4 Mach Environment

See the section “General Mach Information” at the end of the *A Programmers’s Guide to the Mach User Environment* to find the current include file names and paths needed to compile code using the Mach system calls.

```
        mem[0] = CHILD_CHANGED;
        printf("\n");
        printf("CHILD: lock = %d\n", *lock);
        printf("CHILD: changing lock to %d\n", CHILD_WAIT);
        printf("\n");
        *lock = CHILD_WAIT;
        while (*lock == CHILD_WAIT);
                /* wait for parent to change lock */
        if ((ret = vm_deallocate(task_self(), lock,
                sizeof(int), TRUE)) != KERN_SUCCESS) {
                mach_error("vm_deallocate returned value of ", ret);
                printf("Exiting.\n");
                exit();
        }
        if ((ret = vm_deallocate(task_self(), mem,
                MAXDATA * sizeof(char), TRUE)) != KERN_SUCCESS) {
                mach_error("vm_deallocate returned value of ", ret);
                printf("Exiting.\n");
                exit();
        }
        printf("CHILD: Finished.\n");
}
```

```
        TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate returned value of ", ret);
        printf("Exiting with error.\n");
        exit();
}
if ((ret = vm_inherit(task_self(), lock, sizeof(int),
        VM_INHERIT_SHARE)) != KERN_SUCCESS) {
        mach_error("vm_inherit returned value of ", ret);
        printf("Exiting with error.\n");
        exit();
}
*lock = NO_ONE_WAIT;
if ((ret = vm_allocate(task_self(), &mem, sizeof(int) * MAXDATA,
        TRUE)) != KERN_SUCCESS) {
        mach_error("vm_allocate returned value of ", ret);
        printf("Exiting with error.\n");
        exit();
}
mem[0] = COPY_ON_WRITE;

printf("value of lock before fork: %d\n", *lock);
pid = fork();
if (pid) {
        printf("PARENT: copied memory =  %d\n",
                mem[0]);
        printf("PARENT: changing to %d\n", PARENT_CHANGED);
        mem[0] = PARENT_CHANGED;
        printf("\n");
        printf("PARENT: lock = %d\n", *lock);
        printf("PARENT: changing lock to %d\n", PARENT_WAIT);
        printf("\n");
        *lock = PARENT_WAIT;
        while (*lock == PARENT_WAIT);
           /* wait for child to change the value */
           /* beware of optimizing compilers */
        printf("PARENT: copied memory = %d\n",
                mem[0]);
        printf("PARENT: lock = %d\n", *lock);
        printf("PARENT: Finished.\n");
        *lock = PARENT_WAIT;
        exit();
}
while (*lock != PARENT_WAIT);
        /* wait for parent to change lock */
        /* beware of optimizing compilers */
printf("CHILD: copied memory = %d\n", mem[0]);
printf("CHILD: changing to %d\n", CHILD_CHANGED);
```

3.3.2 Programming Example III, cowtest.c

```
/*
 * This program demonstrates the use of vm_inherit and copy on write
 * memory. A child and parent process will share memory, polling this
 * memory to see whos turn it is to proceed.  First some memory is allocated,
 * and vm_inherit is called on this memory, the variable 'lock'.  Next more
 * memory is allocated for the copy on write test. A fork is executed, and
 * The parent then stores new data in the copy on write memory
 * previously allocated, and sets the shared variable signaling to the
 * child that he is now waiting.  The child, polling the shared variable,
 * realizes it is his turn.  The child prints the value of the variable
 * lock and a value of the copy on write memory as the child sees it.
 * You will notice that the value of the lock is what the parent
 * set it to be, but the value of the copy on write memory is the original
 * value and not what the parent changed it to be.
 * The parent then awakes and prints out the two values once more.
 * The program then ends with the parent signaling the child via the
 * shared variable lock.
 ********************************************************/
#include <mach.h>
#include <stdio.h>

#define NO_ONE_WAIT 0
#define PARENT_WAIT 1
#define CHILD_WAIT 2
#define COPY_ON_WRITE 0
#define PARENT_CHANGED 1
#define CHILD_CHANGED 2

#define MAXDATA 100

main(argc, argv)
        int     argc;
        char    *argv[];
{
        int             pid;
        int             *mem;
        int             *lock;
        kern_return_t   ret;

        if (argc > 1) {
                printf("cowtest takes no switches.  ");
                printf("This program is an example of copy on write \n");
                printf("memory and of the use of vm_inherit.\n");
                exit();
        }
        if ((ret = vm_allocate(task_self(), &lock, sizeof(int),
```

Through execution of this program, the user will notice that the changes to the copied memory are not seen from the child to the parent; that is when the parent changes the value, the child does not see this change. On the other hand, any change in the shared memory is noticed by both tasks.

The copied memory in this example is actually copy-on-write memory. That is, the memory is never really copied until one of the tasks desires to write in this region.

3.3.1 Virtual Memory, Inheritance

`vm_inherit` allows a task to specify how the various regions of its memory will be passed to any child tasks that it forks. By default all memory is passed to the child as a logical copy (actually copy-on-write). `vm_inherit` allows a task to specify that certain page-regions of its memory are to be shared with any children it subsequently forks, or are not to be passed at all to that child.

