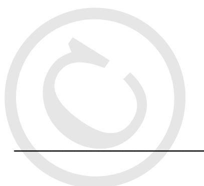
Student-notities

Linux/UNIX system calls in C

Voorbeeld-hoofdstuk

10. Introductie sockets

Nijmegen



Copyright © AT Computing 1997, 2001, 2003
Versie: 4a

Student-notities

■ De student-notities in dit voorbeeld-hoofdstuk zijn fragmenten uit het dictaat dat bij deze cursus wordt meegeleverd. □

Het maken van een pipe gebeurt via de system call `pipe` waarmee in één klap het hele verbindingspad wordt gecreëerd: zowel de beide uiteinden als de eigenlijke verbinding ertussen. Deze verbinding wordt dan door de kernel achter de schermen gerealiseerd met alle eigenschappen zoals we die van pipes kennen: first-in first-out met een maximale inhoud.
Omdat de system call `pipe` beide uiteinden (de filedescriptors) in één keer aflevert en de open filedescriptors alleen via de system calls `fork` en `exec` van het ene proces aan het andere kunnen worden doorgegeven, is het dus uitsluitend mogelijk een anonieme pipe te leggen tussen twee processen die een familierelatie met elkaar hebben.

Bij het socket-mechanisme heeft men de mogelijkheid om I/O-kanalen tot stand te brengen tussen *willekeurige processen*, die dus geen afstammingsrelatie hoeven te hebben, en die zelfs niet op hetzelfde systeem hoeven te leven. Teneinde dit te realiseren, heeft men het creëren van een verbinding uiteengerafeld in drie deelstappen: men maakt nu niet in één keer een “kabel” met twee contactstekers (zoals bij de `pipe` system call), maar elk van de twee contactpunten onafhankelijk van elkaar (in elk van de processen één).

Later wordt dan een verzoek aan de kernel gericht om twee van dergelijke 'stekers' (*sockets* genaamd) te verbinden. Daartoe is het dan wel nodig om een dergelijk contactpunt te kunnen aanwijzen vanuit een willekeurig proces (en vanuit een willekeurig systeem). Vandaar dat er als extra eigenschap aan de contactpunten een naam (adres) kan en moet worden gegeven. Bovendien moet aan beide kanten van de verbinding — dus in de processen die beide sockets opzetten — nog een stap gedaan worden om de verbinding ook echt te leggen: het aanvragen respectievelijk het aannemen van de verbinding.
De op een dergelijke wijze in afzonderlijke stappen gecreëerde verbinding lijkt heel veel op wat we vanouds kennen als een pipe: de twee uiteinden worden elk benoemd door een filedescriptor, en het verkeer gaat volgens het first-in first-out mechanisme. Maar er zijn ook verschillen. Zoals gezegd, het dataverkeer kan tussen twee computers over een netwerk plaatsvinden; bovendien is elk van beide filedescriptors zowel te gebruiken voor lezen als voor schrijven, er zijn dus als het ware twee pipes, één in elke richting. Met andere woorden: sockets zijn bidirectioneel.

© AT Computing — The Linux/UNIX experts

Procescommunicatie via sockets

Communicatie tussen willekeurige processen

Netwerkwijd

Aparte system calls voor:
- maken van een ‘eindpunt’ (socket)
- leggen van een verbinding tussen twee eindpunten

vergelijk: `pipe()` system call in één klap

Sockets lijken op pipes:
- eindpunten zijn filedescriptors
- first-in-first-out dataverkeer

