UCSD Embedded Linux online
Week 5

Slide 1

It's a "net-centric" world, as the marketers like to say. Everything's connected to a network and that's becoming increasingly true of embedded devices. So it's time to turn our attention to network programming.

Slide 2

But before we get into network programming, there's one outstanding item from last week that we need to address. I chose to put this issue off because there was a lot of material in the last class.

Last week's Posix threads exercise introduced a different way of accessing the peripheral devices. Instead of accessing the I/O ports directly from User space, this example makes use of Kernel space device drivers that further isolate us from the hardware.

While other OSes treat devices as files, "sort of", Linux goes one step further in actually creating a directory for devices. Typically this is /dev. One consequence of this is that the redirection mechanism can be applied to devices just as easily as to files.

Devices come in four "flavors": Character, Block, Pipe, and Network. The principal distinction between character and block is that the latter, such as disks, are randomly accessible, that is, you can move back and forth within a stream of characters. With character devices the stream typically moves in one direction only. Block devices, of necessity, have file systems associated with them whereas character devices don't. But in both cases, I/O data is viewed as a "stream" of bytes.

Pipes are pseudo-devices that establish uni-directional links between processes. One process writes into one end of the pipe and another process reads from the other end.

Network devices are different in that they handle "packets" of data for multiple protocol clients rather than a "stream" of data for a single client. This necessitates a different interface between the kernel and the device driver. Network devices are not nodes in the /dev directory.

From here on out we'll only be dealing with character devices.

Slide 3

User Space programs can access I/O at two different levels. We'll be using the low-level I/O APIs that include the kernel services shown in this slide.

The read and write calls are fairly obvious. read transfers data from the device (or file) to memory while write transfers in the opposite direction. The purpose of the open function is to establish a connection, or "path" between the device/file and the application. In the case of a physical device, this is what creates the connection to the device driver. The value returned by open is a *file descriptor*, an integer index, which is used as an argument to all other I/O calls to identify the connection. When a connection is no longer needed, the close function frees up the resources used by that connection.

ioctl may be thought of as a general-purpose "escape" mechanism. Every device has some peculiar characteristics that don't fit neatly into the stream-of-bytes model. A printer has an out-of-paper status condition. A serial port or modem has a data rate parameter. ioctl provides an open-ended mechanism for handling all of these device-specific conditions. Each device driver is allowed to define its own set of ioctl commands and any parameters those commands might need.

Slide 4

The Mini2440 Linux implementation provides a collection of three device drivers as shown here to access the peripheral devices that are relevant to our particular application. Take a look at trgdrive.c in the posix/ directory. Note how initAD() and initDigIO() simply open the corresponding devices. readAD() reads a text string from the ADC and translates it to binary. Set, clear, and write DigOut use the ioctl mechanism to control the LEDs. getDigIn() reads a text string representing the state of the six pushbuttons and translates it to a bit mask.

The network and LCD versions of the thermostat program also use the device driver approach as described here. We'll look at device drivers in a little more detail later on in the class. Our motivation right now is simply to show how device drivers can be utilized from user space applications.

Slide 5

So let's move on to networking. The "socket" interface, first introduced in Berkeley versions of Unix, forms the basis for most network programming. Sockets are an extension of the named pipe concept that explicitly supports a client/server model wherein multiple clients may attach to, and use the services of, a single server. Both the client and server may exist on the same machine. This simplifies the process of building client/server applications. You can test both ends on the same machine before distributing the application across a network.

Slide 6

Here are the basic steps that the server process goes through to establish communication. We start by creating a socket and then *bind* it to a name. For local sockets, the name is a file system entry often in /tmp or /usr/tmp. For network sockets it is a *service identifier* consisting of a port number and access point. Clients use this name to access the service.

Next, the server creates a connection queue with the *listen* service and waits for client connection requests with the *accept* service. When a connection request is received, the accept service returns a new socket, which is then used for this connection's data transfer. When the transaction is complete the newly created socket is closed.

Slide 7

This slide illustrates graphically how the server process sets up a connection.

The server may very well spawn a new thread or process to service the connection while it goes back and waits for additional client requests. This is what allows multiple clients to be simultaneously attached to the server.

