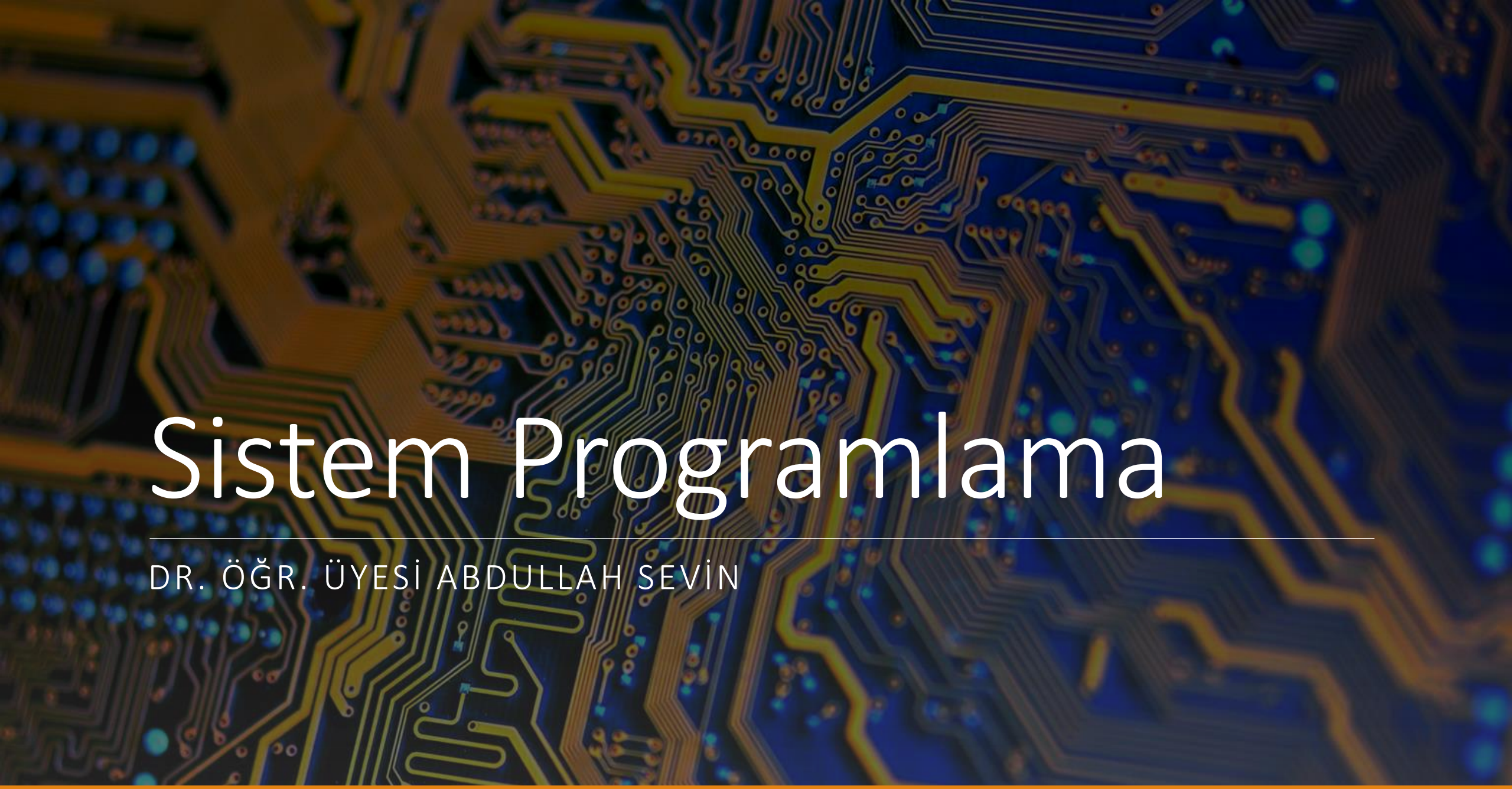


Sistem Programlama

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, bir dosya oluşturulurken izinleri sınırlandıran 9 bitlik bir sayıdır.
- Sistem çağrısı ile bir dosya oluşturulduğunda, umask'te "1" olan bitler izinlerden çıkarılır (kapatılır).
- Dosya oluştururken belirtilen mode ile umask şöyle hesaplanır:

```
gerçek_mode = mode & ~umask
```

- x = (a & m) → m'de 1 olan bitler a'dan alınır.
- x = (a & ~m) → m'de 0 olan bitler a'dan alınır (yani m'deki 1'ler kapatılır).

