A STUDY OF ADAPTIVE LOAD BALANCING ALGORITHMS FOR DISTRIBUTED SYSTEMS

VOL II

IAN DERRICK JOHNSON

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

January 1988

LIST OF CONTENTS

VOLUME 2.

Appendix A : Detailed Simulation Results (using the Independent Process Model) 3

Appendix B : Detailed Simulation Results (using the Cooperating Process Group Model) 19

Appendix C : Program for the Simulated System 42

APPENDIX A

Detailed Simulation Results (using the Independent Process Model)

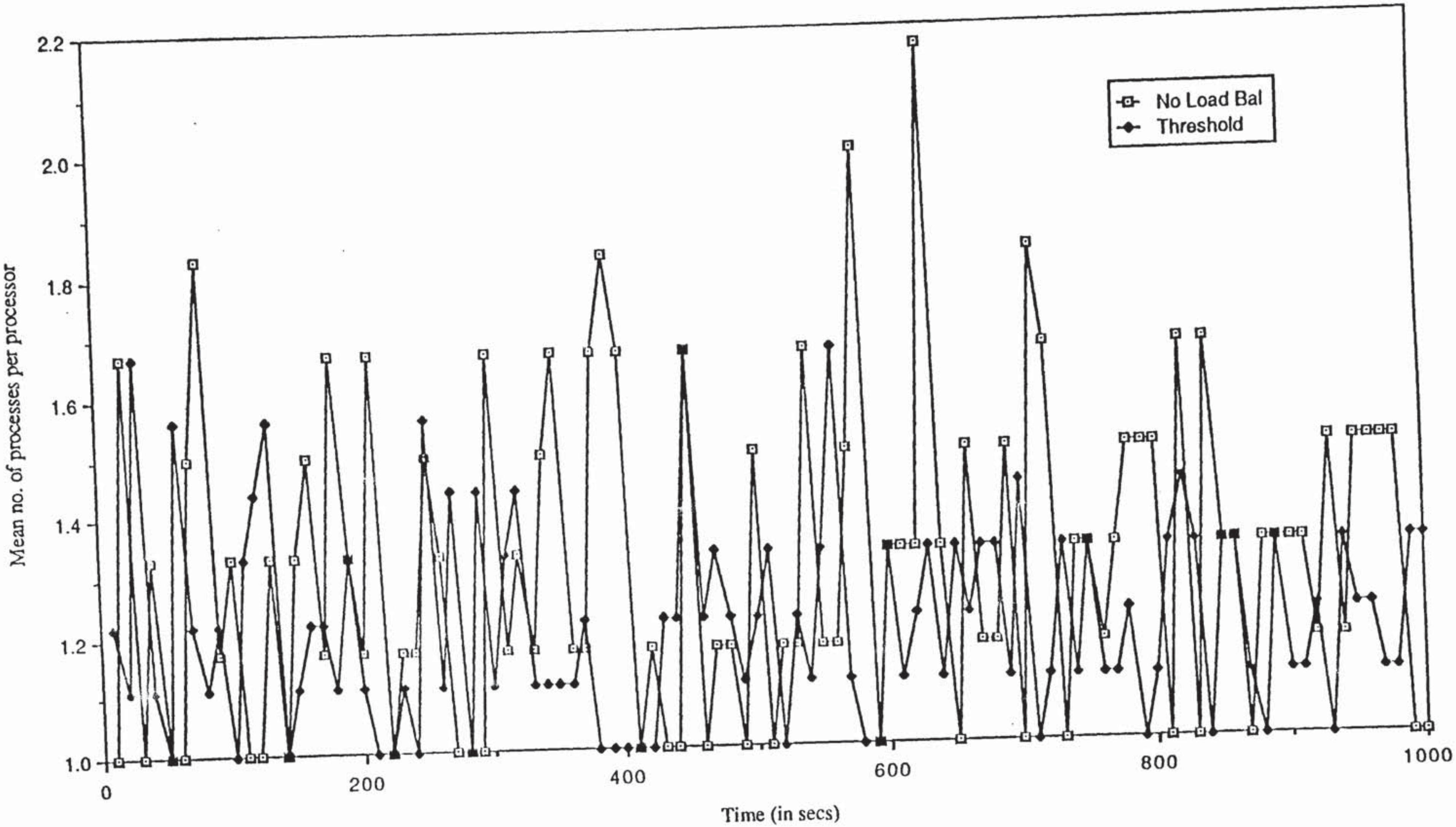

Fig. A.1.1 **Mean Load** - No Load Balancing vs Threshold using the Independent Process Model (Load Value = 0.2)

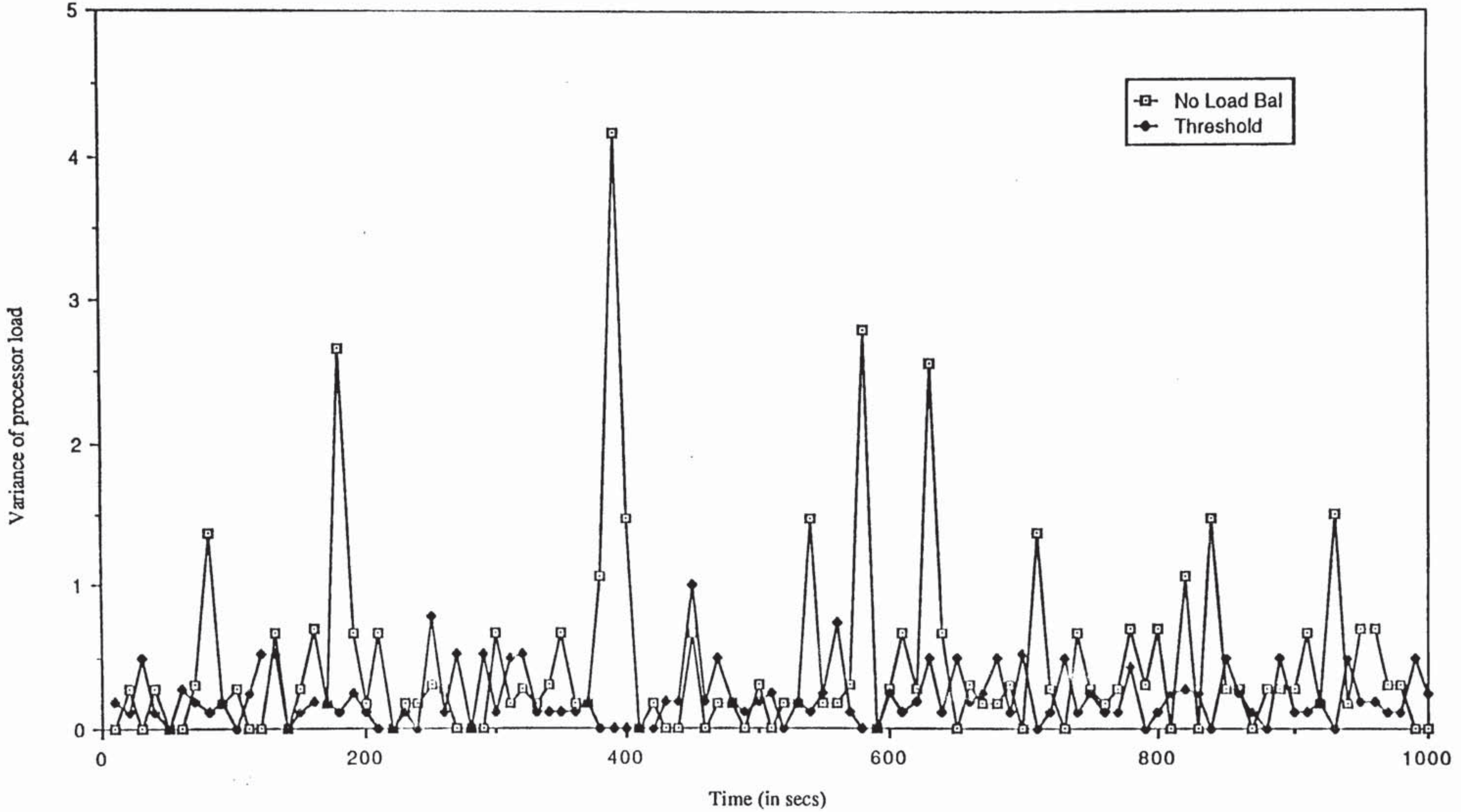

Fig. A.1.2 **Load Variance** - No Load Balancing vs Threshold using the Independent Process Model (Load Value = 0.2)