The inheritance parameter may be set to `VM_INHERIT_SHARE`, `VM_INHERIT_COPY` or `VM_INHERIT_NONE`. The `size` parameter is measured in bytes but only integral numbers of pages are dealt with. The pages are selected by rounding down the start address to a page boundary and then rounding up the end address to a page boundary.

```
int             *lock;
kern_return_t   ret;

if ((ret = vm_inherit(task_self(), lock, sizeof(int),
   VM_INHERIT_SHARE)) != KERN_SUCCESS) {
   mach_error("vm_inherit returned value of ", ret);
   exit(1);
}
```

3.2 Virtual Memory Copying

The vm_copy primitive is another alternative to vm_read. In either case the task port used as the first parameter may specify the caller's address space or that of some other task. If another task's port is used vm_read copies memory from that task's address space to the caller's address space. On the other hand, vm_copy moves the memory from one part of the designated task's address space to another section of that address space. The fact that these primitives do not actually copy the data regions, but only map the regions as copy-on-write pages, means that they actually have some use even in copying data in the caller's own address space. If a task wants multiple virtual memory references to the same data, it can use either of these primitives to set this up. No data is actually copied until one of the virtual memory areas is modified.

The destination region must be allocated prior to the call to vm_copy.

```
int             *data1, *data2;
kern_return_t   rtn;

/* note that data2 was previously allocated. */

if ((rtn = vm_copy(task_self(), data1, vm_page_size, data2)) !=
   KERN_SUCCESS) {
   mach_error("vm_copy returned value of ", rtn);
   exit(1);
}
```

An complete example of a program using vm_copy can be found in the Mach example directory as vm_copy.c.

3.3 Inheritance of Shared versus Copied Memory

This final virtual memory example illustrates the use of vm_inherit, and the difference between copied and shared memory. The problem posed is to have two memory regions, one inherited and shared by a child task and one region simply copied by the UNIX fork call. To show the difference between the two acquisition methods, the parent and child will take turns printing out and changing the values of the two regions.

The first step towards a solution is to allocate and fill with data two regions of memory. The address of the memory that is to be shared by the child task is passed to vm_inherit, using the VM_INHERIT_SHARED flag. The shared memory is used in this program as a lock to regulate whether the child or parent process is to proceed. After forking, the child will wait while the parent prints out the value of the shared memory and the value of the copied memory. After the parent is finished, he changes the lock causing himself to wait and signaling the child to continue. The child prints the contents of the two regions then changes the lock and waits. The lock change signals the parent to once again print the memory values. The parent changes the lock and exits. The child notices the lock change, deallocates the two memory regions, and exits. Be aware that code which loops on a lock that some other task is going to modify is sometimes deleted by optimizing compilers. Either the variable should be tagged as ”volitile” if your compiler understands that construct or optimizing may need to be turned off.

```
                printf("vmread: Exiting.\n");
                exit();
        }
        printf("Successful vm_read.\n");

        if (vm_page_size != data_cnt) {
                printf("vmread: Number of bytes read not equal to number");
                printf("available and requested.  \n");
        }
        min = (vm_page_size < data_cnt) ? vm_page_size : data_cnt;

        for (i = 0; (i < min); i++) {
                if (data1[i] != data2[i]) {
                        printf("vmread: Data not read correctly.\n");
                        printf("vmread: Exiting.\n");
                        exit();
                }
        }
        printf("Checked data successfully.\n");

        if ((rtn = vm_deallocate(task_self(), data1,
                vm_page_size)) != KERN_SUCCESS) {
                mach_error("vm_deallocate returned value of ", rtn);
                printf("vmread: Exiting.\n");
                exit();
        }

        if ((rtn = vm_deallocate(task_self(), data2,
                data_cnt)) != KERN_SUCCESS) {
                mach_error("vm_deallocate returned value of ", rtn);
                printf("vmread: Exiting.\n");
                exit();
        }
}
```

3.1.4 Programming Example II, vm_read.c

```
/*
 *
 * This program is a test of vm_allocate, vm_read and vm_deallocate.
 * First some memory is allocated, and filled with data.  vm_read is
 * then called, with reading starting at the previously allocated chunk.
 * The contents of the two pieces of memory, one retreived by vm_allocate, and
 * one by vm_read is compared. vm_deallocate is then used to rid of the
 * two chunks of memory.
 *
 *************************************************************/

#include <mach.h>
#include <stdio.h>

main(argc, argv)
        int     argc;
        char    *argv[];
{
        char            *data1, *temp;
        char            *data2;
        int             data_cnt, i, min;
        kern_return_t   rtn;



        if (argc > 1) {
                printf("vm_read takes no switches.  ");
                printf("This program is an example vm_read\n");
                exit();
        }

        if ((rtn = vm_allocate(task_self(), &data1, vm_page_size,
                TRUE)) != KERN_SUCCESS) {
                mach_error("vm_allocate returned value of ", rtn);
                printf("vmread: Exiting.\n");
                exit();
        }

        temp = data1;
        for (i = 0; (i < vm_page_size); i++)
                temp[i] = i;
        printf("Filled space allocated with some data.\n");
        printf("Doing vm_read....\n");
        if ((rtn = vm_read(task_self(), data1, vm_page_size, &data2,
                &data_cnt)) != KERN_SUCCESS) {
                mach_error("vm_read returned value of ", rtn);
```

```
char            *data1;
kern_return_t   rtn;

if ((rtn = vm_deallocate(task_self(), data1,
   vm_page_size)) != KERN_SUCCESS) {
   mach_error("vm_deallocate returned value of ", rtn);
   printf("vmread: Exiting.\verb+\+n");
   exit(1);
}
```

