CMU-ITC-85-037

LWP User Manual

24 June 85 13:12

Jonathan Rosenberg
(Larry Raper)

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Table of Contents

Preface

This document describes several packages of programs that have been designed for use by the VICE group. The packages are all based on the LWP package, which is a suite of lightweight process functions usable from a C program. The LWP package is described first, followed by sections for each of the additional utilities.

1. The LWP Package

The LWP package implements primitive functions providing basic facilities that enable procedures written in C, to proceed in an unsynchronized fashion. These separate threads of control may effectively progress in parallel, and more or less independently of each other. This facility is meant to be general purpose with a heavy emphasis on simplicity. Interprocess communication facilities can be built on top of this basic mechanism, and, in fact, many different IPC mechanisms could be implemented. The functions described here currently are available on the SUN Microsystems Workstation under Unix 4.2.[1]

In order to set up the environment needed by the lightweight process support, a one-time invocation of the **LWP_InitializeProcessSupport** function must precede the use of the facilities described here. The initialization function carves an initial process out of the currently executing C procedure. The process id of this initial process is returned as the result of the **LWP_InitializeProcessSupport** function. For symmetry a **LWP_TerminateProcessSupport** function may be used explicitly to release any storage allocated by its initial counterpart. If used, it must be issued from the process created by the **LWP_InitializeProcessSupport** function.

Upon completion of any of the lightweight process functions, an integer value is returned to indicate whether any error conditions were encountered.

A global variable **lwp_debug** can be set to activate or deactivate debugging messages tracing the flow of control within the LWP routines. To activate debugging messages, set **lwp_debug** to a non-zero value. To deactivate, reset it to zero. All debugging output from the LWP routines is sent to stdout.

Macros, typedefs, and manifest constants for error codes needed by the lightweight process mechanism reside in the file **/itc/itc/nfs/include/lwp.h**. A process is identified by an object of type **PROCESS**, which is defined in the include file.

The process model supported by the operations described here is based on a non-preemptive priority dispatching scheme. (A priority is an integer in the range [0..**LWP_MAX_PRIORITY**], where 0 is the lowest priority.) Once a given lightweight process is selected and dispatched, it remains in control until it voluntarily relinquishes its claim on the CPU. Relinquishment may be either explicit (**LWP_DispatchProcess**) or implicit (through the use of certain other LWP operations). In general,

---

[1] The LWP package was originally developed by Larry Raper for use in implementing the SNA network protocol on the Sun.

all LWP operations that may cause a higher priority process to become ready for dispatching, preempt the process requesting the service. When this occurs, the priority dispatching mechanism takes over and dispatches the highest priority process automatically. Services in this category (where the scheduler is guaranteed to be invoked in the absence of errors) are

LWP_CreateProcess

LWP_WaitProcess

LWP_MwaitProcess

LWP_SignalProcess

LWP_DispatchProcess

LWP_DestroyProcess

The following services are guaranteed not to cause preemption (and so may be issued with no fear of losing control to another lightweight process):

LWP_InitializeProcessSupport

LWP_NoYieldSignal

LWP_CurrentProcess

LWP_ActiveProcess

1.1. Key Design Choices

The package should be small and fast;

All processes are assumed to be trustworthy -- processes are not protected from each other's actions;

There is no time slicing or preemption -- the processor must be yielded explicitly.

