Simulation Environment Overview

15-441 Project 2, Spring 2007

1 Overview

In this document, we describe the simulation environment which you will be using in project 2 for this class. The simulator implements the basic components of an operating system kernel, as well as the socket, transport, link and physical layers. You will be responsible for adding network layer to the kernel. The details of your project assignments can be found in the project handout.

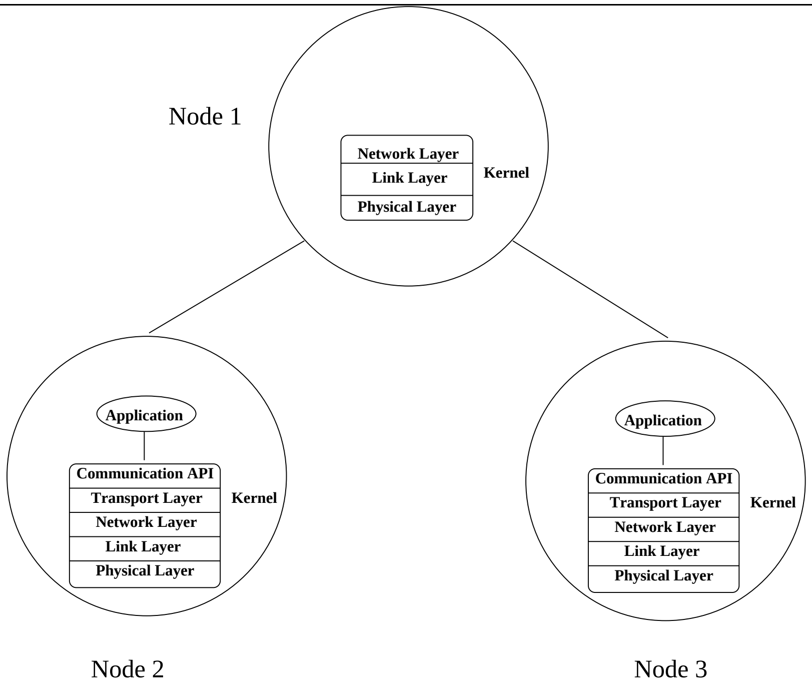

Figure 1: Logical view of Simulator: Applications run on simulated nodes.

Figure 1 shows a logical picture of a sample simulated network, whereas Figure 2 shows the real picture. In the logical view of the simulator, each node has its own operating system kernel, and the applications on the node run on this kernel. In reality, however, each node in the network is a separate UNIX process running on the real OS kernel. An application running on top of a node is a UNIX process separate from the kernel process. The fact that each node is implemented as a separate process enables you to simulate communications between nodes even though all the nodes are actually running on the same machine. Applications are implemented as separate processes so that they can be started after the simulation is already running (i.e. the kernel on each node is running) and so that more than one application can be run on the same node.

In the real world, user applications invoke kernel services via special a special 'trap' instruction which suspends execution of the user program and switches to executing the kernel. The kernel

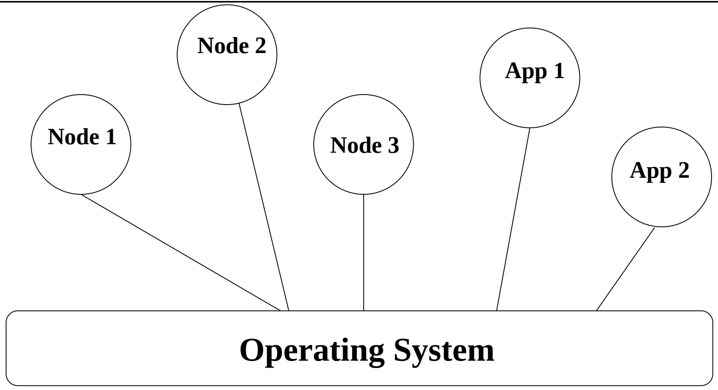

Figure 2: Real view of Simulator: Each node is implemented as a separate UNIX process. Each application running on a node is in a separate process.

can read and write the user's memory to fetch system call parameters and store the results of the system call. In the simulator, user applications and the kernel on the nodes communicate using Inter-Process Communication (IPC) primitives. For each user process that belongs to a kernel, the kernel creates a thread to handle system call requests from the user process.