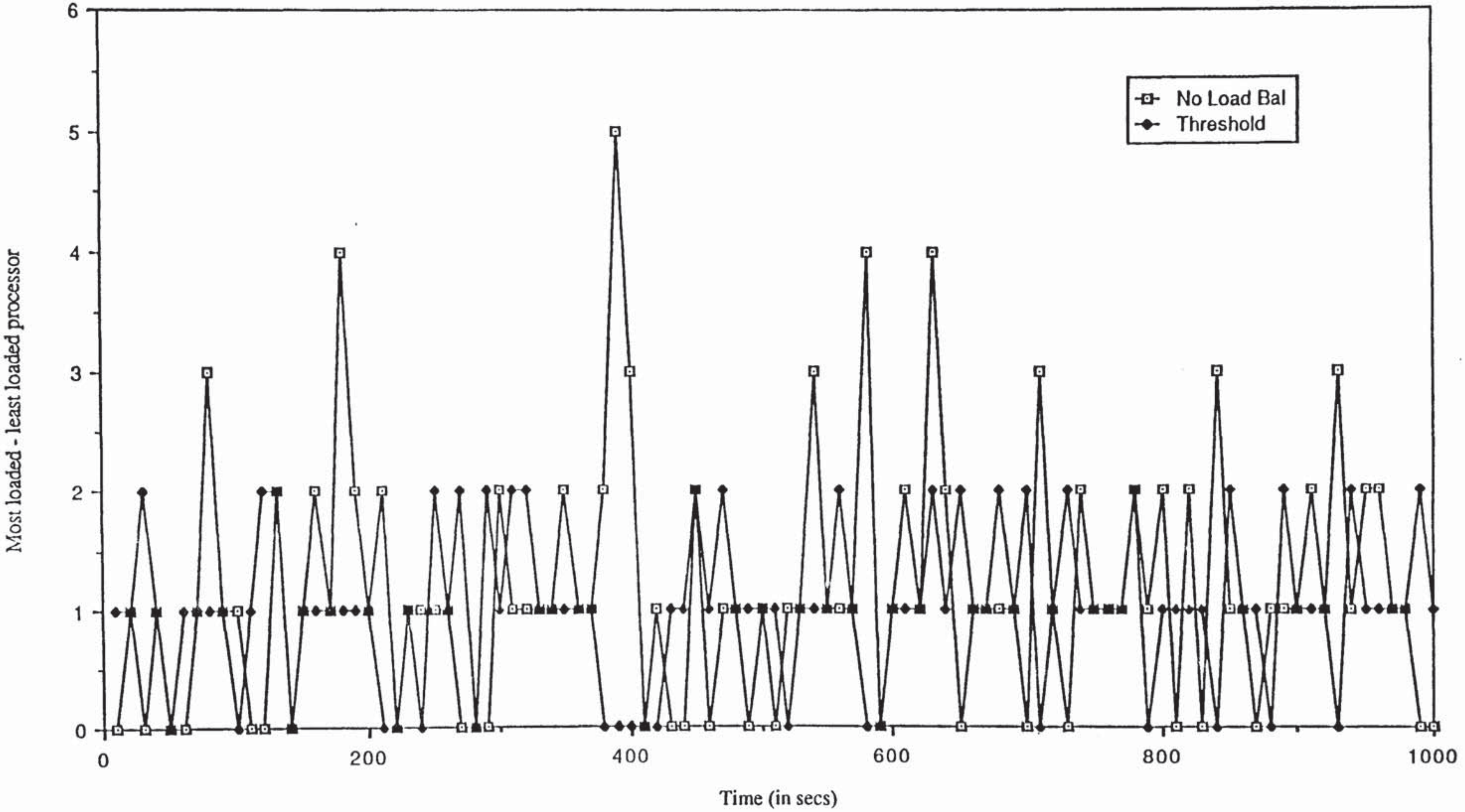

Fig. A.1.3 **Load Difference** - No Load Balancing vs Threshold using the Independent Process Model (Load Value = 0.2)

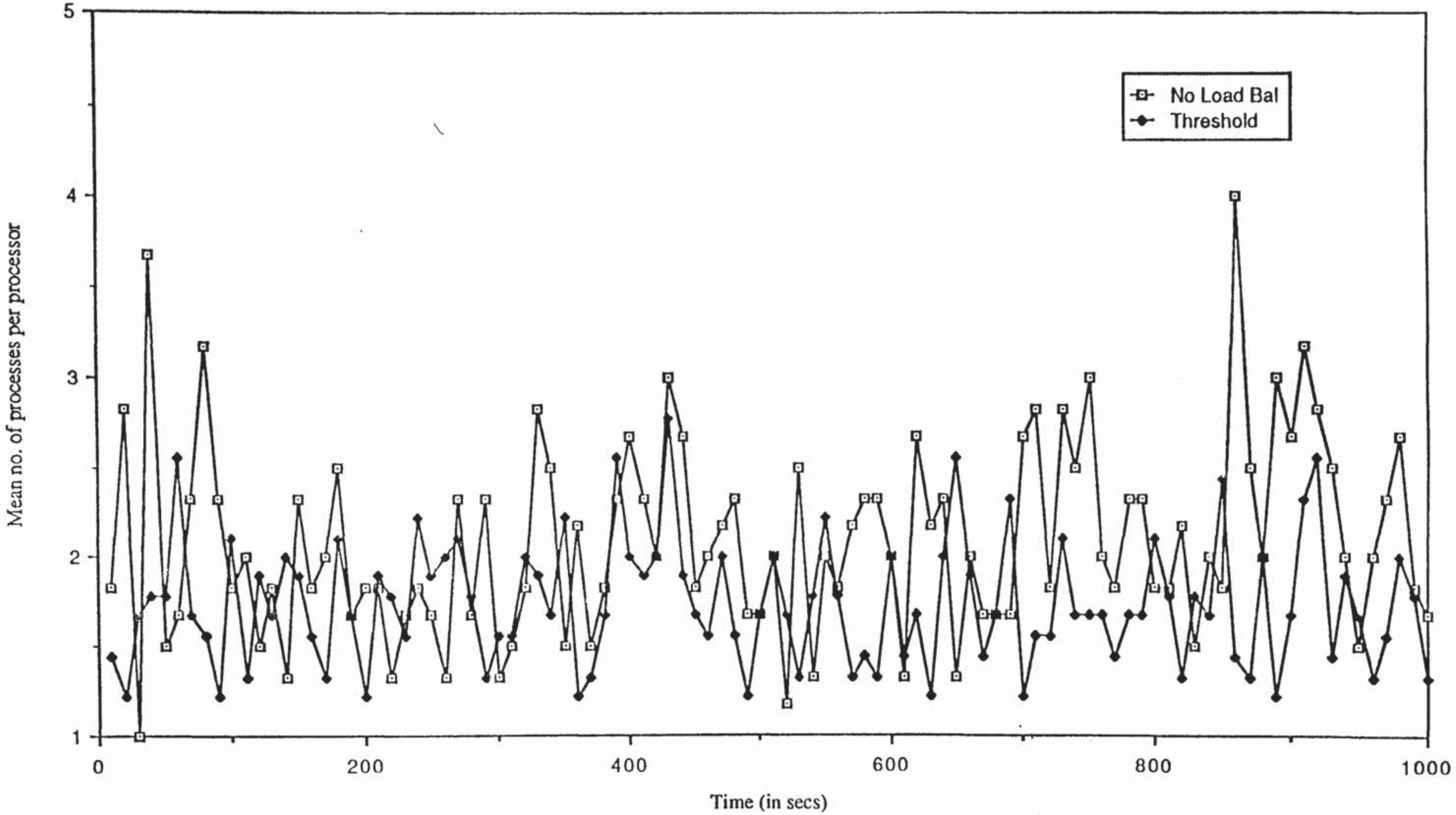

Fig. A.2.1 **Mean Load** - No Load Balancing vs Threshold using the Independent Process Model (Load Value = 0.5)

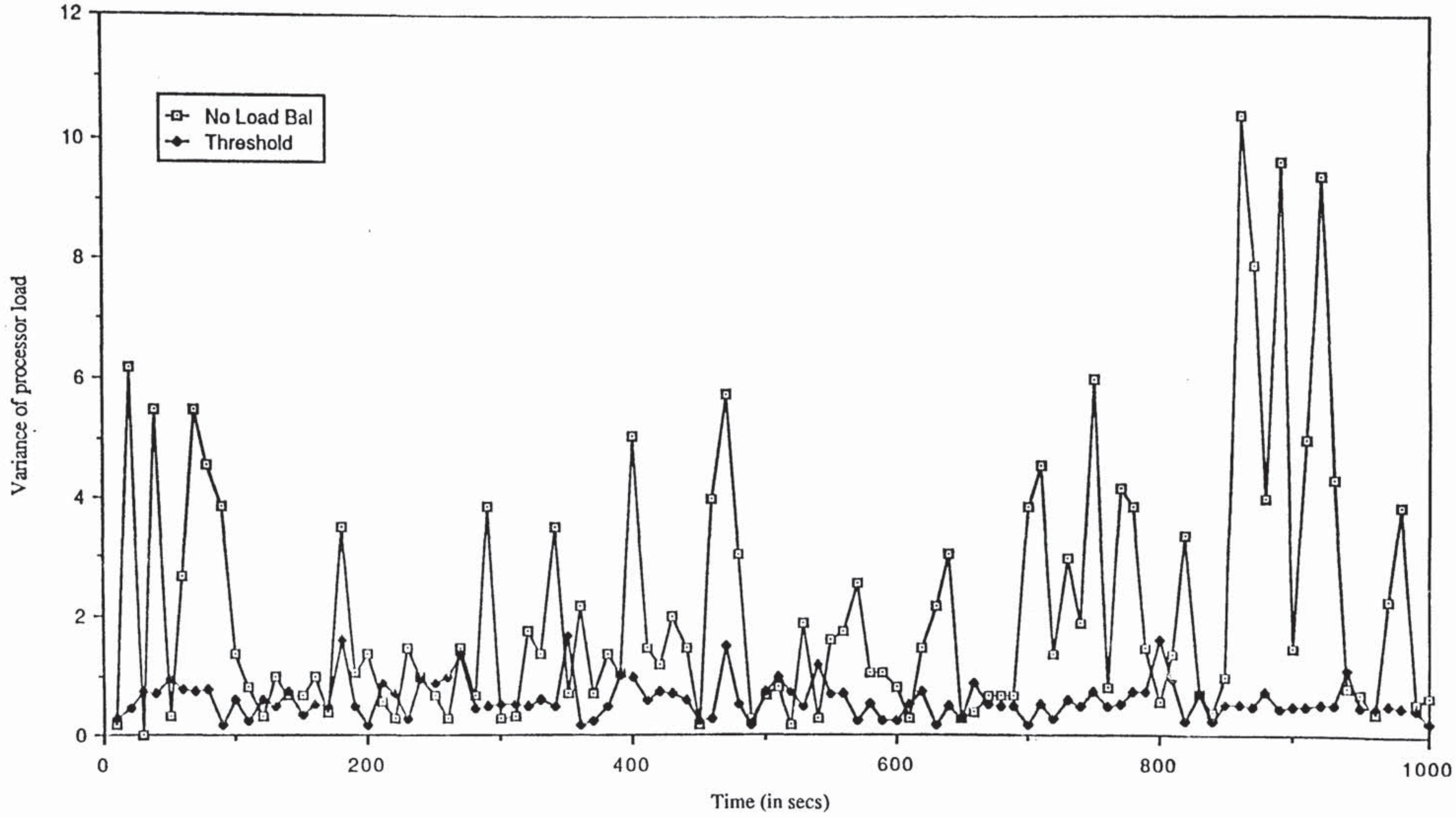

Fig. A.2.2 **Load Variance** - No Load Balancing vs Threshold
using the Independent Process Model (Load Value = 0.5)

