Karadeniz Teknik Üniversitesi
Bilgisayar Mühendisliği Bölümü
Bilgisayar Sistemleri Laboratuarı

1. SÜREÇLER ARASI HABERLEŞME (INTER PROCESS COMMUNICATION - IPC)

Süreçler arası haberleşme, verinin süreçler arasındaki transferi olarak tanımlanabilir. Süreçler aynı bilgisayarda koşabileceği gibi, ağ üzerinde bağlı farklı bilgisayarlar üzerinde de koşabilir. IPC bir sürecin diğer bir süreci kontrol edebilmesini olanaklı kılar. Birçok süreç aynı anda aynı veriyi tutarsızlık problemi oluşturmadan kullanabilir. IPC çok süreçli sistemlerde kullanılması gereken bir kavramdır. IPC'nin kullanımına örnek olarak bir web istemcinin (Explorer yada firefox gibi bir web browser), web sunucusundan talep ettiği web sayfasını HTML kod şeklinde geri alması verilebilir. Bir dizindeki dosyaların isimlerinin yazıcıdan çıktısını almak için kullanılan ls | lpr, IPC'ye verilebilecek diğer bir örnek olacaktır. Shell ls ve lpr süreçlerini ayrı olarak oluşturur ve her ikisini birbirine "|" sembolü ile gösterilen pipe yoluyla bağlar. Pipe, iki süreç arasında tek yönlü bir haberleşmeyi olanaklı kılan bir yapıdır. Ls süreci çıktı verisini pipe'a yazar ve lpr süreci ilgili veriyi pipe'dan okur.

Deney föyünde beş farklı IPC tekniğinden bahsedilecektir:

- Paylaşımlı Bellek (Shared Memory): Süreçler belirli bir bellek bölgesine yazma ve okuma yaparak haberleşmeyi gerçekleştirir.
- Haritalanmış bellek (Mapped Memory): Paylaşımlı belleğe benzer bir yapıdır. Dosya sisteminde bulunan dosya ile ilişkilendirmesi ile shared memory kavramından farklılık göstermektedir.
- Pipe'lar bir süreçle diğer bir süreç arasında sıralı haberleşme sağlar.
- FIFO'lar pipe'lara benzer yapıdadır. Fakat bu yöntemde birbiri ile ilişkili olmayan süreçlerde aralarında haberleşebilmektedir.
- Soketler ilişkili olmayan süreçler arasındaki ve hatta farklı bilgisayarlardaki süreçler arasında haberleşmeyi sağlar.

2. IPC TEKNİKLERİ

2.1. Paylaşımlı Bellek (Shared Memory)

Süreçler arası haberleşmedeki en basit teknik paylaşımlı bellek kullanımıdır. Bu yöntem aynı anda birden çok sürecin paylaşımlı bir bellek ortamına erişimini sağlar. Bütün süreçlerin malloc çağrısı ile geri dönen gösterici değerlerinin aynı olması, aynı bellek bölgesini erişimlerini sağlar. Süreçlerden biri bellek bölgesini değiştirdiğinde, diğer süreçlerde bu değişikliği göreceklerdir.

2.1.1. Hızlı yerel haberleşme (Fast Local Communication)

Süreçlerin hepsi aynı bellek bölgesini paylaştığı için, paylaşımlı bellek kavramı en hızlı IPC tekniğidir. Paylaşımlı bellek bölgesine erişim, sürecin kendine ait bellek bölgesine erişimi kadar hızlı gerçekleşir. Herhangi bir sistem çağrısını kullanmaya gerek yoktur. Aynı zamanda veriyi gereksizce kopyalamanın da önüne bu yöntemin kullanımıyla geçilir.

Yalnız kernel, paylaşımlı belleğe olan erişimler esnasında herhangi bir senkronizasyon yapmayacağı için, süreçler kendi aralarında senkronizasyonu gerçekleştirmek zorundadır. Örneğin birden fazla sürecin aynı bellek bölgesine aynı anda yazmasına engel olunmalıdır. Böyle yarış durumlarından kaçınmanın yolu semaforları kullanmaktır.

2.1.2. Bellek Modeli

Paylaşımlı bir bellek segmentini kullanabilmek için, bir sürecin bu bölgeyi ayırması gerekir. Segmenti kullanmak isteyen diğer süreçler, bölgeye eklenmelidir (attach). Süreç, kullanımı sona erdiğinde, segmenti ayırır.

