4.5 Mach Include Files

When writing a program that uses Mach facilities some of the following include files may be needed:

```
mach.h : needed for all Mach programs
mach/message.h : if any message structures are used
mach_error.h : if mach_error() is used
servers/env_mgr.h : if environment manager server is used
servers/netname.h : if netname server is used
cthreads.h : if C threads package is used
```

4.6 Mach Information/Questions

A Mach news group exists to keep the user community up to date on new Mach releases. This news group is called 'comp.os.mach'.

If you have questions relating to the use of any Mach facilities, or any comments on this tutorial, send mail to `machlib@wb1.cs.cmu.edu`. Comments on this tutorial will be greatly appreciated.

4 General Mach Information

4.1 Structure of the Mach Tree

Directories of interest include:

```
/usr/mach/man : Mach manual pages
/usr/mach/bin : Mach system programs such as MiG
/usr/mach/etc : Mach servers such as the environment manager
/usr/mach/include : Mach include files
/usr/mach/lib : Mach libraries such as libmach.a and libthreads.a
```

4.2 Where to Find Examples and Manuals

You can print copies of the Mach manuals and documents for yourself using the .ps files in the `/usr/mach/doc` directory. There is subdirectory `examples` in `doc` which contains the example programs used in the two tutorial documents. See the `lpr` UNIX manual entry on how to print .ps files.

4.3 Setting Up Search Paths

To compile or load a Mach program your paths must be set to look in the proper place for Mach include files and libraries. By typing `source /usr/mach/lib/machpaths` the appropriate Mach directories will be added to your existing paths.

4.4 Mach Libraries

The Mach library, -lmach, must be loaded with most of the examples given in this tutorial. Any program using Mach kernel interface functions must be loaded with libmach and the Mach version of `crt0`. For example, assuming that /usr/mach/lib is on your LPATH before any other version of `crt0`:

```
cc -o simp_ipc simp_ipc.c -lmach
```

Two threads libraries exist: libthreads.a and libco_threads.a. The 'co' version of this library works on all current Mach kernels. This version does not actually use Mach threads; it uses coroutines. This library does not provide real parallelism, a single UNIX process is running. The other threads library, libthreads.a, uses Mach threads.

```
        {
                j = ((int)random()%range) + (int)' ';
                password[i] = (char)j;
        }
        password[25] = '\0';
        return(KERN_SUCCESS);
}
```

3.5 User Side

The following code shows an example of how to use the server.

```
/* This is an example of a program using the random server */
#include <mach.h>
#include <servers/netname.h>
#include "random.h"
#include <mach_error.h>
#include <stdio.h>


main()
{
    int          number;
    string25     password;
    port_t       serv_port;
    kern_return_t retcode;

        printf("Looking up port for random server\n");
        retcode = netname_look_up(name_server_port,"","RandomServerPort",
                        &serv_port);
        if (retcode != KERN_SUCCESS)
        {    mach_error("error in looking up port for random server",retcode);
             printf("Random server may not be running\n");
             exit();
        }
        printf("Calling get_random\n");
        retcode = get_random(serv_port,&number);
        if (retcode == KERN_SUCCESS)
                printf("Result from get_random is %d\n",number);
        else mach_error("Error from get_random is",retcode);
        printf("Calling get_secret\n");
        retcode = get_secret(serv_port,password);
        if (retcode == KERN_SUCCESS)
                printf("Result from get_secret is %s\n",password);
        else mach_error("Error from get_secret is",retcode);
}
```

```
        msg_simple = In0P->Head.msg_simple;
        if ((msg_size != 24) || (msg_simple != TRUE))
                { OutP->RetCode = MIG_BAD_ARGUMENTS; return; }
    #endif TypeCheck

        OutP->RetCode = get_random(In0P->Head.msg_request_port, &OutP->num);
        if (OutP->RetCode != KERN_SUCCESS)
                return;

        msg_size = 40;

        OutP->numType = numType;
        OutP->Head.msg_simple = TRUE;
        OutP->Head.msg_size = msg_size;
    }
```