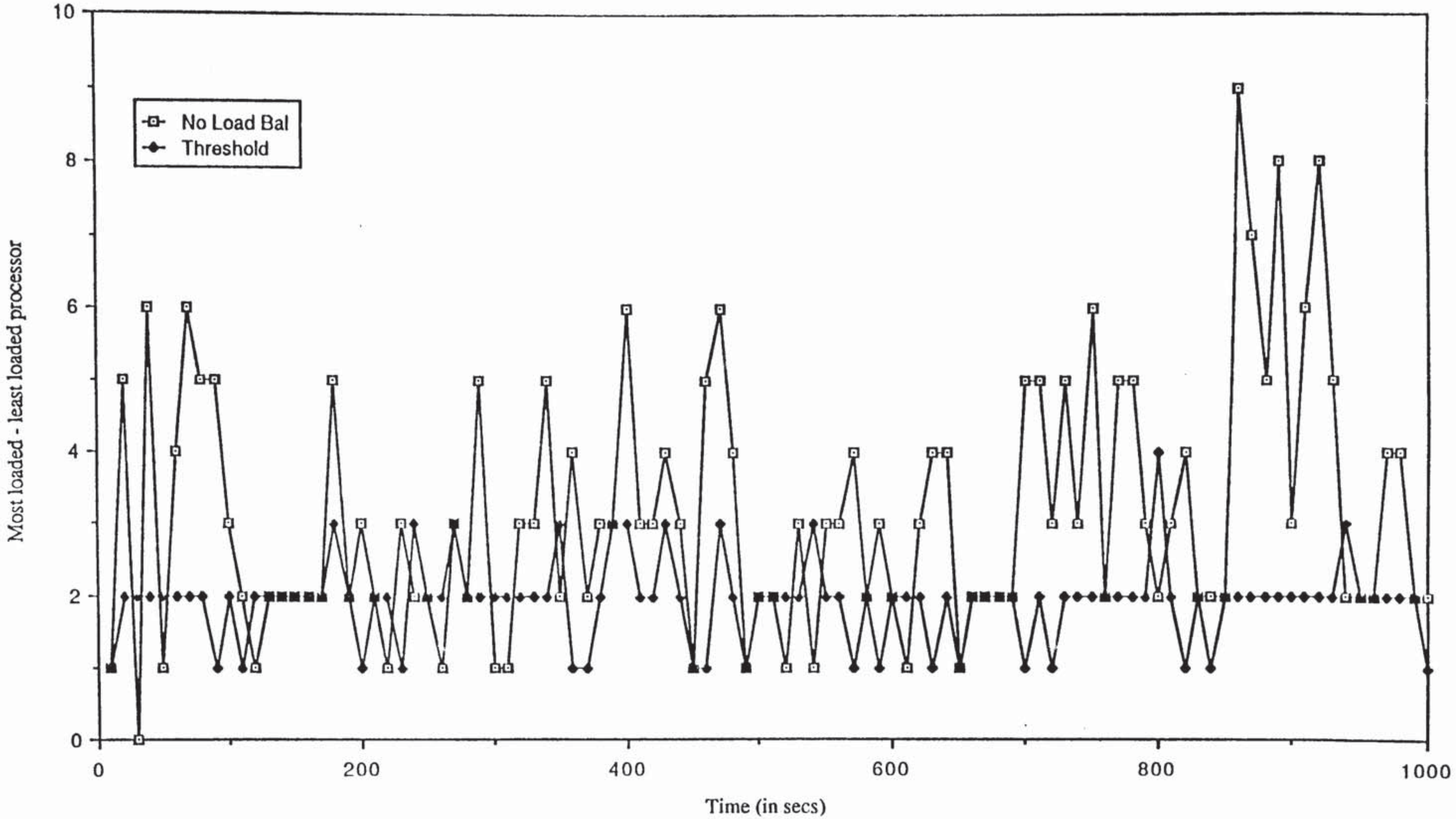

Fig. A.2.3 **Load Difference** - No Load Balancing vs Threshold using the Independent Process Model (Load Value = 0.5)

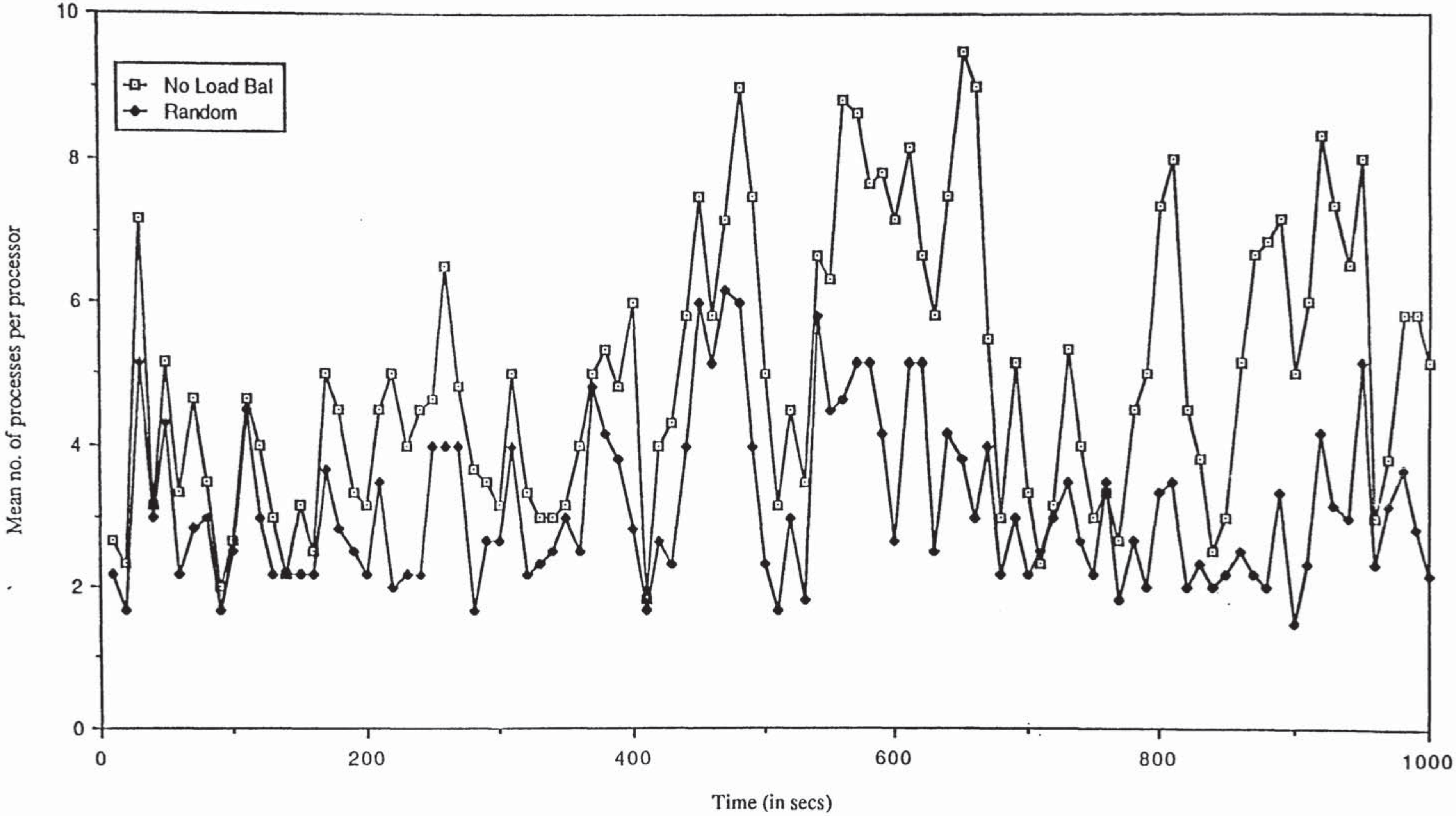

Fig. A.3.1 **Mean Load** - No Load Balancing vs Random
using the Independent Process Model (Load Value = 0.8)