Linux teki bellek modelini anlayabilmek için, tahsis etmek ve bağlamak kavramlarını anlamak gerekir. Linux'de her sürecin sanal belleği sayfalara bölünmüştür. Her süreç kendi bellek adresinden bu sanal bellek sayfalarına olan bir harita tutar. Sanal bellek sayfaları o anki veriyi içermektedir. Her sürecin kendi adres uzayı olmasına rağmen, bir çok sürecin haritaları aynı sayfayı işaret edebilir. Böylelikle belleğin paylaşımı sağlanmış olur.

Yeni bir paylaşımlı bellek segmentini tahsis etmek sanal bellek sayfalarının yaratılmasına sebep olur. Bütün süreçler aynı paylaşımlı segmente erişmek isteyeceği için, süreçlerden yalnızca biri tahsis işlemini gerçekleştirir. Var olan bir segmentin tahsis edilmesi yeni sanal bellek sayfalarını oluşturmaz. Bunun yerine var olan sayfalar için bir tanımlayıcı döndürülür. Bir sürecin paylaşımlı bir bellek segmentini kullanması için, ona eklenmesi gerekir. Böylelikle sürecin kendi sanal belleğinden segmentin paylaşımlı sayfalarına erişimi gösteren ilişkilendirmeler, haritasına eklenir. Sürecin segmenti kullanımı sona erdiğinde , bu harita girişleri kaldırılacaktır. Paylaşımlı bellek segmentine erişim isteyen süreç kalmadığı zaman, bölgeyi son kullanan süreç sanal bellek sayfalarını geri vermelidir.

Bütün bellek segmentleri sistemin sayfa ölçüsünün katları şeklinde tahsis edilir. Sayfa ölçüsü, bellekteki sayfanın byte cinsinden büyüklüğüdür. Linux sistemlerde sayfa ölçüsü 4 KB'dır. Bu değeri öğrenebilmek için getpagesize fonksiyonu kullanılabilir.

2.1.3. Tahsis etmek (Allocation)

Süreç paylaşımlı bellek segmentini shmget (Shared memory get) fonksiyonunu kullanarak tahsis eder. Fonksiyonun ilk parametresi hangi segmentin yaratılacağını gösteren tamsayı bir anahtar değeridir. Diğer süreçler aynı paylaşımlı bellek bölgesine aynı anahtar değerini vererek erişebilir. IPC_PRIVATE özel sabitinin anahtar değeri olarak kullanımı, yeni bir bellek segmentinin yaratılacağını garanti eder. İkinci parametresi segmentteki byte miktarını belirtirr. Segmentler sayfaların kullanımı ile tahsis edildiği için, segment için ayrılacak olan bellek miktarı sayfa büyüklüğünün katları olmalıdır. Üçüncü parametre ise shmget fonksiyonuna geçirilecek olan opsiyonları belirleyen bayrak (flag) değerleridir.

- IPC_CREAT – Bu bayrak değeri yeni bir segmentin yaratılması gerektiğini söyler.
- IPC_EXCL – IPC_CREAT ile beraber kullanılır. shmget'in aynı segment anahtarının önceden kullanılmış olması durumunda başarısız sonuç döndürmesini sağlar. Aksi takdirde shmget, yeni bir segment oluşturmak yerine, anahtar değeri ile belirtilen ve önceden oluşturulmuş olan segmenti geri döndürür.
- Mode bayrakları – bu değer 9 bit değerinden oluşur. Bu bit değerleri ilgili segmente owner, group, world erişimlerini düzenler. Erişim haklarını tanımlamanın en kolay yolu <sys/stat.h> dosyasında tanımlı olan sabitleri kullanmaktır. Örneğin S_IRUSR ve S_IWUSR paylaşımlı bellek bölgesine owner'ın yazma ve okuma haklarını belirtir. S_IROTH ve S_IWOTH ise diğerleri için olan okuma ve yazma haklarını belirtecektir.

Yalnızca owner tarafından okunabilen ve yazılabilen paylaşımlı bellek segmentinin ayrılabilmesi için shmget fonksiyonunun kullanımı aşağıdaki şekildedir:

```
int segment_id = shmget (shm_key, getpagesize(), IPC_CREAT | S_IRUSR, S_IWUSER) ;
```

Eğer çağrım başarılı olursa, fonksiyon ilgili segment bölgesini gösteren bir tanımlayıcı döndürecektir.