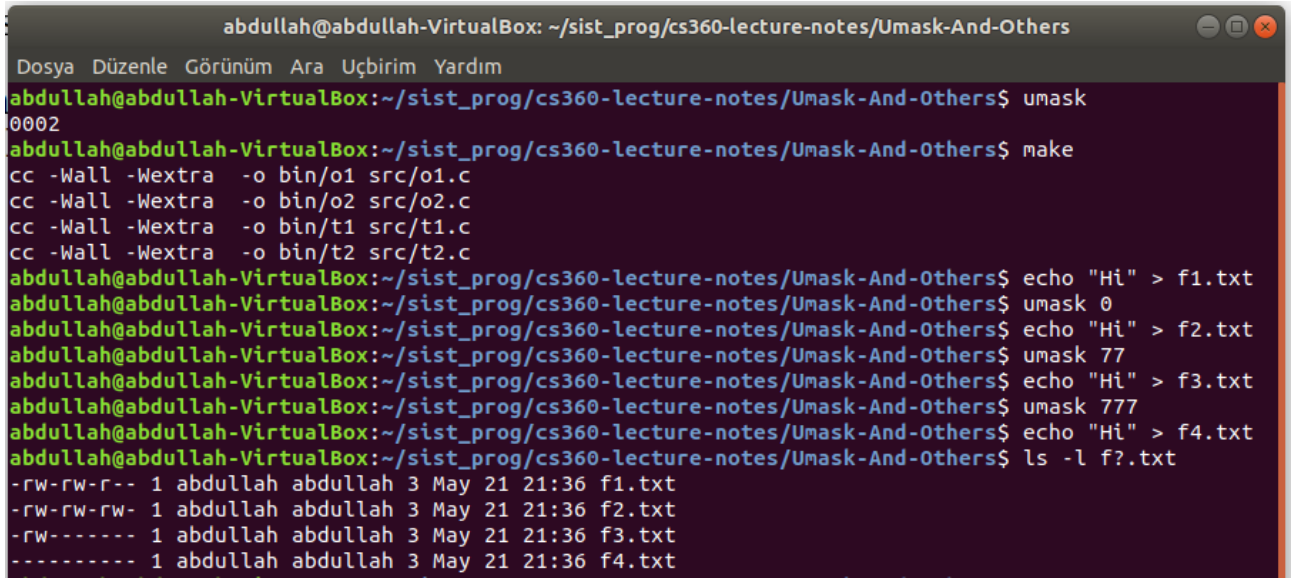
Umask

- mask, dosya oluşturulurken aşırı yetki verilmesini engeller.
- Varsayılan olarak:
  - Düz dosyalar (text/data) genelde 0666 ile oluşturulur.
  - Dizinler ve çalıştırılabilir dosyalar 0777 ile oluşturulabilir.
- Umask: 0022 touch dosya.txt komutu çalıştırıldığında:
  - Varsayılan 0666
  - Gerçek izin: 0666 & ~0022 = 0644
- **Kullanıcı**, koruma modlarını umask ile kendi istezine göre düzenleyebilir.
- read, write, execute =>

umask User Access Group Access Other

Umask

- Aşağıdaki örneklerde, sistem çağrısı yapmayacağız, sadece aynı şeyi yapan, ancak mevcut terminalden **umask** komutunu kullanacağız.
- Kabuğumuza **umask** yazarsak, o zaman bana şu anki **umask**'ı (sekizlik tabanda) söyleyecektir:




```
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 açtığında, 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 verdiğinden, 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