3.1.3 Virtual Memory Reading

`vm_read` makes it possible for one task to read the virtual memory of another. In the example below, a task is reading its own memory. The first parameter to `vm_read` is the task whose address space is to be read. Note the parameter `address`, which is the first address to be read, must be on a page boundary. Size is in bytes and must be an integral number of pages. The data read is returned in a newly allocated region. The size in bytes of this new region is also returned. `vm_deallocate` should be used on the region returned by `vm_read` when it is no longer needed.

```
char            *data1, *data2;
int             data_cnt;
kern_return_t   rtn;

if ((rtn = vm_read(task_self(), data1, vm_page_size, &data2,
   &data_cnt)) != KERN_SUCCESS) {
   mach_error("vm_read returned value of ", rtn);
   printf("vmread: Exiting.\verb+\+n");
   exit(1);
 }
```

3 Use of Virtual Memory

3.1 Allocation, Deallocation, and Reading

The program for this section will demonstrate vm_allocate, vm_read, and vm_deallocate. The purpose of this example is to check vm_read to be sure the data was read correctly.

The first step in solving this problem is to get a chunk of memory and fill it with data. vm_allocate is used to get the virtual memory. Data is then stored in it. The next step is to call vm_read. vm_read allows a task to read another task's virtual memory. Passing the address of the previously allocated memory as a starting point, vm_read is called. Note that our example is a simplified use of vm_read since a task is reading its own memory. vm_read returns a newly allocated region containg the data read. Note that the parameter size, which is the number of bytes to be read, must be an integral number of pages. vm_read can be checked by comparing the data received with the previously allocated chunk. If both spaces contain the same data, vm_read worked correctly. To clean up before ending this example program, both allocated spaces must be deallocated by calling vm_deallocate. Note that the data returned by vm_read must be deallocated.

3.1.1 Virtual Memory Allocation

vm_allocate allocates a region of virtual memory, placing it in the specified task's address space. The size parameter is the number of bytes to allocate which is rounded to an integeral number of virtual pages. If last parameter is TRUE the kernel allocates a region of memory at the next convenient location and returns the virtual address as the second parameter. If the last parameter is FALSE, the kernel allocates memory starting at the address specified by the second parameter. vm_page_size is a global constant defined via mach.h. A page of newly allocated memory is zero-filled when it is first touched.

```
char            *data1;
kern_return_t   rtn;

if ((rtn = vm_allocate(task_self(), &data1, vm_page_size,
   TRUE)) != KERN_SUCCESS) {
   mach_error("vm_allocate returned value of ", rtn);
   printf("vmread: Exiting.\verb+\+n");
   exit(1);
}
```