2.1.4. İlişkilendirme ve ilişki kesme (Attachment and Detachment)

Paylaşımlı bir bellek bölgesinin süreç tarafından kullanılabilmesi için , shmat (Shared memory Attach) fonksiyonunun kullanılması gerekir. Fonksiyonun ilk parametresi, shmget fonksiyonu tarafından döndürülen segment tanımlayıcısıdır. İkinci argüman ise, paylaşımlı belleğin haritalanacağı sürecin adres uzayıdır. NULL değeri seçilirse, Linux uygun bir adres seçecektir. Üçüncü parametre ise bayrak değeridir :

- SHM_RND ikinci parametrede belirtilen adres, sayfa ölçüsünün katı olacak şekilde aşağıya yuvarlanması gerektiğini söyleyen bayrak değeridir.
- SHM_RDONLY : segmentin yalnızca okumaya açık olacağını gösterir.

Eğer fonksiyon çağrımı başarılı olursa bağlanılan paylaşımlı segmentin adresi geri döner. Fork çağrısı ile oluşturulan süreçler bağlanılan paylaşımlı segmentleri miras alırlar. Eğer isterlerse ilgili segmentlerle olan ilişkilerini kesebilirler.

Sürecin bellek segmentini kullanımı sona erdiğinde, bellek ile olan ilişkilendirme shmdt (shared memory detach) fonksiyonunun kullanımı ile koparılmalıdır. Alacak olduğu parametre shmat fonksiyonu tarafından döndürülen adres değeridir. Eğer ilişkinin koparıldığı segmeneti kullanan son süreç ise, bellek segmenti taşınır.

2.1.5. Paylaşımlı bellek bölgesinin kontrolü ve geri verilmesi

shmctl (Shared memory control), paylaşımlı bellek segmenti hakkında bilgi döndürür ve ilgili segmenti değiştirebilir. İlk parametresi paylaşımlı bellek bölgesinin tanımlayıcısıdır. Paylaşımlı bellek segmenti hakkında bilgi edinebilmek için ikinci parametre olarak IPC_STAT değerinin ve shmid_ds yapısına bir göstericinin geçirilmesi gerekir. Bir segmenti kaldırabilmek için IPC_RMID ikinci parametre, NULL değeri ise üçüncü parametre olarak geçirilir. Her bir bellek segmentinin kullanımı sona erdiğinde shmctl kullanılarak explicit olarak geri verilmelidir.

