Network Programming

Lecture 6—Advanced Sockets II: Routing Sockets, Key Management Sockets, and Threads

Lei Wang
lei.wang@dlut.edu.cn

Dalian University of Technology

Dec 22, 2008

Part 3. Advanced Sockets II: Routing Sockets, Key Management Sockets, and Threads

Chapter 18: Routing Sockets

Routing Sockets Introduction

`AF_ROUTE` domain: The only type of socket is **raw socket**. Three types of operations supported on a routing socket:

- A process can send a message to the kernel by writing to a routing socket. (e.g. add and delete route)
- A process can read a message from the kernel on a routing socket. (e.g. kernel notify a process that an ICMP redirect has been received)
- A process can use the `sysctl` function to either dump the routing table or list all configured interfaces.

Datalink Socket Address Structures

```
struct sockaddr_dl {
  uint8_t      sdl_len;
  sa_family_t  sdl_family;   /* AF_LINK */
  uint16_t     sdl_index;    /* system assigned index, if > 0 */
  uint8_t      sdl_type;     /* IFT_ETHER, etc. from <net/if_types.h> */
  uint8_t      sdl_nlen;     /* name length, starting in sdl_data[0] */
  uint8_t      sdl_alen;     /* link-layer address length */
  uint8_t      sdl_slen;     /* link-layer selector length */
  char         sdl_data[12]; /* minimum work area, can be larger;
                                contains i/f name and link-layer address */
};
```

Reading and Writing

Message type	To kernel?	From kernel?	Description	Structure type
RTM_ADD	•	•	Add route	rt_msghdr
RTM_CHANGE	•	•	Change gateway, metrics, or flags	rt_msghdr
RTM_DELADDR		•	Address being removed from interface	ifa_msghdr
RTM_DELETE	•	•	Delete route	rt_msghdr
RTM_DELMADDR		•	Multicast address being removed from interface	ifma_msghdr
RTM_GET	•	•	Report metrics and other route information	rt_msghdr
RTM_IFANNOUNCE		•	Interface being added or removed from system	if_announcemsghdr
RTM_IFINFO		•	Interface going up, down, etc.	if_msghdr
RTM_LOCK	•	•	Lock given metrics	rt_msghdr
RTM_LOSING		•	Kernel suspects route is failing	rt_msghdr
RTM_MISS		•	Lookup failed on this address	rt_msghdr
RTM_NEWADDR		•	Address being added to interface	ifa_msghdr
RTM_NEWMADDR		•	Multicast address being joined on interface	ifma_msghdr
RTM_REDIRECT		•	Kernel told to use different route	rt_msghdr
RTM_RESOLVE		•	Request to resolve destination to link-layer address	rt_msghdr

Figure: 18.2 Types of messages exchanged across a routing socket.

Example: Fetch and Print a Routing Table Entry

- `route/getrt.c`

Chapter 19: Key Management Sockets

Key Management Sockets Introduction

IPsec needs a mechanism to manage secret encryption and authorization keys. RFC 2367 introduces a generic key management API for IPsec and other security services.

- New protocol family: `PF_KEY`: only type of socket supported in this domain is a raw socket.
- Superuser privilege required.
- SA (Security Association) and SADB (Security Association database)
- More than one SA can apply to a single stream of traffic.
- SADB may be used for more than just IPsec; e.g. OSPFv2, RIPv2, so `PF_KEY` sockets are not specific to IPsec.

Key Management Message Header

```
struct sadb_msg {
 u_int8_t sadb_msg_version;   /* PF_KEY_V2 */
 u_int8_t sadb_msg_type;      /* see Figure 19.2 */
 u_int8_t sadb_msg_errno;     /* error indication */
 u_int8_t sadb_msg_satype;    /* see Figure 19.3 */
 u_int16_t sadb_msg_len;      /* length of header + extensions / 8 */
 u_int16_t sadb_msg_reserved; /* 0 on transmit, ignored on receive */
 u_int32_t sadb_msg_seq;      /* sequence number */
 u_int32_t sadb_msg_pid;      /* process ID of source or dest */
};
```

Type of Messages

Message type	To kernel?	From kernel?	Description
SADB_ACQUIRE	•	•	Request creation of an SADB entry
SADB_ADD	•	•	Add a complete security database entry
SADB_DELETE	•	•	Delete an entry
SADB_DUMP	•	•	Dump the SADB (debugging)
SADB_EXPIRE		•	Notify of expiration of an entry
SADB_FLUSH	•	•	Flush the entire database
SADB_GET	•	•	Get an entry
SADB_GETSPI	•	•	Allocate an SPI to create an SADB entry
SADB_REGISTER	•		Register as a replier to SADB_ACQUIRE
SADB_UPDATE	•	•	Update a partial SADB entry

Figure: 19.2 Types of messages exchanged across a PF_KEY socket

Types of SAs

Security Association Type	Description
SADB_SATYPE_AH	IPsec authentication header
SADB_SATYPE_ESP	IPsec encapsulating security payload
SADB_SATYPE_MIP	Mobile IP authentication
SADB_SATYPE_OSPFV2	OSPFv2 authentication
SADB_SATYPE_RIPV2	RIPv2 authentication
SADB_SATYPE_RSVP	RSVP authentication
SADB_SATYPE_UNSPECIFIED	Unspecified; only valid in requests

