10 bytes from the open file descriptor.
ls: cannot access f1.txt: No such file or directory
Read returned 10: 3 Jim Plank
```

- ❑ ls komutu, remove() çağrısından sonra f1.txt dosyasının gerçekten gittiğini gösterir. Ancak, işletim sistemi, dosyanın son dosya tanıtıcısı kapatılana kadar dosyayı silmez. Bu nedenle read() çağrısı başarılı olur.



# Open/remove/read

---

❑ bin/o2'yi tekrar deneyin -- f1.txt kaldırıldığı için şu anda mevcut değil:

```
UNIX> bin/o2
Opening f1.txt and putting "Fun Fun" into s.
Removing f1.txt
Listing f1.txt, and reading 10 bytes from the open file descriptor.
ls: cannot access f1.txt: No such file or directory
Read returned -1: -1 Fun Fun
```

- ❑ Ne oldu? İlk olarak, open() çağrısı başarısız oldu ve -1 döndürdü. Böylece, read() çağrısı da başarısız oldu ve -1 döndürdü.
- ❑ Okuma çağrısı başarısız olduğundan, s baytlarının üzerine hiçbir zaman yazılmadı - dolayısıyla bunları yazdığımızda, "Fun Fun" aldık.

# Rename, time

---

- ❑ **rename**(char \*f1, char \*f2) şu şekilde çalışır: UNIX> mv f1 f2
- ❑ **utime**: Bu sistem çağrısı, bir dosyanın inode'unun zaman alanlarını değiştirmenizi sağlar.
- ❑ Yasa dışı olmalı gibi görünüyor (örneğin, ödevini zamanında bitirmiş gibi görünmek için bir program yazılabilir...), Her zaman olduğu gibi, kılavuz sayfasını okuyun.
- ❑ Zaman değerleriyle çalışırken, birkaç veri yapısının farkında olmanız gerekir:
- ❑ **time\_t**: Bu, zamanın başlamasından bu yana geçen saniye sayısını içeren bir long değişkeni (1 Ocak 1970). time() sistem çağrısı, makinenizdeki geçerli saati bir time\_t olarak döndürür.
- ❑ **struct timeval**. Bu, aşağıdaki tanıma sahiptir.
- ❑ Gettimeofday() ögesini çağırarak şimdiki zamanı alabilirsiniz.

```
struct timeval {  
    long tv_sec;           /* seconds */  
    long tv_usec;         /* microseconds */  
};
```

# time

---

- ❑ **struct tm.** Bu, aşağıdaki tanıma sahiptir.
- ❑ `time_t` veri türleri, `struct tm` veri türleri ve diziler arasında dönüşüm yapan prosedürlerin bir listesini görmek için "man ctime" yapın.
- ❑ Yararlı olanlar `ctime()`, `localtime()`, `mktime()`, `asctime()` ve `strftime()`'dir.

```
struct tm {  
    int tm_sec;           /* seconds */  
    int tm_min;           /* minutes */  
    int tm_hour;          /* hours */  
    int tm_mday;          /* day of the month */  
    int tm_mon;           /* month */  
    int tm_year;          /* year */  
    int tm_wday;          /* day of the week */  
    int tm_yday;          /* day in the year */  
    int tm_isdst;         /* daylight saving time */  
};
```

# time

---

## ❑ `time()`, `localtime()`, `asctime()` ve `mktime()`

```
#include <time.h>

time_t time(time_t *tloc);           /* Returns or fills in the current time as a time_t */
struct tm *localtime(const time_t *clock); /* Turns a time_t into a (struct tm *) */
char *asctime(const struct tm *timeptr); /* Returns a printable string from a (struct tm *) */
time_t mktime(const struct tm *timeptr); /* Turns a (struct tm *) into a time_t */
```

- ❑ `time()` ile, ona bir NULL veya 0 argümanı vererseniz, argümanı yok sayar. Bir şey sizin için bir işaretçi döndürdüğünde her zaman dikkatli olmalısınız.
- ❑ Kılavuz sayfası, `malloc()` ile oluşturulduğunu belirtmedikçe, `malloc()` ile oluşturulmadığını varsaymalısınız.

# time

- ❑ Aşağıda localtime() ile gösterelim.
- ❑ Program src/t1.c'dir.