Sockets zijn bidirectioneel

Laag bovenop netwerkprotocollen;
meestal TCP/IP, andere protocollen ook mogelijk

Zeer geschikt voor client-server toepassingen

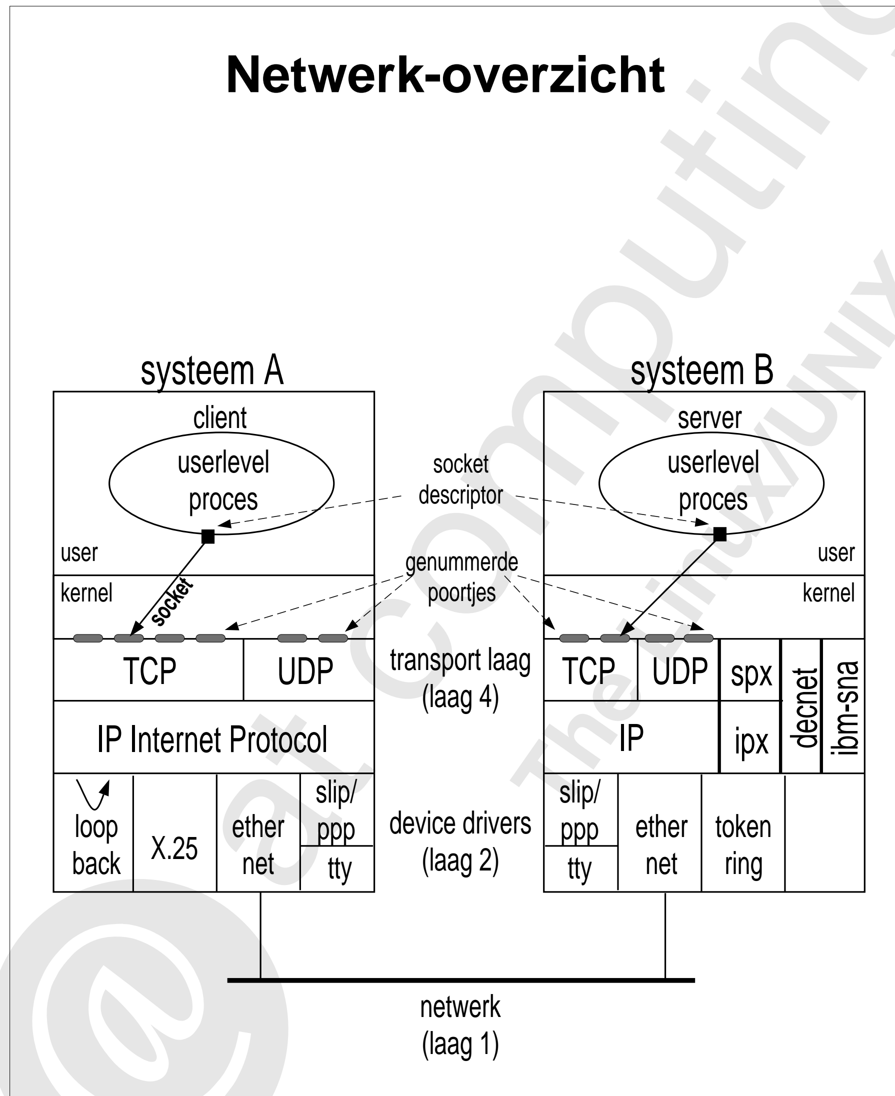
Figuur 1.

@ computing The Linux/UNIXperts

Student-notities

In dit overzichtplaatje zie je dat de genummerde TCP- en UDP-poortjes zich aan de bovenzijde van de betreffende transportlaag bevinden. Zo'n poortje wordt geopende door een lokaal proces; dit levert een filedescriptor af, waarmee dat proces vervolgoperaties kan uitvoeren. Zo'n koppeling tussen proces en lokaal poortje wordt een *socket* genoemd.
Nadat beide processen een lokaal poortje geopend hebben, ligt er nog steeds geen verbinding tussen de twee poortjes. Een verbinding wordt altijd gelegd tussen een *client*-proces (verlangt een bepaalde dienst) en een *server*-proces (levert een bepaalde dienst). Het client-proces neemt het initiatief om het poortje van het server-proces op te zoeken en zodoende de verbinding tussen de poortjes te leggen. Daarvoor moet de client wel weten achter welk poortnummertje de server zich heeft verschanst.

© @ at computing The Linux/UNIXers



Figuur 2.

© @ at computing The Linux/UNIXperts

Student-notities

Er zijn verschillende soorten sockets. De belangrijkste zijn *stream* sockets (de TCP transport-laag bij TCP/IP) en *datagram* sockets (de UDP transport-laag bij TCP/IP).

Sockets van type stream

Een socketverbinding van het type *stream* heeft de volgende eigenschappen:

- Tussen beide sockets moet een verbinding opgebouwd worden alvorens er data getransporteerd kan worden.
- Het is byte-serieel transport.
- Boodschappen worden gegarandeerd bij de geadresseerde afgeleverd; er raken geen boodschappen kwijt zonder kennisgeving.
- Boodschappen worden foutvrij bij de geadresseerde afgeleverd: de inhoud van de boodschap is gegarandeerd correct.
- Boodschappen worden in dezelfde volgorde afgeleverd als waarin ze verzonden werden.
- Boodschappen worden onderweg nergens verdubbeld: eenmaal versturen betekent ook eenmaal ontvangen.
- De grenzen tussen verschillende boodschappen (tussen verschillende `write` system calls) zijn voor de ontvanger niet zichtbaar. Deze eigenschap is bekend van de klassieke pipes.
- Een socket is bidirectioneel: op elk van beide sockets kan zowel geschreven als gelezen worden, er is dus in wezen in ieder van beide richtingen onafhankelijk van elkaar datatransport mogelijk ("full-duplex").