Figure: 19.3 Types of SAs

Example: Dumping the Security Association Database

- `key/dump.c`

Creating a Static Security Association

SA Extensions:

```
struct sadb_sa {
  u_int16_t sadb_sa_len;       /* length of extension / 8 */
  u_int16_t sadb_sa_exttype;   /* SADB_EXT_SA */
  u_int32_t sadb_sa_spi;       /* Security Parameters Index (SPI) */
  u_int8_t  sadb_sa_replay;    /* replay window size, or zero */
  u_int8_t  sadb_sa_state;     /* SA state, see Figure 19.7 */
  u_int8_t  sadb_sa_auth;      /* authentication algorithm, see Figure
  u_int8_t  sadb_sa_encrypt;   /* encryption algorithm, see Figure 19.8
  u_int32_t sadb_sa_flags;     /* bitmask of flags */
};
```

Dynamically Maintaining SAs

- A daemon registers itself with the kernel using the SADB_REGISTER message, specifying the type of SA it can handle in the sadb_msg_satype.
- If a daemon can handle multiple SA types, it sends multiple SADB_REGISTER messages, each registering a single type.
- In the SADB_REGISTER reply message, the kernel includes a supported algorithms extension, indicating what encryption and/or authentication mechanisms are supported (by an sadb_supported structure).

```
struct sadb_supported {
  u_int16_t sadb_supported_len;      /* length of extension + algorithms / 8 */
  u_int16_t sadb_supported_exttype;  /* SADB_EXT_SUPPORTED_{AUTH, ENCRYPT} */
  u_int32_t sadb_supported_reserved; /* reserved for future expansion */
};
                                     /* followed by algorithm list */

struct sadb_alg {
  u_int8_t sadb_alg_id;              /* algorithm ID from Figure 19.8 */
  u_int8_t sadb_alg_ivlen;           /* IV length, or zero */
  u_int16_t sadb_alg_minbits;        /* minimum key length */
  u_int16_t sadb_alg_maxbits;        /* maximum key length */
  u_int16_t sadb_alg_reserved;       /* reserved for future expansion */
};
```

Chapter 26: Threads

Thread Introduction

Unix traditional process paradigm: `fork` new processes, but there are problems with `fork`:

- `fork` is expensive. Memory is copied from the parent to the child, all descriptors are duplicated in the child, and so on.
- IPC is required to pass information between the parent and child after the `fork`. Parent to child is easy, but not the case vice versa.

Threads help with both problems:

- *lightweight processes*—"lighter weight" than a process: 10–100 times faster
- All threads within a process share the same global memory—sharing is easy, **but** along with this simplicity comes the problem of *synchronization*.

Thread Introduction contd.

Shared

All threads within a process share the following:

- Global variables
- Process instructions
- Most data
- Open files (e.g. descriptors)
- Signal handlers and signal dispositions
- Current working directory
- User and group IDs

Thread Introduction contd.

Its own

All threads within a process share the following:

- Thread ID
- Set of registers, including program counter and stack pointer
- Stack (for local variables and return addresses)
- `errno`
- Signal mask
- Priority

One analogy is to think of signal handlers as a type of thread, i.e. the main flow of execution (one thread) and a signal handler (another thread); both threads share the same global variables, but each has its own stack.

Basic Functions: Creation and Termination

- `pthread_create` Function

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func) (void *), void *arg);
```

- `pthread_join` Function

```
int pthread_join(pthread_t tid, void ** status);
```

- `pthread_self` Function

```
pthread_t pthread_self(void);
```

- `pthread_detach` Function

```
int pthread_detach(pthread_t tid);
```

- `pthread_exit` Function

```
oid pthread_exit (void *status);
```

All functions defined in `<pthread.h>`

Example: `str_cli` Function and TCP Echo

- `threads/strclithread.c`

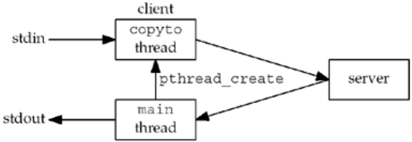

Figure: 26.1 Recoding `str_cli` to use threads.

- `threads/tcpserv01.c`

Mutexes: Mutual Exclusion

- A typical problem with *concurrent programming* or *parallel programming*
- In terms of Pthreads, it is a variable of type `pthread_metex_t`, used with the following two functions:

```
int pthread_mutex_lock(pthread_mutex_t * mptr);
int pthread_mutex_unlock(pthread_mutex_t * mptr);
```

Condition Variables

- We need something to let us go to sleep waiting for some condition to occur.
- Mutex provides mutual exclusion and the condition variable provides a signaling mechanism.
- In terms of Pthreads, it is a variable of type `pthread_cond_t`, used with the following two functions:

```
int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);

int pthread_cond_signal(pthread_cond_t *cptr);
```

Summary

- Creation of threads are normally faster than creation of a new process with `fork`.
- All threads in a process share global variables and descriptors. (Cause synchronization problems, so mutexes and condition variables)
- Thread safe issue.