3.1.2 Virtual Memory Deallocation

vm_deallocate affects only the memory of task specified as the first parameter. This function reliquished access to the memory specified in the parameters: address and size. Other tasks which have access to this physical memory may continue to use it. Note the size is expected in bytes and is rounded up to give a page boundary. Never use vm_deallocate on memory that has been acquired by UNIX malloc.

```
                if ((err = env_del_port(environment_port, port_name))
                        != KERN_SUCCESS) {
                        mach_error("PARENT: env_del_port returned ", err);
                        exit(1);
                }
                if ((err = port_deallocate(task_self(), my_port))
                        != KERN_SUCCESS) {
                        mach_error("PARENT: port_deallocate returned ", err);
                        exit(1);
                }
                printf("PARENT: Finished successfully.\n");
        }
        else  printf("Error from fork.\n");
}
```

```
        for (i = 0; i < MAXDATA; i++)
                printf("%d ", msg_rcv.inline_data[i]);
        printf("\n");
        setup_simp_reply(&msg_xmt, &msg_rcv);
        if ((ret = msg_send(&msg_xmt, MSG_OPTION_NONE,
                0)) != SEND_SUCCESS){
                mach_error("PARENT: msg_send returned value of ", ret);
                exit(1);
        }
        printf("PARENT: Successful msg_send.\n");
}




main (argc, argv)
        int             argc;
        char            **argv;

{
        kern_return_t   err;
        port_t          my_port;
        env_name_t      port_name;
        int             fret;

        if (argc > 1) {
                printf("no arguments to simp_ipc\n");
                exit(1);
        }
        /* create a port name that both the child and parent will know */
        sprintf(port_name, "ipc_test_%d", getpid());

        /* create and register port for parent to receive on */

        if ((my_port = Register(port_name)) == PORT_NULL)
                exit(1);

        /* fork returns 0 if child, and the child's ID to the parent.  */
        fret = fork();
        if (fret == 0) { /* child process */
                if ((my_port = LookFor(port_name)) == PORT_NULL)
                        exit(1);
                child_routine(my_port);
                printf("CHILD: Finished successfully.\n");
        }
        else if (fret >0) { /* parent process */
                parent_routine(my_port);
```

```
}



/* This routine is called to demonstrate the passing of a simple message.
 * Please see program comment for order of events. */

void child_routine(my_port)
        port_t          my_port;

{
        msg_return_t    ret;
        int             i;
        struct simp_msg_struct  msg_xmt, msg_rcv;

        setup_simp_request(&msg_xmt, my_port);
        if ((ret = msg_rpc(&msg_xmt.h, MSG_OPTION_NONE, sizeof(msg_xmt), 0,
                0)) != RPC_SUCCESS) {
                mach_error("CHILD: msg_rpc returned value of ", ret);
                exit(1);
        }
        printf("CHILD: Successful msg_rpc.\n");
}



void parent_routine(my_port)
        port_t          my_port;

{
        msg_return_t    ret;
        int             i;
        int             x;
        msg_header_t    msg_xmt;
        struct simp_msg_struct msg_rcv;

        msg_rcv.h.msg_local_port = my_port;
        msg_rcv.h.msg_size = sizeof(msg_rcv);
        if ((ret = msg_receive(&msg_rcv.h, MSG_OPTION_NONE, 0)) !=
                RCV_SUCCESS) {
                mach_error("PARENT: msg_receive returned value of ", ret);
                exit(1);
        }
        printf("PARENT: Successful msg_receive.\n");
        printf("PARENT: Data..");
```

```
 * associated with the given name.
 */

port_t LookFor(name)
        env_name_t      name;

{
        port_t          result;
        kern_return_t   error;

        if ((error = env_get_port(environment_port, name,
                &result)) != KERN_SUCCESS) {
                        mach_error("CHILD: env_get_port returned ",
                                error);
                        exit(1);
                }

        printf("CHILD: Successful env_get_port.\n");
        return(result);
}




/* This routine is used by the parent to create a port, and to associate the
 * port name with the port via the environment manager.
 * port_allocate is used to allocate a port, and then env_set_port is called
 * passing the name of the port, and the newly allocated port.   */

port_t Register(name)
        env_name_t              name;

{
        port_t          result;
        kern_return_t   error;

        if ((error = port_allocate(task_self(), &result)) != KERN_SUCCESS) {
                mach_error("PARENT: port_allocate returned value of ", error);
                exit(1);
        }
        if ((error = env_set_port(environment_port, name,
                result)) != KERN_SUCCESS) {
                mach_error("PARENT: env_set_port returned value of ", error);
                exit(1);
        }
        printf("PARENT: Successful env_set_port.\n");
        return(result);
```

```
{
        int i;

        msg_xmt->h.msg_local_port = thread_reply();
        msg_xmt->h.msg_remote_port = my_port;
        msg_xmt->h.msg_size = sizeof(struct simp_msg_struct);
        msg_xmt->h.msg_id = 0x12345678;
        msg_xmt->h.msg_type = MSG_TYPE_NORMAL;
        msg_xmt->h.msg_simple = TRUE;

        msg_xmt->t.msg_type_name = MSG_TYPE_INTEGER_32;
        msg_xmt->t.msg_type_size = 32;
        msg_xmt->t.msg_type_number = MAXDATA;
        msg_xmt->t.msg_type_inline = TRUE;
        msg_xmt->t.msg_type_longform = FALSE;
        msg_xmt->t.msg_type_deallocate = FALSE;
        for (i = 0; i < MAXDATA; i++)
                msg_xmt->inline_data[i] = i;

}


/* This procedure is used to set up the reply message that the parent is
 * sending to the child.  Note that the remote_port of the received message
 * designates where the reply message will be sent. No data is sent in this
 * message, so the size of the message is simply the size of the message
 * header.  */

void setup_simp_reply(msg_xmt, msg_rcv)
        msg_header_t            *msg_xmt;
        struct simp_msg_struct  *msg_rcv;

{
        msg_xmt->msg_remote_port = msg_rcv->h.msg_remote_port;
        msg_xmt->msg_local_port = PORT_NULL;
        msg_xmt->msg_id = 0x12345678;
        msg_xmt->msg_size = sizeof(msg_header_t);
        msg_xmt->msg_type = MSG_TYPE_NORMAL;
        msg_xmt->msg_simple = TRUE;
}



/* This procedure is used by the child to get the communication port.
 * The child got the name as part of its inherited static variable space.
 * Port rights, however, are not inherited across forks. env_get_port,
 * a utility of the environment manager is called to return the port
```

2.12 Programming Example I, simp_ipc.c

```
/*
 * This program is an illustration of MACH message passing from a child
 * to the parent process and back.  In this example, the child is passing
 * a simple message where the data is stored in the message. The program
 * allocates a port to use for communication.  The environment manager
 * is used to register the port with a name that both the parent
 * and child know. The program forks a child process which
 * then uses env_get_port to acquire the port needed for communication.
 * A message, containing the data the parent needs, is formed by the child
 * and sent with msg_rpc to the parent.  msg_rpc does a send and a receive
 * using the same message buffer.  The parent does a receive on the
 * established communication port receiving the message from the child.
 * Upon receiving the child's message, the parent constructs and sends
 * a confirmation or reply message back to the child indicating he received
 * the child's message and data.  The call to msg_rpc by the child
 * receives the parent's reply.  The child then tells the environment
 * manager the communication port is no longer needed, and calls
 * port_deallocate.
 *
 ********************************************************/

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

#define MAXDATA 20

/*  simple message structure */
struct simp_msg_struct {
        msg_header_t    h;
        msg_type_t      t;
        int             inline_data[MAXDATA];
};



/* This routine is used to set up the message containing the data that
 * the child will send to the parent.  Here the data is a simple array of
 * integers.  */

void setup_simp_request(msg_xmt, my_port)
        struct simp_msg_struct  *msg_xmt;
        port_t          my_port;
```

```
msg_xmt.t.msg_type_deallocate = FALSE;
/* set msg_xmt.out_of_line_data to point to the data */
```

```
};
struct simp_msg_struct  msg_xmt;

if ((ret = msg_rpc(&msg_xmt.h, MSG_OPTION_NONE, sizeof(msg_xmt), 0,
   0)) != RPC_SUCCESS) {
   mach_error("CHILD: msg_rpc returned value of ", ret);
   exit(1);
}
```

2.11 A Non-Simple Message

Messages are *non-simple* if they contain ports or out-of-line data. The next example shows how to construct a data containing out-of-line data. The most common reasons for sending data out-of-line are that the data block is very large or is of variable size. In-line data is copied by the sender into the message structure and then often copied out of the message by the receiver. Out-of-line data, however, is mapped by the kernel from the address space of the sender to the address space of the receiver. No actual copying of data is done unless one of the two tasks subsequently modifies the data. This is an example of copy-on-write data sharing.

