The UBC OSI Distributed Application Programming Environment [1]

User Manual

December 1, 1990

Gerald W. Neufeld

Murray W. Goldberg

Barry J. Brachman

The OSI Laboratory

Department of Computer Science

University Of British Columbia

[1] This work was made possible by grants from the Canadian National Science and Engineering Council, the UBC Center for Integrated Systems Research and UBC Research Services and Industry Liason.

Contents

Acknowledgement

The authors would like to acknowledge the work of Yueli Yang for her contribution to the design and development of portions of the programming environment. We would also like to recognize the constructive suggestions and comments of EAN Group member Eric Lau and OSIWARE employee Duncan Stickings.

Introduction

This manual describes DAPE (the Distributed Application Programming Environment) and gives instructions for its use. DAPE is a software package which provides the remote operations service defined by the CCITT standard X.219 (ISO standard 9072-1) using the protocol defined by the CCITT standard X.229 (ISO standard 9072-2). It also supports communication through use of a Presentation layer interface. These protocol layers, as well as the OSI transport, session and association control layers are also provided. DAPE provides access to CASN1, an ASN.1 compiler. This compiler enables DAPE applications to encode application level data into an external representation suitable for transmission over a heterogenous computer network. All DAPE applications execute in the Threads sub-kernel environment. Threads provides a convenient coding environment for a group of cooperating processes. Finally, DAPE provides an object storage facility for persistent storage of variable size C data objects.

This manual is divided into parts. The first part describes the Threads environment. The features of Threads are outlined and the Threads application interface is detailed. The second part discusses the communication facilities of DAPE. The main sections of this part discuss application association establishment, release and information transfer using remote operations or presentation data transfer primitives. This part also describes the functionality and use of CASN1, the ASN.1 compiler. The final part of this manual presents the persistent object storage facility. Examples are given throughout the manual.

Part I

Operating System Support

Chapter 1

The Threads Sub-Kernel

1.1 Introduction

Threads is a sub-kernel running inside a UNIX process. The purpose behind writing Threads was to provide a pleasant and convenient coding environment for communication protocols. The result though is such an environment for any set of cooperative communicating processes.

The threads environment has the following properties. First, light-weight processes may be created and destroyed at will. All threads processes run in a shared memory space. Simple interprocess communication primitives have been supplied. A sleep facility has been provided. Memory management routines have been provided to allow quick memory allocation. A threads process has access to all existing libraries and UNIX system calls, though the more popular blocking primitives have been replaced so only the calling threads process is blocked, not the entire UNIX process.

A program written to run in the threads environment has few differences from one written to run directly under UNIX. The main difference, of course, is that the capabilities mentioned above are all available for use. All such programs must include the header file "os.h". Also, instead of a program having to have a "main()" routine, a program designed to run under threads must instead have a "mainp()" routine. This gives access to argv and argc just as "main()" does. It should be noted that very little other than process creation

should be done in the "mainp()" routine. "mainp()" is not actually a threads process, and therefore the proper task for it is the creation of other threads processes. Once the "mainp()" routine returns, the created routines will be allowed to run, and "mainp()" will never be heard from again.