Slide 8

The client is, perhaps not surprisingly, somewhat simpler. It begins by creating a socket and naming it to match the server's publicly advertised name. Next, it attempts to connect to the server. If the connection request succeeds, the client proceeds to transfer data as necessary. When the transaction is complete, the client closes the socket.

Slide 9

Again, here's a graphical representation of the client process.

Slide 10

A socket is created by calling the socket system service, which returns a descriptor for later use in accessing the socket.

A socket is characterized by three attributes that determine how communication takes place. The *domain* specifies the communication medium. The most commonly used domains are AF_UNIX for local file system sockets and AF_INET for Internet connections.

The domain determines the format of the socket name or address. For AF_INET, the address is in the form of a "dotted quad" as in 198.162.1.50. A network computer may support many different network services. A specific

service is identified by a "port number". Established network services like ftp, http, etc have defined port numbers, usually below 1024. Local services may use port numbers above 1023.

Some domains, AF_INET for example, offer alternate communication mechanisms. SOCK_STREAM is a sequenced, reliable, connection-based, two-way byte stream. This is the basis for TCP and is the default for AF_INET domain sockets. SOCK_DGRAM is a *datagram* service. It is used to send relatively small messages with no guarantee that they will be delivered or that they won't be reordered by the network. This is the basis of UDP. The AF_UNIX domain only supports streams.

The protocol is usually determined by the domain and type and you don't have a choice. So the protocol argument is usually zero.

Slide 11

We'll start with a fairly trivial example to illustrate client/server interaction. You'll need to create a new Eclipse makefile project and point it at target_fs/home/src/network. Create two make targets in the project named "server" and "client" where the targets are respectively "server" and "client." Note incidentally that this makefile does not have an "all" target. So when you created it, you should have seen an error message, "No rule to create target 'all'".

Take a look at netserve.c. Notice first of all the three new header files at lines 12 to 14 that prototype the network services and data structures. At line 24 we create a server_socket that uses streams. Next we need to *bind* this socket to a specific network address. That requires filling in a sockaddr_in structure, server_addr. The function inet_aton() takes a string containing a network address as its first argument, converts it to a binary number and stores it in the location specified by the second argument, in this case the appropriate field of server_addr. inet_aton() returns zero if it succeeds. In this example the network address is passed in through the compile-time symbol SERVER so that we can build the server to run either locally through the loopback device or across the network.

The port number is 16 bits and is passed in through another compile-time symbol in the makefile, PORT, that is currently set to 4201. The function htons() is one of a small family of functions that solves the problem of transferring binary data between computer architectures with different byte ordering policies. The Internet has established a standard "network byte order", which happens to be Big Endian. All binary data is expected to be in network byte order when it reaches the network. htons() translates a short (16 bit) integer from "host byte order", whatever that happens to be, to network byte order. There is a companion function, ntohs() that translates back from network byte order to host order. Then there is a corresponding pair of functions that do the same translations on long (32 bit) integers.[1]

Now we *bind* server_socket to server_addr with the bind() function. Finally, we create a queue for incoming connection requests with the listen() function. A queue length of one should be sufficient in this case because there's only one client that will be connecting to this server.

Now we're ready to *accept* connection requests. The arguments to accept() are:

- The socket descriptor.
- A pointer to a sockaddr structure that accept() will fill in.
- A pointer to an integer that currently holds the length of the structure in the second argument. accept() will modify this if the length of the client's address structure is shorter.

accept() blocks until a connection request arrives. The return value is a socket descriptor to be used for data transfers to/from this client. In this example the server simply echoes back text strings received from the client until the incoming string begins with "q".

[1] Try to guess the names of the long functions.

Now look at netclient.c. netclient determines at run time whether it is connecting to a server locally or across the network. We start by creating a socket and an address structure in the same manner as in the server. Then we *connect* to the server by calling connect(). The arguments are:

- The socket descriptor.
- A pointer to the sockaddr structure containing the address of the server we want to connect to.
- The length of the sockaddr structure.