Each node has its own operating system kernel. Some nodes utilize all the layers of the network stack implemented in your kernel, and there are applications running on top of them (Nodes 2 and 3 in the figure). These nodes represent end-systems or communication endpoints. Other nodes, e.g. Node 1, only use the physical, link and network layers of the network stack. These nodes are routers. They are only responsible for forwarding packets, and since forwarding is a function provided by the network layer, they do not need to use the layers above the network layer. Endpoints on the other hand, do need to have all layers of the network stack since packets that are sent and received by the application layer need to undergo processing by all layers below the application layer.

In this handout, we will use `$PDIR` to denote the project directory. The project directory for Project 2 will be: `/afs/andrew.cmu.edu/course/15/441-sp07/project2/`.

2 Building the kernel and running a network simulation

The support code for your projects provides an environment that emulates a simple machine with hardware-level network devices and a system call interface. The support code also includes a socket layer and a simple transport layer implementation. The support code is provided to you as a set of libraries: `libkernel.a`, and `libuser.a`. `libkernel.a` is to be linked with your network layer code to build a kernel. `libuser.a`, is to be linked with the applications that run on your kernel.

When your simulated kernel boots, the support code will initialize its data structures, such as those representing the "hardware", and then call the `kernel_init()` routine. The `kernel_init()` routine provided in the templates includes code for initializing the transport layer. In this function, you will add any initialization code that is necessary for your portion of the kernel. This would include things like telling the support code which function it should call when it receives a packet, and telling the support code which functions it should call when the user program wants to send data over the network. (We discuss both in more detail later.)

You will be using the simulator to simulate a network. Typically, a network consists of more than one node (otherwise it is not very interesting). A sample network configuration is shown in Figure 3.

A script `$PDIR/kernel/startkernel.pl` will be provided to help you bring your network up when you start the simulation. This script reads a network configuration file (see Section 2.2)

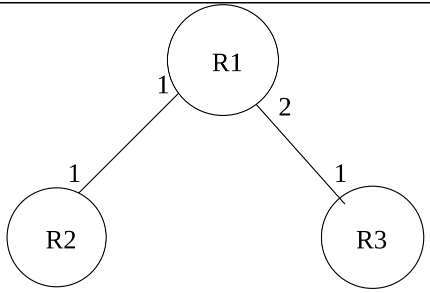

Figure 3: A sample network configuration.

that you specify, and launches the appropriate number of kernels. Each kernel is started in its own `xterm` window. An optional second argument (`debug`) may be specified to `startkernel.pl` so that it runs each kernel within `gdb`. If you don't specify this option, problems may be difficult to debug since when a kernel crashes, the xterm window corresponding to that kernel will close.

2.1 Debug output

The kernel can be supplied with a `-v` flag on startup in order to control the level of debug output it provides. The `-v` option can be specified with or without a number after it. If no number is specified after the `-v` flag, full debugging output is enabled. Be aware that there is no space between the `-v` and the number. This is not a mistake. Note that the number after the `-v` flag is actually a bitwise or of several possible flags.

```
#define VERBOSE_SYS  1            // print syscall entry and exit
#define VERBOSE_PATH 2            // print paths taken by packets
#define VERBOSE_ERROR 4           // print extra error information
#define VERBOSE_OTHER 8
#define VERBOSE_ALL 0xffffffff
```

For example, executing `./kernel -f network.cfg -n 1 -v3` would output information relating to both syscall entry and exit as well as the path taken by all packets. Also, if the `-v` flag is supplied to `./startkernel.pl`, `./startkernel.pl` will forward the `-v` flag to the kernels.

2.2 Network configuration file

As mentioned above, you need to create a network configuration file to run a simulation. This configuration file specifies each node in the network along with all of its interfaces and their respective addresses, as well as all the links that exist between each node and other nodes in the network.

We use the network from Figure 3 to illustrate how network configuration files are built. Interface 1 on node R1 is connected to interface 1 on node R2, and interface 2 on node R1 is connected to interface 1 on node R3.

The network configuration file for this network is the following:

```
# Configuration for Router 1
Router 1 {
     1  1.1.1.1  255.255.255.0
     2  1.1.2.1  255.255.255.0
     1:1 2:1
     1:2 3:1
}

# Configuration for Router 2
```

```
Router 2 {
     1  1.1.1.2  255.255.255.0
     2:1 1:1
}

# Configuration for Router 3
Router 3 {
     1  1.1.2.2  255.255.255.0
     3:1 1:2
}
```

As usual, lines that start with a "#" are comments and will be ignored by the simulator. The configuration file is comprised of a number of clauses, one for each node in the network (i.e. Router 1, Router 2, ...). The clause for a node begin with a description of the interfaces on that node. For each interface, we specify the interface number (which must be greater than zero, and less than 17), the IP address, and the netmask.