The following is a simple example of code written to use threads:

```
#include "standards.h"
#include "os.h"

/*****************************************************************/
/* this process awaits a message from other processes, prints the      */
/* message to stdio, and replies to the sender. This process will      */
/* terminate when the received message begins with  the character "!". */

PROCESS writer()
{
    PID  who;
    int  msglen;
    char *buf;
    char printbuf[100];
    char firstchar;

    do
        {
        /* await a message from some other process                     */
        buf = Receive( &who, &msglen );

        /* remember the first character so we can test it later        */
        firstchar = buf[0];

        /* null terminate the message                                  */
        buf[msglen - 1] = (char)0;

        /* send message to stdio indicating message arrival            */
        sprintf(printbuf, "Received message %s, length %d\n", buf, msglen);
        WriteN( 1, printbuf, strlen(printbuf) );

        /* return a reply to unblock sender                            */
        strcpy( buf, "thank you" );
        if( ( Reply(who, buf) ) < 0)
```

```
            lilerr("writer on rpl");
        }
    while(firstchar != '!');
}


/*************************************************************************/
/* this is a termination subroutine called when the reader terminates  */
printThis( string )
char *string;
{
    WriteN( 1, string, strlen( string ) );
}

/*************************************************************************/
/* this process reads some text from stdin and sends the text to the   */
/* writer process for printing to stdio. This process will terminate   */
/* when the first character of the read text is a "!". Note that this  */
/* process is passed a single argument on creation.                    */

PROCESS reader( arg )
char *arg;
{
    PID  writer;
    char buf[100];
    char *rplbuf;
    int  numread;
    char printbuf[100];
    char firstch;

    /* indicate that printThis is to be called when this proc finishes */
    OnTermination( printThis, "Goodbye Cruel World" );

    /* print out the create argument, just for fun                     */
    WriteN( 1, arg, strlen(arg) );

    /* ask Threads for the PID of the "writer" process                 */
    writer = NameToPid("writer");

    do
        {
        /* read text from stdin and record the first character         */
        numread = Read(0, buf, 100);
```

```
        firstch = buf[0];

        /* send the text to the writer for writing to stdio            */
        if ( (rplbuf = Send(writer, buf, numread)) < 0)
            lilerr("reader on send");

        /* write (to stdio) the reply received from the writer         */
        sprintf( printbuf, "got reply %s\n", rplbuf );
        WriteN( 1, printbuf, strlen(printbuf) );
        }
    while(firstch != '!')
}

mainp(argc, argv)
int argc;
char **argv;
{
    /* create two processes, each with a 3k stack size, at normal priority */
    Create(reader, 3000, "reader", "readers arg", NORM);
    Create(writer, 3000, "writer", 0, NORM);
}
```

1.2 Process Management, Scheduling and Context Switching

Threads process scheduling is performed on a round-robin, multi-priority basis. All ready processes of a common priority level are scheduled on a round-robin basis. A process will only be allowed to run if there are no higher priority ready processes.

Threads scheduling is neither time-slicing nor preemptive. Context switches are performed only on the basis of threads system calls made by a threads process. The system calls which potentially cause a context switch are the following: Pexit(), Send(), Receive(), Reply(), Read(), Write(), ReadN(), Recvfrom(), WriteN(), Accept(), Connect() and Sleep(). Also, a process may explicitly release control of the processor by calling Sched(). The Sched() threads call requires no parameters. This property of knowing when a context

switch is likely to occur is actually very useful when programming cooperating processes. A critical section can be written with little worry about other process interference. If some operation on shared memory is performed, the writer only needs to be cautious about which threads system calls are made during the critical section.

Lack of time slicing is only a problem in the case of non-cooperating processes (for example - a multi-user system). In this case a threads process may easily neglect or refuse to relinquish control of the cpu for as long as it wishes. This would be a serious problem for the other users (processes) in the system.

Context switches are implemented through the switching of process stacks. Each process has its own stack. When a process is about to give up control of the system, two things happen. First, most processor registers are saved onto the process' stack. Next, the actual stack pointer is saved into a variable in the process' process control block. To load the next ready process, the processor registers are loaded from the ready process' process control block, the ready process' saved stack pointer is decremented to reflect the popping of the registers, and the decremented saved stack pointer is loaded into the hardware stack pointer. Many of these steps require the use of assembly language. The effect of this is that when the Threads context switching subroutine returns, the return will occur to a location taken from the newly installed stack, and therefore a context switch occurs. This necessitates the creation of a "fake" stack at process creation time. This fake stack will cause a completed process (running off the end or returning) to return to a system subroutine which does all necessary clean-up.