When connect() returns we're ready to transfer data. The client prompts for a text string, writes this string to the socket and waits to read a response. The process terminates when the first character of the input string is "q".

You can run both programs in separate shell windows as shown here. But also remember that you can debug either of these programs by setting the host launch configuration to the network project and selecting one of these programs as the application.

Slide 12

One other piece of business we didn't get to last week is an easier way to specify include paths for Makefile projects. We can *export* the include paths from the measure project as shown here and then *import* them into the network project as described in the next slide.

You can put the file anywhere you like. You'll be reading it in the next slide. The file gets a .xml extension

Slide 13

Now we'll import the include paths into the network project. The process is almost identical only this time you're reading the file instead of writing it.

Slide 14

Now let's run our client/server pair across the network. Create a new project target as described here and then rebuild. In this case, the client runs on the workstation and the server on the target. Don't forget to delete the server object files before rebuilding.

You can debug the server on the target by setting the target launch configuration to the network project and selecting server as the application.

Slide 15

Moving on to a more practical example, our thermostat may very well end up in a distributed industrial environment where the current temperature must be reported to a remote monitoring station and setpoint and limit need to be remotely settable. Naturally we'll want to do that over a network.

Copy monitor.c and thermostat.c from the Posix/ directory to network/. Actually, thermostat.c doesn't require any changes.

Slide 16

This slide shows the changes required in monitor.c to replace the current serial connection with a network connection. You'll want to isolate the process of setting up the net server and establishing a connection in a seperate function called createServer(), which consists roughly of lines 18 to 55 from netserve.c. The reason for doing it this way will become apparent soon.

createServer()'s return value is the client socket number returned by accept(). It should return a -1 if anything fails. Call this function just before the while (1) loop and check the return value for error.

Replace the fgets() call with a read() call that reads the client socket.

Two other features that weren't in our previous monitor program are a case for "?" and an "OK" response to the client acknowledging a parameter change. The "OK" response to a parameter change is necessary for compatibility with our network client that requires a response to every message it sends.

The "?" is a way to send the current temperature over the network. Actually, we might want to query any of the thermostat's parameters. Think about how you might extend the command protocol to accomplish that.

Slide 17

Create the make targets shown here and then build the simulation version of the thermostat. The net thermostat works with the net client that we've already built and tested. You'll need *three* shell windows to run the simulated net thermostat: one for the devices program (start this first), another for netclient (start this last), and a third for thermostat_s.

When you're happy with the simulation, build the target version and test it there. Remember to run netclient remote to get the client to talk to the remote server.

Slide 18

As the net thermostat is currently built, only one client can be connected to it at a time. It's quite likely that in a real application, multiple clients might want to connect to the thermostat simultaneously. Remember from our discussion of sockets that the server process may choose to spawn a new process in response to a connection request so it can immediately go back and listen for more requests.

In our case we can create a new monitor thread. That's the idea behind the createServer() function. It can be turned into a server thread whose job is to spawn a new monitor thread when a connection request comes along. Then it goes back and accepts another connection.

Here, in broad terms, are the steps required to convert the net thermostat to service multiple clients:

1. Copy monitor.c to a file called multimon.c and work in that file
2. Recast createServer() as a thread function. This means that its return value is void * and it takes a void * argument, neither of which we will use. Replace any error returns with return NULL.
3. In the createThread() function, change the arguments of the pthread_create() call to create the server thread instead of the monitor thread.
4. In the terminateThread() function, change the arguments of the pthread_cancel() call to cancel the server thread instead of the monitor thread.
5. Go down to the last four lines in createServer(), following the comment "Accept a connection". Bracket these lines with an infinite while loop.
6. Replace the return client_socket; statement with a call to pthread_create() to create a monitor thread.
7. Add a case 'q': to the switch statement in the monitor thread. 'q' says the client is terminating (quitting) the session. This means the monitor thread should exit.

Slide 19

Now, here are the tricky parts. Where does the thread object come from for each invocation of pthread_create(). A simple way might be to estimate the maximum number of clients that would ever want to access the thermostat and then create an array of pthread_ts that big. The other, more general, approach is to use dynamic memory allocation, the malloc() function.