2.1.6. Örnek Program

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

```
  /* Attach the shared memory segment.  */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size.  */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Deatch the shared memory segment.  */
  shmdt (shared_memory);

  /* Reattach the shared memory segment, at a different address.  */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

2.3. Haritalanmış bellek (Mapped Memory)

Bu teknikte, farklı süreçler paylaşılan bir dosya aracılığıyla haberleşebilmektedir. Her ne kadar haritalanmış belleğin, paylaşımlı bellek segmentinin bir isim ile kullanılması gibi düşünülse de arada bazı teknik farklılıklar vardır. Haritalanmış bellek bir dosya ile sürecin belleği arasında bir ilişkilendirme oluşturur. Linux dosyaları her biri sayfa büyüklüğünde olan parçalara böler ve sonra onları sanal bellek sayfalarına kopyalar. Böylece sürecin adres uzayında kullanılabilir olurlar. Süreç, dosyanın içeriğini olağan bir bellek erişimi ile okuyabilir. Aynı zamanda belleğe yazarak dosyanın içeriğini de değiştirebilir. Bu da dosyalara hızlı erişime izin verir.

Haritalanmış bellek, bir dosyanın tüm içeriğini tutabilmek için bir buffer ayırma gibi düşünülebilir. Ardından dosyanın tüm içeriği buffer'a okunur ve eğer buffer da bir değişiklik varsa yapılan değişiklik tekrar bufferdan geriye dosyaya yazılır. Linux dosya yazma ve okuma işlemlerini kullanıcı yerine halledecektir.

2.3.1. Olağan bir dosyayı haritalama

Bir dosyayı bir sürecin belleğine haritalayabilmek için mmap (Memory mapped) fonksiyonu kullanılır. ilk argüman sürecin adres uzayında dosyanın haritalanmasını isteyeceğiniz adres değeridir. NULL değeri Linux'ün uygun bir adres değerini başlangıç adresi olarak seçmesini sağlar. İkinci argüman haritanın byte cinsinden büyüklüğüdür. Üçüncü argüman haritalanmış adres aralığındaki korumayı belirtir. Koruma PROT_READ, PROT_WRITE ve PROT_EXEC in OR'lanmasından oluşur. Dördüncü parametre ise ek seçenekler belirten bir bayrak değeridir. Beşinci argüman haritalanma yapılacak olan dosyaya ilişkin bir dosya göstericisidir. Son argüman ise haritalanmanın dosyanın başından itibaren nereden başlayacağını belirten ofset değeridir. Bir dosyanın sadece belirli bir kısmını yada hepsini belleğe haritalamada kullanılabilir. Uygun ofset ve uzunluk değerleri bu amaçla seçilebilir.

Fonksiyona parametre olarak geçirilecek olan bayrak değerlerine ilişkin değerler şu şekildedir:

- MAP_FIXED: Linux talep edilen adresi dosyayı haritalamada kullanır.
- MAP_PRIVATE: bellek aralığına yazmalar ilişkili dosyaya geri yazılmaz. Bunun yerine dosyanın özel bir kopyasına yazılır. Diğer süreçler bu yazmaları görmeyecektir.
- MAP_SHARED: Yazmalar bufferlama yerine hemen dosyaya yansıtılır. IPC için haritalanmış belleğin kullanılması durumunda bu mod kullanılır. bu mod MAP_PRIVATE ile birlikte kullanılmaz.

Eğer fonksiyon çağrımı başarılı olursa, belleğin başlangıcını işaretleyen bir gösterici geri döndürür. Başarısızlık durumunda MAP_FAILED döner.

Bellek haritalama bittiğinde, munmap ile serbest bırakılması gerekir. Fonksiyona başlangıç adresi ve haritalanmış bellek bölgesinin uzunluğu parametre olarak geçirilir. Linux bir süreç sonlandığında haritalanmış bölgelerdeki haritalamayı otomatik olarak geri alır.

2.3.2. Örnek programlar

Bellek haritalanmış bölgelerin, dosyaların okunması ve yazılmasına ilişkin kullanımlarına iki programla örnek verilecektir. İlk program rasgele bir sayı üretmekte ve bu sayıyı bellek haritalanmış dosyaya yazmaktadır. İkinci program sayıyı okumakta, yazmakta ve bellek haritalanmış dosyada bu değeri ikiyle çarparak yer değiştirmektedir.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
```

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* Return a uniformly random number in the range [low,high].  */

int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;

  /* Seed the random number generator.  */
  srand (time (NULL));

  /* Prepare a file large enough to hold an unsigned integer.  */
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  /* Create the memory-mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Write a random integer to memory-mapped area.  */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* Release the memory (unnecessary since the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

mmap-write programı dosyayı açar, eğer önceden yoksa yaratır. open fonksiyonuna verilen üçüncü argüman dosyanın okuma veya yazma için açıldığını belirtir. Dosyanın büyüklüğü bilinmediği için lseek fonksiyonu, dosyanın tam sayı bir değer tutabilecek kadar büyük olup olmadığı tespit etmek için kullanılır. ardından dosya pozisyonunu başa alır. Program dosyayı haritalar ve dosya tanımlayıcısını kapar çünkü daha ihtiyaç duyulmayacaktır. Program rasgele bir sayıyı haritalanmış belleğe yazar. Ve böylece aslında dosyaya yazmıştır. Ve haritalamayı geri alır. Munmap çağrımı gereksizdir çünkü Linux program sonlandığında otomatik olarak haritalamayı geri alacaktır.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

  /* Open the file.  */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Create the memory-mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
  close (fd);

  /* Read the integer, print it out, and double it.  */
  sscanf (file_memory, "%d", &integer);
  printf ("value: %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
  /* Release the memory (unnecessary since the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

mmap-read dosyadaki sayıyı okur ve iki katı değeri dosyaya geri yazar. Öncelikle dosyayı açar, yazma ve okuma için haritalar. Dosyanın işaretli olmayan bir tamsayıyı tutabilecek büyüklükte olduğu varsayımı yapıldığı için, lseek'i tekrar kullanmaya gerek yoktur. Program sscanf kullanarak değeri bellekten dışarı okur, biçimler ve iki katı değeri sprintf'i kullanarak tekrar yazar. Bu örnek programın koşması için komut satırından girilmesi gereken ifade aşağıdaki şekildedir. Tmp dizini altındaki integer-file isimli dosya haritalanmaktadır.

```
%./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% /mmap-read /tmp/integer-file
value : 42
% cat /tmp/integer-file
84
```

Dikkat edilmesi gereken husus 42 değerinin diskteki bir dosyaya write çağrımı kullanılmadan yazılmış olmasıdır. Okuma için de read çağrımı kullanılmamıştır.