After we have described the interfaces for a node, we describe how these interfaces are connected to other nodes. The notation X:Y refers to interface Y on node X. Thus, the line "`1:1 2:1`" in the configuration entry for node R1, shown above, specifies that interface 1 on R1 should be connected to interface 1 on R2. For this course, all links will be point-to-point. Hence you should make sure not to connect a single interface to multiple remote interfaces.

Note that in this configuration, R2 and R3 are actually end points, not routers. However, the simulator requires the word "Router" for each node in the configuration file.

This sample configuration file is provided in `$PDIR/kernel/network.cfg`. You can modify the sample or create your own configuration for testing purposes.

3 Building and running user programs

User programs run on the simulated nodes. Each user program is run as a separate user process as shown in Figure 1. All user programs used with the simulator must be linked against the user library (`libuser.a`) we provide (see the template `Makefile` in `$PDIR/utils` for more details). In most respects, the user programs that run with the simulator are just like user programs that run with the OS's network stack. There are, however, three important differences:

1. The entry point for the user programs must be named `Main()` instead of `main()`. Our support code defines `main()`. After the support code has completed its initialization, it will invoke your `Main()` function. The interface for `Main()` function is exactly the same as `main()`. That is, the usual `argc` and `argv` are still there.

2. User programs must be run with "`-n` *`i`*" as the first argument. This argument is to specify that this user program should be run on node *`i`*. Note that the `Main()` function will *not* see this argument (i.e. the simulator will strip this argument before calling `Main()`).

3. Calls to the socket API must use capitalized names rather than standard names. For example, when your user program wants to create a socket, it must call `Socket()` rather than `socket()`.

4. The user program must be single-threaded. The kernel's system call handling model does not support multi-threaded user programs.

4 Interacting with the link layer

In your projects you will be adding a network layer to the simulator. The network layer transmits and receives packets from the network with the help of the link layer. In this section, we describe the interface between the link layer and the network layer.

4.1 Initialization

Before your network layer can receive any packets from the link layer, you must tell the link layer which function it should call when packets arrive. To do so, use hw_interfaces_register(). The prototype for this function is given in $PDIR/include/hw_interfaces.h.

4.2 The network interface list

As explained earlier, the kernel boot code reads the network configuration file (Section 2.2) and creates a list of networking interfaces on the node. In this subsection, we describe this data structure, in case your network layer needs to access it.

Each element on this list is a struct ifnet defined in $PDIR/include/if.h:

```
struct ifnet {
  TAILQ_ENTRY(ifnet)      if_next;

  int                     if_index;       /* interface number */
  struct sockaddr_in      if_addr;        /* address of interface */
  struct sockaddr_in      if_netmask;     /* netmask of if_addr */
  int                     if_mtu;         /* MTU of interface */

  void (*if_start)(struct ifnet *ifp, struct pbuf *p);

  pthread_mutex_t         if_mutex;       /* Lock for accessing outgoing
                                           * interface on this device */
  struct hwif             *if_hwif;       /* hardware device */
};
```

The head of this list can be accessed by calling the function ifnet_listhead() provided by the simulator. The TAILQ_ENTRY() macro is a macro defined in $PDIR/include/queue.h that is useful for creating linked lists. Iterating over the interface list can be done as follows:

```
struct ifnet *ifp = ifnet_listhead();

for( ; ifp; ifp = TAILQ_NEXT(ifp, if_next)) {
  printf(‘‘interface index: %d\n’’, ifp->if_index);
}
```

4.3 Handing packets to the network interface for transmission

Once your forwarding layer has completely built a packet and has determined which interface the packet should be sent out on, the forwarding layer can send this packet by calling the if_start() routine of the appropriate interface. (The prototype for if_start() is given in $PDIR/include/if.h). For example, if your forwarding layer has consulted the forwarding table, and determined that the current packet should be forwarded through interface 1, you would do the following:

```
struct ifnet *ifp;
struct pbuf *pkt;                          /* packet to be sent */

/* ifp = code to find interface 1 here */

ifp->if_start(ifp, pkt);     /* send the packet */
```

Note that the link layer will free the buffer after it has finished transmission of the packet, whether transmission succeeds or not. For this reason, must not free the buffer yourself after passing it to the link layer.