The client_socket will have to be passed to the newly created monitor thread. OK, we can probably do that with the thread's parameter.

But the biggest problem is, how do you recover a pthread_t object when a monitor thread terminates? The monitor thread itself can't return it because the object is still being used. If we ignore the problem the thermostat will eventually become unresponsive either because all elements of the pthread_t array have been used, or we run out of memory because nothing is being returned to the dynamic memory pool.

Slide 20

Here are some thoughts. Associate a flag with each pthread_t object whether it is allocated on a fixed array or dynamically allocated. The flag, which is part of a data structure that includes the pthread_t, indicates whether its associated pthread_t is free, in use, or "pending," that is, able to be freed. The server thread sets this flag to the "in use" state when creating the monitor. The structure should probably also include the socket number. It is this "meta" pthread_t object that gets allocated and passed to the monitor thread.

Now the question is, when can the flag be set to the "free" state? Again, the monitor can't do it because the pthread_t object is still in use until the thread actually exits. Here's where the third state, PENDING, comes in.

The monitor sets the flag to PENDING just before it terminates. Then we create a separate thread, let's call it "resource", that does the following:

1. Wakes up periodically and checks for monitor threads that have set their flags to PENDING
2. Joins the thread
3. Marks the meta pthread object free
4. Goes back to sleep

Remember that when one thread joins another, the "joiner" (in this case resource) is blocked until the "joinee" (monitor) terminates. So when resource continues from the pthread_join() call, the just-terminated monitor's pthread_t object is guaranteed to be free.

It's quite likely that by the time the resource thread wakes up, the monitor thread has already exited, in which case pthread_join() returns immediately with the error code ESRCH meaning that *thread* doesn't exist.

Slide 21

Create the resource thread function. When a monitor thread is ready to exit, it sets the flag field of its meta_pthread_t object to PENDING and exits. When the resource thread wakes up, it scans the entire list or array of meta_pthread_ts looking for any that have a flag value of PENDING. It joins the corresponding thread and sets the flag to FREE on return from pthread_join().

The createThread() function now needs to create both the server thread and the resource thread. Likewise, terminateThread() needs to cancel and join both of those. And of course, any monitor threads that are running also need to be cancelled and that should be done before the resource thread is cancelled.

There's a separate makefile called Makefile.multi to build the multiple client versions of netthermo. This requires two more make targets in the network project. You decide what to name them. The build command for these targets is shown on the last line here. The –f option is the way of specifying a different makefile. And of course, add SERVER=REMOTE to build the target version.

So how do you test multiple monitor threads? Simple. Just create multiple shell windows and start netclient in each one. When you run thermostat under the debugger, you'll note that a new thread is created each time netclient is started.

Slide 22

While programming directly at the sockets level is a good introduction to networking, and a good background to have in any case, most real world network communication is done using higher level application protocols. HTTP, the protocol of the World Wide Web is probably the most widely used. No big surprise there. After all, virtually every desktop computer in the world has a web browser. Information rendered in HTTP is accessible to any of these computers with no additional software.

HTTP is a fairly simple synchronous request/response ASCII protocol over TCP/IP as shown here. The client sends a request message consisting of a header and, possibly, a body separated by a blank line. The header includes what the client wants along with some optional information about its capabilities. Each of the protocol elements is a line of text terminated by CR/LF. The single blank line at the end of the header tells the server to proceed.

Slide 23

Here's an example of a typical HTTP request packet. The first line starts with a *method token*, in this case GET, telling the server what "method" to use in fulfilling this request. This is followed by the "resource" that the method acts on, in this case a file name. The server replaces the "/" with the default file index.html. The remainder of the first line says the client supports HTTP version 1.1

The Host: header specifies to whom the request is directed, while User-Agent: header identifies who the request is from. Next come several headers specifying what sorts of things this client understands in terms of media types, language, encoding, and character sets. The Accept: header line is actually much longer than shown here.

The Keep-Alive: and Connection: headers are artifacts of HTTP version 1.0 and specify whether the connection is "persistent", or is closed after a single request/response interaction. In version 1.1 persistent connections are the default.

This example is just a small subset of the headers and parameters available.