```c
#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).

```
abdullah@abdullah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ bin/o1
Opening the file:
Doing chmod
Doing write
abdullah@abdullah-VirtualBox:~/sist_prog/cs360-lecture-notes/Umask-And-Others$ ls -l f1.txt
---------- 1 abdullah abdullah 3 May 21 22:08 f1.txt
abdullah@abdullah-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 sıfırlanmamış(truncate) veya değiştirilmemiştir. Bu nedenle değiştirme süresi değişmez.
- chmod() komutu başarılı oldu, ancak write() sistem çağrısına -1 dosya tanıtıcısı döndürülür, 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:

```c
#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, bir listeleme yapar ve ardından açık dosyadan 10 bayt okumaya çalışır.
- Soru şu: f1.txt dosyasını kaldırdığımızda ne olur?

```
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 yazdırdığımızda, "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() öğesini 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 fonksiyonların 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 verirseniz, 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.

```c
/* 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 değiştiriyor iseniz, a'ya (struct tm *) dönüştürmek ve onu değiştirmek genellikle daha iyidir.
- src/t2.c programı, 9 yıl boyunca şimdiki zamana art arda bir yıl ekleyerek gösterir:

time

```c
/* 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     # Leap year handled correctly
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     # Leap year handled correctly
Year +8:   1898098232. Diff: 31536000. Sat Feb 23 12:30:32 2030
Year +9:   1929634232. Diff: 31536000. Sun Feb 23 12:30:32 2031
```

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. Yapılan işlemleri görebilmeniz için görsel bir assembler yazılmıştır.
- **Kayıtçılar-Registers:**
- Çoğu makineden 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 kayıtçı 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 kayıtçı vardır:
- değeri her zaman sıfır olan **g0.**
- değeri her zaman bir olan **g1.**
- değeri her zaman -1 olan **gm1.**
- Son olarak, kullanıcının doğrudan erişemeyeceği iki özel kayıtçı 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, komutların ardışıl çalıştırılmasından oluşur. Bu komut döngüsü olarak bilinir.
- Komut döngüsü 4 genel aşamadan oluşur:

1. Kodu çöz (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 program belleğinde saklanır ve **pc** kayıtçısı tarafından işaret edilen komut, yürütme için **IR**'ye yüklenir.
- Başka bir deyişle, **pc** (bir sonraki işlenecek komut) 0x2040 değerini içeriyorsa, **IR** 0x2040 bellek adresinden başlayarak 4 baytın içerdiği komutu yürütür.
- Assembly kodu, komutların okunabilir bir kodlamasıdır. **Assembler** adı verilen bir program, assembly kodunu programı oluşturan uygun 0'lara ve 1'lere dönüştürür.
- -S bayrağıyla gcc'yi çağırırsanız, C programı için assembler kodu içeren bir .s dosyası üretecektir.
- -S bayrağı olmazsa, komutlar direkt üretilir.
- Örn: gcc –S p1.c

Memory <-> Register komutlar

- ld mem -> %reg Kayıtçı değerini bellekten yükleyin. (Load)
- st %reg -> mem Kayıtçı değerini belleğe kaydedin. (Store)
- 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 (yani adres) 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 sabit bir değeri kayıtçıya ata
- **Tüm aritmetik işlemler kayıtçıdan kayıtçıya 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 r 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 ekleme ve çıkarma yapmanızı sağlayan iki özel komut vardır:
- push %reg veya push #val — Bu, %reg veya #val değerini sp'den çıkarır
- pop %reg vey 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 "karşılaştır-compare" ve "dal-branch" komutları da vardır, ancak bunların üzerinden henüz geçmeyeceğiz.
- Son olarak, gerçekte kod olmayan, ancak değişkenler için **belleğin ayrılması gerektiğini** belirten "**yönergeler**" de vardır.
- Bu derleyicide, bu tür direktiflerden sadece biri:
- **.globl i** Globals segmentinde i değişkeni için 4 bayt ayırın

Kontrol komutları

- **Program sayacı (pc)**, 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.