The fields that change values from those in the simple message example are `msg_simple`, `msg_type_inline`, and possibly `msg_type_deallocate`. See Section 2.6 for details on `msg_remote_port` and `msg_local_port`. An example program of non-simple message passing can be found in the Mach examples directory. This example is not included in this document, but can be found in the file `ool_ipc.c` in the Mach examples directory.

```
struct ool_msg_struct {
   msg_header_t    h;
   msg_type_t      t;
   int             *out_of_line_data;
};
struct ool_msg_struct   msg_xmt;
port_t          comm_port;


msg_xmt.h.msg_local_port = thread_reply();
msg_xmt.h.msg_remote_port = comm_port;
msg_xmt.h.msg_size = sizeof(struct ool_msg_struct);
msg_xmt.h.msg_id = 0x12345678;
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_simple = FALSE;

msg_xmt.t.msg_type_name = MSG_TYPE_INTEGER_32;
msg_xmt.t.msg_type_size = 32;
msg_xmt.t.msg_type_number = MAXDATA;
msg_xmt.t.msg_type_inline = FALSE;
msg_xmt.t.msg_type_longform = FALSE;
```

```
msg_rcv.h.msg_local_port = comm_port;
if ((ret = msg_receive(&msg_rcv.h, MSG_OPTION_NONE, 0)) !=
   RCV_SUCCESS){
   mach_error("CHILD: msg_receive returned value of ", ret);
   exit(1);
}
```

2.9 Setting up a Reply Message

At this point a message has already been received in the structure `msg_rcv`. A reply message is to be constructed and sent to the sender of `msg_rcv`. Note that the reply message, `msg_xmt` is simply a `msg_header_t` since no data is required. The `msg_remote_port` field, where to send the message, is set to the remote port of the previously received message. The earlier `msg_receive` call set the remote port field of `msg_rcv` to the `msg_local_port` field specified by the sender. See the comment in Section 2.6 about setting up the `msg_local_port` field.

```
struct simp_msg_struct {
   msg_header_t    h;
   msg_type_t      t;
   int             inline_data[MAXDATA];
};
msg_header_t            msg_xmt;
struct simp_msg_struct  *msg_rcv;


msg_xmt.h.msg_remote_port = msg_rcv->h.msg_remote_port;
msg_xmt.h.msg_local_port = PORT_NULL; /* no reply expected */
msg_xmt.h.msg_id = 0x12345678;
msg_xmt.h.msg_size = sizeof(msg_header_t);
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_simple = TRUE;
```

2.10 RPC, Send/Receive

`msg_rpc` does a `msg_send` followed by a `msg_receive` using the same message buffer. `msg_size` is, as usual, the size of the message that is being sent. The third parameter to `msg_rpc` represents the maximum size of the message buffer for the message to be received. In this case it is the size of the message being sent because we know that the reply message is smaller.

```
msg_return_t    ret;
struct simp_msg_struct {
   msg_header_t    h;
   msg_type_t      t;
   int             inline_data[MAXDATA];
```

```
msg_xmt.t.msg_type_number = MAXDATA;
msg_xmt.t.msg_type_inline = TRUE;
msg_xmt.t.msg_type_longform = FALSE;
msg_xmt.t.msg_type_deallocate = FALSE;
/* fill the msg_xmt.inline_data array with the desired data */
```

2.7 Sending Messages

The first parameter to `msg_send` is the address of a `msg_header_t`. This message will be sent to the port indicated by the `msg_remote_port` field. Send rights to `msg_local_port` are given to the receiver so that it may send a reply message.

```
msg_return_t     ret;
struct simp_msg_struct {
   msg_header_t     h;
   msg_type_t       t;
   int              inline_data[MAXDATA];
};
struct simp_msg_struct  msg_xmt;

if ((ret = msg_send(&msg_xmt.h, MSG_OPTION_NONE, 0)) != SEND_SUCCESS){
   mach_error("CHILD: msg_send returned value of ", ret);
   exit(1);
}
```

2.8 Receiving Messages

`msg_receive` is used to retrieve the next message from a port specified in the `msg_remote_port` field. The field `msg_size` must be set to the size of the buffer for the message and thus the maximum permitted size of the message being received. If the message that is queued on the port is too big, the receive will fail. When `msg_receive` returns, the `msg_remote_port` field will be set to the sender's `msg_local_port` field, or the port that reply messages are expected on and `msg_size` will be set to the size of the message that was received.

```
msg_return_t     ret;
struct simp_msg_struct {
   msg_header_t     h;
   msg_type_t       t;
   int              inline_data[MAXDATA];
};
struct simp_msg_struct  msg_rcv;

msg_rcv.h.msg_size = sizeof(msg_rcv);
```

```
                    &comm_port)) != KERN_SUCCESS) {
   mach_error("env_get_port returned ", error);
   exit(1);
}
```