2.3.3. Bir dosyaya paylaşımlı erişim (Shared Access to a file)

Farklı süreçler aynı dosya ile ilişkilendirilmiş bellek haritalanmış bölgelerin kullanımı ile haberleşebilir. MAP_SHARED bayrağı belirtilirse, bu bölgeye yapılan yazımlar hemen alttaki dosyaya iletilecektir ve diğer süreçlere görünür yapılacaktır. Eğer bu bayrak belirtilmezse, Linux yazımları dosyaya transfer etmeden önce bufferlayabilir.

Bunun dışında, bufferlanmış yazımları diskteki dosyaya yazması için msync'yi çağırarak Linux zorlanabilir. Bu fonksiyonun ilk iki parametresi bellek haritalanmış bölgeyi belirtir. Üçüncü parametre ise şu bayrak değerlerini alabilir:

- MS_ASYNC: güncelleme işlemi kaydedilmiştir fakat fonksiyon dönmeden koşması gerekli değildir.
- MS_SYNC: güncelleme hemen olur; güncelleme işlemi yapılıncaya kadar msync'ye çağrım bloklanır.
- MS_INVALIDATE: diğer bütün dosya haritalamaları geçersiz kılınır, böylelikle onlar yapılan güncellemeyi görebilir.

Paylaşımlı bellek bölgelerinde olduğu gibi, bellek haritalanmış bölgelerin kullanıcıları yarış durumlarını engellemek için bir prosedür takip etmelidir. Örneğin birden fazla sürecin haritalanmış belleğe aynı anda erişimine engel olmak için bir semafor kullanılabilir.

2.4. PIPE

Pipe, tek yönlü haberleşmeye izin veren bir haberleşme aracıdır. Pipe'ın bir tarafından yazılan veri çıkış ucundan okunur. Pipe lar seri aygıtlardır. Pipe'daki veri her zaman yazıldığı sırada okunur. Tipik olarak, tek bir süreç altındaki iki thread arasındaki haberleşmede veya ana ve çocuk süreç arasındaki haberleşmelerde kullanılır.

shell'de, | sembolü bir pipe oluşturur. Örneğin "ls | less" komutu shell in iki çocuk süreç oluşturmasına sebep olur. Shell aynı zamanda ls'in çıkışını less'in girişine bağlayacak bir pipe'da oluşturur.

Pipe'ın veri kapasitesi limitlidir. Yazan süreç, okuyan süreçten daha hızlıysa, pipe daha fazla veriyi tutamaz hale geldiğinde yazan süreç bloklanır ve okuyan sürecin pipe!ı belirli bir oranda boşaltmasını bekler. Tersi durumda, okuyan sürecin okuyabileceği veri yazan süreç tarafından henüz yazılmamışsa bloklanacaktır. Böylece, pipe iki süreci kendiliğinden senkronize edecektir.