Sockets van type datagram

Een socketverbinding van het type *datagram* heeft de volgende eigenschappen:

- Het is niet noodzakelijk dat er een vaste koppeling bestaat tussen de beide sockets. Het is mogelijk om bij het versturen van datagrammen vanaf een bepaald socket voor ieder afzonderlijk datagram op te geven aan welk ander socket de boodschap gericht moet worden: datagram sockets zijn niet "connection oriented".
- Ook deze sockets zijn bidirectioneel, full-duplex.
- De grenzen tussen verschillende boodschappen zijn voor de ontvanger
- Een datagram socket is niet betrouwbaar: volgorde van ontvangst is niet gegarandeerd, foutvrije aankomst is niet gegarandeerd, meervoudige aankomst van een datagram is niet uitgesloten.

© AT Computing
The Linux/UNIXperts

Stream en datagram sockets

Karakteristieken *stream* sockets (TCP):

- Connection oriented: eerst verbinding opzetten, dan datatransport
- Boodschappen worden gegarandeerd afgeleverd
- Boodschappen worden niet verminkt
- Boodschappen komen in goede volgorde aan
- Boodschappen worden niet verdubbeld
- De grenzen tussen boodschappen zijn niet zichtbaar voor lezer
- Socket-type: `SOCK_STREAM`

Karakteristieken *datagram* sockets (UDP):

- Connectionless: verbinding hoeft niet opgebouwd te worden
- Boodschappen kunnen verloren gaan, in de verkeerde volgorde aankomen of verdubbeld worden
- De grenzen tussen boodschappen zijn zichtbaar voor lezer
- Socket-type: `SOCK_DGRAM`

Figuur 3.

© AT Computing The Linux/UNIXperts

Student-notities

De system calls die zijn gedefinieerd voor de afzonderlijke stappen in het opbouwen van een verbinding tussen twee sockets zijn:

`socket` Voor het creëren van een uiteinde: er wordt een *socket-descriptor* (in de vorm van een filedescriptor) geretourneerd. Deze system call moet zowel door het server-proces alsook het client-proces uitgevoerd worden.

`bind` Voor het geven van een specifieke naam (een poortnummer en eventueel IP-adres) aan een socket.

`listen` Om aan de kernel kenbaar te maken dat naar dit socket een verbinding gelegd mag worden vanuit een ander socket.

`connect` Voor het leggen van een verbinding tussen een eigen socket (typisch van een client) en een ander socket (van de server).

`accept` Om een verbinding aan te nemen die vanuit een ander socket met system call `connect` wordt aangevraagd, zodat er daadwerkelijk datatransport kan gaan plaatsvinden tussen een eigen `listen`-ing socket en een `connect`-ing socket in een ander proces.

`read/write` Hierna kan het feitelijke datatransport plaatsvinden met behulp van de normale system calls `read` en `write`. Daarbij worden dan de socket-descriptors als gewone filedescriptors gebruikt.

`close` Hierbij kan één van beide partijen de verbinding weer verbreken.

De hier geschetste stappen geven één methode weer voor het opbouwen van een verbinding tussen twee sockets; er zijn nog andere mogelijkheden en er zijn nog meer system calls die in samenhang met sockets gebruikt kunnen worden, zoals we later zullen zien.

© AT Computing — The Linux/UNIX experts

Overzicht socket system calls

socket()

Creëer een eindpunt

connect()

Zet verbinding op naar ander socket met specifiek poortnummer

bind()

Geef specifiek poortnummer aan socket, zodat partner verbinding hiernaar kan opzetten

listen()

Maak kenbaar dat een `connect` gedaan mag worden vanaf ander socket (jargon: maak socket *passief*)

accept()

Accepteer `connect` vanuit een ander socket

read() en **write()**

Datatransport na opbouwfase

Figuur 4.

© at computing The Linux/UNIXperts

Student-notities

Om tussen twee processen een verbinding tot stand te brengen zullen beide processen natuurlijk als eerste stap een socket van hetzelfde socket-type moeten creëren met de `socket` system call. Daarna moet elk van beide processen een bepaalde volgorde van stappen doorlopen om uiteindelijk tussen de beide sockets een verbinding te leggen. Eén van beide partijen moet zich passief opstellen (deze partij wordt de *server* genoemd), en is daarme te vergelijken met een winkelier die zijn winkeldeur heeft geopend en op klanten staat te wachten. De actieve partij (de *client*) is degene die de verbinding naar de server opbouwt.