4.4 Getting packets received by the network interface

Assuming you have initialized the link layer properly, the link layer will call one of your functions (call it the "input handler") whenever a network interface receives a packet from the network. As indicated by the prototype of the initialization function (hw_interfaces_register()), the link layer will call your input handler with three arguments: a struct ifnet indicating on which interface the packet was received, a struct pbuf * pointing to the packet. Note that your code is responsible for freeing the buffer, in case of any errors. You may assume that once a packet leaves the IP layer for the transport layer, the transport layer will be responsible for freeing it.

4.5 The pbuf structure

A packet sent or received by an application is processed by several different layers in the network stack. In real BSD-style implementations, an mbuf structure is used for passing the packet between the different layers. In projects 2, you will be using a pbuf structure for building and passing packets between network stack layers. The pbuf structure is simplified version of the BSD mbuf.

The definition of the pbuf structure is the following (given in $PDIR/include/pbuf.h):

```
struct p_hdr {
           struct  pbuf *ph_next;    /* next buffer in chain */
           struct  pbuf *ph_nextpkt; /* next chain in queue/record */
           caddr_t ph_data;          /* location of data */
           int     ph_len;           /* amount of data in this mbuf */
           int     ph_type;          /* type of data in this mbuf */
           int     ph_flags;         /* flags; see below */
   };

   struct pbuf {
           struct p_hdr p_hdr;
           char         p_databuf[PHLEN];
   };
   #define p_next    p_hdr.ph_next
   #define p_nextpkt p_hdr.ph_nextpkt
   #define p_data    p_hdr.ph_data
   #define p_len     p_hdr.ph_len
   #define p_type    p_hdr.ph_type
   #define p_flags   p_hdr.ph_flags
   #define p_dat     p_databuf
```

The pbuf's must be allocated and deallocated using the routines p_get() and p_free() declared in $PDIR/include/pbuf.h. Since a pbuf contains less than 512 bytes of data (PHLEN is defined as 512 minus header length), an MTU-sized packet (1500 bytes in your projects) will consist of 4 pbuf structures linked together by the p_next field in each pbuf – this is called a pbuf chain. The p_nextpkt field can be used to link multiple packets together on a queue. By convention, only the first pbuf in a pbuf chain should be used to link to another pbuf chain (through p_nextpkt).

The field p_data points to the location where the packet data starts within the p_databuf buffer. Why implement pbufs this way? Suppose your transport layer has built a UDP packet with 20 bytes of data and an 8-byte UDP header. Before this packet gets sent on the wire, it will have to go through network and link layer processing. If you place the data at the beginning of the pbuf, the network layer will have to allocate a new pbuf in which to store the 20-byte IP header and prepend this pbuf to the packet. However, if you were clever enough to leave 20 bytes of space at the beginning of the p_databuf buffer, you could simply subtract 20 from the value of p_data and then copy the 20-byte IP header to the address indicated by this pointer. An example of a packet consisting of multiple pbuf structures is shown in Figure 4.

The field p_len is the length of data contained in the pbuf; it is not the total length of the packet. p_type is managed by the pbuf allocation code and p_flags is presently not used at all by the kernel.