1.2. A Simple Example

```
# include "/itc/itc/nfs/include/lwp.h"

static read_process(id)
  int *id;
{
  LWP_DispatchProcess();        /* Just relinquish control for now */

  for (;;) {
    /* Wait until there is something in the queue */
    while (empty(q)) LWP_WaitProcess(q);
    /* Process queue entry */
    LWP_DispatchProcess();
  }
}

static write_process()
{
  ...

  /* Loop & write data to queue */
  for (mesg=messages; *mesg!=0; mesg++) {
    insert(q, *mesg);
    LWP_SignalProcess(q);
  }
}

main(argc, argv)
  int argc; char **argv;
{
  PROCESS *id;

  LWP_InitializeProcessSupport(0, &id);
  /* Now create readers */
  for (i=0; i<nreaders; i++)
    LWP_CreateProcess(read_process, STACK_SIZE, 0, i, "Reader", &readers[i]);
  LWP_CreateProcess(write_process, STACK_SIZE, 1, 0, "Writer", &writer);
  /* Wait for processes to terminate */
  LWP_WaitProcess(&done);
  for (i=nreaders-1; i>=0; i--) LWP_DestroyProcess(readers[i]);
}
```

Editorial Note:

These definitions are found in the C header file "/itc/itc/nfs/include/lwp.h". Those header files are the authoritative source of these definitions, and will be more up-to-date than this manual.

1.3. Constants and Data Structures

```
/************************************************************\
*                                                            *
*    Information Technology Center                           *
*    Carnegie-Mellon University                              *
*                                                            *
*    (c) Copyright IBM Corporation, 1985                     *
*                                                            *
\************************************************************/

# define LWP_SUCCESS     0
# define LWP_EBADPID     -1
# define LWP_EBLOCKED    -2
# define LWP_EINIT       -3
# define LWP_EMAXPROC    -4
# define LWP_ENOBLOCK    -5
# define LWP_ENOMEM      -6
# define LWP_ENOPROCESS  -7
# define LWP_ENOWAIT     -8
# define LWP_EBADCOUNT   -9
# define LWP_EBADEVENT   -10
# define LWP_EBADPRI     -11
# define LWP_NO_STACK    -12

/* Maximum priority permissible (minimum is always 0) */
# define LWP_MAX_PRIORITY 1

/* Maximum # events that process may wait on at any time */
# define LWP_MAX_EVENTS 20

typedef struct lwp_pcb *PROCESS;

struct lwp_context {  /* saved context for dispatcher */
  char *topstack;     /* ptr to top of process stack */
};

struct lwp_pcb {                /* process control block */
 char       name[32];           /* ASCII name */
 int        rc;                 /* most recent return code */
 char       status;             /* status flags */
 char       *eventlist[LWP_MAX_EVENTS];   /* ptr to array of eventids */
 int        eventcnt;           /* no. of events in eventlist array */
 int        wakevent;           /* index of eventid causing wakeup */
 int        waitcnt;            /* min number of events awaited */
 char       blockflag;          /* if (blockflag), process blocked */
 int        priority;           /* dispatching priority */
 PROCESS    misc;               /* for LWP internal use only */
 char       *stack;             /* ptr to process stack */
 int        stacksize;          /* size of stack */
 int        (*ep)();            /* initial entry point */
```

```
 char      *parm;            /* initial parm for process */
 struct lwp  context
        context;          /* saved context for next dispatch */
 PROCESS     next, prev;        /* ptrs to next and previous pcb */
};

# ifndef LWP  KERNEL
# define LWP  ActiveProcess    (lwp  cpptr+0)

# define LWP  SignalProcess(event)     LWP  INTERNALSIGNAL(event, 1)
# define LWP  NoYieldSignal(event)     LWP  INTERNALSIGNAL(event, 0)

extern
# endif
 PROCESS     lwp  cpptr;   /* pointer to current process pcb */

struct  lwp  ctl {             /* LWP control structure */
  int       processcnt;        /* number of lightweight processes */
  char      *outersp;          /* outermost stack pointer */
  PROCESS   outerpid;          /* process carved by Initialize */
  PROCESS   first, last;       /* ptrs to first and last pcbs */
  char      dsptchstack[800];    /* stack for dispatcher use only */
};

# ifndef LWP  KERNEL
extern
# endif
    char lwp  debug;       /* ON = show LWP debugging trace */
```

1.4. LWP Runtime Calls

LWP_InitializeProcessSupport

Initialize LWP support & start initial process

Call:

int LWP_InitializeProcessSupport(**in** *int priority,* **out** *PROCESS * pid)*

Parameters:

priority	Priority at which initial process is to run.
pid	The process id of the initial process will be returned in this parameter.