Threads makes the assumption that there is always at least one runnable process in the system. This assumption is satisfied by the doIO process which runs at the lowest priority and is guaranteed never to block.

The headers for the process management subroutines are as follows:

```
PID MyPid()
```

```
PID NameToPid(name)
char *name;

int PExists(pid)
PID pid;

PID Kill(pid)
PID pid;

Pexit()

PID Create(addr, stksize, name, arg, prio)
int (*addr)();
int stksize;
char *name;
int arg;
PPRIO prio;

OnTermination( subr, arg )
int (*subr)();
int arg;
```

MyPid() requires no parameters and returns the process identifier of the calling process.

NameToPid() requires a single parameter which is a pointer to a null terminated character string. This string represents a process name (see Create()). This routine searches the existing processes in the system for one with name *name*. The process identifier (PID) of the first process found with such a name is returned.

PExists() requires the single argument *pid*. This routine checks for a process with a process identifier of *pid*. If such a process exists in the system, the integer 1 (one) is returned. Otherwise the integer 0 (zero) is returned.

Kill() requires the single argument *pid*. Kill() searches for the process identified by *pid*, and if found, removes all traces of this process from the system returning the PID of the killed process. Otherwise, a zero is returned.

Pexit() causes the calling process to exit from the system. This routine is always

successful and when called is the last instruction executed by the calling process. Note that it is not necessary for a process to make a call to Pexit(). A process may also terminate its own existence by simply returning from, or falling off the end of its main subroutine.

Create() causes the creation of a new Threads process. Create requires five parameters. The first parameter, *addr*, is a pointer to the routine which acts as the main subroutine for the newly created process. In the C language this is accomplished by simply using the name of the desired subroutine (without parens) for this parameter. The *stksize* parameter is the size, in bytes, of the new process' stack. The stack size requirements vary with the depth of subroutine calls made by the new process. They also vary according to the number of local variables and parameters of routines called by the process. A minimum requirement is generally about 2K bytes, though some processes require as much as 10K bytes or more. The third parameter, *name*, points to a text string which acts as a user supplied identifier for this process. This string must be null terminated and currently has a length restriction of 17 bytes including the null-terminator. Any number of processes may be identified by the same name. The name is copied by the Create() routine and therefore the memory containing this routine may be released by the caller on return from Create(). The fourth parameter, *arg*, acts as an argument (parameter) to the newly created process. This argument is passed transparently to the process and may be of any (4 byte or smaller) type, although it should be cast to an integer on call. The main routine for the new process receives this argument as a parameter, or may instead not declare any parameters if no creation-time arguments are required. The final argument to Create() is *prio*. This arguments indicates the priority of the newly created process. The possible process priorities are HIGH, NORM and LOW. Except under unusual circumstances, user processes should be created at normal (NORM) priority.

OnTermination() requires two arguments, *subr* and *arg*. This routine registers a subroutine to be executed for the calling process when that process terminates (returns, exits or is killed). The *subr* parameter identifies the entry point of the subroutine to be called on process termination. As in the case of Create(), the entry point is identified by using the

subroutine name as the argument. The *arg* parameter is a single argument to be provided to the called subroutine. The registration of the subroutine may be cancelled by calling OnTermination() again with a NULL for the *subr* parameter. It should be noted that this subroutine will be called by the process which is terminating only in the cases that this process falls off the end or calls PExit(). If the process is terminated using the Kill() routine, then the termination subroutine is performed by the process making the call to Kill().

1.3 Memory Management

Threads memory management provides fast memory allocation and deallocation. Threads provides two forms of memory management: a general memory allocation scheme, and a per-process allocation scheme.

