Signals and Networks

Session Speaker

Deepak V.
Session Objectives

• To state the purpose and handling of signals in VxWorks
• To state the different types of Network Components
• Overview of Socket
• RPC procedure on a remote machine
Session Topics

• Signals in VxWorks
• Registering a Signal Handler
• Network Interface Layer
• Remote Procedure Call
• Sockets Programming
• RPC Client - Server Model
Signals

• Signal: sent to a process to notify an event
  – IPC with a short message sent to a process or to a group of processes
    • Number identifying the signal (no arguments)
    • Widely used due to its simplicity and efficiency: introduced by the first UNIX and exists for 30 years with minor changes
  – Usually, a process will call a user-space function (signal-handler) in response to a signal

• Purpose
  – Used to make a process aware that a specific event has occurred
  – Used to force a process to execute a signal handler function included in its code

• Analogy: *Signal to process is like interrupts to kernel*
# Signals

<table>
<thead>
<tr>
<th>#</th>
<th>Signal Name</th>
<th>Default Action</th>
<th>Comment</th>
<th>POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>Terminate</td>
<td>Hangup of controlling terminal or process</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt from keyboard</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>Dump</td>
<td>Quit from keyboard</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>SIGILL</td>
<td>Dump</td>
<td>Illegal instruction</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
<td>Dump</td>
<td>Breakpoint for debugging</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>Dump</td>
<td>Abnormal termination</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>SIGIOT</td>
<td>Dump</td>
<td>Equivalent to SIGABRT</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>SIGBUS</td>
<td>Abort</td>
<td>Bus error</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>SIGFPE</td>
<td>Dump</td>
<td>Floating point exception</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>SIGABRT</td>
<td>Terminate</td>
<td>Forced process termination</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>SIGBU</td>
<td>Terminate</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>SIGSEGV</td>
<td>Terminate</td>
<td>Invalid memory reference</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>SIGUSR2</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>SIGPIPE</td>
<td>Abort</td>
<td>Write to pipe with no readers</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>SIGALRM</td>
<td>Abort</td>
<td>Real time clock</td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>SIGTERM</td>
<td>Abort</td>
<td>Process termination</td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>SIGTKFLTR</td>
<td>Abort</td>
<td>Coprocessor stack error</td>
<td>No</td>
</tr>
</tbody>
</table>
## Signals

<table>
<thead>
<tr>
<th>Sig</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
</tr>
<tr>
<td>18</td>
<td>SIGCONT</td>
<td>Continue</td>
</tr>
<tr>
<td>19</td>
<td>SIGSTOP</td>
<td>Stop</td>
</tr>
<tr>
<td>20</td>
<td>SIGTSTP</td>
<td>Stop</td>
</tr>
<tr>
<td>21</td>
<td>SIGTTIN</td>
<td>Stop</td>
</tr>
<tr>
<td>22</td>
<td>SIGTTOU</td>
<td>Stop</td>
</tr>
<tr>
<td>23</td>
<td>SIGURG</td>
<td>Ignore</td>
</tr>
<tr>
<td>24</td>
<td>SIGXCPU</td>
<td>Dump</td>
</tr>
<tr>
<td>25</td>
<td>SIGXFSZ</td>
<td>Dump</td>
</tr>
<tr>
<td>26</td>
<td>SIGVTALRM</td>
<td>Terminate</td>
</tr>
<tr>
<td>27</td>
<td>SIGPROF</td>
<td>Terminate</td>
</tr>
<tr>
<td>28</td>
<td>SIGWINCH</td>
<td>Ignore</td>
</tr>
<tr>
<td>29</td>
<td>SIGIO</td>
<td>Terminate</td>
</tr>
<tr>
<td>29</td>
<td>SIGPOLL</td>
<td>Terminate</td>
</tr>
<tr>
<td>30</td>
<td>SIGPWR</td>
<td>Terminate</td>
</tr>
<tr>
<td>31</td>
<td>SIGUNUSED</td>
<td>Dump</td>
</tr>
</tbody>
</table>
Signals in VxWorks

- VxWorks supports a software signal facility. Signals asynchronously alter the control flow of a task.
- Any task or ISR can raise a signal for a particular task.
- The task being signaled immediately suspends its current thread of execution and executes the task-specified signal handler routine the next time it is scheduled to run.
- The signal handler executes in the receiving task’s context and makes use of that task’s stack. The signal handler is invoked even if the task is blocked.

```c
void normalCode()
{
    ...
}

void mySignalHandler()
{
    ...
}
```

Diagram:

- signal -> normalCode()
- signal -> mySignalHandler()
Signals in VxWorks

- Signals are more appropriate for error and exception handling than as a general purpose intertask communication mechanism.
- In general, signal handlers should be treated like ISRs; no routine should be called from a signal handler that might cause the handler to block.
- Because signals are asynchronous, it is difficult to predict which resources might be unavailable when a particular signal is raised. To be perfectly safe, call only those routines that can safely be called from an ISR.
- If a task has supplied a signal handler for an exception, the default exception handling is not performed.
- A user-defined signal handler is useful for recovering from catastrophic events.
- The `setjmp()` is called to define the point in the program where control will be restored, and `longjmp()` is called in the signal handler to restore that context.
  - Note that `longjmp()` restores the state of the task’s signal mask.
- Signals are also used for signaling software exceptions as well as hardware exceptions.
Signals in VxWorks

• The wind kernel supports two types of signal interface:
  – UNIX BSD-style signals
  – POSIX-compatible signals
    • The POSIX-compatible signal interface, in turn, includes both the fundamental signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b
• For more information about signals, see the reference entry for sigLib
• By default, VxWorks uses the basic signal facility component INCLUDE_SIGNALS which automatically initializes signals with sigInit()
Signals in VxWorks

- Signals are analogous to hardware interrupts
- The basic signal facility provides a set of 31 distinct signals
- A signal handler binds to a particular signal with `sigvec()` or `sigaction()` in much the same way that an ISR is connected to an interrupt vector with `intConnect()`
- Certain signals are associated with hardware exceptions. For example, bus errors, illegal instructions, and floating-point exceptions raise specific signals
- The VxWorks implementation of `sigLib` does not impose any special restrictions on operations on `SIGKILL`, `SIGCONT`, and `SIGSTOP` signals such as those imposed by UNIX
Registering a Signal Handler

- To register a signal handler: `signal (signo, handler)`
  - Signo: Signal number
  - Handler: Routine to invoke when signal arrives (or SIG_IGN to ignore signal)
  - Returns the previously installed signal handler, or SIG_ERR

- The signal handler should be declared as:
  ```
  void sigHandler (int sig); /* signal number */
  ```

- If an exception signal handler returns:
  - The offending task will be suspended
  - A message will be logged to the console

- Exception signal handlers typically call:
  - `exit( )` to terminate the task, or
  - `taskRestart( )` to restart the task, or
  - `longjmp( )` to resume execution at location saved by `setjmp( )`
# Basic Signal Calls

<table>
<thead>
<tr>
<th>POSIX 1003.1b Compatible Call</th>
<th>UNIX BSD Compatible Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal()</td>
<td>signal()</td>
<td>Specifies the handler associated with a signal.</td>
</tr>
<tr>
<td>kill()</td>
<td>kill()</td>
<td>Sends a signal to a task.</td>
</tr>
<tr>
<td>raise()</td>
<td>N/A</td>
<td>Sends a signal to yourself.</td>
</tr>
<tr>
<td>sigaction()</td>
<td>sigvec()</td>
<td>Examines or sets the signal handler for a signal.</td>
</tr>
<tr>
<td>sigsuspend()</td>
<td>pause()</td>
<td>Suspends a task until a signal is delivered.</td>
</tr>
<tr>
<td>sigpending()</td>
<td>N/A</td>
<td>Retrieves a set of pending signals blocked from delivery.</td>
</tr>
<tr>
<td>sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember()</td>
<td>sigsetmask()</td>
<td>Manipulates a signal mask.</td>
</tr>
<tr>
<td>sigprocmask()</td>
<td>sigsetmask()</td>
<td>Sets the mask of blocked signals.</td>
</tr>
<tr>
<td>sigprocmask()</td>
<td>sigblock()</td>
<td>Adds to a set of blocked signals.</td>
</tr>
</tbody>
</table>
Network Programming
1 Introduction

Sockets

UDP Sockets Programming

TCP Sockets Programming

RPC
VxWorks Networking

- Network programming allows users to:
  - Build services
  - Create distributed applications
- VxWorks network programming tools:
  - Berkeley sockets
  - zbuf Socket API
  - Sun RPC (Remote Procedure Call)
Network Components

- NFS
- Tornado tgserv
- rlogin
- telnet
- rsh
- ftp
- RPC
- zbuf API
- Sockets
- TCP
- UDP
- IP
- Ethernet
- Shared Memory
- SLIP/PPP
Network Interface Layer