Completion Codes:

LWP_SUCCESS	All went well
LWP_EINIT	This routine has already been called.
LWP_EBADPRI	Illegal priority specified (<0 or too large)

Initializes the LWP package. In addition, this routine turns the current thread of control into the initial process with the specified priority. The process id of this initial process will be returned in parameter pid. This routine must be called to ensure proper initialization of the LWP routines. This routine will not cause the scheduler to be invoked.

LWP_TerminateProcessSupport

Terminate process support and clean up

Call:

int LWP_TerminateProcessSupport( )

Parameters:

None

Completion Codes:

None

This routine will terminate the LWP process support and clean up by freeing any auxiliary storage used. This routine must be called from within the procedure and process that invoked LWP_InitializeProcessSupport. After LWP_TerminateProcessSupport has been called, LWP_InitializeProcessSupport may be called again to resume LWP process support.

LWP_CreateProcess

Create and start a light-weight process

Call:

int LWP_CreateProcess(**in** *int (* ep)(),* **in** *int stacksize,* **in** *int priority,*
in *char * parm,* **in** *char * name,* **out** *PROCESS * pid)*

Parameters:

ep	This is the address of the code that is to execute the function of this process. This parameter should be the address of a C routine with a single parameter.
stacksize	This is the size (in bytes) to make the stack for the newly-created process. The stack cannot be shrunk or expanded, it is fixed for the life of the process.
priority	This is the priority to assign to the new process.
parm	This is the single argument that will be passed to the new process. Note that this argument is a pointer and, in general, will be used to pass the address of a structure containing further "parameters".
name	This is an ASCII string that will be used for debugging purposes to identify the process. The name may be a maximum of 32 characters.
pid	The process id of the new process will be returned in this parameter

Completion Codes:

LWP_SUCCESS	Process created successfully
LWP_ENOMEM	Not enough free space to create process
LWP_EBADPRI	Illegal priority specified (<0 or too large)
LWP_EINIT	LWP_InitializeProcessSupport has not been called

This routine is used to create and mark as runnable a new light-weight process. This routine will cause the scheduler to be called. Note that the new process will begin execution before this call returns only if the priority of the new process is greater than the creating process.

LWP_DestroyProcess

Destroy a light-weight process

Call:

int LWP_DestroyProcess(**in** *PROCESS * pid)*

Parameters:

pid	The process id of the process to be destroyed

Completion Codes:

LWP_SUCCESS	Process destroyed successfully
LWP_EINIT	LWP_InitializeProcessSupport has not been called

This routine will destroy the specified process. The specified process will be terminated immediately and its internal storage will be freed. A process is allowed to destroy itself (of course, it will only get to see the return code if the destroy fails). Note a process may also destroy itself by executing a **return** from the C routine. This routine calls the scheduler.

LWP_WaitProcess

Wait for event

Call:

int LWP_WaitProcess(**In** *char * event)*

Parameters:

event	The event to wait for. This can be any memory address. But, 0 is an illegal event.

Completion Codes:

LWP_SUCCESS	The event has occured
LWP_EINIT	LWP_InitializeProcessSupport has not been called
LWP_EBADEVENT	The specified event was illegal (0)

This routine will put the calling process to sleep until another process does a call of LWP_SignalProcess or LWP_NoYieldSignal with the specified event. Note that signals of events are not queued: if a signal occurs and no process is woken up, the signal is lost. This routine invokes the scheduler.

LWP_MwaitProcess

Wait for a specified number of a group of signals

Call:

int LWP_MwaitProcess(**in** *int wcount,* **in** *char * evlist[])*

Parameters:

wcount	Is the number of events that must be signaled to wake up this process
evlist	This a null-terminated list of events (remember that 0 is not a legal event). There may be at most LWP_MAX_EVENTS events.