The general scheme works as follows. A request for memory allocation is rounded up to the next size allowed by Threads. These sizes, and the number of such sizes are configurable within Threads. Threads keeps a list of available memory blocks of each size. Initially, each list of memory blocks is empty. When a process requests memory, Threads checks the list to see if there are any appropriately sized blocks on the list. If there are, the first block is dequeued and returned. Otherwise, a request for memory is made to UNIX. When memory is returned from a Threads process, it is not returned to UNIX, but is instead queued onto the appropriate list. This way, once a sufficient free pool of memory has been established, subsequent allocations and deallocations are very fast. If a request is made for a block of memory larger than the largest configured size, the request is passed directly to UNIX. When this block is freed, the free request is also passed directly to UNIX. If more memory is required from UNIX and UNIX cannot satisfy the request, all free memory blocks are returned to UNIX, and the process of building the free block pool begins again. This can help recover from serious memory fragmentation. Memory allocated in this way is not associated with any Thread process. If the process that allocated the memory dies, exits or gets killed, the memory still exists.

The calls to allocate and release this type of memory are as follows:

```
BYTE *Malloc( size )
int size;

Free( mem )
BYTE *mem;
```

The parameter to Malloc(), *size*, indicates the number of bytes required. Malloc() will return the address of the allocated memory. The parameter to Free() is the address of the memory to be freed. This address must have been previously returned by Malloc().

Threads also has a per-process memory facility. Each Threads process has associated with it a stack of memory frames. If a process exits or is killed, all of the memory allocated from its stack frames is returned to the system. Per-process memory may be allocated only from a process' top memory frame. There are also operations to create, destroy and swap frames. Frames may also be passed from one process to another. An example of one way that this is useful follows. Say process A wishes to create a linked list of structures and pass it to another process - process B. Process A can create a new memory frame, and allocate all of the linked list nodes from this top frame. It then can pass this frame (and all the nodes contained within) to process B. Process B consumes the list and can free all of the storage associated with the list using a single call which frees the memory frame. This feature is especially useful for complex structures which would be time consuming to free. Its implementation is very efficient in that freeing a memory frame (no matter how many blocks have been allocated on it) requires little more time than freeing one memory block.

The headers of the subroutines which operate on per-process memory are as follows:

```
NewFrame()

FreeFrame()

SwapFrame()
```

```
void *PopFrame()

PushFrame( frame )
void *frame;

int TransferTempMem( topid )
PID topid;

BYTE *TempMalloc( size )
int size;

FreeTempMem()
```

NewFrame() creates a new memory frame and pushes it onto the calling thread's memory frame stack.

FreeFrame() frees all memory associated with the calling process' top memory frame, pops the frame and discards it.

SwapFrame() swaps the top two memory frames of the calling process. If zero or one frames exist for this process then SwapFrame() has no effect.

PopFrame() pops and returns a pointer to the top memory frame of the calling process. This routine should be used with caution, generally in conjunction with PushFrame(). The reason for caution is that a memory frame which is not currently on any process' memory stack is essentially an orphan. This memory will not be returned to the system should its creator or owner exit.

PushFrame() takes a pointer to a frame and pushes it on the calling thread's memory stack. The PopFrame() / PushFrame() pair can be used to transfer per-process memory from one process to another, or to perform stack rearrangement functions. Note that a stack frame which does not currently reside on any process' stack is in danger of becoming uncollectable garbage. Normally, when a memory frame exists on some process' memory stack, the memory is returned to the system when the process exits or dies. Note also that

a memory frame cannot exist on more than one memory stack at a time. If an application wishes to move the top memory frame from one process to another, this may be done using PopFrame(), PushFrame() and inter-process communication, but it is preferable to use TransferTempMem() instead.

TransferTempMem() takes as an argument the PID *topid*. This operation transfers the top memory frame of the calling process to the top of the memory stack of process *topid*. This routine avoids the time interval between a PopFrame() and a PushFrame() when a memory frame does not belong to any process.

TempMalloc() takes an integer parameter *size*. This routine allocates memory from the top memory frame of the calling process. This memory cannot be released using Free(). Per-process memory is instead returned to the system using FreeFrame() (discussed previously) or FreeTempMem(). An important feature of TempMalloc() is that if no memory frame currently exists on the calling process' memory stack, a new one is created and pushed. In this case, the allocated memory is taken from the new frame.