Wil de actieve partij een verbinding kunnen opbouwen, dan zal het passieve socket aanwijsbaar moeten zijn. Dat betekent dat de eigenaar van het passieve socket (het server-proces) voor dit socket een specifiek adres moet uitkiezen en dit moet koppelen aan het socket. Dit gebeurt met de `bind` system call.

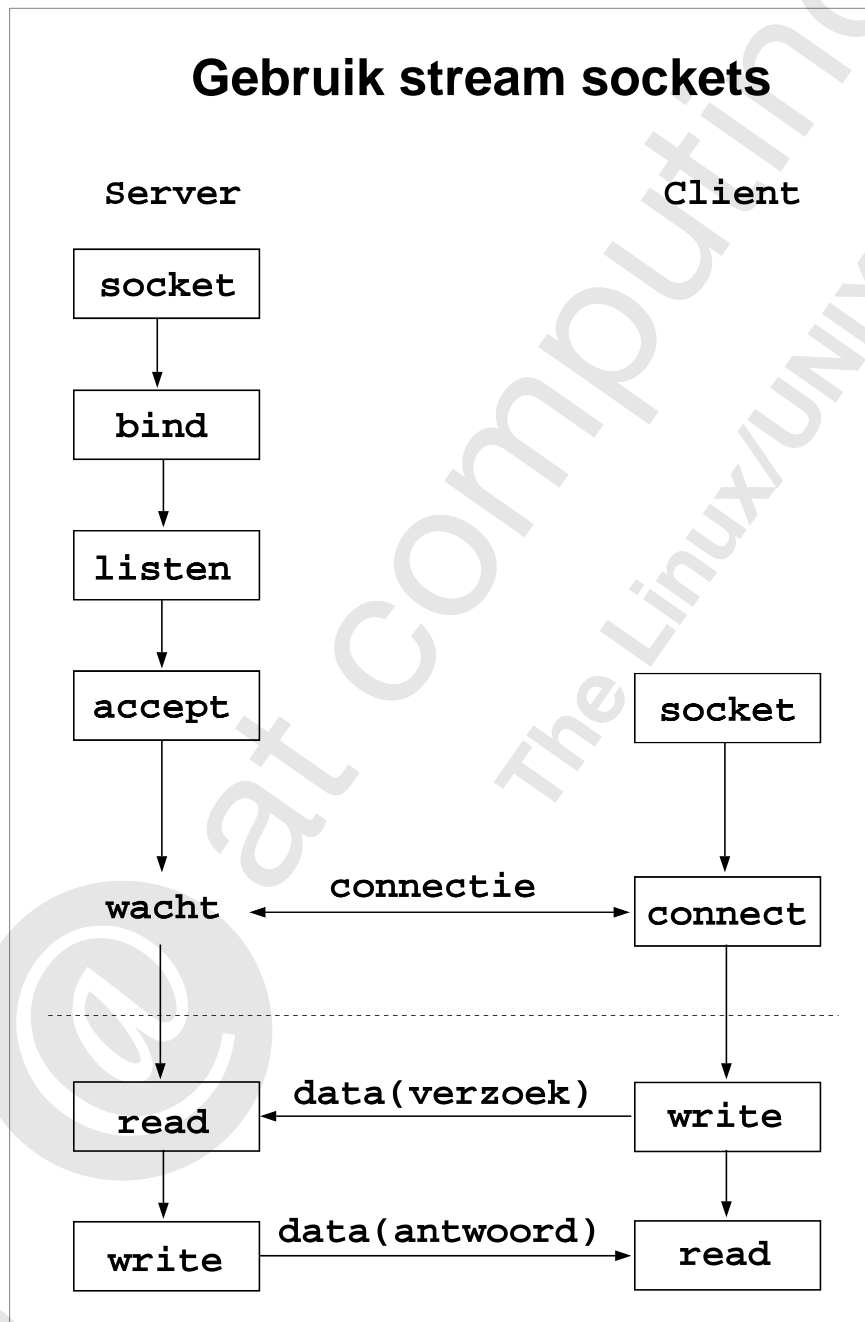
Op een of andere manier moet de eigenaar van het actieve socket (het client-proces) dan kennis hebben van het adres waar het verbinding mee wil zoeken. Dat adres kan op verschillende manieren worden doorgegeven, bijvoorbeeld bij het opstarten van het client-programma als commandoregel-argument. Maar een betere strategie is dat programma's die een bepaalde service bieden (de servers) een algemeen bekend adres — een zogenaamd *well-known port number* — op hun passieve socket plakken, zodat dat adres ofwel (hard ingecodeerd) al bekend is bij de client) ofwel door die client kan worden opgezocht (meestal in de file `/etc/services`).