Completion Codes:

LWP_SUCCESS	The specified number of appropriate signals has occurred
LWP_EBADCOUNT	There are too few events (0), or too many events (more than LWP_MAX_EVENTS), or wcount >the number of events in evlist
LWP_EINIT	LWP_InitializeProcessSupport has not been called

This routine allows a process to wait for wcount signals of any of the signals in evlist. Any number of signals of a particular event is only counted once. The scheduler will be invoked.

LWP_SignalProcess

Signal an event

Call:

int LWP_SignalProcess(**In** *char * event)*

Parameters:

event	The event to be signaled. An event is any memory address except 0

Completion Codes:

LWP_SUCCESS	The signal was a success (a process was waiting for it)
LWP_EBADEVENT	The specified event was illegal (0)
LWP_EINIT	LWP_InitializeProcessSupport was not called
LWP_ENOWAIT	No process was waiting for this signal

This routine causes event to be signaled. This will mark all processes waiting for only this event as runnable. The scheduler will be invoked. Signals are not queued: if no process is waiting for this event, the signal will be lost and LWP_ENOWAIT will be returned.

LWP_NoYieldSignal

Signal an event, but don't yield

Call:

int LWP_NoYieldSignal(**in** *char * event)*

Parameters:

event	The event to be signaled. An event is any memory address except 0

Completion Codes:

LWP_SUCCESS	The signal was a success (a process was waiting for it)
LWP_EBADEVENT	The specified event was illegal (0)
LWP_EINIT	LWP_InitializeProcessSupport was not called
LWP_ENOWAIT	No process was waiting for this signal

This routine causes event to be signaled. This will mark all processes waiting for only this event as runnable. This call is identical to LWP_SignalProcess except that the scheduler will not be invoked -- control will remain with the signalling process. Signals are not queued: if no process is waiting for this event, the signal will be lost and LWP_ENOWAIT will be returned.

LWP_DispatchProcess

Yield to the scheduler

Call:

int LWP_DispatchProcess( )

Parameters:

None

Completion Codes:

LWP_SUCCESS	All went well
LWP_EINIT	LWP_InitializeProcessSupport has not been called

This routine is a voluntary yield to the LWP scheduler.

LWP_CurrentProcess

Get the current process id

Call:

int LWP_CurrentProcess(**out** *PROCESS * pid)*

Parameters:

pid	The current process id will be returned in this parameter

Completion Codes:

LWP_SUCCESS	The current process id has been returned
LWP_EINIT	LWP_InitializeProcessSupport has not been called

This routine will place the current process id in the parameter pid.

LWP_ActiveProcess

Yield current process id

Call:

PROCESS LWP_ActiveProcess( )

Parameters:

None

Completion Codes:

None

This is a macro that yields the current process id. If LWP_InitializeProcessSupport has not been called. the invocation yields 0. It exists because people may find it more convenient than LWP_CurrentProcess.

2. The Lock Package

The lock package contains a number of routines and macros that allow C programs that utilize the LWP abstraction to place read and write locks on data structures shared by several light-weight processes. Like the LWP package, the lock package was written with simplicity in mind -- there is no protection inherent in the model.

In order to use the locking mechanism for an object, an object of type **struct Lock** must be associated with the object. After being initialized, with a call to **LockInit**, the lock is used in invocations of the macros **ObtainReadLock**, **ObtainWriteLock**, **ReleaseReadLock** and **ReleaseWriteLock**.

The semantics of a lock is such that any number of readers may hold a lock. But only a single writer (and no readers) may hold the clock at any time. The lock package guarantees fairness: each reader and writer will eventually have a chance to obtain a given lock. However, this fairness is only guaranteed if the priorities of the competing processes are identical. Note that no ordering is guaranteed by the package.

In addition, it is illegal for a process to request a particular lock more than once, without first releasing it. Failure to obey this restriction may cause deadlock.