2.6 Setting up a Simple Message

A message consists of a fixed length header defined by the structure `msg_header_t` followed by a variable number of typed data items. A message is *simple* if it does not contain any out-of-line data (pointers) or any ports. The `msg_remote_port` field must contain the port to which the message is to be sent. In this case it is `comm_port`. The `msg_local_port` field should be set to the port or port set on which a reply message is expected. `thread_reply()`, which returns the thread's reply port is used as the reply port. An example of using a port set for the reply port can be found in *A Programmer's Guide to the Mach User Environment*.

```
typedef struct {
           unsigned int    :24,
                           msg_simple : 8;
           unsigned int    msg_size;        /* in bytes */
           int             msg_type;        /* NORMAL, EMERGENCY */
           port_t          msg_local_port;
           port_t          msg_remote_port;
           int             msg_id;          /* user supplied id */
} msg_header_t;


struct simp_msg_struct {
   msg_header_t    h;
   msg_type_t      t;
   int             inline_data[MAXDATA];
};
struct simp_msg_struct  msg_xmt;
port_t          comm_port;


msg_xmt= &msg_xmt_data;
msg_xmt.h.msg_local_port = thread_reply();
msg_xmt.h.msg_remote_port = comm_port;
msg_xmt.h.msg_size = sizeof(struct simp_msg_struct);
msg_xmt.h.msg_id = 0x12345678;
msg_xmt.h.msg_type = MSG_TYPE_NORMAL;
msg_xmt.h.msg_simple = TRUE;


msg_xmt.t.msg_type_name = MSG_TYPE_INTEGER_32;
msg_xmt.t.msg_type_size = 32;
```

2.3 Port Deallocation

`port_deallocate` is used to relinquish a task's access to a port. If the task has ownership and receive rights to the port, the `port_deallocate` destroys the port and notifies (on their notify ports) all the other tasks that have send rights to the port.

```
port_t          my_port;
kern_return_t   error;

if ((error = port_deallocate(task_self(), my_port)) != KERN_SUCCESS) {
   mach_error("PARENT: port_deallocate returned value of ", error);
   exit(1);
}
```

2.4 Environment Manager Server/Checking in a Port

The Environment Manager is used as a repository for named ports. `env_get_port` can be used to associate a name with a port. Note that the port must have been previously acquired either through a message, or from `port_allocate`, or be one of the special system ports that are aquired on task creation. Name has been set to a string.

```
env_name_t      name;
port_t          comm_port;
kern_return_t   error;


if ((error = env_set_port(environment_port, name,
                        comm_port)) != KERN_SUCCESS) {
   mach_error("PARENT: env_set_port returned value of ", error);
   exit(1);
}
```

2.5 Environment Manager Server/Looking up a Port

`env_get_port` can be used to look up a port when the name of the port is known. If `env_set_port` has not been called to associate a port with the given name, `env_get_port` will fail.

```
env_name_t      name;
port_t          comm_port;
kern_return_t   error;

/* Name has been previously set to a desired string. */

if ((error = env_get_port(environment_port, name,
```

Next the parent takes a message structure and fills in the header fields needed by msg_receive: msg_remote_port, representing the port on which the message is to be received, and msg_size, the maximum size of the expected message. With this message structure, the parent calls msg_receive. msg_receive returns the message queued on the port designated by the msg_remote_port field. Remember that the child's message contained a port to send a reply message to, the msg_local_port field. Upon return from msg_receive, the msg_remote_port field is set to the child's msg_local_port field, the expected reply port.

In our example, the parent is going to send a message back to the child indicating that it received the message containing the data. This reply message contains no data itself; it is just a confirmation. The parent sets the msg_remote_port field of the reply message to the msg_remote_port field of the previously received message. msg_send is now called to send the reply message to the port indicated by the msg_remote_port field.

The earlier call of msg_rpc by the child will now receive the parent's reply message. Our example is over except for cleaning up. env_del_port is called to let the Environment Manager know the name/port association is no longer needed. port_deallocate is then called by the parent which owns the communication port to destroy it.

Detailed discussion of the various calls used by the example are given next and the complete text of the program is given in Section 2.12.

2.1 Mach Error Printing

mach_error is an error routine that accepts a string and an error value. The string is then printed along with an error string associated with the value.

```
kern_return__t  error;
mach_error("PARENT: port_allocate returned value of", error);
```

2.2 Port Allocation

port_allocate is used to create a port. The first argument to port_allocate is the task the port is to belong to, in this case the process itself or task_self().

```
port_t          result;
kern_return_t   error;

if ((error = port_allocate(task_self(), &result)) != KERN_SUCCESS) {
   mach_error("PARENT: port_allocate returned value of ", error);
   exit(1);
}
```

2 Message Communication Between Processes

The first sample program shows how to pass messages between two tasks. This is a good illustration of the following fundamental Mach features: allocation, deallocation, and use of ports; use of the Environment Manager; setting up message structures; and communication between two processes via messages on ports. In this example the parent task will fork a child task, which will send the parent a message containing data. The parent will then notify the child that he received the data by sending a reply message.

At this point, the reader should be aware that most programmers do not code IPC at this level of detail, but instead use the Mach Interface Generator (MIG) to produce the message handling code. The use of MIG is explained in the *Programmers's Guide to the Mach User Environment*. Users who are new the the probably want to read that document before attempting to write code following these examples.