```
/* This program shows time(), localtime() and asctime(). */

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

int main()
{
    time_t now;
    struct tm *tm1, *tm2;

    /* Set now to the current time,
       use localtime() to convert it to a (struct tm *)
       and print them both out. */

    now = time(0);

    tm1 = localtime(&now);
    printf("Seconds: %ld.  Asctime(tm1): %s\n", now, asctime(tm1));

    /* Add an hour to "now" and do the same, using
       tm2 for the (struct tm *) */

    now += 3600;
    tm2 = localtime(&now);
    printf("Seconds: %ld.  Asctime(tm2): %s\n", now, asctime(tm2));

    /* Print out tm1.  Is that what you expect? */

    printf("Asctime(tm1): %s\n", asctime(tm1));

    /* Print the pointers.*/

    printf("0x%lx 0x%lx\n", (long unsigned int) tm1, (long unsigned int) tm2);
    return 0;
}
```

# time

---

```
UNIX> bin/t1
Seconds: 1645637115.  Asctime(tm1): Wed Feb 23 12:25:15 2022  # You'll note asctime() includes a newline.

Seconds: 1645640715.  Asctime(tm2): Wed Feb 23 13:25:15 2022  # We've added an hour to the time_t, and it's reflected in tm2.

Asctime(tm1): Wed Feb 23 13:25:15 2022                        # You'll note, tm1 has "changed" too.

0x7fb062c017a0 0x7fb062c017a0                                  # Why?  Because localtime() always returns the same pointer.
UNIX>
```

- ❑ Zamanı manipüle ediyorsanız, a'ya (struct tm \*) dönüştürmek ve onu manipüle etmek genellikle daha iyidir.
- ❑ src/t2.c programı, 9 yıl boyunca şimdiki zamana art arda bir yıl ekleyerek gösterir:



# time

---

```
/* This program shows how you can use the (struct tm *) to manipulate time. */

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

int main()
{
    time_t years[10];          /* Now, and every year up to 9 years from now */
    struct tm *tm1;
    int i;

    /* Set years[0] to now, convert to tm1 and print. */

    years[0] = time(0);
    tm1 = localtime(&years[0]);
    printf("Base time: %ld. %14s  %s", years[0], "", asctime(tm1));

    /* Now use the (struct tm *) to increment the years, and print.
       When you run this, you'll see that leap years are handled correctly. */

    for (i = 1; i < 10; i++) {
        tm1->tm_year += 1;
        years[i] = mktime(tm1);
        printf("Year +%d:   %ld. Diff: %ld. %s", i, years[i], years[i]-years[i-1], asctime(tm1));
    }

    return 0;
}
```

# time

---

- ❑ Çalıştırdığımızda, artık yılları doğru bir şekilde ele aldığını göreceksiniz -- time\_t'leri artık olmayan yıllardan farklı:

```
UNIX> bin/t2
```

```
Base time: 1645637432.           Wed Feb 23 12:30:32 2022
Year +1:   1677173432. Diff: 31536000. Thu Feb 23 12:30:32 2023
Year +2:   1708709432. Diff: 31536000. Fri Feb 23 12:30:32 2024
Year +3:   1740331832. Diff: 31622400. Sun Feb 23 12:30:32 2025
Year +4:   1771867832. Diff: 31536000. Mon Feb 23 12:30:32 2026
Year +5:   1803403832. Diff: 31536000. Tue Feb 23 12:30:32 2027
Year +6:   1834939832. Diff: 31536000. Wed Feb 23 12:30:32 2028
Year +7:   1866562232. Diff: 31622400. Fri Feb 23 12:30:32 2029
Year +8:   1898098232. Diff: 31536000. Sat Feb 23 12:30:32 2030
Year +9:   1929634232. Diff: 31536000. Sun Feb 23 12:30:32 2031
```