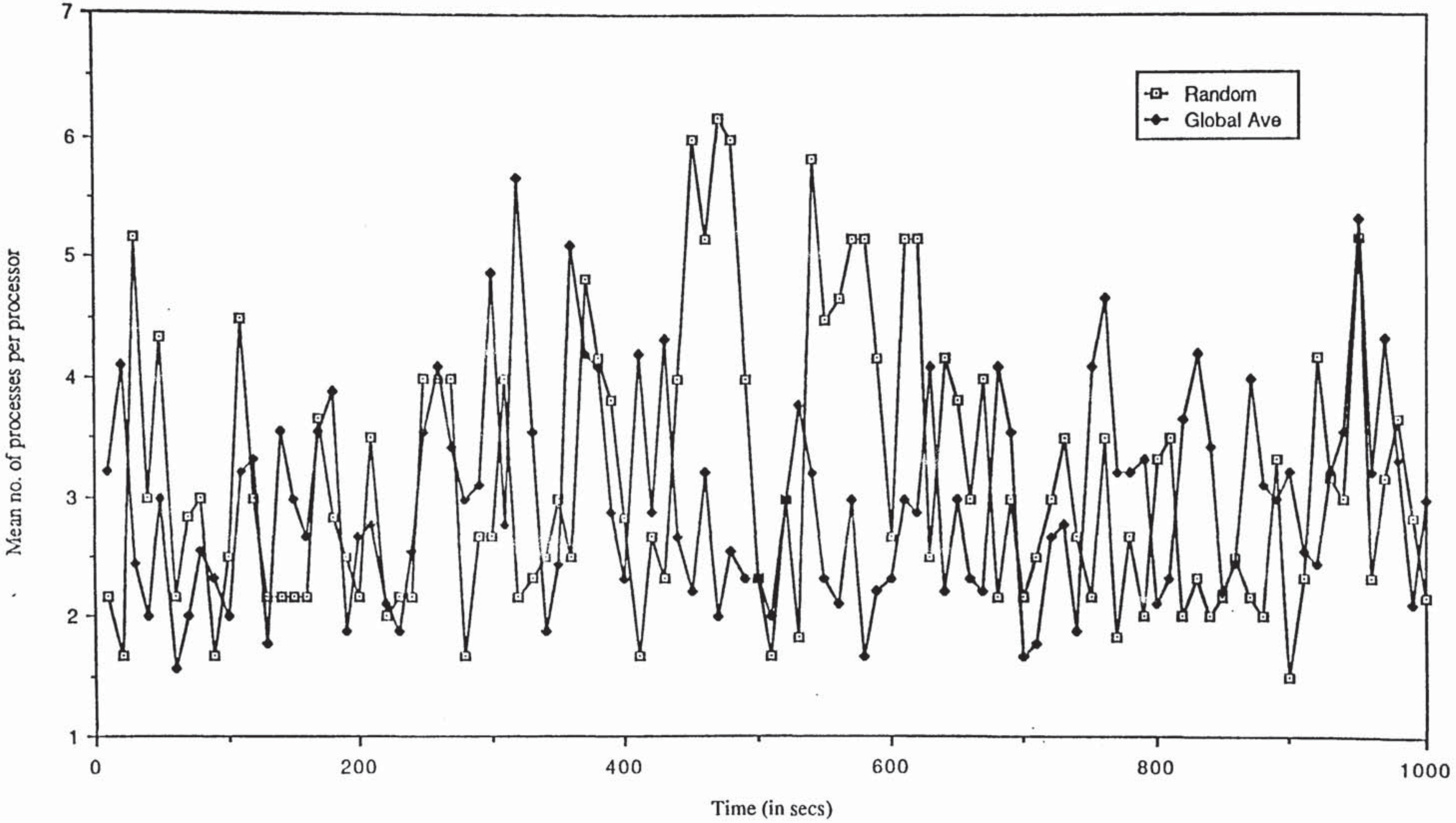

Fig. A.3.2 **Mean Load** - Random vs Global Average using the Independent Process Model (Load Value = 0.8)

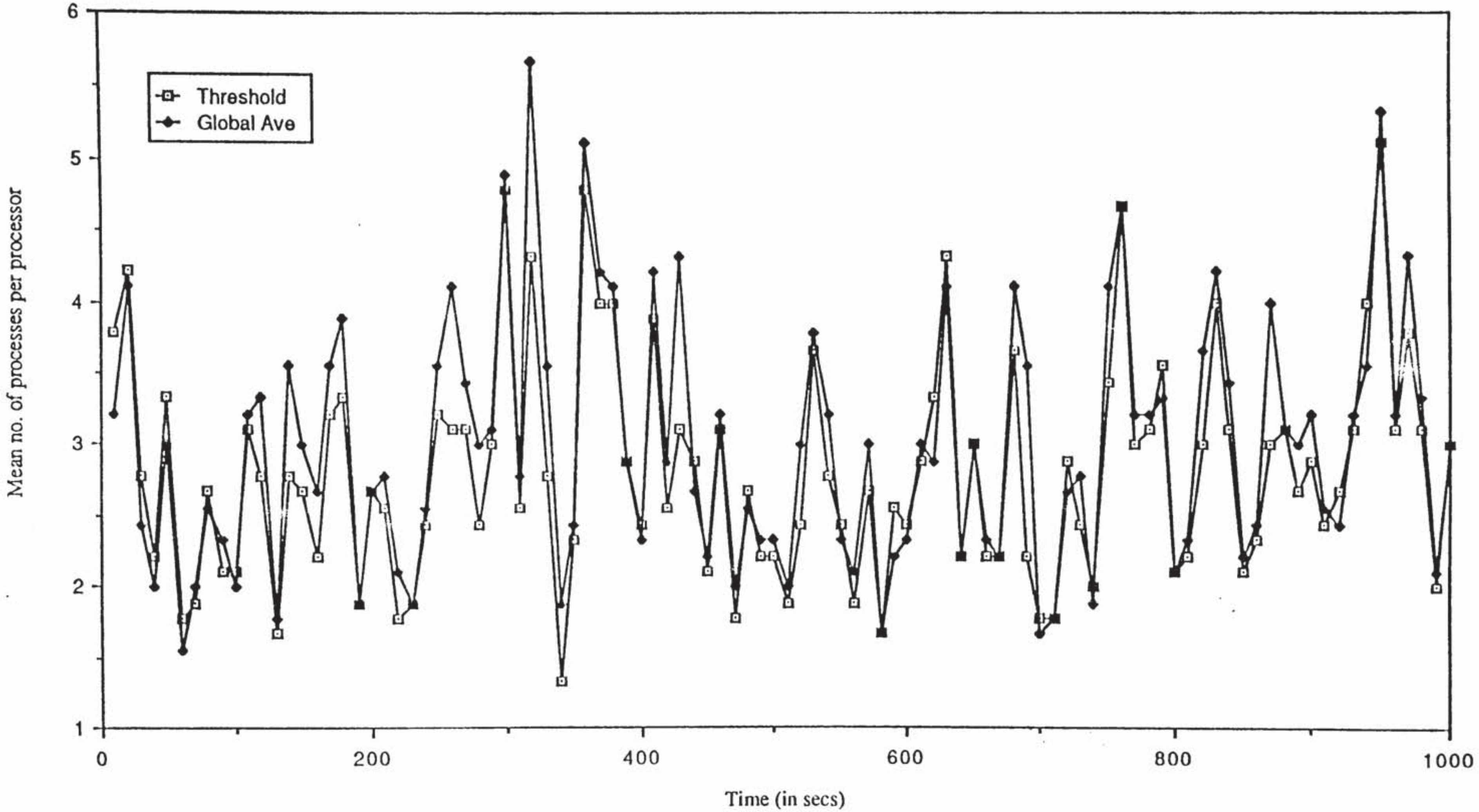

Fig. A.3.3 **Mean Load** - Global Average vs Threshold
using the Independent Process Model (Load Value = 0.8)

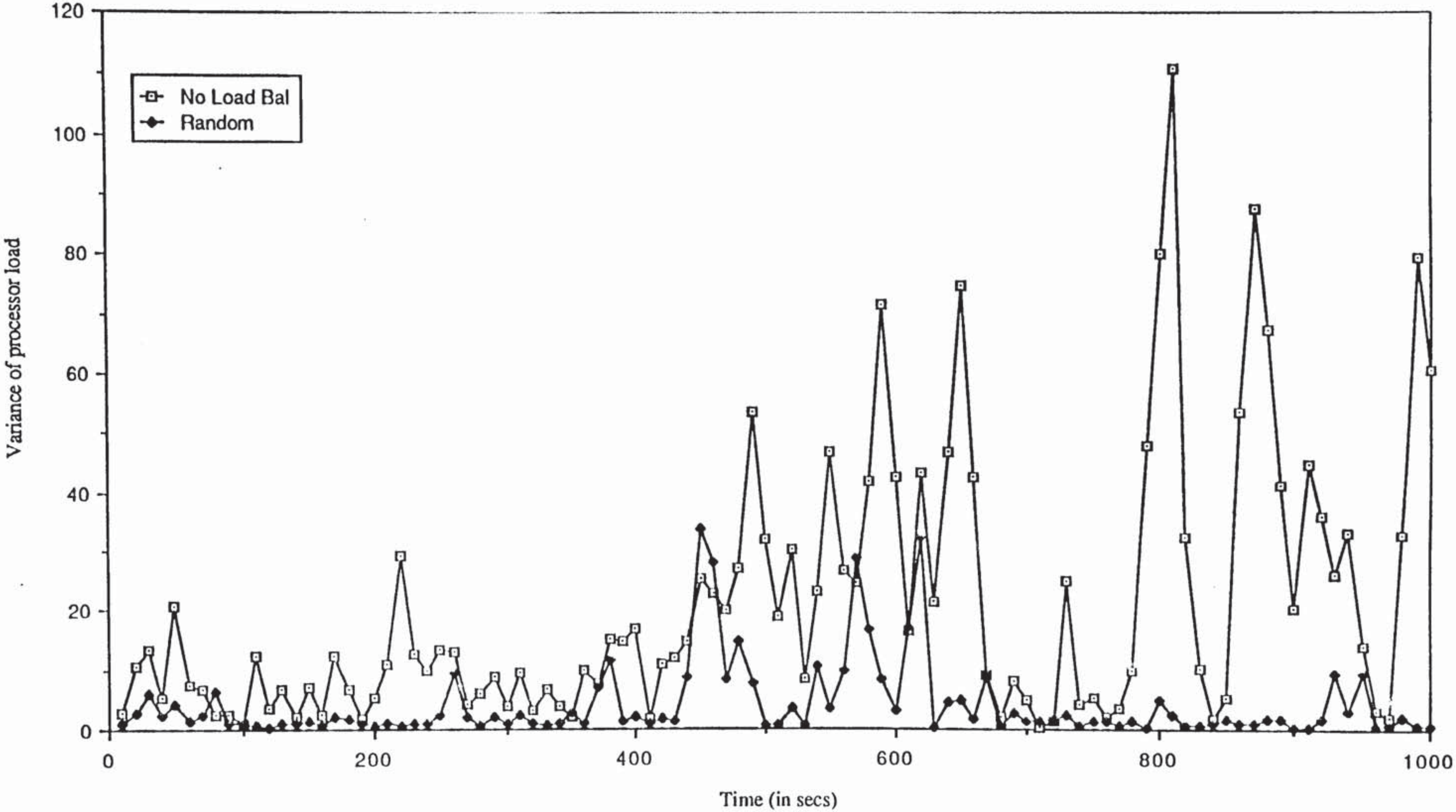

Fig. A.3.4 **Load Variance** - No Load Balancing vs Random
using the Independent Process Model (Load Value = 0.8)