This example uses a Mach version of the UNIX fork utility to create a child task. The UNIX part of the fork creates a complete copy of the parent's address space and prepares the child to begin executing immediately after the fork call. The Mach part of the fork creates two ports for the child task: its task kernel port, defined by `task_self()`; and a notification port, defined by `task_notify()`. The task port is the port that represents that task in calls to the kernel. The notify port is the port on which the task may receive special messages from the kernel. The child task also inherits an exception port, a bootstrap port and some ports for system servers such as the Environment Manager and the Netmsgserver. Access to user defined ports is not inherited through forking. The thread that is created has a thread kernel port, referenced by `thread_self()`, and a thread reply port, referenced by `thread_reply()` created for it. The thread kernel port is the port that represents the thread in kernel call. The thread reply port is a port on which the thread can receive any initialization messages from its parent.

Message passing between the parent and child cannot take place until a port is known by both processes. Before forking, a string is constructed to be used as the name of the communication port and a port is allocated using the `port_allocate` call. Then the Environment Manager function `env_set_port` is called to associate the name with the port. This name is available to both processes after forking since it is a static variable. After the fork the child can acquire send rights to the port using `env_get_port`.

Now that both tasks have access to the communication port, a message is constructed by the child. This message contains a fixed sized message header and a variable sized data portion. When constructing the message, the child sets the `msg_remote_port` field in the header to the communication port established earlier. This field designates the port to which the message is to be sent. Another header field that the child must be sure to set properly is `msg_local_port`. This is the port on which the child will wait for a reply message. In this example, the child will receive the reply message on his thread reply port. The task that receives the message constructed by the child automatically receives send rights to the `msg_local_port`. Since the child task wishes to send a message and then immediately receive a reply message, it uses `msg_rpc` instead of `msg_send` and `msg_receive`. `msg_rpc` does a send followed by a receive using the same message buffer for both calls.

1.2.5 Standard Mach Servers

There are a couple of standard servers that support use of Mach style communications. One is the NetMsgServer. It passes Mach IPC messages between machines. It also provides network-wide port registration and lookup functions. The names of these calls are `netname_check_in` and `netname_look_up`. The man section `netname.3` documents them. The other general purpose Mach server is the Environment Manager. It can register or lookup ports or named strings but does not communicate with other Environment Managers. The functions that it provides are documented in the manual *The Mach Environment Manager* or in the man sections `env_conn.3`, `env_list.3`, and `env_port.3`. In general, one decides to register a port with the NetMsgServer if it is to be known by tasks on arbitrary remote machines within the local network. Ports are registered with the Environment Manager if they are to be used only by tasks which share access to the same Environment Manager. Often such tasks are part of the same creation tree or are performing a computation on a single node.

The examples in this document demonstrate the creation of tasks and threads, message passing between tasks, shared memory communication between tasks and threads, and the use of the virtual memory primitives.

from multiple clients may find one request blocked, but be able to continue working on another request. Creating or destroying a thread is also a much less expensive operation than creating or destroying a task.

1.2.3 Communications

There are two basic ways to communicate between tasks or between threads within a task: shared memory and message passing (IPC). The most obvious and probably most efficient form of communication between two threads in the same task is through shared memory. The most common form of communication between tasks is through message passing. However, threads in the same task may send messages to each other as long as the programmer is careful about which threads are waiting for messages on which ports. Also, it is possible for a task to share memory with tasks that have a common ancestor. Since these tasks will probably be on the same machine this sharing can be efficient. Unrelated tasks can also share memory, but that style of memory sharing is made potentially more complex when two unrelated tasks are not located on the same node. Memory sharing between unrelated tasks is not covered in this tutorial. When two threads/tasks are using the same memory, locking is often needed. Unfortunately, hardware mechanisms for locking memory locations vary from one machine to another. The Mach C threads library package provides machine-independent locking primitives. Tasks that don't use the C threads library must provide their own locking protocols.

1.2.4 Virtual Memory Primitives versus Malloc

The Mach kernel provides a set of primitives to allow a programmer to manipulate the virtual address space of a task. The two most fundamental ones are `vm_allocate` to get new virtual memory and `vm_deallocate` to free virtual memory. The programmer also has available the UNIX functions `sbrk`, `malloc` and `calloc`.

The decision to use one allocation method rather than another should be based on several factors. `sbrk` is now obsolete and only retained for backward compatibility with older UNIX programs. It is not recommended that new programs which expect to use Mach features should use `sbrk`. In fact, `sbrk` calls `vm_allocate` to increase the user's address space. `vm_allocate` always adds new, zero-filled virtual memory in paged-aligned, multiple of page-sized chunks (where a page is currenly 4K or 8K bytes). `Malloc` allocates approximately the size it is asked for (plus a few bytes) out of a pre-allocated heap. `calloc` is the same as `malloc` except that it zeros the memory before returning it. `malloc` and `calloc` are library subroutine calls while `vm_allocate` is a kernel syscall which is somewhat more expensive.

The the most obvious basis on which to choose an allocation function is the size of the desired space. There is one other consideration, however, which is the desirability of page-aligned storage. If the memory that is allocated is to be passed out-of-line in a message, it is more efficient if it is page-aligned. Note that it is essential that the correct deallocation function be used. If memory has been `vm_allocated` it must be `vm_deallocated`, if it was `malloc`ed it must be `freed`. Memory that is received out-of-line from a message has been `vm_allocated` by the kernel.

