Pipe()

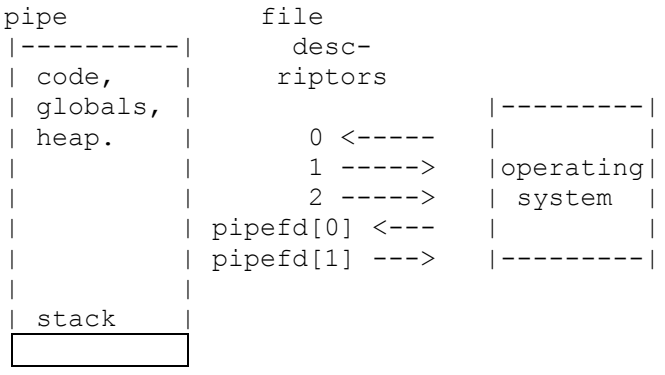
Öncelikle pipe ile ilgili man sayfasını okuyun.Ben **pipe.txt** de SunOS versiyonunu dahil ettim.Bu metnin okuyacağınız en iyi 70 satırı..Bilmeniz gereken her şeyi açık bir şekillde anlatıyor.Bu çok güzel.Sistem çağlrılarından olan Pipe lar , parent-child proseslerine diğerler proseslerle iletişim kurabilmesi için bir yol sunar.Genellikle şu şekilde çağtılırlar:

```
int pipe(int fd[2]);
```

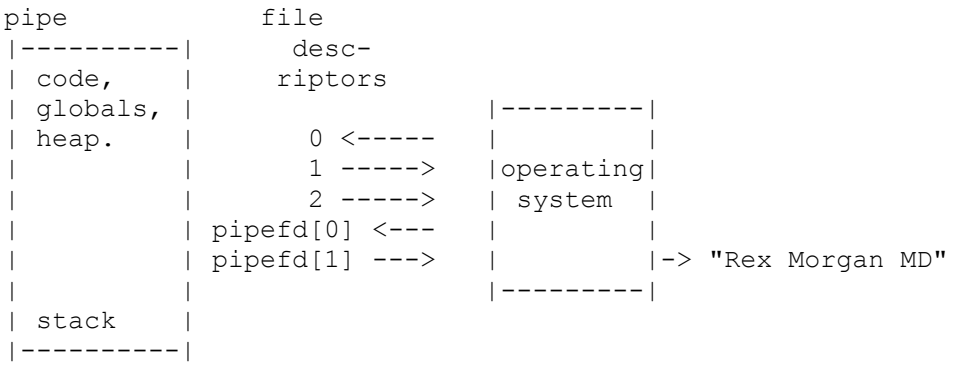
Diğer bir deyişle 2 tamsayı dizisini devredersin .Bu dizileri diğeriyle anlaşabilen iki dosya tanımlayıcıyla doldurur. fd[1] tarafından yazılan herhangi bir şey fd[1] tarafında akunabilir. Bunun tek prosesler için hiç bir faydası yok.Ancak prosesler arasında bir iletişim metodu verir. **pipe.c** ye bakın .pipe() ilk çağrılışı fd[0] ve fd[1] dosya tanımlayıcılarını kurmak içindir. fd[1] tarafından yazılan herhangi bir şey fd[1] tarafında akunabilir.Bunu koymanın diğer yolu , her nerede **write(fd, buf, size)** cağırdıysan , prosesin işletim sistemi için tampon ile belirtilen adreste başlayan boyut baytını gönderir. Fd işletim sistemine bu baytlrla ne yapılacağını söyler.fd Genellikle open() tarafından geri döndürülen bir dosya tanımlayıcıdır.Böylece sizin write() çağrınız işletim sistemine bu baytları dosyaya yazmak için kullanacağını söyler.Anacak diğer dosya tanımlayıcı tipleri de vardır.Örneğin; **write(1, buf, size)** dediğinizde ,bu baytları genellikle bir disk dosyası değil, fakat bunun yerine bir terminal olan bir standart bir çıkış'a yazmak için kullacağız demektir.
Fd pipe sonuna yazma olduğunda , **write()** işletim sistemine bu baytları diğer proselerin onlar için **read()** icra ederek pipe ın sonunu okuyana kadar tamponda tutacağını belirtir.

Bu tüm işletim sistemi üzerinden gerçekleştirilmesi gereken süreçler arası iletişimi tanımlamak için çok önemlidir. Borular (pipe) bunun gerçekleşmesi için temiz bir yoldur.