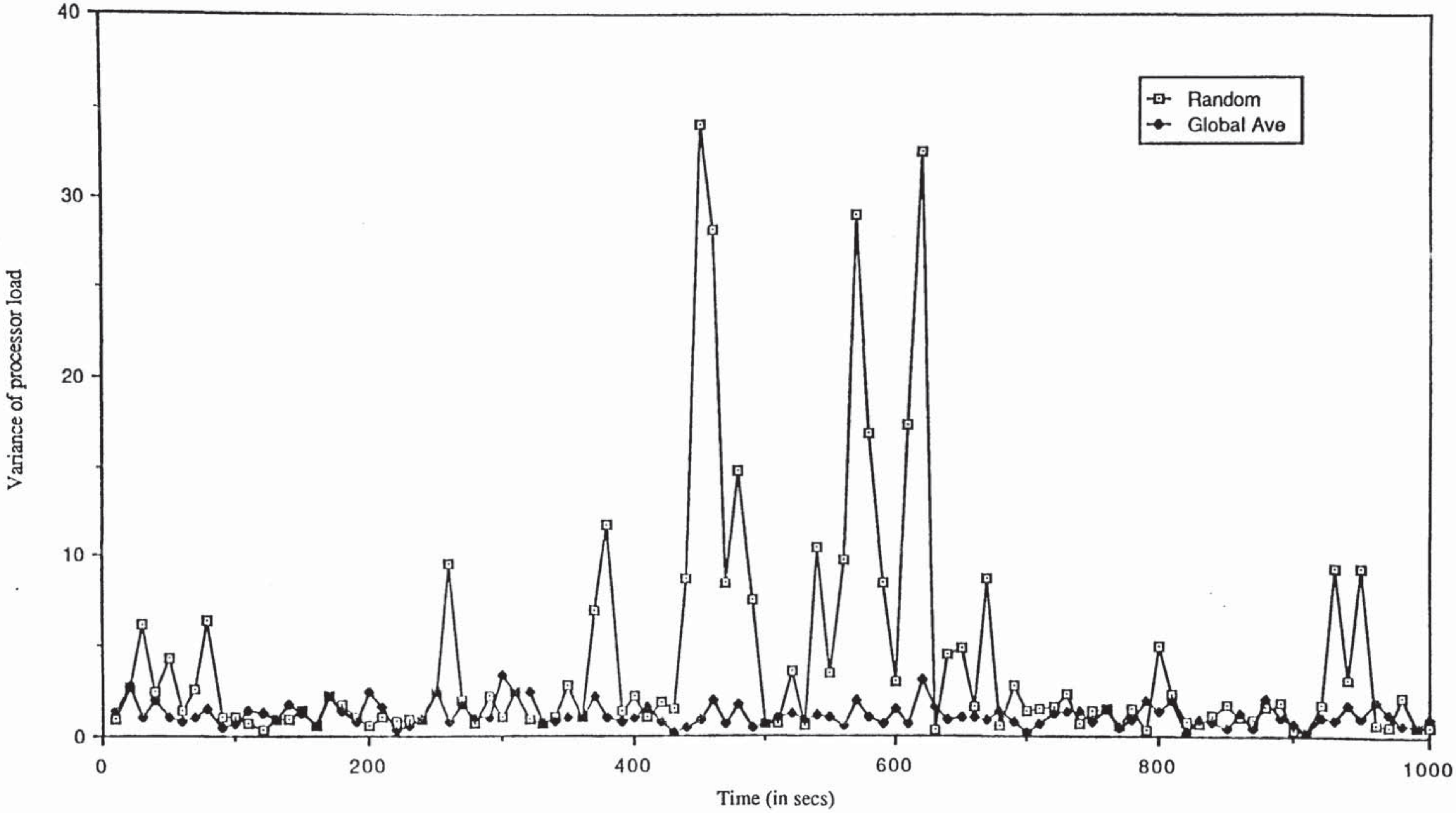

Fig. A.3.5 **Load Variance** - Random vs Global Average using the Independent Process Model (Load Value = 0.8)

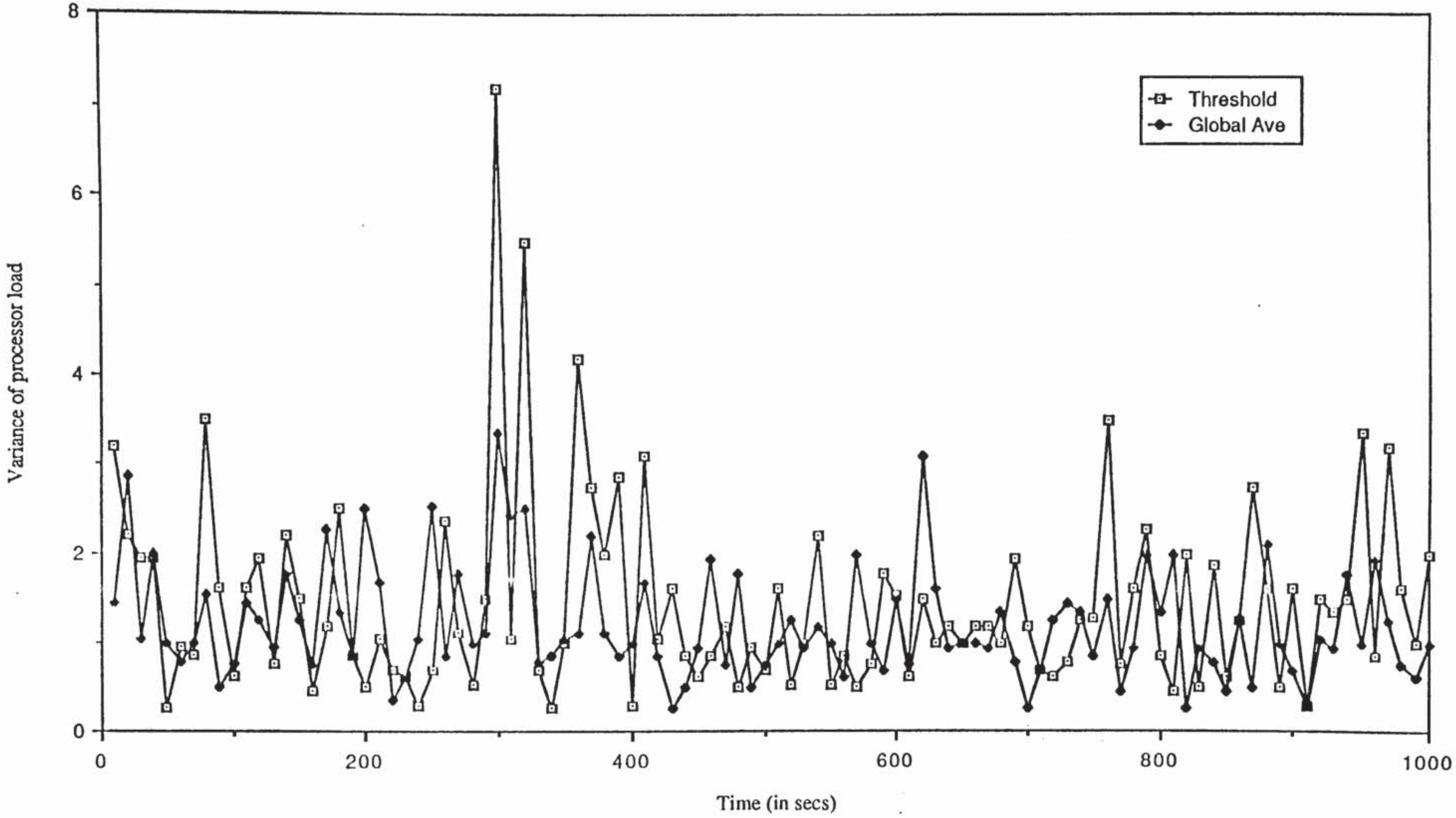

Fig. A.3.6 **Load Variance** - Global Average vs Threshold using the Independent Process Model (Load Value = 0.8)