2.4.1. Pipe'ların oluşturulması

Bir pipe'ı oluşturmak için pipe komutu kullanılır. 2 elemanlı bir tamsayı dizisi olsun. Pipe'a çağrım okuyan dosya tanımlayıcısını sıfırıncı pozisyonda yazma dosya tanımlayıcısını birinci pozisyonda tutar.

```
int pipe_fds[2];
int read_fd;
int write_fd;

pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

2.4.2. Ana ve çocuk süreçler arasındaki haberleşme

pipe fonksiyonu dosya tanımlayıcıları oluşturur. Bunlar yalnızca ilgili süreç ve çocuklarında geçerli olacaktır. Sürecin dosya tanımlayıcısı ilişkide olmayan diğer süreçlere geçirilemez. Bununla birlikte süreç fork'u çağırdığında, dosya tanımlayıcıları çocuk sürece de kopyalanacaktır. Böylece, pipe'ların yalnızca ilişkide olan süreçleri biribirine bağladığı söylenebilir.

Verilen örnek programda, fork çağrısı yeni bir çocuk süreç oluşturur. Çocuk süreç pipe dosya tanımlayıcısını miras alır. Ana süreç pipe'a bir string yazar ve çocuk süreç de bunu okur. Örnek program dosya tanımlayıcılarını FILE* stream lerine fdopen kullanarak çevirir.

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

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
```

```
   between each.  */

void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count) {
    /* Write the message to the stream, and send it off immediately.  */
    fprintf (stream, "%s\n", message);
    fflush (stream);
    /* Snooze a while.  */
    sleep (1);
  }
}

/* Read random strings from the stream as long as possible.  */

void reader (FILE* stream)
{
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while (!feof (stream)
         && !ferror (stream)
         && fgets (buffer, sizeof (buffer), stream) != NULL)
    fputs (buffer, stdout);
}

int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    FILE* stream;
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Convert the read file descriptor to a FILE object, and read
       from it.  */
    stream = fdopen (fds[0], "r");
    reader (stream);
    close (fds[0]);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
    /* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    writer ("Hello, world.", 5, stream);
    close (fds[1]);
  }

  return 0;
}
```

main'in başlangıcında, fds iki elemanlı bir dizi olarak tanımlanır. Pipe çağrımı bir pipe oluşturur ve okuma ve yazma dosya tanımlayıcılarını bu diziye yerleştirir. Ardından program fork çağırımı ile çocuk süreç oluşturur. Pipe'ın okuma tarafı kapandıktan sonra, ana süreç pipe'a string leri yazmaya başlar. Pipe'ın yazma tarafı kapandıktan sonra, çocuk stringleri pipe'dan okur.

Writer fonksiyonunda yazma gerçekleştikten sonra ana süreç fflush fonksiyonunu çağırarak pipe'daki verinin hızlı iletimini sağlar. Aralarında ilişki olmayan süreçlerin haberleşebilmesi içinse pipe'lar yerine FIFO'lar kullanılmaktadır.

2.4.5. FIFO

FIFO, aslında dosya sisteminde adı olan bir pipe'dır. Herhangi bir süreç FIFO'yu açabilir yada kapayabilir. Pipe'ın her iki ucundaki süreçler arasında bir ilişkinin olması gerekmez. FIFO lar isimlendirilmiş pipe lar olarak çağrılır. Mkfifo komutunun kullanımı ile FIFO oluşturulabilir. FIFO için yolun tanımlanması

gerekir. Örneğin /tmp/fifo'da FIFO oluşturabilmek için komut satırından aşağıdaki ifadeyi girmek yeterli olacaktır.

```
% mkfifo /tmp/fifo
       Bir pencereden FIFO dan okuyabilmek için aşağıdaki komutu çalıştırın
% cat < /tmp/fifo
       ikinci pencerede, aşağıdaki komutu kullanarak FIFO'ya yazın.
%cat > tmp/fifo
```

Ardından bazı ifadeler yazılsın. Enter tuşuna her basıldığında, yazılan ifade FIFO dan geçecek ve ilk pencerede görünecektir. FIFO yu ctrl+D tuşlarına basarak kapayın. FIFO'yu kaldırabilmek için "rm /tmp/fifo" komutu yeterli olacaktır.

FIFO aslında disk üzerinde bir dosyadır. Kodlama esnasında mknod() fonksiyonunu uygun parametrelerle kullanarak FIFO oluşturulabilir. Aşağıda FIFO oluşturan mknod() çağrısına örnek verilmiştir.

```
mknod ("myfifo", S_IFIFO | 0644, 0)
```

oluşturulan FIFO'ya yukarıdaki örnekte myfifo ismi verilmiştir. ikinci argümanda FIFO'nun oluşturulması esnasındaki erişim hakları belirlenmiştir. son olarak da bir aygıt numarası parametre olarak verilmiştir. FIFO oluşturulduğu için aslında bu değer önemsizdir.

FIFO oluşturulduktan sonra herhangi bir süreç open() sistem çağrısını kullanarak FIFO'yu kullanmaya başlayabilir. Aşağıda verilen örnekte iki süreçten biri veriyi FIFO kullanarak göndermekte, diğeri ise FIFO'daki veriyi çekmektedir.