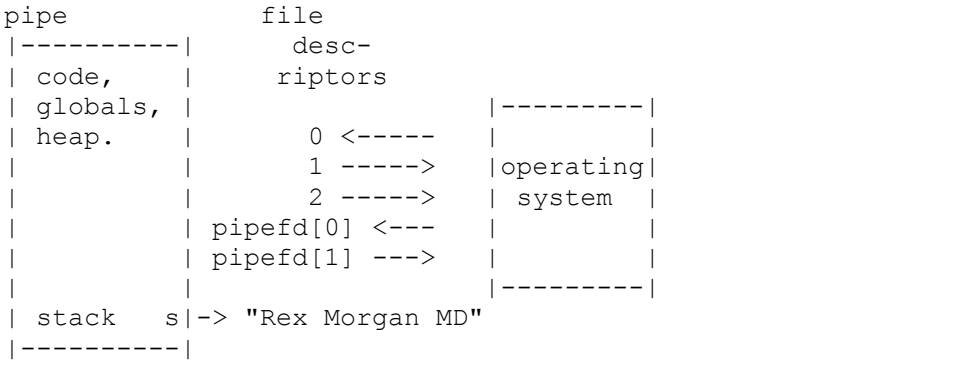
Pipe.c ye geri dönelim . pipe ı ilk çalıştırdımızda , 5 tane açık dosya tanımlayıcına sahip çalışan prosesi görebiliriz.Standar giriş(0), standar çıkış(1), standart hata(2) , pipe sonun okuma(**pipefd[1]**) ve pie sonun yazmak(**pipefd[1]**).Bu dosya işaretçilerin her biri işletim sitemi için bir işaretçidir.Biz onu görselleştirebiliriz :



Şimdi ilk çağrımız : "**write(pipefd[1], s2, strlen(s2));**",
Bu onu tampon bölgede tutan işletim sistemine bir dize yollar: “Rex Morgan MD”



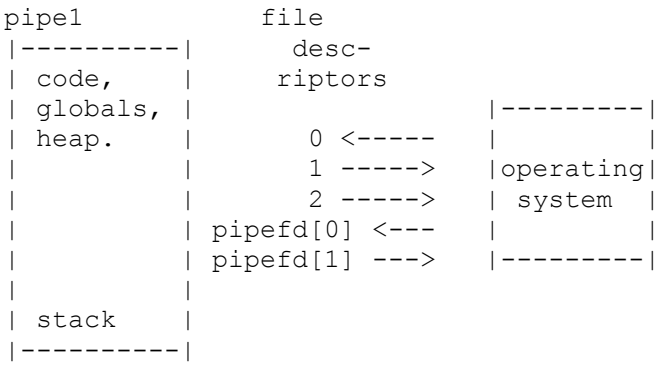
Ardından pipe dan 1000 karakter okumaya çalışan çağrımızı kullanırız:
"**i = read(pipefd[0], s, 1000);**" , Bu işletim sisteminden "Rex Morgan MD" dizesini alır ve onu s değişkenine atar :



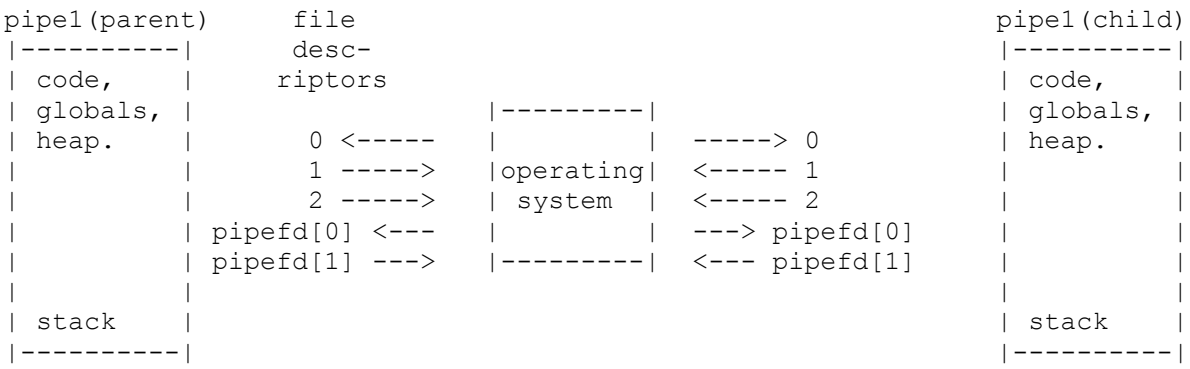
Bu pipe ların çok kolay bir kullanımı bunun yanında işe yaramaz.ancak bu bize bir proses içinde pipe ların kullanımı gösterir.