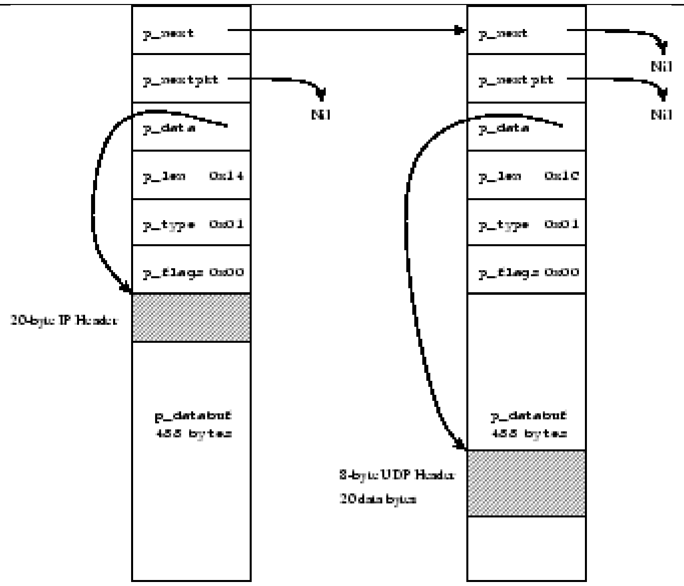

Figure 4: pbuf: A 48-byte IP packet spread out over 2 pbuf structures. There is a 20-byte IP header, an 8-byte UDP header, and 20-bytes of user data. The IP header starts at the beginning of the first pbuf's p_databuf, while the UDP header and data bytes start in the middle of the second pbuf's p_databuf. Placing data in the middle of p_databuf and modifying p_data to point to it is a clever way to leave space for headers, or to push and pop headers, without requiring additional pbufs.

The prototype for `struct pbuf` and other utility functions are given in `$PDIR/include/pbuf.h`. Some of the routines which you might find useful in creating your network layer are `p_strip()`, `p_prepend()`, `p_get()`, `p_free()`, `p_pktlen()`, `p_freep()`, and `p_copyp()`.

5 Interacting with the transport layer

The transport layer sits between the socket layer and the network layer. The support code provides a simple implementation of TCP and UDP. In this section, we describe the interface between the transport layer and the network layer.

5.1 Handing packets to the transport layer

Once your forwarding layer has decided that the packet is destined to *this* host, it should strip off the IP header and send the rest of the packet to the appropriate transport layer. For TCP packets, the `protocol` filed in the IP header is set to `IPPROTO_TCP`, and for UDP packets, the field is set to `IPPROTO_UDP`.

As of now, the only transport layer implemented is UDP. A UDP packet should be passed to the UDP layer by calling the `udp_receive()` routine. The prototype for `udp_receive()` is given in `$PDIR/include/udp.h`. TCP packets should be silently dropped and freed for now.

5.2 Getting packets from the transport layer

When a user program wants to transmit data, the transport layer will receive the data through the socket layer. The transport layer will then pass the packet to the forwarding layer by calling the `ip_output()` routine. The prototype for `ip_output()` is given in
`$PDIR/kernel/ipforward.h`.

You must implement the ip_output() routine. Your ip_output() routine should prepend an IP header with the fields set appropriately, and then send the packet on the appropriate interface after looking up the forwarding table.

Note that if the IP_NOROUTE bit is set in the flags parameter, the behavior of ip_output() changes significantly. Instead of looking up the forwarding table to find out which interface to send the packet on, it looks up the network interface list described in section 4.2. It uses the source IP address and the netmask of each interface along with the destination IP address to choose which interface to send the packet on. As explained in the project handout, your routing daemon will use this option.

6 Socket API

The socket layer provides an API (application program interface) for user programs to access the networking functionality of the kernel. For user programs to interface to the simulator, you can use the socket API. The prototypes are defined in $PDIR/include/Socket.h (this header file should be included by user programs, not your kernel).

Observe that the first letter of each call is *capitalized*. This is to distinguish them from the actual Linux system calls, which will go into the Linux kernel upon invocation. All your user programs will be linked against a library provided by us (libuser.a) so that when they invoke these capitalized calls, the corresponding handlers in our simulated kernel (and not the Linux kernel) are invoked.

The simulator Socket API supports Socket(), Close(), Bind(), Connect, Accept(), Read(), and Write() functions for TCP. Similarly, it supports Socket(), Close(), Sendto(), Recvfrom() and Setsockopt() functions for UDP. Note: Unlike an operating system kernel, the simulator has no way of cleaning up after a user process once it exits. Please make sure to always Close() your socket descriptors before exiting to recycle simulator kernel resources.

6.1 The Socket() call

The Socket() call accepts three arguments: *family, type, and protocol*. It supports the following three combinations of family and type: (1) AF_INET/SOCK_STREAM: this combination specifies that the user wants to create a TCP socket, (2) AF_INET/SOCK_DGRAM: this combination specifies that the user wants to create a UDP socket, and (3) AF_ROUTE/SOCK_RAW: this combination specifies that the user wants to create a routing socket.

6.2 The Accept() call

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.

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.