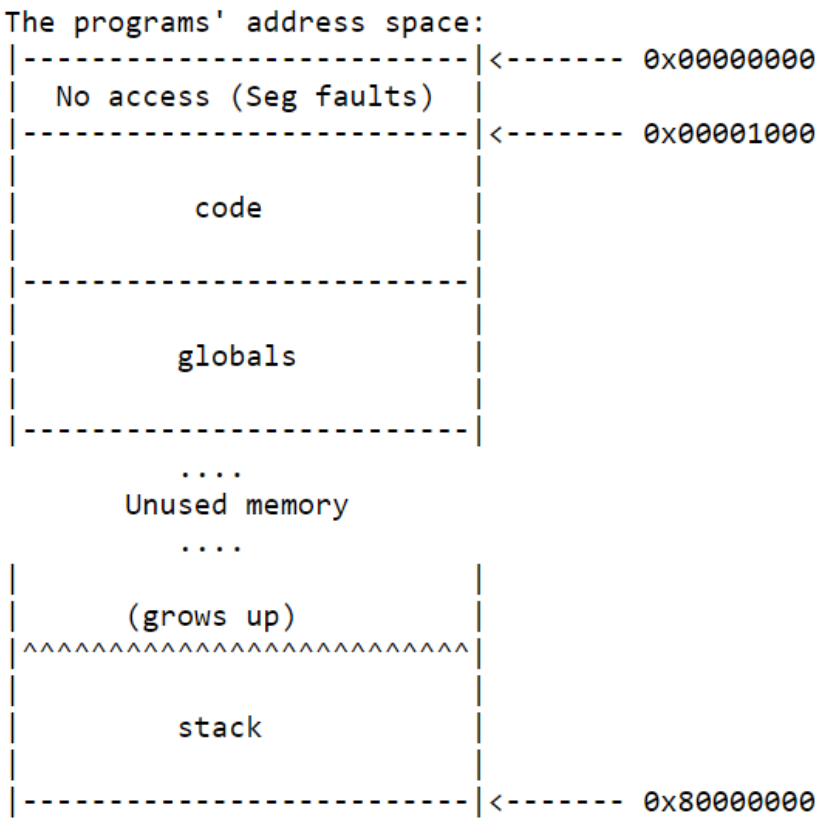
Adres Alanı

- Her programın belleğine ilişkin görünümüne "**adres alanı**" denir.
- Tipik olarak bir adres alanı 4 bölüme ayrılır: **Kod, globaller, yığıt-heap ve yığın-stack**.
- **Kod kısmı**, komutlardan başka bir şey içermez.
- **Globaller**, global değişkenlerin depolandığı yerdir
- **Yığıt-heap**, malloc ile depolamanın yapıldığı alanı ifade eder.
- **Yığın-stack**, fonksiyonlar 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** de 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:
No access (Seg faults)
0x00000000
0x00001000
code
globals
....
Unused memory
....
(grows up)
stack
0x80000000

Kod

- C derleyicisi, C kodunu alır ve onu komutlara çevirir. Assembler 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 derste optimize edilmemiş kod üreteceğiz, bu da her *C ifadesinin bağımsız olarak derleme koduna çevrildiği anlamına gelir*.
- Derleyici optimizasyonunu başka bir konu şimdilik.