• Network interface layer enables machines attached to the same physical media to communicate
• Three types of network interface provided by VxWorks allows applications to communicate using:
  – An ethernet cable
  – The VMEbus (shared memory)
  – A serial line
• VxWorks supports SLIP, CSLIP and PPP protocols for communicating over a serial line
• PPP (Point-to-Point Protocol)
• (C)SLIP ((Compressed) Serial Line Internet Protocol)
Network Components

[Schematic diagram showing network components including NFS, Tornado tgtsvr, rlogin, telnet, rsh, ftp, RPC, Sockets, TCP, UDP, IP, Ethernet, Shared Memory, and SLIP/PPP.]
Internet Protocol

• IP (Internet Protocol) layer routes data from machine to machine, forwarding through gateways as needed
• Routing tables are used to make decisions
• IP also fragments and reassembles IP datagrams when necessary
Network Components
Transport Layer

- Transport layer gets data from application to application
- Two protocols available - UDP and TCP
- UDP (User Datagram Protocol)
  - Connectionless
  - Message (datagram) oriented
  - Unreliable
  - Stateless
- TCP (Transmission Control Protocol)
  - Connection based
  - Stream oriented
  - Reliable
  - Flow controlled
Ports

- Abstract destination point within a node

- TCP/UDP intertask communication:
  - Data is sent by writing to a remote port
  - Data is received by reading from a local port
Network Components

Diagram showing network components:

- Network Driver
  - NFS
  - Tornado
tgtsrv
  - rlogin
  - telnet
  - rsh
  - ftp

- zbuf
  - API

- Sockets
  - TCP
  - UDP

- IP
  - Ethernet
  - Shared Memory
  - SLIP/PPP
Application Programmer’s Interface

- Application Programmer’s Interface (API) to the network protocols
- The zbuf API is a proprietary socket interface which uses buffer loaning to improve performance by minimizing data copies
- The zbuf API is also called Zero-Copy TCP
Network Components
Remote Procedure Call

• All other applications, including user defined applications, are built on top of sockets
• RPC (Remote Procedure Call) is a higher level API for communicating over the network
Packet Encapsulation

- Application bandwidth will be lower than the bandwidth of the physical media over which the data passes (e.g., Ethernet)
- Sending many small packets will give poor performance
Network Programming

Introduction

2 Sockets

UDP Sockets Programming

TCP Sockets Programming

RPC
Socket Overview

• Programmatic interface to internet protocols
• Protocol specified when socket created (e.g., UDP or TCP)
• Server binds its socket to a well known port
• Client’s port number dynamically assigned by the system
Ports

• Socket address consists of:
  – An internet address (node socket is on)
  – A port number (destination within that node)

• Port identified by a short integer. VxWorks port usage conventions:
  0 - 1023 Reserved for system services (e.g., rlogin, telnet, etc.)
  1024 - 5000 Dynamically allocated
  > 5000 User defined

• Unique to each machine and protocol
Socket Address

- Generic socket address:

```c
struct sockaddr {
    u_short    sa_family; /* address family */
    char       sa_data[14]; /* protocol specific address data */
};
```

- Socket address structure used by Internet Protocol:

```c
struct sockaddr_in {
    short       sin_family;   /* AF_INET */
    u_short     sin_port;     /* port number */
    struct in_addr sin_addr; /* internet address */
    char        sin_zero[8]; /* padding, must be zeroed out */
};
```

VxWorks supports only Internet sockets.

sockaddr_in and in_addr structures defined in in.h
Network Byte Ordering

- Fields in the struct sockaddr_in must be put in network byte order (big-endian)
- Macros to convert long/short integers between the host and the network byte ordering:
  - htonl() host to network long
  - htons() host to network short
  - ntohl() network to host long
  - ntohs() network to host short

- Macros defined in wind/target/h/netinet/in.h
- Big Endian: Most significant byte stored at lowest address in memory and goes first across the network
Caveat - User Data

- To send data in a system independent way:
  1. Sender converts data from its system-dependent format to some standard format
  2. Receiver converts data from the standard format to its system-dependent format
- The standard format used must handle:
  - Any standard data types used (e.g., int, short, float, etc.).
  - Data structure alignment
- One such facility, XDR, will be discussed in the RPC section of this chapter
Creating a Socket

```c
int socket (domain, type, protocol)
```

- **domain**  Must be PF_INET
- **type**    Typically SOCK_DGRAM (UDP) or SOCK_STREAM (TCP)
- **protocol**  Socket protocol (typically 0)

- Opens a socket (analogous to open() for files)
- Returns a socket file descriptor or ERROR