```
# Leap year handled correctly
```

```
# Leap year handled correctly
```

# Soru 1)

---

1-) Dizindeki dosyaları ve alt dizinleri aç (test1) ve chmod 'ları readonly yapın?

# Assembler

---

- ❑ Assembly kodu, 4 baytlık işaretçileri olan ve kayan nokta içermeyen hayali bir makinede çalışan bir RISC assembly kodudur. Eylem halinde görebilmeniz için bunun için görsel bir assembler yazılmıştır.
- ❑ **Kayıtçılar-Registers:**
- ❑ Çoğu makineden biraz farklı olan ama neredeyse tüm tek işlemcili işlemcilere örnek teşkil eden genel bir bilgisayar mimarisini varsayacağız.
- ❑ Makinemizin CPU'da 8 genel amaçlı register olduğunu varsayacağız. Hepsi 4 bayttır ve kullanıcı tarafından okunabilir veya yazılabilir.
- ❑ İlk beşi **r0**, **r1**, **r2**, **r3**, **r4** olarak adlandırılır. Son üç kayıt özeldir:
  - ❑ Altıncı, **sp** olarak adlandırılır ve "yığın işaretçisi" olarak adlandırılır.
  - ❑ Yedinci, **fp** olarak adlandırılır ve "çerçeve işaretçisi" olarak adlandırılır.
  - ❑ Sekizincisi **pc** olarak adlandırılır ve "program sayacı" olarak adlandırılır.

# register

---

- ❑ Ek olarak, bilgisayarda her zaman aynı değerleri içeren üç salt okunur kayıt vardır:
- ❑ değeri her zaman sıfır olan **g0**.
- ❑ değeri her zaman bir olan **g1**.
- ❑ değeri her zaman negatif olan **gm1**.
- ❑ Son olarak, bilgisayarın ayrıca kullanıcının doğrudan erişemeyeceği iki özel kaydı vardır:
- ❑ **IR** -- Komut kayıtçısı (**Instruction Registers**). Şu anda yürütülmekte olan komutu tutar.
- ❑ **CSR** -- Kontrol durumu kayıtçısı (**Control Status Register**). Mevcut ve önceki komutların yürütülmesine ilişkin bilgileri içerir.

# Komut döngüsü

---

- ❑ Bilgisayarın çalışması, komutları ardışıl çalıştırılmasından oluşur. Bu komut döngüsü olarak bilinir.
- ❑ Komut döngüsü 4 genel aşamadan oluşur:
  1. Kod çözme komut (IR'de)
  2. Komutu yürütün
  3. Bir sonraki komutu belirleyin ve bilgisayarı buna göre güncelleyin
  4. Bir sonraki komutu IR'ye yükleyin.
- ❑ Komut nedir? Diğer her şey gibi, 0'lar ve 1'ler dizisidir. Tüm komutlarımızın 32 bit olduğunu varsayacağız.



# Komutlar

---

- ❑ Komutlar, bir programın belleğinin bir parçası olarak saklanır ve pc kayıtçısı tarafından işaret edilen komut, yürütme için IR'ye yüklenen komuttur.
- ❑ Başka bir deyişle, bilgisayar 0x2040 değerini içeriyorsa, IR 0x2040 bellek adresinden başlayarak 4 baytın içerdiği komutu yürütür.
- ❑ Assembly kodu, komutların okunabilir bir kodlamasıdır. Derleyici adı verilen bir program, assembly kodunu programı oluşturan uygun 0'lara ve 1'lere dönüştürür.
- ❑ -S bayrağıyla gcc'yi çağırırsanız, C programı için assembler'ı içeren bir .s dosyası üretecektir.
- ❑ -S bayrağı olmadan direkt olarak komutları üretir.
- ❑ Örn: gcc -S p1.c

# Memory <-> Register komutlar

---

- ❑ `ld mem -> %reg` Kayıtçı değerini bellekten yükleyin.
- ❑ `st %reg -> mem` Kayıtçı değerini belleğe kaydedin.
- ❑ Belleği adreslemenin birkaç yolu vardır:
- ❑ `st %r0 -> i` : `r0` kayıtçısının değerini `i` global değişkeninin global hafıza konumuna kaydedin.
- ❑ `st %r0 -> [r1]` : `r1` kayıtçısındaki değeri bir bellek konumuna işaretçi olarak kabul edin ve `r0`'ın değerini bu bellek konumunda saklayın.
- ❑ `st %r0 -> [fp+4]` : Çerçeve işaretçisinin değerini bir bellek konumuna işaretçi olarak kabul edin ve `r0`'ın değerini bu konumdan 4 bayt sonra bellek konumunda saklayın.