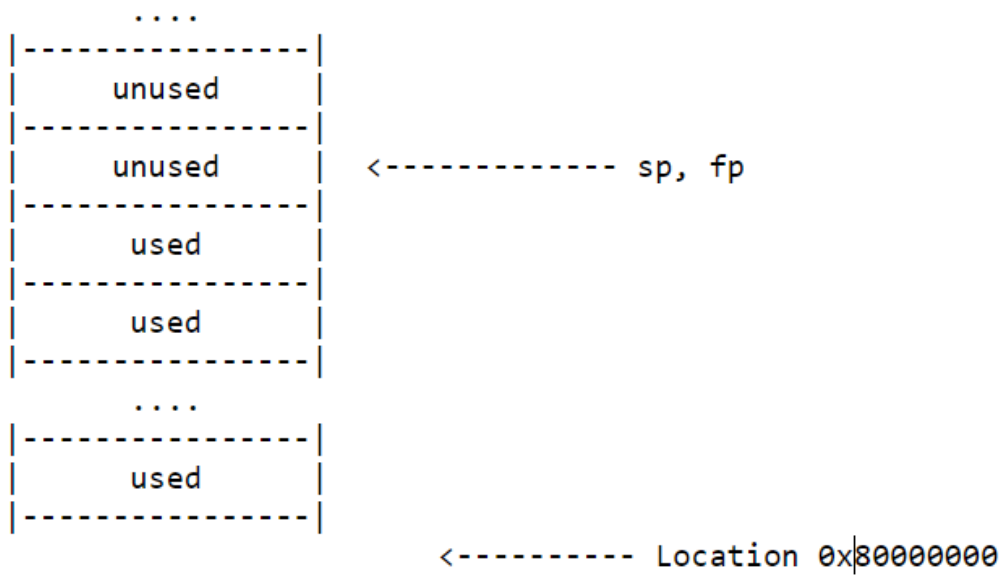
```
    .globl i
    .globl j
main:
    mov #3   -> %r0
    mov #2   -> %r1
    add %r0,%r1   -> %r1
    st  %r1 -> j
    st  %r0 -> i
    ret
```

