- Neben *AF_INET* und *AF_INET6* wird durch den POSIX-Standard auch noch *AF_UNIX* genannt.
- *PF_UNIX* bzw. *AF_UNIX* stehen für UNIX-Domain-Sockets.
- Ähnlich zu den Pipes bieten sie eine Interprozess-Kommunikation innerhalb eines Rechners auf Basis der BSD-Socket-Schnittstelle an.
- Anders als Pipes sind sie bidirektional.
- Als Adresse wird ein Dateiname verwendet. Eine Socket-Datei wird durch einen entsprechenden *bind*-Systemaufruf implizit erzeugt. Die Datei wird aber nicht automatisch entfernt, wenn der Dienst endet.
- Das Dateisystem kann hier den Zugriffsschutz übernehmen.

- Für UNIX-Domain-Sockets gibt es aus **#include** <sys/un.h> die Datenstruktur **struct** *sockaddr_un* mit folgenden Komponenten:

sa_family_t sun_family	Adressfamilie, hier immer *AF_UNIX*
char *sun_path*[]	Pfadname der Socket-Datei

- Bei *bind* kann ein Zeiger auf eine entsprechende Datenstruktur übergeben werden.
- Noch einfacher ist es, die Funktion *parse_hostport* entsprechend zu erweitern, so dass auch Pfadnamen unterstützt werden.

hostport.c

```
bool parse_hostport(char* input, hostport* hp, in_port_t defaultport) {
   if (input[0] == '/' || input[0] == '.') {
      /* special case: UNIX domain socket */
      hp->domain = PF_UNIX;
      hp->protocol = 0;
      struct sockaddr_un* sp = (struct sockaddr_un*) &hp->addr;
      sp->sun_family = AF_UNIX;
      strncpy(sp->sun_path, input, sizeof sp->sun_path);
      hp->namelen = sizeof(struct sockaddr_un);
      return true;
   }
   // regular hostports ...
}
```

- Wegen der objekt-orientierten Socket-Schnittstelle genügt eine entsprechende Erweiterung der *parse_hostport*-Funktion, um UNIX-Domain-Sockets zu unterstützen.

```
thales$ ls
MXP             lockmanager.o  mxprequest.c   mxpresponse.h  mxpsession.o
Makefile        mutexd         mxprequest.h   mxpresponse.o
lockmanager.c   mutexd.c       mxprequest.o   mxpsession.c
lockmanager.h   mutexd.o       mxpresponse.c  mxpsession.h
thales$ mutexd ./socket &
[1] 3000
thales$ ls -l socket
srwxrwxr-x 1 borchert sai 0 Jul  6 13:42 socket
thales$ cd ../connect
thales$ connect ../mutexd-multiplexed/socket
S
id Andreas
Swelcome
quit
thales$
```

- Für interaktiv nutzbare Netzwerkdienste stand *telnet* zur Verfügung. Dieser lässt sich aber nicht für UNIX-Domain-Sockets verwenden.
- Entsprechend wird ein verallgemeinerter Ansatz namens *connect* benötigt...

connect.c

```
int main(int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s hostport\n", cmdname);
      exit(1);
   }
   char* hostport_string = *argv++; --argc;
   hostport hp;
   if (!parse_hostport(hostport_string, &hp, 0)) {
      fprintf(stderr, "%s: invalid hostport: %s\n", cmdname,
         hostport_string);
      exit(1);
   }
   int sfd = socket(hp.domain, SOCK_STREAM, hp.protocol);
   if (sfd < 0) {
      perror("socket"); exit(1);
   }
   if (connect(sfd, (struct sockaddr*) &hp.addr, hp.namelen) < 0) {
      perror(hostport_string); exit(1);
   }
   // ...
}
```

connect.c

```
struct pollfd fds[] = {
   {sfd, POLLIN, 0}, /* wait for input from the socket */
   {0, POLLIN, 0}, /* wait for input from stdin */
};
char buf[BUFSIZ];
while (poll(fds, sizeof(fds)/sizeof(fds[0]), -1) > 0) {
   for (int index = 0; index < sizeof(fds)/sizeof(fds[0]); ++index) {
      if (fds[index].revents) {
         ssize_t nbytes = read(fds[index].fd, buf, sizeof buf);
         if (nbytes < 0) { perror("read"); exit(1); }
         if (nbytes == 0) exit(0);
         int outfd = (index == 0? 1: sfd);
         size_t written = 0;
         while (written < nbytes) {
            ssize_t outbytes = write(outfd,
               buf + written, nbytes - written);
            if (outbytes < 0) {
               perror("write"); exit(1);
            }
            written += outbytes;
         }
      }
   }
}
```

Es gibt zahlreiche Techniken zur lokalen Kommunikation und Synchronisation:

- *pipe*: unidirektional, Prozesse müssen miteinander verwandt sein.
- Benannte Pipes: über das Dateisystem, die Pipe-Datei muss explizit angelegt werden.
- UNIX-Domain-Sockets: bidirektional, deutlich einfacher im Vergleich zu benannten Pipes.
- Message Queues mit *msgsnd*, *msgrcv* etc. – sehr unhandlich wie alle System-V-IPC-Mechanismen
- Gemeinsame Speicherbereiche mit *mmap* – da fehlt noch die Synchronisierung...