Aug. 1986 or in the *Mach Kernel Interface Manual*. The latter document gives the complete calling semantics for all the Mach system calls.

1.2.1 Ports, Port Names and Port Sets

Recently a new abstraction has been added to Mach: the *port set*. A `port set` is a group of ports which can be received on in parallel. That is, a thread can do a `msg_receive` call on a port set and will receive the first message that appears on any of the ports in the set. In earlier versions of Mach there was only one port set, which was the set of all `enabled` ports. Port sets are only used for receiving messages, you can not send to a port set. When ports are created they are not a member of any port set but may be added to a port set by the call `port_set_add`. A port may be a member of only one port set at a time, and the task must have receive rights to a port before it can enter it into a port set. A port set cannot be sent in a message. If a task wishes to transfer an entire port set to another task, each of the ports must be sent as a separate port with receive rights and then the receipient must redefine a new port set with all the ports in it.

A `port name` is a new term used to refine the way in which ports are referred to. The use of the term port or the type `port_t` implies that the task has at least send rights for the port. The term `port name` and type `port_name_t` implies that the task may not have any rights to the port and could be holding its name for some other task. The only place where the distinction is important in code is the type used during message passing. If the type `port_t` is used, the kernel will map the port rights to the receipient of the message. If the type `port_name_t` is used no rights mapping is done and the agrument gets passed as a simple integer. All three types `port_t, port_name_t` and `port_set_name_t` are defined to be the same basic C type and may be used interchangeabley in C code. This allows for backwards compatibility with code that was written when only `port_t` existed and allows primitives to work for either ports, port sets or port names.

1.2.2 Tasks versus Threads

Mach tasks have independent address spaces and typically communicate by sending messages to each other. Separate tasks can be used to perform parts of a computation on different workstations connected by a network. The port and message passing facilities of Mach have been designed to allow transparent communication between tasks whether they are on the same node or on two separate nodes in a network. All message operations are location independent and, in theory, it is impossible to tell whether a message has been sent to or received from a task on the same machine or a remote one. In practice, however, the timing and failure modes are different between local messages and remote messages. System services such as remote file access and network message communication are themselves implemented as tasks communicating via messages.

Threads, on the other hand, share their memory and access rights with the other threads in a task. They often communicate within a task through shared memory locations. Threads are intended to allow separate execution units to work in parallel on the same problem. This gives a user an easy way to get parallel computation on a multi-processor. On a single processor, multiple threads may simplify the structure of a program that is logically doing several different functions. Multiple threads are also useful if some of a program's actions may cause a line of exection to be blocked, while other lines of execution could usefully continue. For example, server that handles requests

1 Introduction

1.1 Tutorial Documents

This document is one of two tutorials designed to teach basic Mach programming skills. This manual explains the use of the Mach kernel calls. It begins with an introduction to the basic Mach abstractions of ports, messages, virtual memory, tasks and threads. It then contains a number of simple programs which send and receive Mach messages, and use virtual memory.

There is a companion document to this one, *A Programmer's Guide to the Mach User Environment* that explains the use of higher level methods for implementing multi-threaded programs and interprocess communication. Before writing programs that use the system calls directly, the user should be aware that the methods outlined in the other document may be used to solve his problem more simply.

The final section of *A Programmer's Guide to the Mach User Environment* describes where to find the mach environment on-line at CMU and how to use it.

1.2 Basic Mach Concepts

In many ways the Mach operating system can be viewed as an extension of the UNIX operating system. Existing 4.3bsd programs which do not use knowledge about internal UNIX data structures will continue to function in Mach. However, Mach provides a number of new features not available in traditional UNIX systems. The primary motivation for the differences between Mach and UNIX was a desire to better support multiprocessors and to provide a solid foundation for distributed computing.

In order to use Mach's new features, the programmer needs to be familiar with four fundamental Mach abstractions:

- A *task* is an execution environment, including a paged virtual address space and protected access to system resources such as processors and ports. In general for a task to be useful, it must have at least one thread executing within it. Thus when we speak of communicating with a task, it means to communicate with a thread running in that task. A task with one thread is the Mach equivalent of a traditional process.
- A *thread* is the basic unit of execution. It consists of a processor state, an execution stack and a limited amount of per thread static storage. It shares all other memory and resources with all the other threads executing in the same task. A thread can only execute in one task.
- A *port* is a communication channel - a logical queue of messages protected by the kernel. Only one task can receive messages from a port, but all tasks that have access to the port can send messages.
- A *message* is a typed collection of data objects used in communication between threads.

This tutorial presents and explains several simple programs which make use of these Mach abstractions to solve simple programming problems. A more detailed explanation of the basic Mach abstractions can be found in the Unix Review article *Threads of a New System*, Richard F. Rashid,

A Programmer's Guide to the Mach System Calls

Linda R. Walmer
Mary R. Thompson

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

Version of: December 28, 1989

Abstract

This document is one of two tutorials designed to teach basic Mach programming skills. This manual explains the use of the Mach kernel calls. The companion document to this one, *A Programmer's Guide to the Mach User Environment* explains the use of higher level methods for implementing multi-threaded programs and interprocess communication. Before writing programs that use the system calls directly, the user should be aware that the methods outlined in the other document may be used to solve his problem more simply.

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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or Department of the Navy, Space and Naval Warfare Systems Command, or Carnegie-Mellon University, unless designated by other documentation.

Contents