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

 #define FIFO_NAME "american_maid"

 main()
 {
     char s[300];
     int num, fd;

     /* don't forget to error check this stuff!! */
     mknod(FIFO_NAME, S_IFIFO | 0666, 0);

     printf("waiting for readers...\n");
     fd = open(FIFO_NAME, O_WRONLY);
     printf("got a reader--type some stuff\n");

     while (gets(s), !feof(stdin)) {
         if ((num = write(fd, s, strlen(s))) == -1)
             perror("write");
         else
             printf("speak: wrote %d bytes\n", num);
     }
 }
```

speak isimli program öncelikle bir FIFO oluşturmakta, ardından open() çağrısı ile açmaya çalışmaktadır. Eğer pipe'ın diğer ucu başka süreçler tarafından okunma halinde ise open() çağrısı bloklanarak bekleyecektir.

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

#define FIFO_NAME "american_maid"

main()
{
    char s[300];
    int num, fd;

    /* don't forget to error check this stuff!! */
    mknod(FIFO_NAME, S_IFIFO | 0666, 0);
```

```
        printf("waiting for writers...\n");
        fd = open(FIFO_NAME, O_RDONLY);
        printf("got a writer:\n");

        do {
            if ((num = read(fd, s, 300)) == -1)
                perror("read");
            else {
                s[num] = '\0';
                printf("tick: read %d bytes: \"%s\"\n", num, s);
            }
        } while (num > 0);
    }
```

tick programı eğer pipe'a veri yazan kimse yoksa open () çağrısı yapması durumunda bloklanacaktır.

2.5. Soketler

Soketler iki yönlü haberleşme aygıtlarıdır. Aynı makinedeki farklı süreçler arasında yada uzak makinelerdeki süreçlerin haberleşmesinde kullanılabilir. Şu ana kadar verilmiş olan haberleşme teknikleri içerisinde, uzak makinelerdeki süreçler arasında da haberleşmeyi olanaklı kılan tek yöntemdir. Soket oluşturulacağı zaman, üç parametrenin belirtilmesi gerekir: haberleşme stili, namespace ve protokol. Haberleşme stili soketin iletilen veriye nasıl davranacağını belirler. Aynı zamanda taraflar arasındaki paket iletiminin nasıl gerçekleştirileceğini belirler. Namespace, soket adreslerinin nasıl yazılacağını belirler. Soket adresi soket bağlantısının bir ucunu işaretler. Örneğin lokal namespace için soket adresleri dosya isimleri iken, internet namespace için soket adresi ağa bağlı olan uzak makinenin IP adresi ve port numarasıdır. Son parametre olan protokol verinin nasıl iletileceğini belirler.

Soketler önceki bölümlerde anlatılan haberleşme tekniklerinden daha esnek yapıdadır. Çeşitli sistem çağrılarından soketleri oluşturmada ve haberleşme aracı olarak kullanmakta yararlanılabilir.

Socket – soket oluşturur.

Close – soketi yok eder.

Connect – iki soket arasında bağlantı oluşturur.

Bind – sunucu soketini bir adres ile etiketler.

Listen – bir soketi durumları bekleyecek şekilde konfigüre eder.

Accept – bağlantıyı kabul eder ve bu bağlantı için yeni bir soket oluşturur.

Soketleri oluşturma ve yok etme:

Socket ve close fonksiyonları soketleri oluşturur ve yok eder. Bir soket oluşturuluyorken üç parametre bildirilir, namespace, haberleşme stili, ve protokol. Namespace parametresi için PF_ (Protocol Families) öneki ile başlayan sabitler kullanılır. PF_LOCAL veya PF_UNIX lokal namespace'i tanımlarken, PF_INET Internet namespace'inin kullanılacağını gösterir. Haberleşme stili parametresi için SOCK_ öneki ile başlayan sabitler kullanılır. SOCK_STREAM TCP tabanlı haberleşme stilini tanımlarken, SOCK_DGRAM UDP tabanlı haberleşme stilini ifade eder. Üçüncü parametre protokol ise veriyi iletme ve almak için düşük seviyeli mekanizmayı belirtir. Genellikle 0 olarak kullanımı tercih edilir. Soket çağrısının başarılı olması durumunda soket için olan bir dosya tanımlayıcısı geri döner. Read, write komutları kullanılarak sokete yazılabilir yada okunabilir. Soket'in kullanımı sona erdiğinde ise close komutu ile yok edilir.