2.1. Key Design Choices

The package must be simple and fast: in the case that a lock can be obtained immediately, it should require a minimum of instructions;

All the processes using a lock are trustworthy;

The lock routines ignore priorities;

2.2. A Simple Example

```
# include "lock.h"

struct Vnode {
  ...
  struct Lock lock;  /* Used to lock this vnode */
  ...
};

# define READ  0
# define WRITE  1

struct Vnode *get  vnode(name, how)
  char *name;
  int how;
{
  struct Vnode *v;

  v = lookup(name);
  if (how == READ)
    ObtainReadLock(&v->lock);
  else
    ObtainWriteLock(&v->lock);
}
```

Editorial Note:

These definitions are found in the C header file "/itc/itc/nfs/include/lock.h". Those header files are the authoritative source of these definitions, and will be more up-to-date than this manual.

2.3. Constants, Macros and Data Structures

```
/************************************************************************\
*                                                                        *
*    Information Technology Center                                       *
*    Carnegie-Mellon University                                          *
*                                                                        *
*    (c) Copyright IBM Corporation, 1985                                 *
*                                                                        *
*                                                                        *
\************************************************************************/

/*
    Include file for using Vice locking routines.
*/

struct Lock {
  int           readers_waiting;     /* # readers waiting */
  int           readers_reading;     /* # readers actually with read locks */
  int           writers_waiting;     /* # writers waiting */
  unsigned char     write_locked;        /* !=0 if a writer has it locked */
};

# define READ_LOCK    1
# define WRITE_LOCK    2

# define ObtainReadLock(lock)\
    if (!(lock)->write_locked && !(lock)->writers_waiting)\
      (lock)->readers_reading++;\
    else\
      Lock_Obtain(lock, READ_LOCK)

# define ObtainWriteLock(lock)\
    if (!(lock)->write_locked && !(lock)->readers_reading)\
      (lock)->write_locked++;\
    else\
      Lock_Obtain(lock, WRITE_LOCK)

# define ReleaseReadLock(lock)\
    (!--(lock)->readers_reading && (lock)->writers_waiting ?\
        LWP_SignalProcess(&(lock)->writers_waiting)  :\
        0)

# define ReleaseWriteLock(lock)\
    ((lock)->write_locked--,\
     (lock)->readers_waiting ? LWP_SignalProcess(&(lock)->readers_waiting) : 0,\
     !(lock)->readers_reading && (lock)->writers_waiting ?\
        LWP_SignalProcess(&(lock)->writers_waiting)  :\
        0)

# define CheckLock(lock)\
```

```
    (-((int)(lock)->write_locked) + (lock)->readers_reading)
/* I changed above from:
    (-(lock)->write_locked + (lock)->readers_reading)
  because write_locked wasn't being sign extended.
  The following code was generated:
    movl  a6@(C),a5
    movb  a5@(20),d0
    negb  d0
    andl  # 0xff,d0
    addl  a5@(18),d0
    moveq  # 0x0,d1
    cmpl  # 0x0,d0
    scs  d1
    negb  d1
    movl  d1,a6@(FFFFFFFC)
  (Bob)
*/
```

2.4. Lock Calls

LockInit

Initialize a lock

Call:

LockInit(**out** *struct Lock * lock)*

Parameters:

lock	The (address of the) lock to be initialized

Completion Codes:

None

This routine must be called to initialize a lock before it is used.

ObtainReadLock

Obtain a read lock

Call:

ObtainReadLock(**in out** *struct Lock * lock)*

Parameters:

lock	The lock to be read-locked

Completion Codes:

None

A read lock will be obtained on the specified lock. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.

ObtainWriteLock

Obtain a write lock

Call:

ObtainWriteLock(**In out** *struct Lock * lock)*

Parameters:

lock	The lock to be write-locked

Completion Codes:

None

A write lock will be obtained on the specified lock. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.