3.4 Server procedures

Finally the subsystem implementor must write the proceedures that actually perform the requested operations. For this server the following code will do.

```
/* procedues of random_server */
#include <mach.h>
#include "random_types.h"

long    random();


kern_return_t get_random(serv_port,num)
/* get_random returns a random number between 0 and 2**32 - 1 */
    port_t      serv_port;
    int         *num;
{
        *num = (int) random();
        return(KERN_SUCCESS);
}

kern_return_t get_secret(serv_port,password)
    port_t      serv_port;
    string25    password;
/* get_secret returns a random printable ascii string 25 chars long */
{
    int         i,j;
    int         range = (int)'~' - (int)' ';

        for (i=0;i<25; i++)
```

```
        return TRUE;
    }
```

The two internal routines that do most of the message unpacking are `_Xget_random` and `_Xget_secret`. These routines respectively call `get_random` and `get_secret` with the appropriate parameters. When those routines return, a reply message is created in the buffer that `OutHeadP` points to.

The following is the code for `_Xget_random`;

```
    /* Routine get_random */
    mig_internal novalue _Xget_random(InHeadP, OutHeadP)
        msg_header_t *InHeadP, *OutHeadP;
    {
        typedef struct {
                msg_header_t Head;
        } Request;

        typedef struct {
                msg_header_t Head;
                msg_type_t RetCodeType;
                kern_return_t RetCode;
                msg_type_t numType;
                int num;
        } Reply;

        register Request *In0P = (Request *) InHeadP;
        register Reply *OutP = (Reply *) OutHeadP;
        extern kern_return_t get_random();

    #if TypeCheck
        boolean_t msg_simple;
    #endif TypeCheck

        unsigned int msg_size;

        static msg_type_t numType = {
                /* msg_type_name = */           MSG_TYPE_INTEGER_32,
                /* msg_type_size = */           32,
                /* msg_type_number = */         1,
                /* msg_type_inline = */         TRUE,
                /* msg_type_longform = */       FALSE,
                /* msg_type_deallocate = */     FALSE,
                /* msg_type_unused = */ 0
        };
    #if TypeCheck
        msg_size = In0P->Head.msg_size;
```