Kod

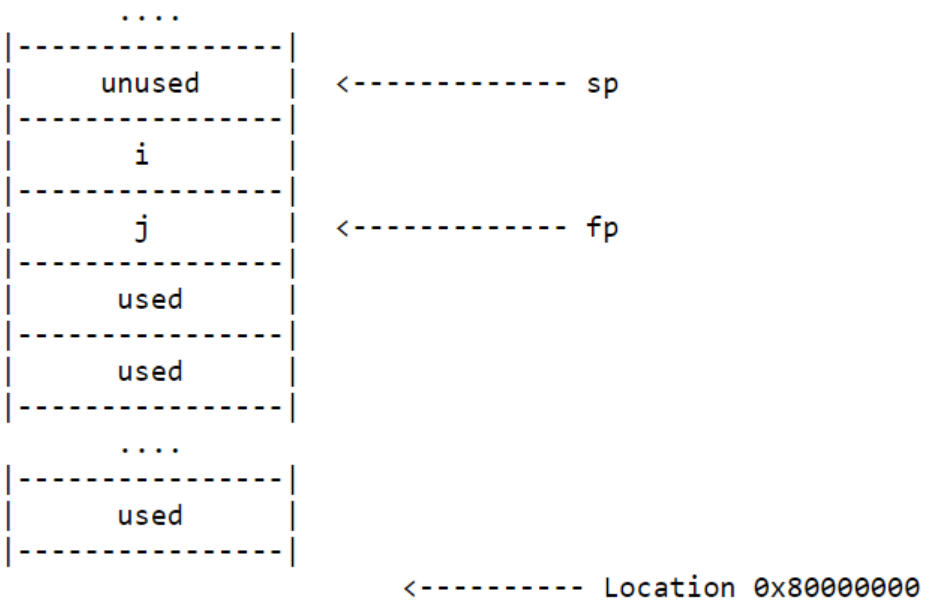
```
int main()
{
  int i, j;

  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** kayıtçıları 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



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!

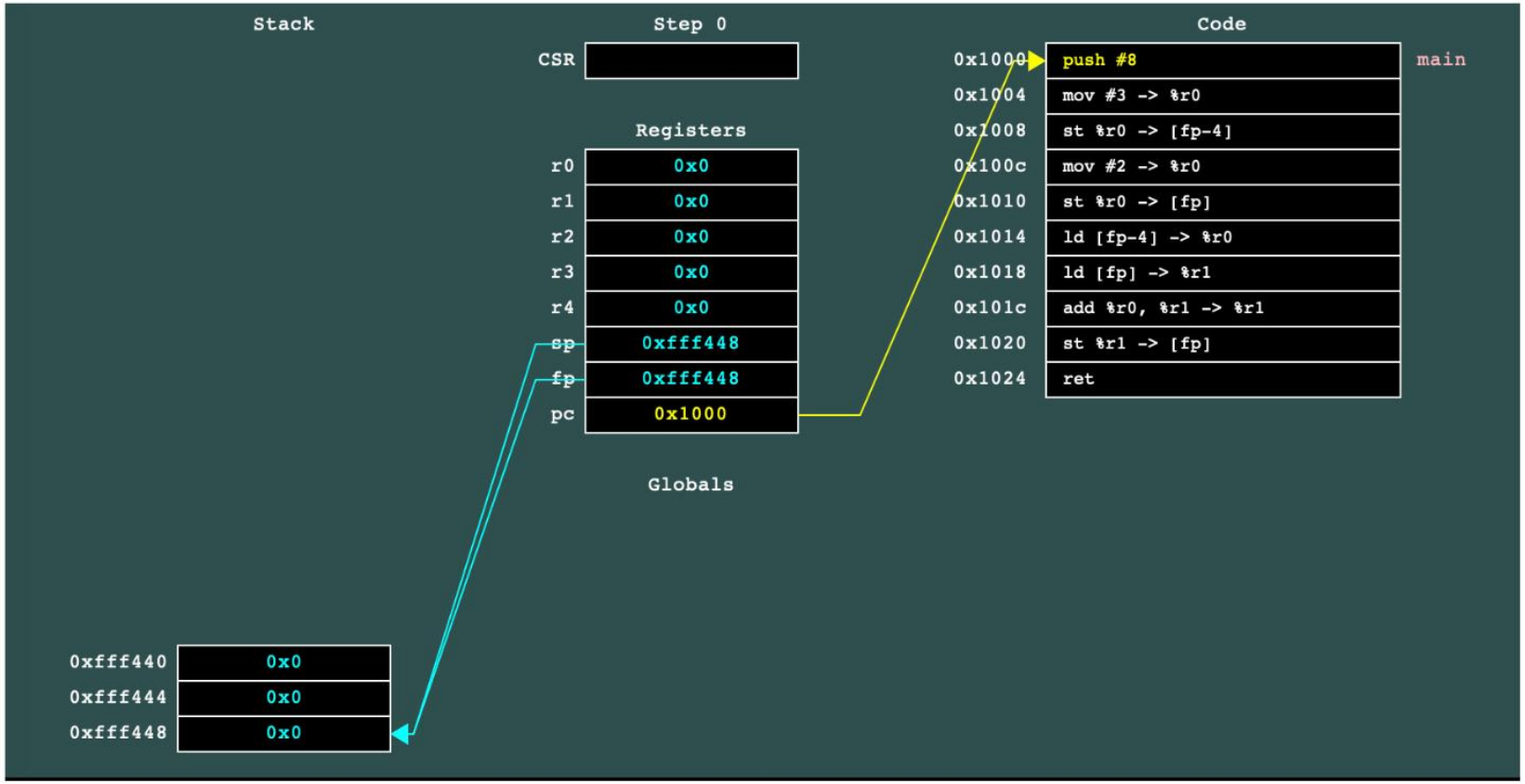
Jassem -- The Visual Assembler

- Assembler anlamanıza 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 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 anlık görübtüsünü 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.
- Yığıt-heap yok. Yığın-stack 0xfff44c'de biter.

Jassem


Stack
Step 0
Code
CSR
Registers
r0
r1
r2
r3
r4
sp
fp
pc
0x0
0x0
0x0
0x0
0x0
0xfff448
0xfff448
0x1000
Globals
0x1000
0x1004
0x1008
0x100c
0x1010
0x1014
0x1018
0x101c
0x1020
0x1024
push #8
mov #3 -> %r0
st %r0 -> [fp-4]
mov #2 -> %r0
st %r0 -> [fp]
ld [fp-4] -> %r0
ld [fp] -> %r1
add %r0, %r1 -> %r1
st %r1 -> [fp]
ret
main
0xfff440
0xfff444
0xfff448
0x0
0x0
0x0

Soru

```
int i;
main(){
int j, k;
i=3;
j=5;
k=i+j;
i++;
}
```