- Speicherbereiche können auch von nicht miteinander verwandten Prozessen gemeinsam genutzt werden.
- Hierzu genügt es, mit *mmap* eine Datei in den eigenen Speicherbereich abzubilden.
- Vorteil: Das Hin- und Herkopieren kann minimiert werden.
- Nachteile:
  - Die Größe des gemeinsamen Speicherbereichs wird zu Beginn festgelegt. Dieser kann später nicht wachsen.
  - Die Synchronisierung muss auf irgendeine andere Weise erreicht werden.

Zur Synchronisation von Prozessen auf dem gleichen Rechner bieten sich u.a. folgende Techniken an:

- Über das Dateisystem, etwa mit *link* (siehe erstes *mutexd*-Beispiel) oder mit *flock*. Nachteil: Wie warten wir darauf, dass uns der Partner etwas mitgeteilt hat?
- Semaphores aus dem System-V-IPC-Mechanismen (ebenso sehr unhandlich). Nachteil wie oben.
- Andere Kommunikation mit impliziter Synchronisierung
- Mutex- und Bedingungsvariablen der POSIX-Threads-Schnittstelle

Letzteres ist vielleicht überraschend. Interessanterweise können Mutex- und Bedingungsvariablen der POSIX-Threads-Schnittstelle auch von mehreren Prozessen mit Hilfe gemeinsamer Speicherbereiche genutzt werden.

```
#include <pthread.h>
// ...
Mutex* mutex; // zeigt in gemeinsamen Speicher
// ...
pthread_mutexattr_t mxattr;
pthread_mutexattr_init(&mxattr);
CHECK(pthread_mutexattr_setpshared, &mxattr, PTHREAD_PROCESS_SHARED);
CHECK(pthread_mutex_init, mutex, &mxattr);
pthread_mutexattr_destroy(&mxattr);
```

- Mit Hilfe von Mutex-Variablen können mehrere Parteien sichergehen, dass nur ein Prozess zu einer Ressource hat.
- Eine Mutex-Variable wird mit *pthread_mutex_init* initialisiert. *CHECK* ist hier ein Makro, das auf Fehler reagiert.
- Als einziges Attribut wird hier *PTHREAD_PROCESS_SHARED* gesetzt. Dies muss gesetzt sein, wenn die Mutex-Variable von mehreren Prozessen gemeinsam genutzt wird.
- Mit *pthread_mutex_destroy* wird sie wieder freigegeben.

```
pthread_mutex_destroy(mutex);
```

```
CHECK(pthread_mutex_lock, mutex);
```

- Durch den Aufruf von *pthread_mutex_lock* wird der Aufrufer blockiert, bis die Mutex-Variable frei ist.
- Danach ist sie vom Aufrufer belegt, bis sie mit *pthread_mutex_unlock* wieder freigegeben wird.

```
CHECK(pthread_mutex_unlock, mutex);
```

```
pthread_cond_t* condition; // zeigt auf gemeinsamen Speicher

pthread_condattr_t condattr;
pthread_condattr_init(&condattr);
CHECK(pthread_condattr_setpshared, &condattr, PTHREAD_PROCESS_SHARED);
CHECK(pthread_cond_init, condition, &condattr);
pthread_condattr_destroy(&condattr);
```

- Bedingungsvariablen erlauben es, auf ein Ereignis zu warten, das durch eine andere Partei signalisiert wird.
- Hier ist ebenso bei der Initialisierung wichtig, dass das Attribut *PTHREAD_PROCESS_SHARED* gesetzt wird.
- Die Freigabe erfolgt mit *pthread_cond_destroy*.

```
pthread_cond_destroy(condition);
```

```
CHECK(pthread_cond_wait, condition, mutex);
```

- Die Operation *pthread_cond_wait* erfolgt immer in Verbindung mit einer Mutex-Variablen, die bereits mit *pthread_mutex_lock* reserviert sein muss.
- In einer atomaren Operation wird dann die Mutex-Variable freigegeben und der aufrufende Prozess (bzw. Thread) in die zugehörige Warteschlange eingereiht.
- Mit *pthread_cond_signal* weckt einen Prozess aus der Warteschlange auf (falls vorhanden). Alternativ gibt es auch die Operation *pthread_cond_broadcast*, mit der alle aus der Warteschlange aufgeweckt werden.

```
CHECK(pthread_cond_signal, condition);
```