3.3 Server message dispatch code

The file randomServer.c contains the server side code that was generated by MIG. It exports the routine random_server that is called by the main program. It checks the message id to determine what message was received, and then unpacks the arguments and calls the appropriate server procedure. The following fragment shows the control logic of the dispatch routine.

```
boolean_t random_server(InHeadP, OutHeadP)
    msg_header_t *InHeadP, *OutHeadP;
{
    register msg_header_t *InP =  InHeadP;
    register death_pill_t *OutP = (death_pill_t *) OutHeadP;

    static msg_type_t RetCodeType = {
            /* msg_type_name = */           MSG_TYPE_INTEGER_32,
            /* msg_type_size = */           32,
            /* msg_type_number = */         1,
            /* msg_type_inline = */         TRUE,
            /* msg_type_longform = */       FALSE,
            /* msg_type_deallocate = */     FALSE,
            /* msg_type_unused = */ 0
    };

    OutP->Head.msg_simple = TRUE;
    OutP->Head.msg_size = sizeof *OutP;
    OutP->Head.msg_type = InP->msg_type;
    OutP->Head.msg_local_port = PORT_NULL;
    OutP->Head.msg_remote_port = InP->msg_reply_port;
    OutP->Head.msg_id = InP->msg_id + 100;
    OutP->RetCodeType = RetCodeType;
    OutP->RetCode = MIG_BAD_ID;

    if ((InP->msg_id > 501) || (InP->msg_id < 500))
            return FALSE;
    else {
            typedef novalue (*SERVER_STUB_PROC)();

            static SERVER_STUB_PROC routines[] = {
                    _Xget_random,
                    _Xget_secret,
            };

            if (routines[InP->msg_id - 500])
                    (routines[InP->msg_id - 500]) (InP, &OutP->Head);
             else
                    return FALSE;
    }
```

```
        msg_header_t *request = (msg_header_t *) requestbuf;
        death_pill_t *reply = (death_pill_t *) replybuf;
        msg_return_t mr;

        /*
         *  Problems with this server loop:
         *      Requests which are not processed successfully
         *      (bad msg_id, type mismatch, whatever) should
         *      be cleaned up; ports & memory should be deallocated.
         *
         *      Replies which are dropped (reply port died or was
         *      full, some problem with rights or memory in the reply)
         *      should also be cleaned up.
         *  But these are hard problems (harder than they might appear)
         *  so we ignore them.
         */

        for (;;) {
                /* receive a request message */

                request->msg_size = sizeof requestbuf;
                request->msg_local_port = service;

                mr = msg_receive(request, MSG_OPTION_NONE, 0);
                if (mr != RCV_SUCCESS)
                        return mr;

                /* ignore notification messages from the kernel */

                if (request->msg_local_port == task_notify())
                        continue;

                /* demux and process the request, generating a reply */

                (void) (*function)(request, &reply->Head);

                /* send the reply, if necessary */

                if ((reply->Head.msg_remote_port != PORT_NULL) &&
                    (reply->RetCode != MIG_NO_REPLY)) {
                        /* don't block if the reply port is full */

                        (void) msg_send(&reply->Head, SEND_TIMEOUT, 0);
                }
        }
}
```

```
        kr = port_set_add(task_self(), port_set, server_port);
        if (kr != KERN_SUCCESS) {
                mach_error("port_set_add", kr);
                exit(1);
        }

        kr = port_set_add(task_self(), port_set, task_notify());
        if (kr != KERN_SUCCESS) {
                mach_error("port_set_add", kr);
                exit(1);
        }

        /* check service port into the name service so clients can find it */

        kr = netname_check_in(name_server_port, "RandomServerPort",
                              PORT_NULL, server_port);
        if (kr != KERN_SUCCESS) {
                mach_error("netname_check_in", kr);
                exit(1);
        }

        /* call the standard service loop; should never return */

        kr = mig_server(port_set, random_server);
        mach_error("mig_server", kr);
        exit(2);
}
```

The following code is the library routine `mig_server`.

```
#include <mach/mach.h>
#include <mach/message.h>
#include <mach/mig_errors.h>

msg_return_t
mig_server(service, function)
        port_name_t service;            /* receive right or port set */
        boolean_t (*function)();        /* server demux & processing */
{
        /*
         *      Buffers should be aligned on 4-byte boundaries,
         *      so that internal fields are aligned properly
         *      for int fetches and stores.
         */
        int requestbuf[MSG_SIZE_MAX/sizeof(int)];
        int replybuf[MSG_SIZE_MAX/sizeof(int)];
```

This server may get some `EMERGENCY_MSGs` from the kernel on its `notify_port()` which it wishes to ignore. The most common sort of `EMERGENCY_MSG` is the notification of a port death. In the case of a server these ports are usually client reply ports. The library routine `mig_server` ignores all messages on the `notify_port()`. A more complicated server might need to take action on some of these messages and would not be able to use
t mig_server.

Note that the function `mach_error_string` is a library routine that returns the string associated with a Mach error code. This function is defined in `mach_error.h` and is included in `libmach.a`.

The following code shows a typical main loop for a MIG server.

```
/****************************************************
 *   Main program for random server
 ***************************************************/

#include <stdio.h>
#include <mach.h>
#include <mach_error.h>
#include <mach/message.h>
#include <servers/netname.h>

extern boolean_t random_server();       /* from randomServer.c */
extern msg_return_t mig_server();       /* from mig_server.c */

main()
{
        port_name_t server_port;
        port_name_t port_set;
        kern_return_t kr;

        /* allocate a service port for receiving request messages */

        kr = port_allocate(task_self(), &server_port);
        if (kr != KERN_SUCCESS) {
                mach_error("port_allocate", kr);
                exit(1);
        }

        /* allocate a port set to hold the service port and notify port */

        kr = port_set_allocate(task_self(), &port_set);
        if (kr != KERN_SUCCESS) {
                mach_error("port_set_allocate", kr);
                exit(1);
        }

        /* put the service port and notify port into the port set */
```

different messages: one for the get_random function which will have a msg_id of 500; one for the reply to get_random which will have a msg_id of 600; one for the function get_secret which will have a msg_id of 501; and one for its reply which will have a msg_id of 601. The msg_id is used by the server to identify which message it has received.