Pipe ların normal kullanım yolu için ilk önce bir pipe yaratılır , ardından çoçuk proseslere (child process)ayrılması sağlanır. Ebeveyn ve çocuğun ikiside benzer dosya tanımlayıcılarına sahip olduğundan , pipe üzerinden iletişim kolayca uygulanır. Buraya not etmemiz gereken çok önemli bir noktada pipe larda full dublex yerine half dublex kabul görmüştür.Aslında çoğu durumda pipe üzerinden iletişim tek yönlüdür.program borusu için doğru yol, kullanılmayan pipefd leri kapatmak için parent ve child proseslere sahip olmaktır.Eğer bunu yapmazsanız çok ilginç şeyler olur.Örnek için aşağıya bakın.

Şimdi pipe1.c ye bakın .
Tekrar **pipe()** çağrıldıktan sonra sistem şu şekilde görünür :



Şimdi **, fork()** çağrıldığında pipe1 i çiftleyen yeni bir proses olusturulur.dosya tanımlayıcıları genellikle çiftlenir dolayısıyla onlar da işletim sistemindekilere benzer bir işaretçi olurlar.hal şimdi şu şekilde olur :



Şimdi parent proses standart girişten gelen metnin satırlarını okumak için **fgets()** çağırır ve bunları pipe ‘ a yazarchild prosesler pipe dan okur ve her satırı standart çıkışa yazar.Bu kod sana kolay görünmeli. **pipefd[1]** için her yazımın işletim sistemine gittiğini görmelisin .

pipe1() çalıştırmayı deneyelim :

```
UNIX> pipe1
How bout them Vols!
     1  How bout them Vols!
Give him six!
     2  Give him six!
Juice em, Big Dog, Juice em!
     3  Juice em, Big Dog, Juice em!
< CNTL-D >
UNIX>
```

İyi görünüyor. Şimdi, benzerini standart çıkış dosyasına yapmayı deneyelim:

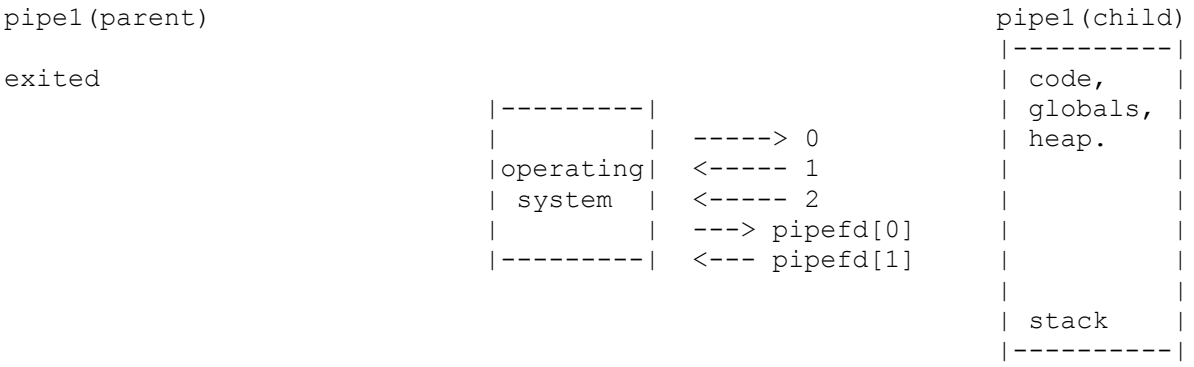
```
UNIX> pipe1 > output
How bout them Vols!
Give him six!
```

```
Juice em, Big Dog, Juice em!
< CNTL-D >
UNIX> cat output
UNIX>
```

Hmmm. Bu bir problem gibi görünüyor.**fork()** dosya tanımlayıcılarını çiftlediğinden beri,child proseslerin çıkışa yazdığını varsaydık çünkü standart çıkışın nerede olduğuna yönlendirildi.Bu doğru probelm **ps x** (or **ps aux | grep $USER**) yaparken görünebilir :

```
UNIX> ps aux | grep plank
...
plank     6277  0.1  0.3  760  576 pts/22   S 09:40:25  0:00 grep plank
plank     6241  0.0  0.2  684  436 pts/22   S 09:39:02  0:00 pipe1
plank     6244  0.0  0.2  700  452 pts/22   S 09:39:23  0:00 pipe1
...
UNIX>
```

Neler oluyor? Bu en iyi aşağıdaki resim ile açıklanabilir .Parent proses CNTL-D kodlandığında , **pipefd[1]** yi kapatır ve çıkar.Sistemin durumu şuan şu şekilde görünülüyor:



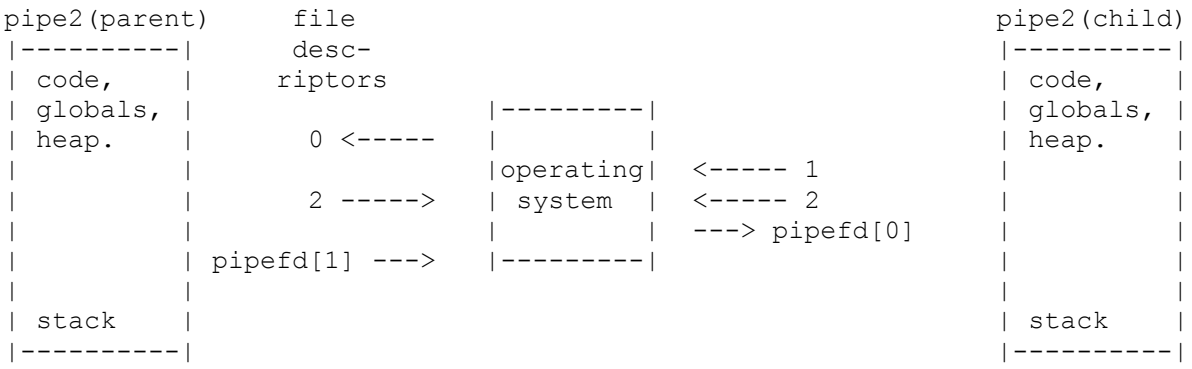
Not olarak child proses şuan hala açık **pipefd[1]**' e sahip.Böylece bu pipe[0] da okumak için bekler ve işletim sistemi hiçbir prosesin **pipefd[1]**' e yazmayacağını bilmez.Dolayısıyla child proses orada hiçbir şey yapmadan oturur.Çıkış dosyasında hiçbir şey yoktur , çünkü child proses yazmak için çıkışı tamponlayan printf kullanır .tampon dolmadığında henüz **write(1, ...)** yi gerçekleyemez ve böylelikle biz çıkış dosyasında hiçbir şey göremeyiz. ps x emri görüldüğünde , orada 2 proses vardır – yukarıda çağırdığımız her iki **pipe1** den bir child proses. (i.e. "**pipe1**" and "**pipe1 > output**").
Şundan emin olabilirsin ki , daha fazlasını okuduğunda burada neler olduğunu anlayacaksın . Child proses **pipefd[0]** ' dan okuma kısmında sılı durumdadır. **read()** çağrısı geri dönüş yapmayacak çünkü orada okuyacak hiçbir şey yok ve **pipefd[1]**' den beri kapalı.işletim sistemini **read()** çağrısından değer olarak 0 dönmesini sağlayamaz.Dolayısıyla bu proses askıda kalır.

Şimdi bu iki child prosesi öldürelim :

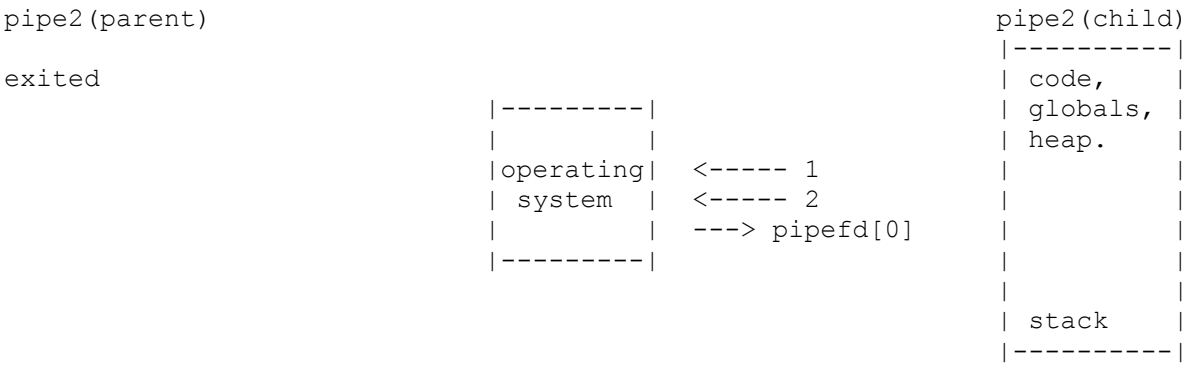
```
UNIX> kill 6241 6244
```

Ve pipe2.c ye bak.pipe2.c kullanılmayacak olan dosya tanımlayıcıoları kapatan parent prosese sahip.parent ve child proseslerin her ikisi birden kendi döngülerine

girdiklerinde,sistemin durumu aşağıdaki gibi görülür, kullanılmayan dosya tanımlayıcılarından dolayı,