- Symbolic constants for socket types are defined in wind/target/h/sys/socket.h
- Sockets functions are declared in sockLib.h, which also includes sys/socket.h
Binding a Socket

• To bind a socket to a well known address:

  STATUS bind (sockFd, pAdrs, adrsLen)

  sockFd  Socket descriptor returned from socket( )

  pAdrs  Pointer to a struct sockaddr to which to bind this socket

  adrsLen  sizeof (struct sockaddr)

• Typically only called by server
Introduction

Sockets

3 UDP Sockets Programming

TCP Sockets Programming

RPC
UDP Socket Overview

```
fd = socket (PF_INET, SOCK_DGRAM, 0)
bind (fd, (struct sockaddr *)&saddr, sizeof(saddr))

loop
  recvfrom (fd, ...) /* Wait for request */
  /* Perform service */
  sendto (...) /* Send reply */

select (...) /* Wait for reply with timeout */
```
UDP Socket Overview

• UDP is a connectionless, message-oriented protocol
• The server binds its socket to a well known port and reads requests from its socket in an infinite loop
• If client does not bind its socket to an address, a port number is dynamically assigned
• UDP is unreliable - the request or the reply may get lost. The client must implement some kind of timeout to detect lost packets
• This is typically done by calling select( ) to wait for a reply with a timeout
Sending Data on UDP Sockets

```c
int sendto (sockFd, pBuf, bufLen, flags, pDestAdrs, destLen)
```

sockFd  SOCKET to send data from
pBuf Address of data to send
bufLen Length of data in bytes
flags Special actions to be taken
pDestAdrs Pointer to struct sockaddr containing
destination address
destLen sizeof (struct sockaddr)

- Returns the number of bytes sent or ERROR
- flags may be MSG_DONTROUTE to send a message without consulting routing table (only used by diagnostic or routing programs)
Receiving Data on UDP Sockets

int recvfrom (sockFd, pBuf, buflen, flags, pFromAdrs, pFromLen)

sockFd  Socket to receive data from
pBuf    Buffer to hold incoming data
buflen  Maximum number of bytes to read
flags   Flags for special handling of data
pFromAdrs  Pointer to struct sockaddr. Routine
            supplies internet address of sender
pFromLen  Pointer to integer. Must be initialized to
            sizeof (struct sockaddr)

• Blocks until data available to receive
• Returns number of bytes received or ERROR
• flags may be MSG_PEEK to look at data without removing it from the queue
• UDP is a datagram delivery service. Consequently, unread bytes in a packet
  will be lost
Network Programming

Introduction

Sockets

UDP Sockets Programming

TCP Sockets Programming

RPC
TCP Socket Overview

• TCP is connection based (like making a phone call)
• Concurrent servers are often implemented:

Clients send connection requests to Master’s socket

New socket created dynamically for each connection, serviced by work task
TCP Socket Overview

• The master server’s socket is never used for data transfer, only for connection requests
• A new socket file descriptor is created dynamically for each incoming client connection
• The new connection is handled by a server work task which is either spawned dynamically or pre-spawned at initialization time as one of a number of such tasks
• This allows many clients to be serviced concurrently
• Concurrent servers are advantageous when:
  – Service requires blocking: Several clients can be handled at once, with server slaves overlapping blocking and non blocking activities
  – Different requests need servicing at different priorities
TCP Server Overview

/* master server */
masterFd = socket (PF_INET, SOCK_STREAM, 0)
/* fill in server's sockaddr struct */
bind (...) /* bind to well-known port */
listen (...) /* configure request queue */
FOREVER
{
    clientFd = accept (masterFd, ...)
    taskSpawn (... , slaveSrv, clientFd, ...)
}
/* slave server */
slaveSrv(clientFd, ...)
{
    read (clientFd, ...) /* read request */
    serviceClient ()
    write (clientFd, ...) /* send reply */
    close (clientFd)
}
TCP Client Overview

1 /* TCP Client */
2 fd = socket (PF_INET, SOCK_STREAM, 0)
3
4 /* fill in sockaddr with server’s address */
5
6 connect (fd, ...) /* request service */
7
8 write (fd, ...) /* send request */
9 read (fd, ...) /* read reply */
10
11 close (fd) /* terminate connection */
Server Initialization