The file `mach/std_types.defs` defines some frequently used types. In this case the relevant defintions are:

```
type int         = MSG_TYPE_INTEGER_32;
type port_t      = MSG_TYPE_PORT;
type boolean_t   = MSG_TYPE_INTEGER_32;
```

The MIG defs file is processed by `cpp` before it is processed by MIG. Thus cpp strips off the comments and handles the `#include` directives.

The syntax of the type declaration is to define the C type name first and then say that it is equal to an IPC type name. The set of defined IPC types can be found in the MIG document or in the file `<mach/message.h>`. For string and unstructured types the number of *bits* in the type must follow the name.

The import statement gives the name of a header file to be included in the generated code. This header file must define any non-standard C types used by the interface. In this case it consists of the following definition:

```
typedef char    string25[25];
```

The routine declarations specify the name of the routine and the order and types of the arguments. The first argument is the port to which the message will be sent. The specification `in`, `out` or `inout` may precede the name of any other parameter and specify in what direction the argument is to be passed. Any unspecified parameter, except the first one, is assumed to be an `in` parameter.

MIG generates three C files from the definition file. The file `random.h` defines the functions to be called by a client of the server and should be `#included` into code that calls those functions. `randomUser.c` is the code to create and send the messages to the client and then wait to receive the reply message. When the reply message is received, any `out` or `inout` parameters are taken out of the message and returned as function parameters to the caller. This file should be linked with the user of the server. `randomServer.c` is the server side of the message interface. It unpacks the request message, calls a function provided by the server implementor to execute the request and then creates the reply message. The server implementor must write a main program and the functions that execute the requests. There is a library routine, `mig_server`, that provides a simple version of a send and receive loop. In the example, the user writes the code which sets up the ports and then calls `mig_server` to do the rest. If a server needs more control over possible error conditions, it can provide its own receive and send loop.

3.2 Server main program

In this example the server waits on a port_set containing the `ServerPort` and `notify_port()`. The thread reply port is not included in this set. When ports are created they are not a member of any port set by default. Thus they must be added explicitly to `port_set`.

3 MIG - The Mach Interface Generator

Much multi-task communication takes the form of one or more tasks requesting services or responses from another task. This in fact is a description of a Mach server process. Since the creation and reading of messages requires a lot of repetitious code, it should come as no great surpise that Mach provides a compiler to produce a remote procedure call interface to IPC message passing. A complete description of MIG including an example of its use can be found in *MIG - the Mach Interface Generator* by Draves, Jones and Thompson.

A brief example of the use of MIG follows. The problem that is to be solved is to write a simple server that will return either a random integer or a random string. The user interface to this server is to consist of two calls:

```
ret_code = get_random(server_port,num)
       port_t  server_port;
       int     num;

ret_code = get_secret(server_port,password)
       port_t  server_port;
       string25 password;
```

3.1 MIG Definition file

The subsystem implementor must first write a MIG definition file to specify the details of the procdure arguments and the messages to be used. MIG understands different kinds of routines and many obscure options in the way messages are to be formatted, sent and received. But for this simple case it is enough to define the name of the server, the types of the arguments that are being used, and the routines that are desired. The following MIG definition file will suffice to do this:

```
subsystem random 500;

#include <mach/std_types.defs>

type string25   = (MSG_TYPE_STRING_C,8*25);

import "random_types.h";

routine get_random(
                requestport     server_port     : port_t;
                out             num             : int);

routine get_secret(
                requestport     server_port     : port_t;
                out             password        : string25);
```