- ❑ `st %r0 -> [sp]--` veya `st %r0 -> ++[sp]` : `sp`'nin değerini bir hafıza konumuna bir işaretçi olarak kabul edin, `r0`'ın değerini bu hafıza konumuna kaydedin ve `--` veya `++`.

# Register <-> Register komutlar

---

- ❑ `mov %reg -> %reg` Bir kayıtçının değerini diğerine kopyala
- ❑ `mov #val -> %reg` kayıt edin veya değerini bir sabit olarak ayarlayın.
- ❑ **Tüm aritmetik işlemler kayıtçıdan kayıtçıya kayda gider:**
  - ❑ `add %reg1, %reg2 -> %reg3:` reg1 & reg2'yi ekleyin ve toplamı reg3'e koyun.
  - ❑ `sub %reg1, %reg2 -> %reg3` Reg2'yi reg1'den çıkarın.
  - ❑ `mul %reg1, %reg2 -> %reg3` reg1 & reg2'yi çarpın.
  - ❑ `idiv %reg1, %reg2 -> %reg3` reg2'nin tamsayı bölümünü reg1'e yapın.
  - ❑ `imod %reg1, %reg2 -> %reg3` reg1 modunu reg2 yapın.

# Register <-> Register komutlar

---

- ❑ Yığın işaretçisinde toplama ve çıkarma yapmanızı sağlayan iki özel talimat vardır:
- ❑ `push %reg` veya `push #val`      Bu, `%reg` veya `#val` değerini `sp`'den çıkarır
- ❑ `pop %reg` veya `pop #val`      Bu, `%reg` veya `#val` değerini `sp`'ye ekler

# Kontrol komutları

---

- ❑ **jsr a** Komut a'dan başlayan alt programı çağırın.
- ❑ **ret** Bir alt programdan dönüş.
- ❑ Ayrıca for ve if deyimlerini uyguladığınız "karşılaştıır-compare" ve "dal-branch" komutları da vardır, ancak bunların üzerinden henüz geçmeyeceğiz.
- ❑ Son olarak, gerçekten kod olmayan, ancak değişkenler için belleğin ayrılması gerektiğini belirten "yönergeler" de vardır.
- ❑ Bu derleyicide, bu tür direktiflerden sadece biri:
- ❑ `.global i` Globals segmentinde `i` değişkeni için 4 bayt ayırın

# Kontrol komutları

---

- ❑ Program sayacı, komut kayıtçısının değerini yüklemek için nereye gitmesi gerektiğini gösterir.
- ❑ Normal komutlarda bir sonraki komutun yüklenebilmesi için bilgisayar 4 artırılır.
- ❑ Kontrol komutlarında, bilgisayar yeni bir değer alır ve makinenin alt programları çağırmasına, «if-then» ifadeleri yapmasına vb. izin verir.



# Adres Alanı

---

- ❑ Her programın belleğine ilişkin görünümüne "adres alanı" denir.
- ❑ Tipik olarak bir adres alanı 4 bölüme ayrılır: Kod, globaller, yığıt-heap ve yığın-stack.
- ❑ Kod, komutlardan başka bir şey içermez.
- ❑ Globaller, global değişkenlerin depolandığı yerdir
- ❑ Yığıt-heap, malloc ile depolamanın yapıldığı alanı ifade eder.
- ❑ Yığın-stack, prosedürler için yerel değişkenler ve bağımsız değişkenler gibi geçici depolama içindir.

# Adres Alanı

---

- ❑ Genel olarak, bir işlem belleği büyük bir bayt dizisi gibi ele alır; ancak baytlar, kayıtların boyutu olduğu için mantıksal olarak her biri 4 baytlık birimler halinde düzenlenir.
- ❑ Bu belleğin 0x80000000 boyutunda olduğunu varsayıyoruz. Kod, 0x1000 adresinde başlar. Globaller, 4096'nın (0x1000) katı olan ilk adresten başlayarak kodu takip eder.
- ❑ Gerçek bir makinede yığın, kodu takip eder veya 4096'nın katı olan başka bir adreste başlar. (Jassem ile yığın yoktur.)
- ❑ Bir program yürütülürken yığın-stack büyüyecek ve küçülecektir, ancak kod ve globaller aynı boyutta kalacaktır.
- ❑ Yığın, 0x80000000 adresinden başlayarak ve daha düşük bellek adreslerine doğru büyüyerek arkadan öne doğru büyür (stack için). Globaller ve yığın arasında kullanılmayan bellek var:

# Adres Alanı

The programs' address space:



# Kod

---

- ❑ C derleyicisi, C kodunu alır ve onu komutlara çevirir. Çeviri tarafından üretilen assembly kodu, makine komutlardan ve direktiflerinden oluşur. Örneğin, aşağıdaki kod:

```
int i;  
int j;  
  
int main()  
{  
    i = 3;  
    j = 2;  
    j = i + j;  
} /* I a
```

```
.globl i                                / Allocate  
.globl j  
main:  
    mov #3 -> %r0                        / i = 3  
    st %r0 -> i  
    mov #2 -> %r0                        / j = 2  
    st %r0 -> j  
    ld i -> %r0                          / j = i + j  
    ld j -> %r1  
    add %r0,%r1 -> %r1  
    st %r1 -> j  
    ret
```

# Kod

---

- ❑ C'deki her komutun assembler'da karşılık gelen bir dizi komutu vardır.
- ❑ Derleyiciniz akıllı değilse verimsiz kod üretebilir. Örneğin, şunu görebilirsiniz:
- ❑ Aynı şekilde çalışır ve daha az yönergeye sahiptir.
- ❑ -O bayrağıyla gcc'yi çağırırsanız, kodunuzu daha az yönerge olacak şekilde optimize etmeye çalışır.
- ❑ Bununla birlikte, normalde, gcc basitçe, optimize edilmemiş kod üretir.
- ❑ Bu derst optimize edilmemiş kod üreteceğiz, bu da her C ifadesinin bağımsız olarak derleme koduna çevrildiği anlamına gelir.
- ❑ Derleyici optimizasyonunu başka bir konu şimdilik.

