Simulation Environment Updates

15-441 Projects 2 and 3, Fall 2003

1 Socket Layer

1.1 Userspace API

Our `Accept()` differs from the standard accept in one significant way. `Accept()` returns 0 (instead of a new file descriptor as in UNIX) upon success, and -1 upon failure. Thus, `Accept()` does not create a new file descriptor (unlike the Berkeley Socket specification), and uses the same file descriptor for the subsequent socket calls.

1.2 Interface to Transport Layer

Herein, we briefly describe what each of the system call handlers in the socket layer does. This information supplements the documentation provided in the "Callbacks" section of the original simulator handout. If your question is unanswered here, please check if you question is answered there.

Note that when we say "calls the transport-specific handler", it is implicit that if no such handler exists, then the function returns `ENOTSUP` to the user.

`Socket()` Allocates a file descriptor for the socket, and then calls the transport-specific handler. If the transport-specific handler returns an error, the socket is destroyed. Otherwise the socket is returned to the user.

`Close()` Calls the transport-specific handler.

`Bind()` Verifies that the address length argument to the call is correct, and then calls the transport-specific handler.

`Connect()` Verify that the address is not `NULL`, and that the address length is correct. Then call the transport-specific handler.

`Accept()` Verify that the from address length is correct, then call the transport-specific handler.

`Sendto()` Verifies that the address length argument to the call is correct, and then calls the transport-specific handler.

`Write()` Calls the transport-specific handler.

`Recvfrom()` First verifies that the from address length argument is correct. If the receive buffer is empty, and the `MSG_NOBLOCK` flag is set in the arguments, return immediately with no data. If the receive buffer has data, give data to the user, along with the address of the sender of that data. Otherwise, sleep until the receive buffer has data, and then return that data and the sender's address. Before returning any data to the user, call the transport-specific handler, indicating the number of bytes that were read, and whether or not a packet was removed from the receive buffer.

`Read()` As with `Recvfrom()`, except that the call always blocks for data, and the address of the sender is not returned.

`Setsockopt()` Calls the transport-specific handler.

1.3 `so_state` field of `struct socket`

This field is used by the socket layer internally. Your transport layer should not use or modify it.

1.4 `enqueue_data()`

The transport layer should call `enqueue_data()` to place a packet in the receive buffer for a socket. Ownership of the buffer passes to the socket layer. The socket layer will free the buffer after the data has been read. Accordingly, the transport layer should not modify or free the buffer after calling `enqueue_data()`. Additionally, the transport layer must not pass stack memory to `enqueue_data()`. The program behavior after doing so would be undefined.

2 Concurrency

The simulator uses threads to handle the concurrency inherent in network communication. Specifically, the simulator maintains one thread per user application, one thread per network interface, and a timer thread.

The simulator guarantees that only one thread is runnable at any given time. Hence a thread is never pre-empted. Consequently, functions run until they complete, or they voluntarily suspend their thread of execution (see "Blocking and waking up processes" in the original simulator handout for details on how to do this).

The per-application threads are used to execute system calls. Note that when an application calls invokes a system call, (e.g. `Read()`), the application code is suspended until the system call completes.

The per-interface threads are used to receive data from the network. When the simulator calls your input function (e.g. `labelfwd_input()`), that function executes in the context of the thread for the interface on which the data was received.

The timer thread is used to implement timers. When the simulator calls a timer callback, the callback executes in the context of the timer thread.

3 Link Layer

The MTU is given by `ETHERNET_MTU`, defined in `if.h`. The link layer will reject packets large than this size.

4 Limitations

Given the semantics of our `Accept()` call, and the lack of a `Select()` call, it is infeasible for a single application process running on our simulator to service multiple connections in a reasonable way. Thus you should not attempt to do this.

Note: we can still test multiple connections using multiple application processes. Thus this limitation *does not* change the project requirements.