The first line of the definition file states that the name of the subsytem is to be `random` and the messages that are created will start with the message id of 500. This interface will send four

```
    while (count != 0)
        condition_wait(done, lock);
    mutex_unlock(lock);
    printf("All %d slaves have finished.\n", nslaves);
    cthread_exit(0);
}

main()
{
    init();
    master((int) random() % 16);  /* create up to 15 slaves */
}
```

```
{
    cthread_init();
    count = 0;
    lock = mutex_alloc();
    done = condition_alloc();
    srandom(time((int *) 0));  /* initialize random number generator */
}

/*
 * Each slave just counts up to its argument, yielding the processor on
 * each iteration.  When it is finished, it decrements the global count
 * and signals that it is done.
 */
slave(n)
    int n;
{
    int i;

    for (i = 0; i < n; i += 1)
        cthread_yield();
    mutex_lock(lock);
    count -= 1;
    printf("Slave finished %d cycles.\n", n);
    condition_signal(done);
    mutex_unlock(lock);
}

/*
 * The master spawns a given number of slaves and then waits for them all to
 * finish.
 */
master(nslaves)
    int nslaves;
{
    int i;

    for (i = 1; i <= nslaves; i += 1) {
        mutex_lock(lock);
        /*
         * Fork a slave and detach it,
         * since the master never joins it individually.
         */
        count += 1;
        cthread_detach(cthread_fork(slave, random() % 1000));
        mutex_unlock(lock);
    }
    mutex_lock(lock);
```

2.10 Yielding the Processor to other Threads

This procedure is a hint to the scheduler, suggesting that this would be a convenient point to schedule another thread to run on the current processor. Calls to `cthread_yield` are unnecessary in an implementation with preemptive scheduling, but may be required to avoid starvation in a coroutine based implementation.

```
int i, n;

/* n is set previously */

for (i = 0; i < n; i += 1)
   cthread_yield();
```

2.11 Exiting a C Thread

`cthread_exit` causes termination of the calling thread. An implicit `cthread_exit` occurs when the top level function of a thread returns. The result parameter will be passed to the thread that joins the caller, or discarded if the caller is detached.

```
cthread_exit(0);
```

2.12 Example V, masterslave.c

```
/*
 * This program is an example of a master thread spawning a number of
 * concurrent slaves.  The master thread waits until all of the slaves have
 * finished to exit.  Once created a slave process doesn't do much in this
 * simple example except loop.  A count variable is used by the master and
 * slave processes to keep track of the current number of slaves executing.
 * A mutex is associated with this count variable, and a condition variable
 * with the mutex.  This program is a simple demonstration of the use of
 * mutex and condition variables.
 */

#include <stdio.h>
#include <cthreads.h>

int count;          /* number of slaves active */
mutex_t lock;       /* mutual exclusion for count */
condition_t done;   /* signalled each time a slave finishes */

extern long random();

init()
```

```
mutex_t lock;
condition_t done;

mutex_lock(lock);
...
while (count != 0)
   condition_wait(done, lock);
...
mutex_unlock(lock);
```

2.7 Signalling a Condition

`condition_signal` is called when one thread wishes to indicate that the condition represented by the condition variable may now be true. If any threads are waiting via `condition_wait`, at least one of them will be awakened. If no threads are waiting, nothing happens.

```
condition_t done;  /* signalled each time a slave finishes */
condition_signal(done);
```

2.8 Forking a C Thread

This function takes two parameters: a function for the new thread to execute, and a parameter to this function. `cthread_fork` creates a new thread of control in which the specified function is executed concurrently with the caller's thread. This is the sole means of creating new threads. A parameter that is larger than a pointer must be passed by reference. Similarly, multiple parameters must be simulated by passing a pointer to a structure containing several components. The call to `cthread_fork` returns a thread identifier that can be passed to `cthread_join` or `cthread_detach`. Every thread must be either joined or detached exactly once.

```
/*  slave is a function that expects an integer parameter  */
/*  see Detaching a C Thread for description of chtread_detach  */

cthread_detach(cthread_fork(slave, random() % 1000));
```