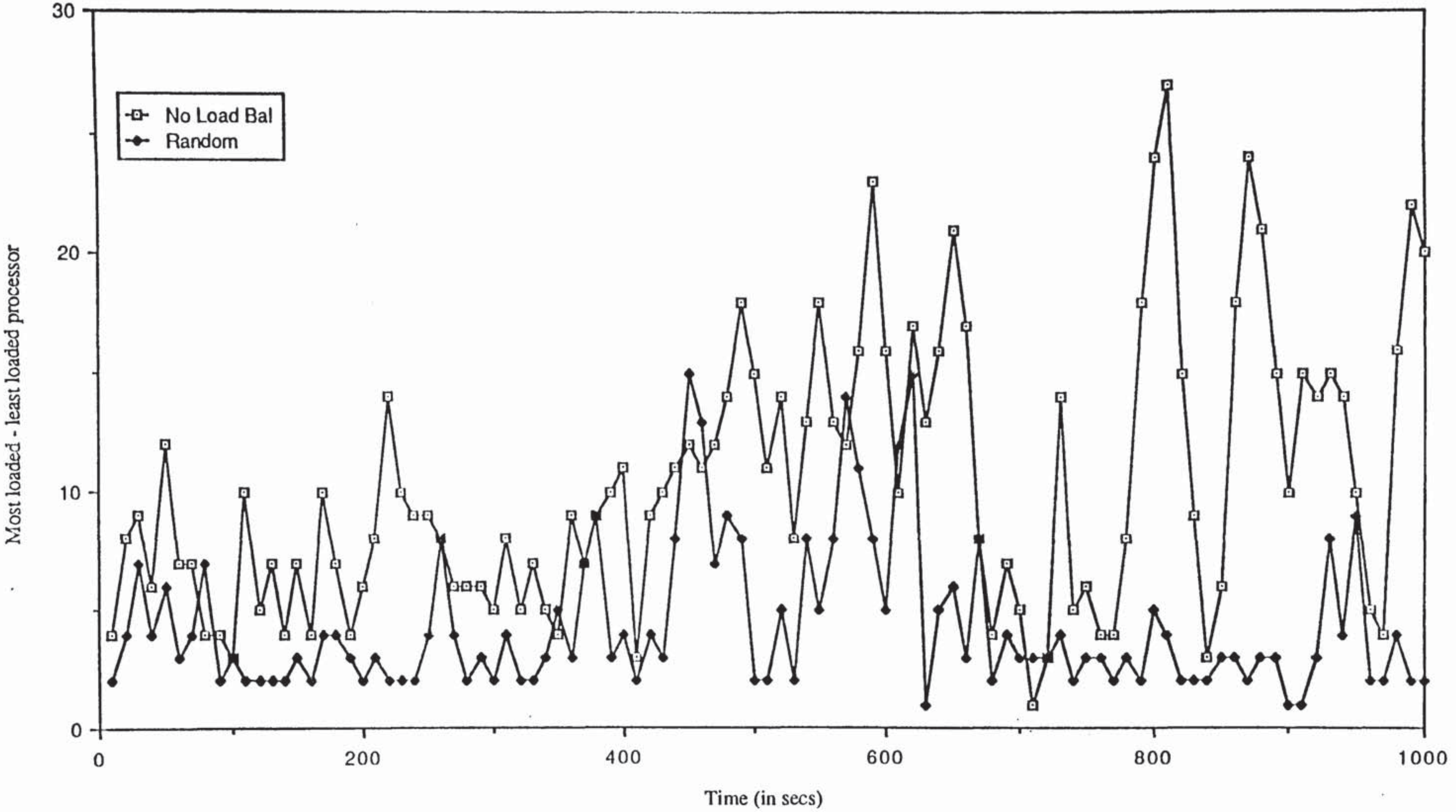

Fig. A.3.7 **Load Difference** - No Load Balancing vs Random using the Independent Process Model (Load Value = 0.8)

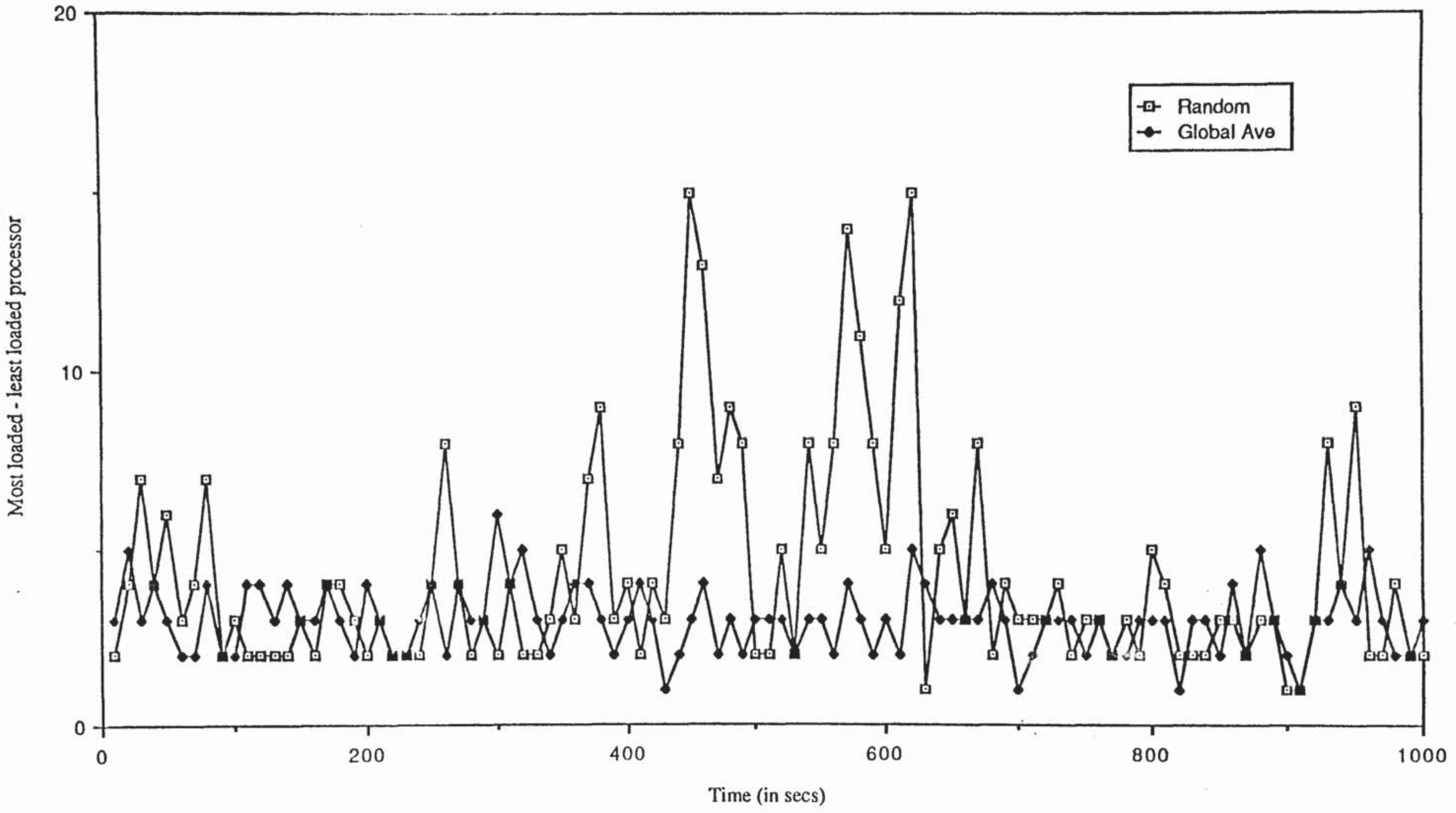

Fig. A.3.8 **Load Difference** - Random vs Global Average using the Independent Process Model (Load Value = 0.8)

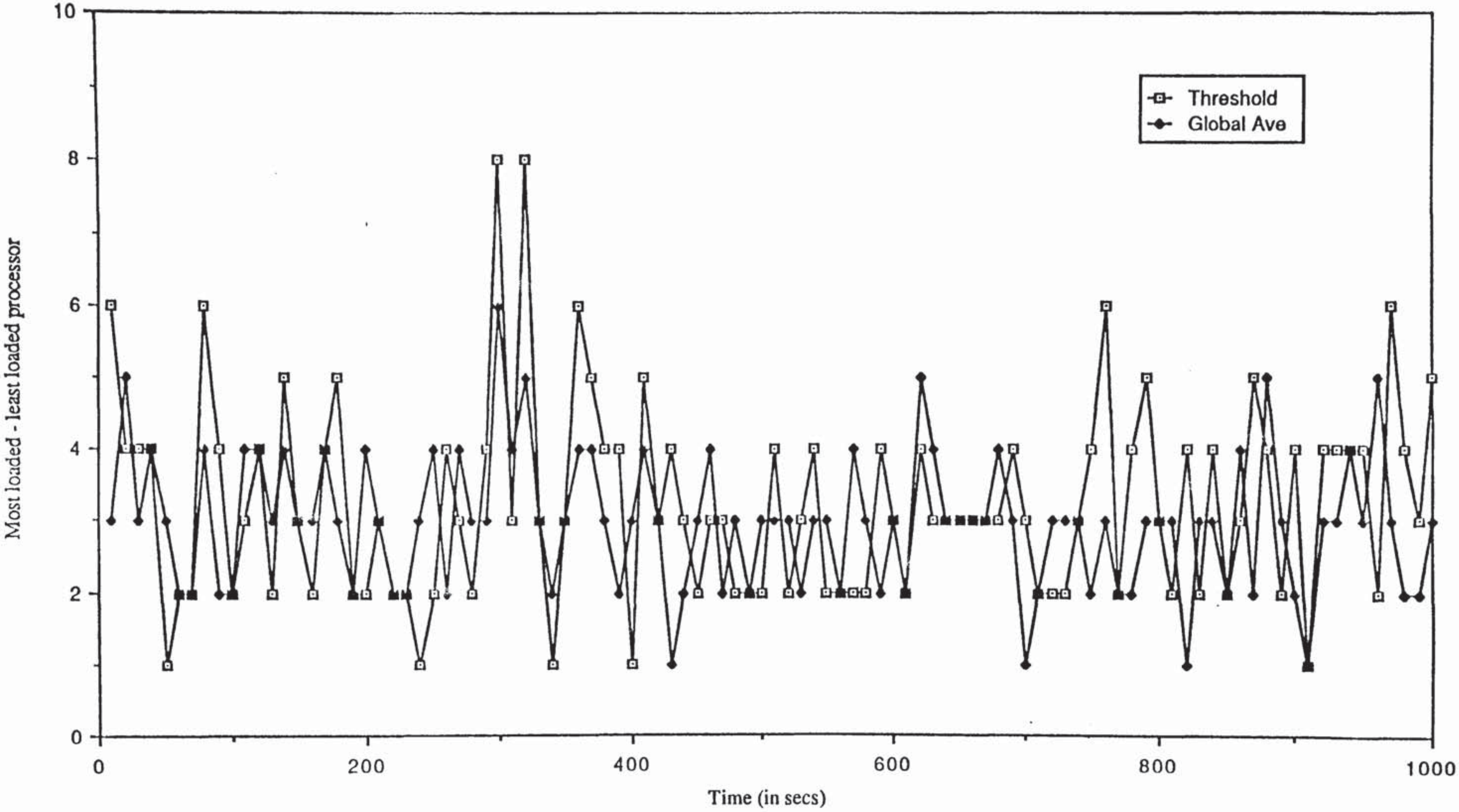

Fig. A.3.9 **Load Difference** - Global Average vs Threshold using the Independent Process Model (Load Value = 0.8)

APPENDIX B

Detailed Simulation Results (using the Cooperating Process Group Model)