De passieve partij moet na het stempelen van een adres op haar socket nog wel de bereidheid kenbaar maken om op dat socket verbindingen te accepteren. Dat heet het "passief maken" van het socket. Dit wordt gedaan met een aanroep van de `listen` system call.

Als dat is gebeurd, kan de actieve partij (client) het initiatief nemen om een verbinding op te bouwen. Het actieve programma doet dat met de `connect` system call. Deze system call heeft het adres van het socket waarmee een verbinding gemaakt moet worden als één van de argumenten.

De passieve partij zal ondertussen een system call aangeroepen hebben waarmee gewacht wordt tot er een verzoek voor een verbinding op het luisterende socket arriveert. Dit gebeurt met de `accept` system call.

Als bovenstaande serie handelingen verricht is kan het feitelijke datatransport beginnen met gewone `read` en `write` system calls.



Figuur 5.

© @ at computing The Linux/UNIXperts

Student-notities

De eigenschappen van het uiteindelijke dataverkeer worden niet alleen bepaald door het socket-*type* (stream of datagram), maar ook door het soort verbinding dat tussen de beide sockets wordt gelegd. Daarbij gaat de keuze in de eerste plaats tussen socket-verbindingen die alleen maar binnen één en hetzelfde UNIX systeem worden gerealiseerd, en verbindingen tussen verschillende computers. In dat laatste geval kun je kiezen tussen verschillende soorten netwerken en/of verschillende protocollen op een netwerk. Reeds bij het creëren van een socket moet worden vastgelegd voor welk soort verbinding dat socket gebruikt gaat worden. Er moet op dat moment worden gespecificeerd in welk *domein* het socket zijn leven gaat leiden. Een domein wordt ook wel een *protocol-familie* of een *adres-familie* genoemd. Afhankelijk van het type UNIX kan er — naast TCP/IP IPv4 en IPv6 — ondersteuning zijn voor andere domeinen zoals XEROX NS, DECnet, Appletalk, IBM-SNA, Novell SPX/IPX, of andere.

Het domein bepaalt ook de vorm die socket-adressen hebben. Voor ieder domein dat ondersteund wordt, is er een apart type structuur, en bovendien is er een overkoepelend (generiek) structuur-type. Alle *socket address structures* hebben als eerste member een `short int` die aangeeft om welk type structuur het gaat, met andere woorden in welk domein (adres-familie) dit socket-adres geldig is. De overige member(s) verschillen per adres-familie. Het generieke structuur-type is als volgt gedefinieerd:

```
struct sockaddr  {
   short sa_family;    /* address family: AF_iets      */
   char  sa_data[14];  /* max. 14 bytes address data, */
                       /* domein afhankelijk           */
};
```

Het eerste member is een `short int` dat aangeeft in welk domein dit socket-adres geldig is. Het tweede member is een reservering voor een hoeveelheid bytes.
Als je socket-adressen van het type `struct sockaddr_in` nodig hebt, moet je de volgende include-file opnemen:

```
#include <netinet/in.h>
```

Het poortnummer en IP-adres in de `struct sockaddr_in` moet door de programmeur gevuld worden in *network byte order* (dat is volgens afspraak *Big Endian*). Om te zorgen dat deze waarden ook op een Little Endian machine op de juiste wijze worden ingevuld, is altijd een set macro's beschikbaar die zorg draagt voor conversie (indien nodig). Er kan in twee richtingen geconverteerd worden voor long int's (letter `l`) en short int's (letter `s`). De letter `n` staat voor network staat en `h` voor host:

```
#include <sys/types.h>
#include <netinet/in.h>

u_long   htonl(u_long  hostlong);
u_short  htons(u_short hostshort);
u_long   ntohl(u_long  netlong);
u_short  ntohs(u_short netshort);
```

Socket domeinen en adressen

Een *socket domein* bestaat uit:

- netwerk-protocol familie
- socket adres-layout

Generieke adres-layout:

```
struct sockaddr {
  short sa_family;
  char  sa_data[...];
};
```

TCP/IP adres-layout (IPv4):

```
struct sockaddr_in {
  short          sin_family;
  u_short        sin_port;
  struct in_addr sin_addr;
  char           sin_zero[8];
};
```

sin_family de adresfamilie; in dit geval **AF_INET**

sin_port *poortnummer*: identificatie van socket binnen host (network byte order)

sin_addr internet host adres (network byte order)

Figuur 6.