6.3 The Recvfrom() call

By default, Recvfrom() is blocking: when a process issues a Recvfrom() that cannot be completed immediately (because there is no packet), the process is put to sleep waiting for a packet to arrive at the socket. Therefore, a call to Recvfrom() will return immediately only if a packet is available on the socket. When the MSG_NOBLOCK bit is set in the flags argument of Recvfrom(), Recvfrom() does not block if there is no data to be read, but returns immediately with a return value of -1, and setting errno to EAGAIN. MSG_NOBLOCK is defined in $PDIR/include/systm.h.

You can find some user level programs written using the Socket API in `$PDIR/utils`.

7 Routing

In order to forward packets, your forwarding layer will need to know which packets will be sent through which links. The simulator provides a way for user space programs to provide the forwarding information to the kernel. As you are responsible for implementing the kernel forwarding code, this section describes the interface that user programs will use to provide the kernel with forwarding information.

These programs communicate with your kernel via a "routing socket". The user programs will call `Socket(AF_ROUTE, SOCK_RAW, 0)` to obtain the routing socket. They will then add entries to the forwarding table by writing messages to the routing socket.

The format of the messages written by the user programs is defined in `$PDIR/include/route.h`, and given below. The user programs will write a message of type `struct rt_msghdr` to the routing socket.

```
struct rt_info {
    struct sockaddr_in rti_dst;         /* destination, only sin_addr.s_addr
                                           field is used in project */
    u_int32_t          rti_index;       /* interface index */
};


struct rt_msghdr {
    u_int16_t   rtm_msglen;
    u_int16_t   rtm_type;       /* Message Types */
    u_int32_t   rtm_errno;      /* set by the kernel, if error */
    struct rt_info     rtm_rti;        /* routing info */

};

/* Message Types */
#define RTM_ADD         0x001     /* Add Route */
#define RTM_DELETE      0x002     /* Delete Route */
#define RTM_CHANGE      0x003     /* Change Metrics or flags */
```

The following values of the `rtm_type` field of the `rt_msghdr` structure are supported: (1) `RTM_ADD`: add an entry to the routing table, (2)`RTM_DELETE:` delete an entry from the routing table, and (3)`RTM_CHANGE:` change an entry in the routing table.

You can find an example user space program (`$PDIR/utils/fdconfig.c`) which uses routing sockets to provide forwarding information to the kernel.

8 Kernel utility functions

Here we describe some utility functions provided by our simulated kernel.

8.1 How to `panic`

The simulated kernel provides the function `panic(char * fmt, ...)`, which causes the kernel to immediately stop running and print out the message passed to it as an argument.

8.2 Using mutexes

We have provided macro wrappers around pthread's mutex functions in `sync.h`. These macros perform error checking and output debugging information if you enable debugging by compiling your code with flags `-DDEBUG` and `-D_GNU_SOURCE`. (Note: the error checking functionalities are provided by error-checking mutexes, which are Linux's extensions to the POSIX standards. Therefore, error-checking mutexes are not portable and only avaliable on Linux.)

When using mutexes for your project, please use what we provided in `sync.h` and not invoke the pthread's mutex functions directly.

8.2.1 Declaration

All mutexes have type `mutex_t`.

8.2.2 Initialization

There are two ways to initialize a mutex, 1) statically at compile time, or 2), dynamically at run time. See the code below:

```
mutex_t myMutex = MUTEX_INITIALIZER; /* Static initialization */

mutex_t myOtherMutex;

...

void myOtherInit(void){

    int r;

    /* mutex_init() takes a pointer to a mutex to be initialized,
       and returns 0 on success, non-zero on failure. */
    r = mutex_init(&myOtherMutex);

    if (r != 0) {
        panic("Failed to initialize myOtherMutex");
    }
}
```

Depending on whether debugging is enabled, the mutex is either initialized as a "fast" mutex, or an "error-checking" mutex.

8.2.3 Locking, unlocking, and destroying

For locking and unlocking mutexes, we provide the macros `MUTEX_LOCK` and `MUTEX_UNLOCK`. When you are finished using a mutex, the mutex should be destroyed with the macro `MUTEX_DESTROY`. These macros take a `mutex_t *` as input. Note: These macros will cause the kernel to panic if the mutex operations fail. Locking and unlocking should never fail, and destroying a mutex will fail only if the mutex is already locked. A failure while performing any of these operations indicate errors in the code; therefore, it is desirable for the kernel to panic on any failures.

If debugging is enabled, these macros will also print out a trace of locking and unlocking to `stderr` showing the the mutexes and the functions trying to lock/unlock them.