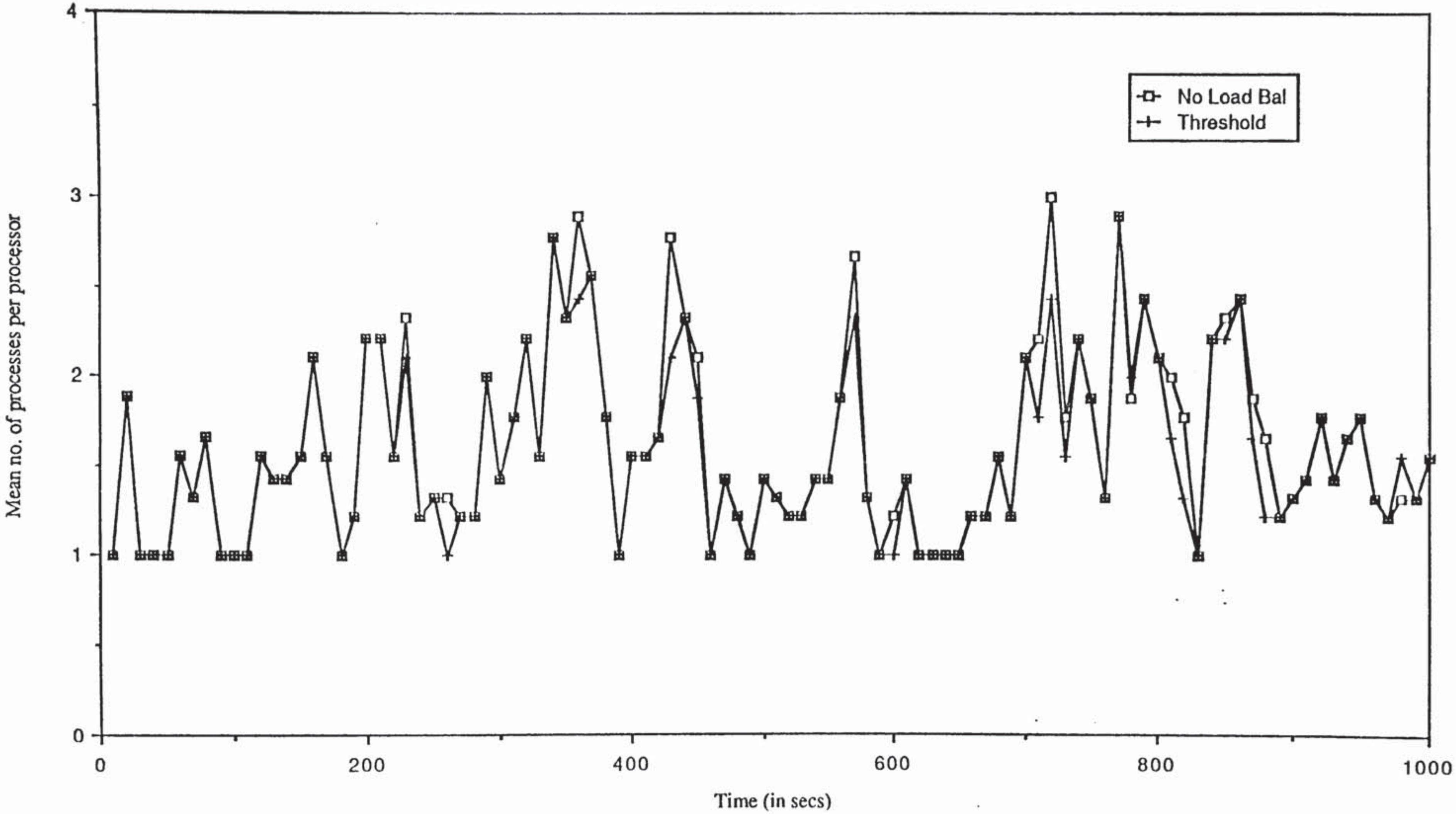

Fig. B.1.1 **Mean Load** - No Load Balancing vs Threshold
using the Cooperating Process Group Model (Load Value = 0.2)

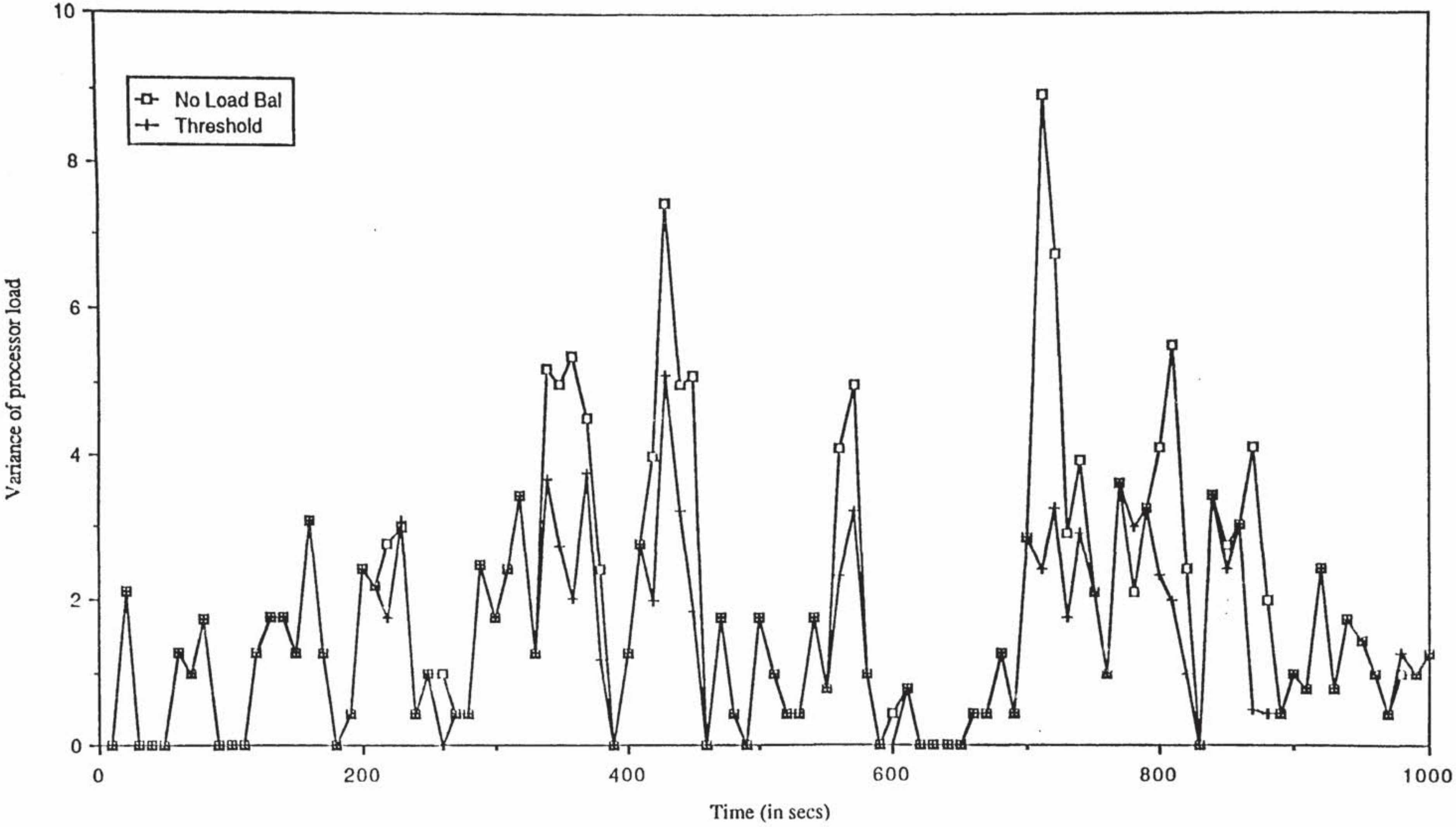

Fig. B.1.2 **Load Variance** - No Load Balancing vs Threshold using the Cooperating Process Group Model (Load Value = 0.2)