FreeTempMem() returns the calling process' per-process memory to the system. The memory from each of the calling process' memory frames (not just the top one as in the case of FreeFrame()) is returned. This routine also pops all memory frames from the calling process leaving it with none.

Finally, there is one routine which is common to both general memory allocation and per-process memory allocation. This is the Realloc() routine. The header for this routine is as follows:

```
BYTE *Realloc( oldptr, newlength )
BYTE *oldptr;
int  newlength;
```

Realloc requires two parameters. *Oldptr* is pointer to memory (previously allocated using Malloc() or TempMalloc()), and *newlength* is an integer. This routine allocates a

new block of memory of length *newlength*, copies the contents of the original memory to the new memory (to the extent of the old or new memory sizes - whichever is smaller), and returns a pointer to the new memory. Realloc() also frees the original memory block. If the original block was per-process memory, the new block will be allocated from the same memory frame as the original.

1.4 I/O

Routines which support I/O in threads include NonBlkRead(), NonBlkWrite(), Read(), Write(), ReadN(), WriteN(), Accept(), Connect(), Recvfrom(), Open(), Close() and Socket(). Each of these routines is meant to replace corresponding UNIX routines (see individual routine descriptions), though some are provided for different reasons than others.

The routines Open(), Close() and Socket() are provided for the reason that threads must keep account of the number of open file descriptors or sockets. Each UNIX process is allowed to have open at any one time no more than *getdtablesize()* descriptors. This causes a problem for the UNIX *accept()* command which allocates a new descriptor. UNIX *accept()* cannot be called if its completion would require more than the available number of descriptors. In order to avoid this situation, each descriptor allocated from and returned to UNIX must be counted, and a check of this number must be made before the threads Accept() makes its call to UNIX accept().

Threads Accept() has a more important job than simply checking the number of available descriptors and calling UNIX accept(). This routine, like the remainder of the above routines (NonBlkRead(), NonBlkWrite(), Read(), Write(), ReadN(), WriteN(), Recvfrom() and Connect()) are re-implemented in threads for a more important reason. Each of these routines has the trait that it has the potential to block the UNIX process once called. This could be a significant problem for the rest of the threads processes running in the system. It would not be appropriate for all Threads processes to have to wait for a single process' I/O.

To avoid this problem, threads replaces common blocking UNIX system calls with similar threads calls. The replacement I/O calls have the effect of blocking the calling threads process without blocking the other processes sharing the same UNIX process. This is accomplished via the UNIX *select()* call. All Read(), Receive(), Accept(), Write() and Connect() calls (with a small exception) place the calling process on the I/O blocked queue. There it stays until its I/O is satisfied. How does the I/O become satisfied? The way this is accomplished is by first marking all I/O descriptors as non-blocking using the *fcntl()* UNIX call. Then, a threads system process called "doIO" periodically checks the I/O blocked queue. If there is anything on this queue, doIO builds read and write masks for use with select(). If select() indicates that any of the descriptors are ready for reading or writing, then the appropriate operation is performed. If the performance of the operation completes the requested threads operation (eg. if the correct number of bytes have been read), then the process making the original call is taken off the I/O blocked queue and readied by doIO.

The interval at which doIO checks the I/O blocked queue depends on the priority at which the doIO process is created. At present, doIO is created at low priority, and therefore I/O is only performed once all higher priority processes have blocked or completed. This seems to be a suitable arrangement as I/O is comparatively slow.