2.9 Detaching a C Thread

`cthread_detach` is used to indicate that a thread will never be joined. A thread may be detached at any time after it is forked, as long as no other attempt at joining or detaching has been made. In the example below, at the time the thread was forked, it was known that it would never be joined and therefore it was detached.

```
/*  slave is a function that expects an integer parameter  */
/*  see Forking a C Thread for description of chtread_fork  */

cthread_detach(cthread_fork(slave, random() % 1000));
```

```
cthread_init();
```

2.2 Allocation of a Mutex Variable

`mutex_alloc` provides dynamic allocation of a mutex variable.

```
mutex_t lock;      /* mutual exclusion for count */
lock = mutex_alloc();
```

2.3 Locking a Mutex Variable

`mutex_lock` attempts to lock the given mutex variable. If the mutex is already locked this call blocks until the mutex is unlocked. If several threads attempt to lock the same mutex concurrently, one will succeed, and the others will block until the mutex is unlocked. A deadlock will result from a thread attempting to lock a mutex it has already locked.

```
mutex_t lock;      /* mutual exclusion for count */
mutex_lock(lock);
```

2.4 Unlocking a Mutex Variable

`mutex_unlock` unlocks the mutex giving other threads a chance to lock it.

```
mutex_t lock;      /* mutual exclusion for count */
mutex_unlock(lock);
```

2.5 Allocation of a Condition Variable

`condition_alloc` provides dynamic allocation of a condition variable.

```
condition_t done;  /* signalled each time a slave finishes */
done = condition_alloc();
```

2.6 Waiting on a Condition

This function unlocks the mutex it takes as a parameter, suspending the calling thread until another thread calls `condition_signal` on the same condition variable. The mutex is then locked and the thread resumes. There is no guarantee that the condition will be true when the thread resumes, therefore `condition_wait` should always be used in the form below.

time the master thread is signalled by a `condition_signal` call, it tests the `count` for a value of zero.

`cthread_init` is the first function called in the example program. This function initializes the C threads implementation and must be called before any of the other cthread functions. If a program is loaded with the Mach version of `crt0`, this call is no longer necessary as it has already been done by `crt0` (or `gcrt0` or `moncrt0`). The `count` which represents the current number of slaves is set to zero. `mutex_alloc` is called to allocate a mutex assigned to the variable `lock`. `condition_alloc` is used to allocate a condition object assigned to the variable `done`. The last initialization call is to the random number generator.

After initialization, the master thread loops creating the number of slave processes desired and incrementing `count` with each creation. `mutex_lock` is called at the beginning of the loop. This call results in either locking the variable `lock`, or blocking until `lock` is unlocked by some other thread. The return of `mutex_lock` signals that the master can now change the variable `count` knowing that no other thread will be accessing this variable until the master unlocks the mutex. `count` is incremented, and a slave is created. To create a slave, the master calls `cthread_fork` followed by `cthread_detach`. `cthread_fork` creates a new thread of control which executes concurrently with the master. `cthread_fork` takes as a parameter a function which the new thread is to execute. Since the master does not intend to later rendez-vous with the slave, `cthread_detach` is called. Once the master has incremented the `count` and created a slave, `mutex_unlock` is called to give the other threads a chance to lock the mutex and consequently access the `count` variable.

Having created the desired number of slaves, the master thread stops looping and waits for all of the slave threads to finish execution. The variable `count` signals the number of slave processes still executing. `mutex_lock` is called so the master may safely access the `count` variable. Now the master thread must wait on the condition `done` by calling `condition_wait`. `condition_wait` unlocks the mutex and suspends the master, letting other threads change the `count` variable. When `condition_wait` returns, the mutex is automatically locked. The master resumes and checks the `count` to see if it is in fact equal to zero. Since there is no gaurantee that the condition will be true when the master is resumed, the `condition_wait` is called in a loop ending when the `count` is zero. Before exiting the master calls `mutex_unlock`. `cthread_exit` is called to terminate the master thread.