ReleaseReadLock

Release a read lock

Call:

ReleaseReadLock(**in out** *struct Lock * lock)*

Parameters:

lock	The lock to be released

Completion Codes:

None

The specified lock will be released. This macro requires that the lock must have been previously read-locked. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.

ReleaseWriteLock

Release a write lock

Call:

ReleaseWriteLock(**In out** *struct Lock * lock)*

Parameters:

lock	The lock to be released

Completion Codes:

None

The specified lock will be released. This macro requires that the lock must have been previously write-locked. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.

CheckLock

Check status of a lock

Call:

int CheckLock(**in** *struct Lock * lock)*

Parameters:

lock	The lock to be checked

Completion Codes:

None

This macro yields an integer that specifies the status of the indicated lock. The value will be -1 if the lock is write-locked, 0 if unlocked, or a positive integer that indicates the numer of readers with read locks. Note that this is a macro and not a routine. Thus, results are not guaranteed if the lock argument is a side-effect producing expression.

3. The IOMGR Package

The IOMGR package contains 3 routines that aid in performing Unix **selects** within the LWP paradigm. After initializing the package, light-weight processes may make calls to IOMGR_Select, which has parameters identical to the Unix **select**. IOMGR_Select, however, puts the calling process to sleep until such time as no user processes are active. At this time the IOMGR process, which runs at the lowest priority, wakes up and coalesces all of the select requests together. It then performs a single **select** and wakes up all processes affected by the result.

3.1. Key Design Choices

The meanings of the parameters, both before and after the call, should be identical to those of the Unix **select**;

A blocking select should only be done if no other processes are runnable.

3.2. A Simple Example

```
void rpc2_SocketListener()
{
  int ReadfdMask, WritefdMask, ExceptfdMask, rc;
  struct timeval *tvp;

  while(TRUE) {
    ...
    ExceptfdMask = ReadfdMask = (1 <<rpc2_RequestSocket);
    WritefdMask = 0;
    rc = IOMGR_Select(8*sizeof(int), &ReadfdMask, &WritefdMask, &ExceptfdMask, tvp);

    switch(rc) {
      case 0:   /* timeout */
          continue;  /* main while loop */

      case -1:  /* error */
          SystemError("IOMGR_Select");
          exit(-1);

      case 1:   /* packet on rpc2_RequestSocket */
          ... process packet ...
          break;

      default:  /* should never occur */
    }
  }
}
```

3.3. IOMGR Calls

IOMGR_Initialize

Initialize the IOMGR package

Call:

int IOMGR_Initialize( )

Parameters:

None

Completion Codes:

LWP_SUCCESS	All went well
LWP_ENOMEM	Not enough free space to create the IOMGR process
LWP_EINIT	LWP_InitializeProcessSupport has not been called
-1	Something went wrong with the TIMER package

This call will initialize the IOMGR package. Its main task is to create the IOMGR process, which runs at priority 0, the lowest priority. The remainder of the processes must be running at priority 1 or greater for the IOMGR package to function correctly.

IOMGR_Finalize

Finalize the IOMGR package

Call:

int IOMGR_Finalize( )

Parameters:

None

Completion Codes:

LWP_SUCCESS	Package finalized okay
LWP_EINIT	LWP_InitializeProcessSupport has not been called

This call cleans up when the IOMGR package is no longer needed. It releases all storage and destroys the IOMGR process.

IOMGR_Select

Perform an LWP select operation

Call:

int IOMGR_Select(**In** *int fds,* **In out** *int * readfds,* **In out** ** writefds,*
In out ** exceptfds,* **In** *struct timeval * timeout)*

Parameters:

fds	Maximum number of bits to consider in masks
readfds	Mask of file descriptors that process wants notification of when ready to be read
writefds	Mask of file descriptors that process wants notification of when ready to be written
exceptfds	Mask of file descriptors that process wants notification of when exceptional condition occurs
timeout	Timeout for use on this select

Completion Codes:

None

This function performs an LWP version of Unix **select**. The parameters have the same meanings as the Unix call. However, the return value will only be -1 (an error occured), 0 (a timeout occured), or 1 (some number of file descriptors are ready). If this is a polling select, it is done and IOMGR_Select returns to the user with the results. Otherwise, the calling process is put to sleep. If at some point, the IOMGR process is the only runnable process, it will awaken and collect all select requests. It will then perform a single select and awaken those processes the appropriate processes -- this will cause return from the affected IOMGR_selects.