© AT Computing - the Linux/UNIXperts

Student-notities

De creatie van een socket vindt plaats via de `socket` system call.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domein, int type, int protocol);
```

Het `domein` geeft aan in welk domein het gecreëerde socket moet leven. Je kunt voor het domein ondermeer één van de symbolische constanten `AF_INET` (IP versie 4) en `AF_INET6` (IP versie 6) opgeven.

Met het argument `type` geef je aan wat voor type socket je wenst. Je kunt hier één van de symbolische constanten `SOCK_STREAM`, `SOCK_DGRAM`, of `SOCK_RAW` opgeven.

Met het argument `protocol` geef je een bij het socket-type passend protocol op waarvoor binnen het domein ondersteuning bestaat. Meestal is hier geen keuze: in de praktijk wordt meestal `0` (het default protocol) ingevuld.
In het internet-domein bijvoorbeeld is het default protocol voor stream sockets TCP (Transport Control Protocol), en voor datagram sockets UDP (User Datagram Protocol). Beide gebruiken als onderliggend protocol IP (Internet Protocol).

De returnwaarde van de `socket` system call is een *socket-descriptor*. Een socket-descriptor dient ter identificatie van het socket en gedraagt zich precies zoals een filedescriptor.
De returnwaarde van de `socket` system call is -1 bij mislukken.

© AT Computing The Linux/UNIX experts

`socket()` system call

Creëer lokaal eindpunt (socket)

```
sock = socket(domein, type, protocol);
```

`domein` Gebruikt domein
Mogelijkheden o.a.:

`AF_INET`	Internet (IPv4)
`AF_INET6`	Internet (IPv6)
`AF_DECnet`	DECNET
`AF_APPLETALK`	APPLETALK
	....

`type` Socket-type
Belangrijkste mogelijkheden:
- **`SOCK_STREAM`** (Internet: TCP)
- **`SOCK_DGRAM`** (Internet: UDP)

`protocol` Protocol; 0 is default protocol

`sock` Socket-descriptor (filedescriptor) voor identificatie van het socket

Figuur 7.

© @ at computing The Linux/UNIXperts

Student-notities

De system call `bind` wordt gebruikt voor het toekennen van een adres aan een socket.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sock, struct sockaddr *naam, int namelen);
```

Hierbij is `sock` een socket-descriptor: de socket-descriptor van het socket dat een specifiek adres moet krijgen.

Het argument `naam` is een pointer naar een structuur die het adres specificeert dat op het socket moet worden gestempeld. Dit adres moet genoteerd zijn in de vorm die hoort bij het domein van het socket `sock`, dus bijvoorbeeld één van de mogelijke soorten socket address-structures `sockaddr_in` (IP versie 4) of `sockaddr_in6` (IP versie 6).

Een internet-socket dat passief gemaakt gaat worden, kan voor het `sockaddr_in` member `sin_addr` de waarde `INADDR_ANY` krijgen, waardoor verbindingen worden aangenomen die gericht zijn aan dit poortnummer op deze machine, onafhankelijk van het netwerkadres waarmee deze machine vanaf het andere socket geadresseerd werd (een machine kan op meer dan één netwerk zijn aangesloten, of zelfs op één netwerk meerdere adressen hebben).

Voor een internet-socket dat actief verbinding gaat maken met een ander socket kan als adres ook `INADDR_ANY` opgegeven worden. In dat geval betekent dit "deze machine". Dezelfde symbolische constante kan gebruikt worden in een system call `connect` met als betekenis de "eigen machine".
Een poortnummer (member `sin_port` in de `sockaddr_in` structuur) met waarde `0` betekent dat het systeem met deze `bind` system call expliciet wordt gevraagd om zelf een portnummer te verzinnen en te koppelen aan dit socket.

Het argument `namelen` is een input parameter: hierin wordt de system call `bind` verteld hoe lang de struct is waarnaar gewezen wordt door het tweede argument. Hier staat dus bijvoorbeeld `sizeof(struct sockaddr_in)`

De returnwaarde van de `bind` system call is `0` als het koppelen van het adres aan het socket slaagt, en -1 in geval van een fout.

AT Computing The Linux/UNIX Xperts

`bind()` system call

Koppel een naam (specifiek poort#) aan socket

```
#include <sys/types.h>
#include <sys/socket.h>

int             sock;
size_t          namelen;
struct sockaddr name;

 ....

bind(sock, &name, namelen);
```

`sock`	Socket-descriptor van socket
`name`	Adres dat aan socket gekoppeld moet worden
`namelen`	Grootte van de structure **`name`**

Poort toegankelijk via *alle* lokale IP-nummers:

```
struct sockaddr_in name;

name.sin_family      = AF_INET;
name.sin_addr.s_addr = htonl(INADDR_ANY);
name.sin_port        = htons(...);
```

Figuur 8.