The headers for these I/O routines are as follows:

```
int Open( file, flags [,mode] )
char *file;
int  flags;
int  mode;

int Close( fd )
int  fd;

int Socket( domain, type, protocol )
int  domain, type, protocol;

int Connect (fd, name, namelen)
```

```
int  fd;
void *name;    /* some address type */
int  namelen;

int Accept (fd)
int  fd;

int Read (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int Recvfrom (fd, buf, numbytes, flags, from, fromlen)
int  fd;
char *buf;
int  numbytes;
int  flags;
char *from;
int  *fromlen;

int ReadN (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int NonBlkRead (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int Write (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int WriteN (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int NonBlkWrite(fd, buf, numbytes)
int  fd;
char *buf;
```

```
int  numbytes;
```

The Open(), Close(), Socket(), Connect() and Accept() routines all provide the same interface and service as their corresponding UNIX routines. As indicated, Open(), Close() and Socket() are only provided to keep track of the number of file descriptors currently in use. The Connect() and Accept() routines are both provided so that a call to one of these blocks only the calling thread rather than the entire UNIX process.

The interface to Read(), ReadN() and NonBlkRead() are the same as for the UNIX read() routine. Read() waits until some data is available for reading, reads the available data, and returns the number of bytes read. Unless some error has occurred, the number of bytes read will be between one and the number requested. ReadN() is similar except that it waits until it reads exactly the number of bytes requested. NonBlkRead() will attempt a non-blocking read and return any data that is immediately available for reading. The number of bytes read will vary between zero and the number of bytes requested. These routines are provided in order that calls made do not block the entire UNIX process.

The interface and service provided by Recvfrom() is the same as the corresponding UNIX routine. This routine is provided so that a call to it does not block the entire UNIX process.

The interface to Write(), WriteN() and NonBlkWrite() are the same as for the UNIX write() routine. Write() waits until it is possible to write some data, writes the data, and returns the number of bytes written. Unless some error has occurred, the number of bytes written will be between one and the number requested. WriteN() is similar except that it waits until it can write exactly the number of bytes requested. NonBlkWrite() will attempt a non-blocking write and return the number of bytes which could be written immediately. The number of bytes written will vary between zero and the number requested. These routines are provided in order that calls made do not block the entire UNIX process.

1.5 Sleep

Processes in the threads sub-kernel have the ability to put themselves to sleep with a timer resolution of CLKRES seconds. In the present implementation CLKRES is set to one tenth of a second.

When the threads Sleep() routine is called, a check is first made to be sure that the caller wants to sleep for more than 0 seconds. If this is not the case, Sleep() returns immediately. Otherwise, a calculation is made of the correct wakeup time by adding the desired sleep duration in seconds to the current time as supplied by the UNIX library function *time(0)*. This value is loaded into the process control block of the calling process, and that process is queued into a blocked queue.

The mechanism by which a process is taken out of the sleep queue and readied will depend on the number of active processes in the system, their cpu intensity, and their state. First of all, it should be mentioned that because threads is not a time-slicing system, there is no guarantee that a sleeping process will be waken up within a deterministic time of the requested wake time. Obviously, some other cpu intensive process could decide not to relinquish control of the cpu for an extended period of time, and therefore the sleeping process would remain asleep until the running process gave up the cpu. This is not really a fault with the sleep logic, but more a by-product of the fact that there is no time-slicing. For the great majority of applications with cooperating processes the lack of time-slicing is not a weakness, but is often rather a benefit. If there are many ready processes in the system, then it is likely that the sleeping process will be waken up as a result of a context switch. Every time a context switch occurs, a check is made of the sleep queue, and if there is a process ready to be waken, it will be placed on the ready queue at that time. If there are few or no ready processes in the system, then the bulk of the time will be spent by threads in the doIO process making the UNIX *select* call. It has been arranged that *select* will time-out every CLKRES seconds, and at this time the sleep queue is checked and appropriate processes readied.

The process of checking the sleep queue for processes to awaken is done at every context switch and therefore must be very fast. This is accomplished by ordering the sleeping processes in order of wake time. When a check is made, first the head of the sleep queue is checked to see if there are any sleeping processes. If there are, it is only necessary to check the wake-time of the head process in the queue, and this is done by means of a simple comparison with the current time. If the process at the head of the queue is ready to be awaken, then the rest in line are checked and readied until one is found whose time has not yet come.