2.5.1. Lokal Soketler

Aynı bilgisayardaki süreçleri bağlayan soketler, PF_LOCAL ve PF_UNIX sabitleri ile beraber lokal namespace'i kullanırlar. Bunlar lokal soketler yada UNIX domen soketleri olarak adlandırılırlar. Dosya isimleri ile belirtilen soket adresleri bağlantılar oluşturulacağı zaman kullanılırlar.

Soketin ismi sockaddr_un yapısında belirtilir. Sun_family alanı AF_LOCAL olarak tanımlanır. Bu sabit değer soketin lokal olduğunu gösterir. Sun_path alanı kullanılacak olan dosya ismidir. Sockaddr_un yapısının boyu SUN_LEN makrosu kullanılarak bulunabilir. Herhangi bir dosya ismi soket oluşturmada kullanılabilir. Yalnız sürecin dizine yazma hakkının olması gerekir. Bir sokete bağlanılabilmesi için, sürecin dosya için okuma hakkına sahip olması gerekir. Farklı bilgisayarlardaki süreçler aynı dosya sistemini paylaşsalar dahi, yalnızca aynı makinedeki süreçler lokal namespace soketleri yardımı ile aralarında haberleşebilir. Lokal namespace için kullanılabilecek protokol 0 değeri ile gösterilir.

İki program kullanılarak soket konusuna örnek verilecektir. Sunucu programı lokal namespace soketi oluşturur ve bağlantılar için dinlemeye başlar. Bir bağlantı talebi aldığında, bağlantıdan gelen mesajı okur ve bağlantı kapanıncaya kadar yazar. Eğer gelen mesajlardan biri "quit" is, sunucu programı soketi kaldırır ve bağlantıyı sonlandırır. Socket-server programı soket dosyasının yerini parametre olarak komut satırından almaktadır.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
```

```
/* Read text from the socket and print it out.  Continue until the
   socket closes.  Return non-zero if the client sent a "quit"
   message, zero otherwise.  */

int server (int client_socket)
{
  while (1) {
    int length;
    char* text;

    /* First, read the length of the text message from the socket.  If
       read returns zero, the client closed the connection.  */
    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;
    /* Allocate a buffer to hold the text.  */
    text = (char*) malloc (length);
    /* Read the text itself, and print it.  */
    read (client_socket, text, length);
    printf ("%s\n", text);
    /* Free the buffer.  */
    free (text);
    /* If the client sent the message "quit", we're all done.  */
    if (!strcmp (text, "quit"))
      return 1;
  }
}

int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  int socket_fd;
  struct sockaddr_un name;
  int client_sent_quit_message;

  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Indicate this is a server.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  bind (socket_fd, &name, SUN_LEN (&name));
  /* Listen for connections.  */
  listen (socket_fd, 5);

  /* Repeatedly accept connections, spinning off one server() to deal
     with each client.  Continue until a client sends a "quit" message.  */
  do {
    struct sockaddr_un client_name;
    socklen_t client_name_len;
    int client_socket_fd;

    /* Accept a connection.  */
    client_socket_fd = accept (socket_fd, &client_name, &client_name_len);
    /* Handle the connection.  */
    client_sent_quit_message = server (client_socket_fd);
    /* Close our end of the connection.  */
    close (client_socket_fd);
  }
  while (!client_sent_quit_message);

  /* Remove the socket file.  */
  close (socket_fd);
  unlink (socket_name);

  return 0;
}
```

İstemci programı lokal namespace soketine bağlanır ve mesaj gönderir. Komut satırından soketin yeri ve gönderilecek mesaj belirtilir.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
/* Write TEXT to the socket given by file descriptor SOCKET_FD.  */
void write_text (int socket_fd, const char* text)
```

```
{
  /* Write the number of bytes in the string, including
     NUL-termination.  */
  int length = strlen (text) + 1;
  write (socket_fd, &length, sizeof (length));
  /* Write the string.  */
  write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  const char* const message = argv[2];
  int socket_fd;
  struct sockaddr_un name;

  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Store the server's name in the socket address.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  /* Connect the socket.  */
  connect (socket_fd, &name, SUN_LEN (&name));
  /* Write the text on the command line to the socket.  */
  write_text (socket_fd, message);
  close (socket_fd);
  return 0;
}
```