© @ computing The Linux/UNIXperts

Student-notities

De system call listen is alleen van toepassing op sockets die als type (tweede argument bij socket) SOCK_STREAM hebben. De bedoeling van de system call listen is om een dergelijk socket passief te maken, dat wil zeggen de bereidheid kenbaar te maken om verbindingen vanuit een ander socket (door middel van een connect) aan te nemen.

```
int listen(int sock, int maxq);
```

Het eerste argument sock is het socket waarop verbindingen aangenomen zullen worden.

Het tweede argument maxq is de maximale hoeveelheid op enig moment nog niet verwerkte connect requests: het bouwt in de kernel aan de server-zijde een soort wachtkamer voor binnengekomen verzoeken waarvoor de server nog geen tijd heeft gehad om ze aan te nemen (met system call accept). In de praktijk wordt hier minstens de waarde 5 gekozen. Staan er op zeker moment maxq connect requests open, dan krijgt een client-programma dat een connect naar dit socket probeert uit te voeren de waarde -1 terug uit de system call connect en bevat errno de waarde ECONNREFUSED.

De returnwaarde van de system call listen is 0 als het gelukt is en -1 in geval van fouten.

© @ at computing The Linux/UNIXperts

`listen()` system call

Maak stream-socket 'passief'

```
listen(sock, maxq);
```

`sock` Socket-descriptor
`maxq` Maximaal aantal uitstaande nog niet geaccepteerde **`connect`**'s
Vuistregel: vul minimaal 5 in

Figuur 9.

© @ at Computing
The Linux/UNIXperts

Student-notities

Vanaf een actief (dus niet via `listen` passief gemaakt) socket kan een verbinding aangevraagd worden naar een passief socket. Dit geldt voor sockets van het type `SOCK_STREAM`. Een stream socket kan maar voor één verbinding worden gebruikt, dus voor iedere andere verbinding is een nieuw stream socket nodig.

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sock, struct sockaddr *naam, int namelen);
```

Hierbij is `sock` de socket-descriptor van het eigen socket van waaruit de verbinding moet worden gelegd.

Het tweede argument `naam` is een input parameter: een pointer naar een struct (van het juiste type) waarin het adres wordt aangereikt van het socket waarmee de verbinding tot stand gebracht moet worden.
Als in deze structuur de constante `INADDR_ANY` wordt ingevuld voor het member `sin_addr` betekent dit dat het adres waarmee de verbinding tot stand gebracht moet worden op de eigen machine ligt.

Het argument `namelen` is een input parameter: de lengte van de structuur waarnaar gewezen wordt door member `naam`.

De returnwaarde van de system call `connect` is `0` als de verbinding tot stand gekomen is (voor stream sockets), en -1 in het geval van fouten.

© @ computing The Linux/UNIXperts

`connect()` system call

Verbinding aanvragen naar een passief socket

```
#include <sys/types.h>
#include <sys/socket.h>

int             sock;
size_t          namelen;
struct sockaddr name;

....

connect(sock, &name, namelen);
```

sock	Socket-descriptor van actieve socket
name	Adres van passieve server-socket
namelen	Lengte van de structure **name**

Verbinding naar socket op *lokale* host:

```
struct sockaddr_in name;

name.sin_family    = AF_INET;
name.sin_addr.s_addr =
                     htonl(INADDR_LOOPBACK);
name.sin_port      = htons(...);
```

Figuur 10.

© at computing The Linux/UNIXperts

Student-notities

Een accept system call mag alleen maar worden uitgevoerd voor een passief socket (dus na een voorafgaande listen). De system call accept blokkeert totdat een client een system call connect doet.

Wanneer — nadat de server een accept en de client een connect heeft uitgevoerd — een socketverbinding is opgezet, zal voor deze verbinding en het feitelijke datatransport geen gebruik gemaakt worden van het passieve ("listen") socket. Er wordt een *nieuwe socket-descriptor* geretourneerd door de system call accept, en het is deze nieuwe socket-descriptor die daarna door de server gebruikt zal worden voor het datatransport met de specifieke klant.

Bij connection-oriented sockets is er bij een socket maar één verbinding met één ander socket: een socket dat een verbinding heeft, heeft exact één *peer* (evenknie) socket. Het passieve (listening) socket kan meerdere verbindingen aannemen via evenzovele accept system calls, maar krijgt zelf nooit een "peer".

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sock, struct sockaddr *naam, int *namelen);
```

Hierbij is sock de socket-descriptor van het luisterende socket.