4. The Timer Package

The timer package contains a number of routines that assist in manipulating lists of objects of type **struct TM_Elem**. **TM_Elem**s (timers) are assigned a timeout value by the user and inserted in a package-maintained list. The time remaining to timeout for each timer is kept up to date by the package under user control. There are routines to remove a timer from its list, to return an expired timer from a list and to return the next timer to expire. This specialized package is currently used by the IOMGR package and by the implementation of RPC2.

A timer is used commonly by inserting a field of type **struct TM_Elem** into a structure. After inserting the desired timeout value the structure is inserted into a list, by means of its timer field.

5. A Simple Example

```
static struct TM_Elem * requests;

...

  TM_Init(&requests);      /* Initialize timer list */
  ...
  for (;;) {
    TM_Rescan(requests);  /* Update the timers */
    expired = TM_GetExpired(requests);
    if (expired == 0) break;
    ... process expired element ...
  }
```

5.1. Timer Calls

TM_Init

Initialize a timer list

Call:

int TM_Init(**out** *struct TM_Elem ** list)*

Parameters:

list	The list to be initialized

Completion Codes:

0	ok
-1	not enough free storage

The specified list will be initialized so that it is an empty timer list. This routine must be called before any other operations are applied to the list.

TM_Final

Finalize a timer list

Call:

int TM_Final(**in out** *struct TM_Elem **list)*

Parameters:

list	The list to be finalized

Completion Codes:

0	ok
-1	* list was 0 or list was never initialized

Call this routine when you are finished with a timer list and the list is empty. This routine releases any auxiliary storage associated with the list.

TM_Insert

Initialize a timer element and insert it into a timer list

Call:

void TM_Insert(**In** *struct TM_Elem * list,* **In out** *struct TM_Elem * elem)*

Parameters:

list	The list into which the element is to be inserted
elem	The element to be initialized and inserted

Completion Codes:

None

The element *elem* is initialized so that the *TimeLeft* field is equal to the *TotalTime* field. (The *TimeLeft* field may be kept current by use of TM_Rescan.) The element is then inserted into the list.

TM_Rescan

Update TimeLeft fields of entries on a timer list and look for expired elements

Call:

int TM_Rescan(**In out** *struct TM_Elem *list)*

Parameters:

list	The list to be updated

Completion Codes:

None

This routine will update the *TimeLeft* fields of all timers on *list*. (This is done by checking the time of day clock in Unix.) This routine returns a count of the number of expired timers on the list. This is the only routine (besides TM_Init that updates the *TimeLeft* field.

TM_GetExpired

Return an expired timer from a list

Call:

*struct TM_Elem * TM_GetExpired(* **in** *struct TM_Elem * list)*

Parameters:

list	The list to be searched

Completion Codes:

None

The specified list will be searched and a pointer to an expired timer will be returned. 0 is returned if there are no expired timers. An expired timer is one whose *TimeLeft* field is less than or equal to 0.

TM_GetEarliest

Return the earliest timer on a list

Call:

*struct TM_Elem * TM_GetEarliest(* **in** *struct TM_Elem * list)*

Parameters:

list	The list to be searched

Completion Codes:

None

This routine returns a pointer to the timer that will be next to expire -- that with a smallest *TimeLeft* field. If there are no timers on the list, 0 is returned.

TM_eql

See if 2 timevals are equal

Call:

unsigned char TM_eql(**in** *struct timeval * t1,* **in** *struct timeval * t2)*

Parameters:

t1	a timeval
t2	Another timeval

Completion Codes:

None

This routine returns 0 if and only if *t1* and *t2* are not equal.