Şimdi , <CNTL-D> kodladığında parent proses sistemi aşağıdaki gibi bırakarak çıkar :



Not: pipefd tamamladı.Böylece işletim sistemi child prosesin 0 döndüren read**(pipefd[0], ...)** metoduna sahiptir ve cild proses zerafetle çıkar.pipe2 yi çağırdığınızda önce pipe1 de olduğu gibi çıkış dosyası doğru olarak oluşturuldu, dolayısıyla etrafta asılı kalan hiçbir child proses kalmadı.

```
UNIX> pipe2 > output
How bout them Vols!
Give him six!
Juice em, Big Dog, Juice em!
< CNTL-D >
UNIX> cat output
     1  How bout them Vols!
     2  Give him six!
     3  Juice em, Big Dog, Juice em!
UNIX>
```

Eğer **‘ps x’** yaparsanız pipe2 proseslerini görmemelisiniz.

SIGPIPE

Yukarıda programda gördüğünüz gibi ,pipe dan okumaya çalıştığınızda , soan yazılmaz ve **read()** 0 döndürür.Pipe ‘ a yazmaya çalıştığınızda, sona okuma olmaz ve **SIGPIPE** sinyali olsturulur.Eğer program kullanılmadıysa program sessizce çıkar.Bu hoş , çünkü bunun anlamı örnek olarak yürütmek :

```
UNIX> cat exec1.c | head -5 | tail -1
```

ve ortadaki prosesi öldür,(the head), diğer ikisi otomatik çıkacak. tail **read()** ‘ e sahip, ve geriye 0 döndürür ve çıkacak , ve cat boş bir pipe ‘ a yazmaya çalışacak, ve dolayısıyla **SIGPIPE** oluşacak ve çıkacak.

pipe3.c ‘ye bak.

Bu diğerleriyle benzer işleri yapar, ancak **SIGPIPE** ‘ı yakalar.(Eğer sinyalleri bilmiyorsan kitaptan 10. bölümü ve sinyalin man sayfasını oku).Test etemk içinse pipe3 ü çalıştır;

```
UNIX> pipe3
Juice em, Big Dog, Juice em!
     1  Juice em, Big Dog, Juice em!
```

Ardından başka bir pencerede child prosesi öldüe , bu daha büyük bir pid’e sahip olacaktır(pid=proses id)

```
UNIX> ps aux | grep plank
...
plank     7064  0.1  0.2  684  452 pts/22   S 09:44:24  0:00 pipe3
plank     7065  0.0  0.2  684  304 pts/22   S 09:44:24  0:00 pipe3
...
UNIX> kill 7065
UNIX>
```

Pipe3 penceresinde hiçbir şey olmadığını göreceksiniz , fakat çocuk kaybolur.(tekrar "**ps aux | grep $USER**" yazarız emin olmak için) Bunun anlamı , açık **pipefd[0]** ‘ a sahip proses mevcut değildir.

```
Give Him Six!
15454: caught a SIGPIPE
UNIX>
```

Write() pipefd[1] için SIGPIPE olusturur.

Bu bir sonraki lab’ı yapaman için sana yeterli bilgiyi vermiş olmalı(Aslında Jsh lab için SIGPIPE ‘a ihtiyacın olmayacaktı). ,Çıkış için yardım almak için programın headsort.c kısmına bakın .—şu şekilde uygulanır :

```
   "head -10 headsort.c | sort"
```

Ve onu bitirmesi için bekler.Eğer biz sınıfta bunu alamazsak , kendiniz kodun üstüne gidin.Bun un Jsh lab için çok yardımcı olduğunu kendiniz bulacaksınız.

Ana işlem diğer proses için bir pipe oluşturmak olduğundan , birini onun çıkışından okumak yada onun girişine yazmak için standart UNIX I/O kütüphanelerine popen ve pclose fonksiyonları sağlandı.Bu iki fonksiyon ardındaki adımları işler.

```
Pipe yaratmak.
Çocuklara dallanmak.
Kullanılmayan pipe sonlarını kapatmak.
Emirleri yürütmek için kabuk işlenir.
Emrin tamamlanması için beklemek.
```

Tabiki **popen** ve **pclose** geniş bir hata kontrolüne sahiptir.Senin lab atamaların için her **popen** ve **pclose** ‘ u kullanma yetkin yok.Ancak eğer sen kaç tane hata beklemeliyim konusunda meraklı isen, bölüm 14 deki örnek koda başvurabilirsin.
Not: **waitpid** kullanmaya yetkin yok.Sadece körü körüne bu örnek kodu takip etmeyin .