```
.globl i
.globl j
main:
    mov #3  -> %r0
    mov #2  -> %r1
    add %r0,%r1  -> %r1
    st  %r1 -> j
    st  %r0 -> i
    ret
```

# Kod

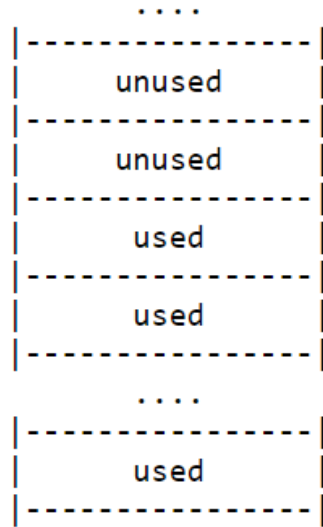
---

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

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

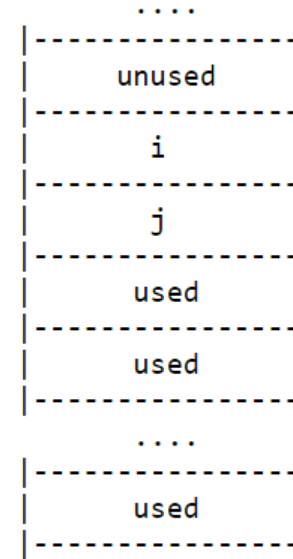
- ❑ Şimdi, çalıştırmak için yandaki koda sahip olduğumuzu varsayalım:
- ❑ i ve j yerel değişkenler olduğundan, geçici depolamadan gelmeleri gerekir: Yığın-stack.
- ❑ Yığın nasıl çalışır? sp ve fp kayıtları tarafından yönetilir.
- ❑ sp ve fp, yığında "çerçeve" olarak bilinen şeyi belirtir. fp, çerçevenin alt tarafını, sp ise üst tarafını gösterir.
- ❑ Yığın işaretçisinin üzerindeki (küçük veya eşit) tüm bellek konumları kullanılmamış kabul edilir. Böylece, sp'yi azaltarak yeni geçici bellek elde edebiliriz, böylece bellek konumlarını mevcut yığın çerçevesine yerleştirebiliriz.

# Kod



<----- sp, fp

<----- Location 0x80000000



<----- sp

<----- fp

<----- Location 0x80000000

❑ Önce

Sonra

❑ İki yerel değişken i ve j'ye yer ayırmak için yığın işaretçisini 8 azaltırız. Bu, geçerli yığın çerçevesinde iki adet 4 baytlık miktar ayırır.

# Kod

---

- ❑ Şimdi, main() kodu önceki gibidir, yalnızca i ve j'ye global değişkenler olarak erişmek yerine, onlara çerçeve işaretçisinin ofsetleri ile erişiriz.

```
main:
    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
```

- ❑ Kodu adım adım çalıştırdığımızda bellek nasıl değişir kaynağa bakalım!



# Jassem -- The Visual Assembler

---

- ❑ Assembler anlamana yardımcı olmak için, Assembly kod programlarını yüklemenizi ve bunlarda adım adım ilerlemenizi sağlayan basit bir görsel Assembler.
- ❑ Grafik betik dili tcl/tk ile yazılmıştır. Bu dosyayı jassem.tcl içindeki bu dizinden alabilirsiniz.
- ❑ tcl/tk ile ilgili güzel şey, Unix, Windows ve Macintosh'ta çalışıyor olmasıdır.
- ❑ jassem.tcl'yi makinelerimizde kullanmak için şunu çalıştırın:
- ❑ UNIX> wish ~/jplank/cs360/bin/jassem.tcl [filename]
- ❑ Wish tüm makinelerimizde kurulu olmalıdır.

# Jassem

---

- ❑ `jassem.tcl`'yi bir Windows veya Macintosh makinesinde kullanmak için `tcl/tk`'yi kurmanız gerekir.
- ❑ Bu ücretsizdir -- [www.scriptics.com](http://www.scriptics.com) adresinden kodu alın.
- ❑ `jassem.tcl`'yi çalıştırırken yaptığınız ilk şey, `p1-g.jas` (yukarıdaki global değişkenleri ekleyen program) veya `p1.jas` (yukarıdaki yerel değişkenleri ekleyen program) gibi bir program yüklemektir.
- ❑ Sistemin bir resmini görmelisiniz - yığın, kayıtlar, globaller ve kod, aşağıdaki gibi:
- ❑ Jassem'de kod `0x1000`'de başlar ve eğer varsa global değişkenler `0x1000`'in bir sonraki katında başlar.
- ❑ Yığın-heap yok. Yığın-stack `0xffff44c`'de biter.

# Jassem