Appendix I
Summary of LWP-Related Calls

Note: The numbers in square brackets indicate the page on which the call is described.

[8]

*LWP_InitializeProcessSupport(***in** *int priority,* **out** *PROCESS *pid)*

[9]

LWP_TerminateProcessSupport()

[10]

*LWP_CreateProcess(***in** *int (*ep)(),* **in** *int stacksize,* **in** *int priority,*
in *char *parm,* **in** *char *name,* **out** *PROCESS *pid)*

[11]

*LWP_DestroyProcess(***in** *PROCESS *pid)*

[12]

*LWP_WaitProcess(***in** *char *event)*

[13]

*LWP_MwaitProcess(***in** *int wcount,* **in** *char *evlist[])*

[14]

*LWP_SignalProcess(***in** *char *event)*

[15]

*LWP_NoYieldSignal(***in** *char *event)*

[16]

LWP_DispatchProcess()

[17]

*LWP_CurrentProcess(***out** *PROCESS *pid)*

[18]

LWP_ActiveProcess()

[23]

*LockInit(***out** *struct Lock *lock)*

[24]

*ObtainReadLock(***in out** *struct Lock *lock)*

[25]

*ObtainWriteLock(***in out** *struct Lock *lock)*

[26]

*ReleaseReadLock(***in out** *struct Lock *lock)*

[27]

*ReleaseWriteLock(***in out** *struct Lock *lock)*

[28]

*CheckLock(***in** *struct Lock *lock)*

[30]

IOMGR_Initialize()

[31]

IOMGR_Finalize()

[32]

*IOMGR_Select(***in** *int fds,* **in out** *int * readfds,* **in out** ** writefds,*
in out ** exceptfds,* **in** *struct timeval * timeout)*

[36]

*TM_Init(***out** *struct TM_Elem * * list)*

[37]

*TM_Final(***in out** *struct TM_Elem * * list)*

[38]

*TM_Insert(***in** *struct TM_Elem * list,* **in out** *struct TM_Elem * elem)*

[39]

*TM_Rescan(***in out** *struct TM_Elem * list)*

[40]

*TM_GetExpired(***in** *struct TM_Elem * list)*

[41]

*TM_GetEarliest(***in** *struct TM_Elem * list)*

[42]

*TM_eql(***in** *struct timeval * t1,* **in** *struct timeval * t2)*

Appendix II
Usage Notes for the ITC SUN Systems

II.1. LWP

In order to use the LWP package it is necessary to include the file **/ itc/ itc/ nfs/ include/ lwp.h** in your C source programs. This file contains the definitions of a process id (**PROCESS**), the error return codes and several auxiliary definitions.

The simplest way to link in the LWP objects is by including the library **/ itc/ itc/ nfs/ lib/ lwp.o** during link editing.

It is also possible to configure your system in such a way that the LWP package will check for stack overflows during the execution of each process. In order to do this all routines that you desire stack checking for must be compiled with the **-p** option to **cc**. (Note that this means that profiling can not be performed when stack checking is in effect.) In addition, your system must be linked by using **ld** directly as follows:

```
ld -X /itc/itc/nfs/lib/start.o {your objects} /itc/itc/nfs/lib/lwp.o ...
```

II.2. Lock

In order to use the lock package it is necessary to include the file **/ itc/ itc/ nfs/ include/ lock.h** in your C source programs.

During link editing the library **/ itc/ itc/ nfs/ lib/ lwp.o** must be included.

II.3. IOMGR

During link editing the library **/ itc/ itc/ nfs/ lib/ iomgr.o** must be included.

II.4. Timer

In order to use the Timer package it is necessary to include the file **/ itc/ itc/ nfs/ include/ timer.h** in your C source programs.

During link editing the library **/ itc/ itc/ nfs/ lib/ timer.o** must be included.