• Before accepting connections, server must:
  – Create a socket (socket( ))
  – Bind the socket to a well known address (bind( ))
  – Establish a connection request queue:
    STATUS listen (sockFd, queueLen)

sockFd Socket descriptor returned from socket( )
queueLen Nominal length of connection request queue
int accept (sockFd, pAdrs, pAdrsLen)

sockFd     Servers socket (returned from socket( ))
pAdrs      Pointer to a struct sockaddr through which the client’s address is returned
pAdrsLen   Pointer to length of address

• Blocks until connection request occurs
• Returns new socket file descriptor (connected to the client) or ERROR
• Original socket, sockFd, is unconnected and ready to accept other connection requests
Requesting Connections

- To connect to the server, the client calls:

  STATUS connect (sockFd, pAdrs, adrsLen)

  - *sockFd*  Client’s socket descriptor
  - *pAdrs*  Pointer to server’s socket address
  - *adrsLen*  sizeof (struct sockaddr)

- Blocks until connection is established or timeout
- Returns ERROR on timeout or if no server is bound to pAdrs
Exchanging Data

• read( )/write( ) may be used to exchange data:

• Caveat: TCP is stream oriented
  – write( ) may write only part of message if I/O is nonblocking
  – read( ) may read more or less than one message
Cleaning up a Stream Socket

• When done using a socket, close( ) it:
  – Frees resources associated with socket
  – Attempts to deliver any remaining data
  – Causes read( ) from peer socket to return 0

• Can also use shutdown( ) to terminate output, while still receiving data from peer socket

• Peer refers to the remote socket to which local socket is connected

• By default, the close( ) call returns immediately; however, TCP attempts to ensure delivery of any outstanding unacknowledged data. Different behaviors of close( ) may be specified by calling setsockopt( ) with the SO_LINGER option
Setting Socket Options

• Options can be enabled on a per socket basis, including:
  – Don’t delay write( )’s to coalesce small TCP packets
  – Enable UDP broadcasts
  – Linger on close( ) until data is sent
  – bind( ) to an address already in use
  – Change the size of the send/receive buffers

• To make a socket non-blocking:
  
  ```c
  int val = 1; /* Set to 0 for blocking I/O */
  ioctl (sock, FIONBIO, &val);
  ```
Setting Socket Options

• If a socket is made non-blocking, a read operation which would have blocked returns ERROR instead and errno is set to EWOULDBLOCK
• A write operation will write the number of bytes which can be buffered or written immediately, and return that number of bytes
• If no bytes can be buffered, the write returns ERROR and errno is set to EWOULDBLOCK
zbuf Socket API

- Improves application performance by minimizing data copies through buffer loaning
- Application must manage buffers
- zbuf application can communicate with a standard socket application
- Supports TCP and UDP protocols
- See zbufLib and zbufSockLib for details
- Proprietary API
Network Programming

Introduction

Sockets

UDP Sockets Programming

TCP Sockets Programming

5 RPC
Overview

• RPC (Remote Procedure Call) provides a standard way to invoke procedures on a remote machine

• For more information about RPC, see:
  – Appendix
  – TCP/IP Illustrated Volume I (Stevens)
  – Power Programming with RPC (O’Reilly & Associates)
  – Documentation and source code can be found in wind/target/unsupported/rpc4.0
RPC Client - Server Model

![Diagram showing the client-server model](image-url)
RPC Client - Server Model

The client calls a local procedure, called the client stub, which hides the details of the network:
1. The client stub builds a network message, using XDR as needed to put parameters in a standard format
2. The message is sent over the network to the server stub
3. The server stub unpacks the message, using XDR as needed
4. A server routine is invoked
5. Return value from the server routine is packaged, using XDR as needed
6. Message is passed back over the network to the client stub
7. The client stub unpacks the message, using XDR as needed
8. The return value is passed back to the client
VxWorks and rpcgen

• rpcgen is a RPC protocol compiler
• From a specification of the remote procedures, rpcgen creates:
  – A client stub
  – A server stub
  – The XDR routines for packing/unpacking data structures. Not created if all parameters/return values are standard data types
  – A header file for inclusion by client and server
• Each VxWorks task accessing RPC calls using code produced by rpcgen must first initialize access

STATUS rpcTaskInit( )
Summary

- VxWorks supports a software signal facility. Signals asynchronously alter the control flow of a task.
- A *signal handler* binds to a particular signal with `sigvec()` or `sigaction()` in much the same way that an ISR is connected to an interrupt vector with `intConnect()`.
- Sockets as the interface to network protocols:
  - UDP transport protocol
  - TCP transport protocol
  - Configurable socket options
- `zbuf` socket API improves application performance by minimizing data copies through buffer loaning.
- Client/server programming strategies are operated for distributed applications.