When the new slave was created via `cthread_fork`, it was given a function to execute and one parameter to pass to that function. In our example, the slave function is given a random number as a parameter. The slave loops this number of times calling a function `cthread_yield`, which yields the processor to other threads. When finished looping, the thread must decrement the `count` variable because it is about to exit. In order to safely access the `count`, `mutex_lock` is called. Once the mutex `lock` is locked, the `count` is decremented. Next `condition_signal` is called to indicate that the condition represented by the condition variable `done` may be true. The slave calls `mutex_unlock` and exits.

2.1 Initializing the C Threads Package

This initialization function must be called before any other C Thread functions. This call is now called automatically by `crt0`, but multiple calls to this routine are harmless.

1 Introduction

This document is one of two tutorials designed to teach basic Mach programming skills. This manual demonstrates the use of the C Threads library primitives in writing a multi-threaded program and the use of the Mach Interface Generator (MIG) to generate remote procedure calls for interprocess communication. It also includes a final section on where at CMU to find the include files and libraries that comprise the Mach environment as well as procedures for obtaining these files and setting up the correct user environment.

The reader should be familiar with the basic Mach abstractions of ports, messages, virtual memory, tasks and threads before reading this document. The introduction to the other tutorial document, *A Programmer's Guide to the Mach System Calls*, explains these concepts.

2 C Threads: Master Thread Spawning Concurrent Slaves

The C threads package is a runtime library that provides a C language interface to a number of low level, Mach primitives for manipulating threads of control. The constructs provided are: forking and joining of threads, protection of critical regions with mutex variables, and synchronization by means of condition variables. For a complete description of the C threads package see the *C Threads* manual by Cooper and Draves. It is highly recommended that a programmer doing multi-threaded applications use the C threads routines rather than the Mach system calls. The C threads package is designed to provide a more natural set of primitives for multi-threaded applications and is carefully optimized to produce the most efficient use of the system calls.

The program at the end of this section is an example of how to structure a program with a single master thread which spawns a number of concurrent slaves. The master thread waits until all the slaves have finished to exit. A random number generator is used to determine the 'life' of the slave processes. Once created the slave processes in this example simply loop calling a cthread function making the processor available to other threads. The random number generator determines the length of this loop. In a more useful version of this program, each slave process would do something while looping.

In order for the master thread to determine when all of the slaves have exited, a `count` variable is needed to keep track of the number of current threads. This `count` is incremented by the master with each creation of a slave. Each slave decrements the `count` when it exits. Because two or more threads may be trying to access the `count` at the same time, a mutex called `lock` is used to provide exclusive access to `count`. If any thread wants to access the `count` variable, it should first lock the mutex. Consequently when the mutex is locked, any thread wanting the `count` variable must wait until the mutex is unlocked.

Condition variables are used to provide synchronization between threads, e.g one thread wishes to wait until another thread has finished doing something. Every condition variable is associated with a mutex. The condition variable represents a boolean state of the shared data that the mutex protects. In this example after all of the slave threads have been created, the master thread waits until the `count` variable is equal to zero. A condition variable `done` is used to represent the possibility that the `count` may equal zero. Just before a slave thread exits, it signals the condition `done` since it may be the last slave executing. The master thread loops waiting on the condition `done`. Each

A Programmer's Guide to the Mach User Environment

Linda R. Walmer
Mary R. Thompson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Version of: 7 November 1989

Abstract

This document is one of two tutorials designed to teach basic Mach programming skills. This manual demonstrates the use of the C Threads library primitives in writing a multi-threaded program and the use of the Mach Interface Generator (MIG) to generate remote procedure calls for interprocess communication.

The reader should be familiar with the basic Mach abstractions of ports, messages, virtual memory, tasks and threads. The introduction to the companion document to this one, *A Programmer's Guide to the Mach System Calls*, explains these concepts.

Comments, suggestions and additions to this document are welcome.

The material developed under this subcontract was or is sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA order 4864, monitored by the Space and Naval Warfare Systems Command under Contract Number N00039-87-C-0251.

Contents