The header of this routine is as follows:

```
Sleep(secs)
long secs;
```

As indicated, this routine puts the calling thread to sleep for the number of seconds indicated in the argument *secs*.

1.6 Signal Handling

Threads provides a way for a threads process to block awaiting the arrival of a UNIX signal. The implementation of this is fairly simple. A call to the SigWait() routine records the signal to be waited for, informs UNIX of a routine to be called on the arrival of this signal (using the UNIX signal() routine), and then blocks the calling thread.

The arrival of this signal causes a global variable to be set. The doIO threads process will check this variable periodically for the arrival of some signal. If one has arrived then each process blocked awaiting that signal is readied.

This method of handling signals is crude in the sense that if signals arrive at very short intervals it is possible that one of them will be missed.

The header of the SigWait() routine is as follows:

```
SigWait( sig )
int sig;
```

As indicated, this call blocks the calling process until the arrival of the signal indicated by *sig*. The valid values for *sig* are those given in the include file <signal.h>.

1.7 Interprocess Communication

Threads processes communicate via Send() / Receive() / Reply() primitives. Because all threads processes share one memory space, no copying of data is done. Instead, a pointer and a length (or actually any two, four byte values) are sent in Send() and Reply().

Send works by first checking to see if the destination process is currently waiting for a message (via Receive()). If so, the pointer and length are transferred, and the receiver is returned to the appropriate ready queue. The sender is placed on the "waiting for reply" queue. If instead the destination process is not waiting for a message, then a check is made to verify that the destination process exists. If it does, the sender is blocked (on the send blocked queue) pending a Receive() operation by the destination.

Receive is very similar to send. First, Threads checks to see if some process is blocked waiting to send to the receiving process. If so, the pointer and length are transferred, and the receiver is readied. The sender is placed on the "waiting for reply" queue. Otherwise, the receiving process is blocked pending some other process sending to it.

Reply is very simple. Unless a problem has occurred, Reply will find that its replied-to process is waiting on the "wait for reply" queue. A verification of this is performed, and the reply pointer is transferred. Also, at this point the original sender is returned to the appropriate ready queue. The Reply() operation may be used by any process. It does not have to be the one which originally received the message.

There is also a routine which allows a potential receiver of messages to test (in a non-

blocking fashion) whether there are messages waiting for it. This is the MsgWaits() subroutine.

The subroutine headers to Send(), Receive(), Reply() and MsgWaits() are as follows:

```
char *Send ( to, msg, len )
PID  to;
char *msg;
int  len;

char *Receive( pid, len )
PID  *pid;
int  *len;

int Reply(pid, msg)
PID  pid;
char *msg;

int MsgWaits()
```

The Send() routine sends a message pointed to by the parameter *msg*, of length *len*, to process *to*. If successful, Send() returns a pointer to the message returned by the Reply() operation. If the proposed destination does not exist, Send() will return the value NOSUCHPROC.

Receive() blocks the calling process until some message is sent to it using Send(). The parameters *pid* and *len* should point to memory locations large enough to hold a PID and integer respectively. The *pid* parameter must point to a valid location. The *len* parameter may have the value NULL if the length of the received message is not required. On return, the *pid* location will contain the PID of the process which sent the message. The *len* location (if provided) will contain the length parameter provided with the sent message. Receive returns a pointer to the message received.

The Reply() operation returns a message to, and unblocks a process which previously performed the Send() operation. The parameter *pid* should contain the PID of the blocked

sender, and the parameter *msg* is a pointer to the returned message (if any). Reply() returns 0 on success or NOSUCHPROC if the destination of the reply does not exist or is not blocked awaiting a Reply().

The MsgWaits() routine returns a 1 if there is at least one message waiting to be received by the calling process. Otherwise, a 0 is returned. This call does not block.