Argument naam is een result-parameter: een pointer naar een struct waarin het adres wordt geretourneerd van het socket waarmee de verbinding tot stand is gebracht.

Het derde argument namelen is zowel een input als een result parameter. Bij aanroep moet de integer waarnaar verwezen wordt door namelen als waarde de lengte van de structuur waarnaar gewezen wordt door naam bevatten, en bij terugkeer heeft de integer waarnaar gewezen wordt door namelen als waarde de actuele lengte van die structuur: deze hangt af van degene die de connect uitvoerde.

De returnwaarde van de accept system call is een socket-descriptor van een nieuw verbonden socket, of -1 in geval van een fout. Bij non-blocking I/O zal, als er nog geen connect uitstaat, de returnwaarde -1 zijn met daarbij in errno de waarde EWOULDBLOCK.

Samengevat wordt de accept system call als volgt aangeroepen:

```
int sock, newsock, namelen;
struct sockaddr name;

 .....

newsock = accept(sock, &name, &namelen);
```

waarna voor de feitelijke verbinding *newsock* wordt gebruikt.

© AT Computing The Linux/UNIX experts

`accept()` system call

Neem binnenkomende **`connect`** aan

```
#include <sys/types.h>
#include <sys/socket.h>

int             sock, newsock;
size_t          namelen;
struct sockaddr name;

   .....

newsock = accept(sock, &name, &namelen);
```

`sock` Socket-descriptor passieve socket

`name` Gevuld met adres van 'peer' socket

`namelen` Grootte van structure **`name`**;
- voor aanroep: maximale grootte
- na aanroep: werkelijke grootte

`newsock` Socket-descriptor van nieuw socket
- datatransport verloopt over **`newsock`**
- **`sock`** kan in gebruik blijven voor een nieuwe **`accept()`**

Figuur 11.

@ at computing The Linux/UNIXperts

Student-notities

Een server-proces kan ervoor kiezen om iedere binnengekomen verbinding (`connect`) aan te pakken en zelf met de betreffende klant te gaan communiceren. Om te zorgen dat je dan met meerdere klanten tegelijk kunt communiceren — en ondertussen ook nog ontvankelijk bent voor nieuw binnenkomende verbindingen — zul je gebruik moeten maken van de system call `select` (wordt later in deze cursus behandeld).

Een andere (eenvoudiger) mogelijkheid is een *concurrent* server-model. Hierbij beperkt het server-proces zich tot het aannemen van nieuwe verbindingen, zoals een winkelier bij zijn winkeldeur staat om nieuwe klanten te verwelkomen. Voor iedere nieuwe verbinding (waarvoor een nieuwe actieve socket-descriptor wordt geretourneerd door de system call `accept`) maakt het server-proces een kind-proces; dit kind-proces verzorgt de communicatie met díe specifieke client, terwijl het server-proces zelf (de ouder) weer wacht op een nieuwe verbinding. Het kind-proces sterft af als de communicatie met de client wordt beëindigd.

Bij dit model moet je wel zorgen dat er geen open socket-descriptors rond blijven zwerven. Tijdens de system call `fork` worden *alle* open descriptors doorgegeven van het ouder-proces naar het kind-proces. Dat betekent dat het kind-proces ook de passieve descriptor erft; deze descriptor heeft het kind-proces niet nodig en moet daarom zo snel mogelijk worden gesloten.
Het ouder-proces daarentegen heeft de actieve descriptor niet meer nodig, nadat deze doorgegeven is aan het kind-proces.

© at computing The Linux/UNIX experts

@ Computing
The Linux/UNIX experts

Voorbeeld: concurrent server

Server-proces kan zelf verbinding afhandelen
òf
per verbinding een kindproces creëren

```
passock = socket(...);
bind(passock, serveradr, ...);
listen(passock, 5);

/* voorkom zombie-procesen */
signal(SIGCHLD, SIG_IGN);

for (ever)
{
  actsock=accept(passock, &clientadr, .);

  if ( fork() ) {
     close(actsock);      /* ouder */
  } else {
     close(passock);      /* kind  */
     read(actsock, reqbuf, sz);
     .....
     write(actsock, respbuf, sz);
     close(actsock);
     exit(0);
  }
}
```

Figuur 12.

Student-notities

In het resterende deel van dit hoofdstuk wordt de generieke functie `getaddrinfo()` uitgebreid behandeld. Met deze functie kan een host-naam en/of service-naam worden vertaald naar een `sockaddr`-structure, zowel voor IP versie 4 als voor IP versie 6.

Verder komt het gebruik van datagram sockets aan de orde (UDP) met de specifieke system calls die daarbij nodig zijn.

© @ at computing The Linux/UNIXperts