Slide 24

It's not really necessary to understand the HTTP protocol in any detail because there are plenty of web servers out there that do it all for you. If, as suggested here, you Google "embedded web server," you get some 7 million hits. One of the first hits is a Wikipedia article on embedded web servers. This in turn leads to the comparison of web servers listed here.

In fact, our target board has a version of the boa embedded web server. It is started at boot time. From a web browser on your workstation, go to the address shown here. Most of the page is in Chinese, but you get the idea.

boa.conf provides configuration information for the program. Among other things, it specifies where the top level HTML files will be found. In this case it's /www. When you open a web page, you normally don't specify a file name. The default is usually index.html. This is also specified in boa.conf.

Slide 25

It turns out that building a basic web server is not that difficult. Really, all it does is serve up files. All the complicated work, the rendering, is done in the browser. There is a simple web server in the network project consisting of two files: webserve.c and webvars.c.

Open webserve.c and go down to the createserver() function around line 214. It looks pretty similar to the same functionality we saw in netserve.c earlier except that it listens on port 80, the one assigned to HTTP. Go on down to the while (1) loop in webmonitor() at line 270. Once the connection is established, the server reads a request message from the client and acts on it. For our rather limited purposes, we're only going to handle two methods: POST and GET.

In the case of GET, the function doGETmethod() near line 183 opens the specified file and determines its content type. If everything is ok, we call responseHeader() to send the success response and then we send the file itself.

Just serving up static HTML web pages isn't particularly interesting, or even useful, in embedded applications. Usually the device needs to report some information and we may want to exercise some degree of control over it. There are a number of ways to incorporate dynamic content into HTML, but for our limited purposes, we're going to take a "quick and dirty" approach suggested by M. Tim Jones in his very useful book *TCP/IP Application Layer Protocols for Embedded Systems*.

A nice feature of HTML is that it's easily extensible. You can invent your own tags. Of course, any tag the client browser doesn't understand it will simply ignore. So if we invent a tag, it has to be interpreted by the server before sending the file out. We'll invent a tag called <DATA> as shown in this slide. Open index.html and you'll see that it has one data tag in it.

The server scans the HTML text looking for a <DATA> tag. data_function is a function that returns a string. The server replaces the <DATA> tag with the returned string. The function parseHTML() around line 124 scans the input file for the <DATA> tag. When one is found, it writes everything up to the tag out to the socket. Then it calls web_var() with the name of the associated data function. web_var(), in webvars.c, looks up and invokes the data function, and returns its string value. The return value from web_var() is then written out and the scan continues.

Needless to say, a data function can be anything we want it to be. This particular example happens to return the current temperature.

The <DATA> tag is how we send dynamic data from the server to the client. HTML includes a simple mechanism for sending data from the client to the server. You've no doubt used it many times while surfing the web. It's the <FORM> tag. Take a look at the one in index.html.

This tells the browser to use the POST method to execute an ACTION named "control" to send three text variables when the user presses a "Go" button. Normally the ACTION is a CGI script. That's beyond the scope of this discussion so we'll just implement a simple function for "control".

Slide 26

To build the web server you'll need two more Eclipse make targets following the same pattern we've seen over and over. Run webthermo_s under the debugger on your workstation so you can watch what happens as a web browser accesses it. You must be running as root in order to bind to the HTTP port, 80, indeed any port number below 1024. In the destination window of your favorite browser enter the line shown here.

The web server is not an assignment. It's really just for your information and edification.

Slide 27

To run our simple web server on the target, you have to stop boa. The ps command lists all of the processes that are currently running along with their process IDs, the first number is each line. The grep command filters the output of ps so we can easily find the boa process. Kill the process ID returned by grep. Then start webserve.

Slide 28

That's it for this week. Let's review. We began with a brief introduction to device drivers and to the "miscellaneous" devices that support the hardware on the board. Then we looked at exporting and importing project properties in Eclipse.

Our main topic was network programming using sockets. We found that sockets operate on a simple client/server paradigm. We built a network aware thermostat and looked at issues involving multiple clients.

Finally we looked at embedded web servers.

Next week we'll look at configuring and building the Linux kernel.