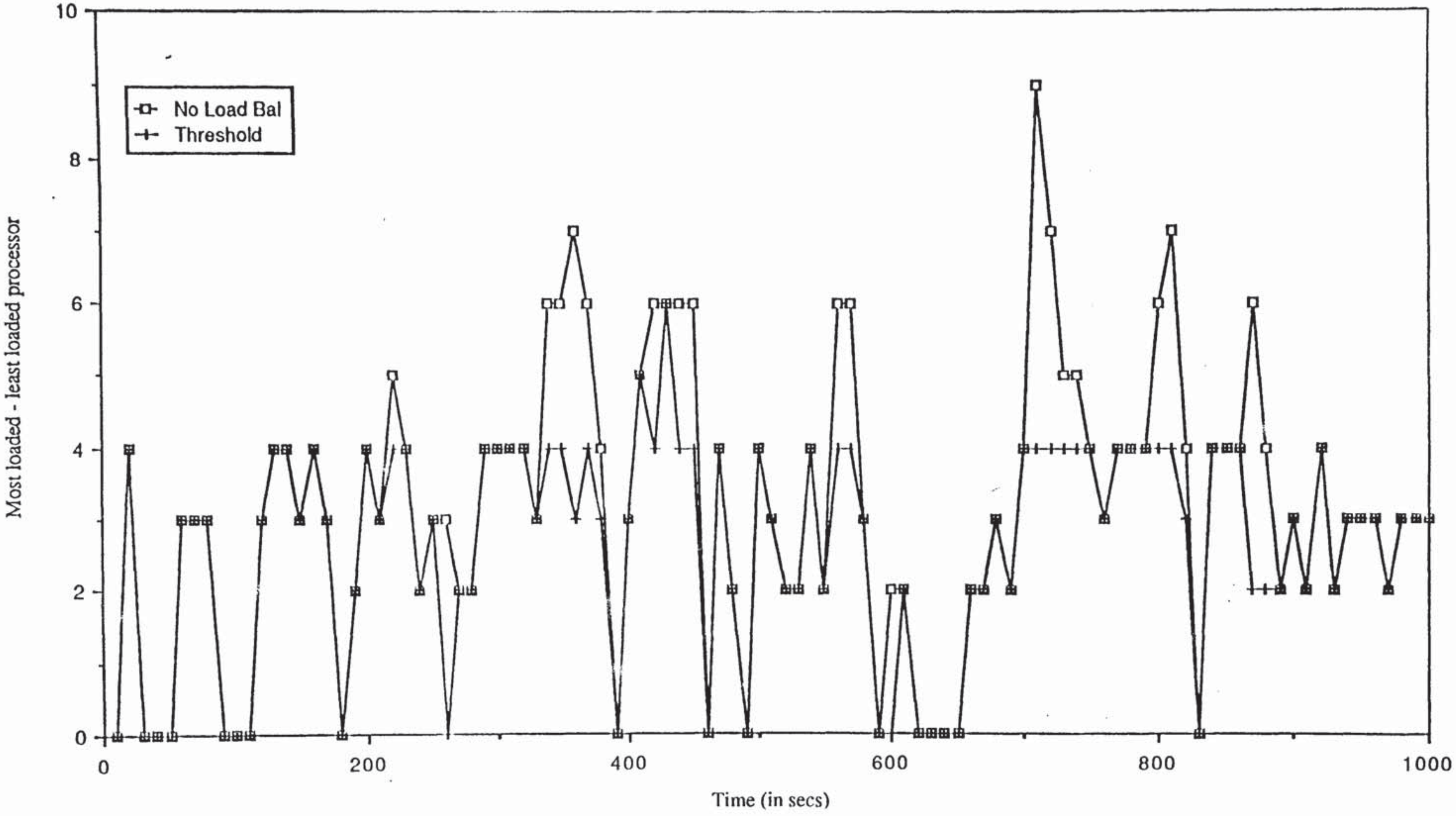

Fig. B.1.3 **Load Difference** - No Load Balancing vs Threshold using the Cooperating Process Group Model (Load Value = 0.2)

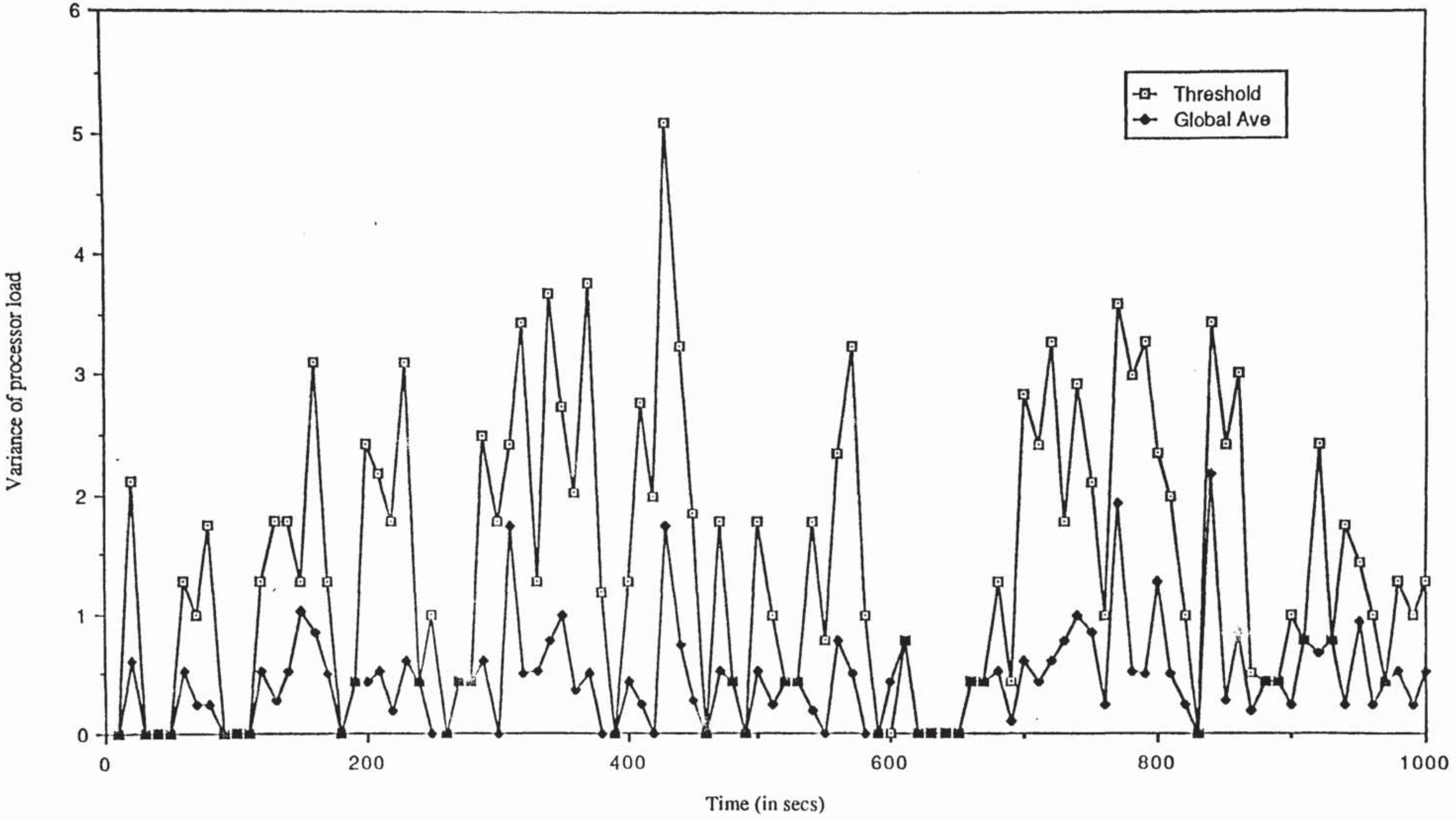

Fig. B.1.4 **Load Variance** - Threshold vs Global Average
using the Cooperating Process Group Model (Load Value = 0.2)

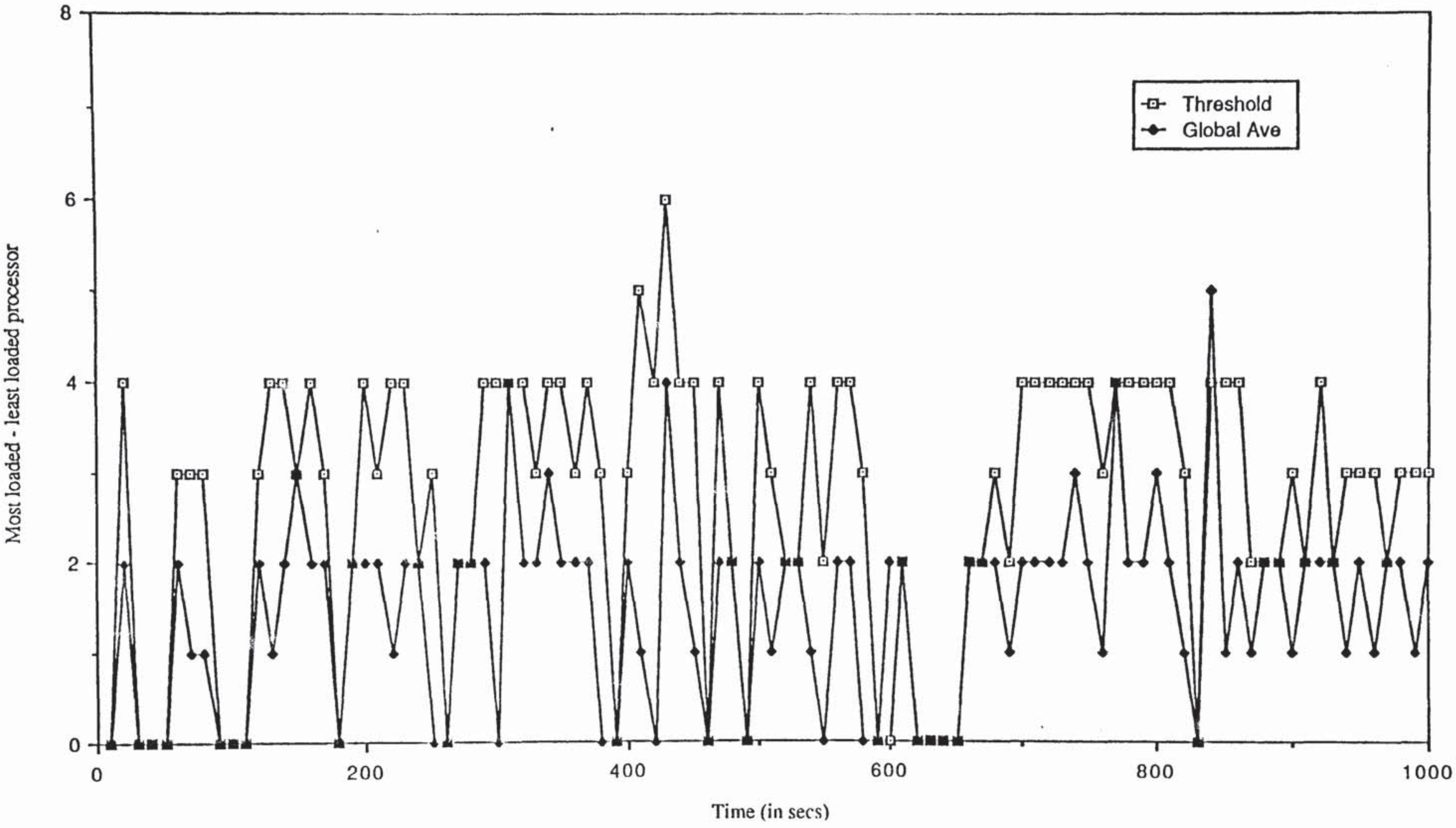

Fig. B.1.5 **Load Difference** - Threshold vs Global Average using the Cooperating Process Group Model (Load Value = 0.2)

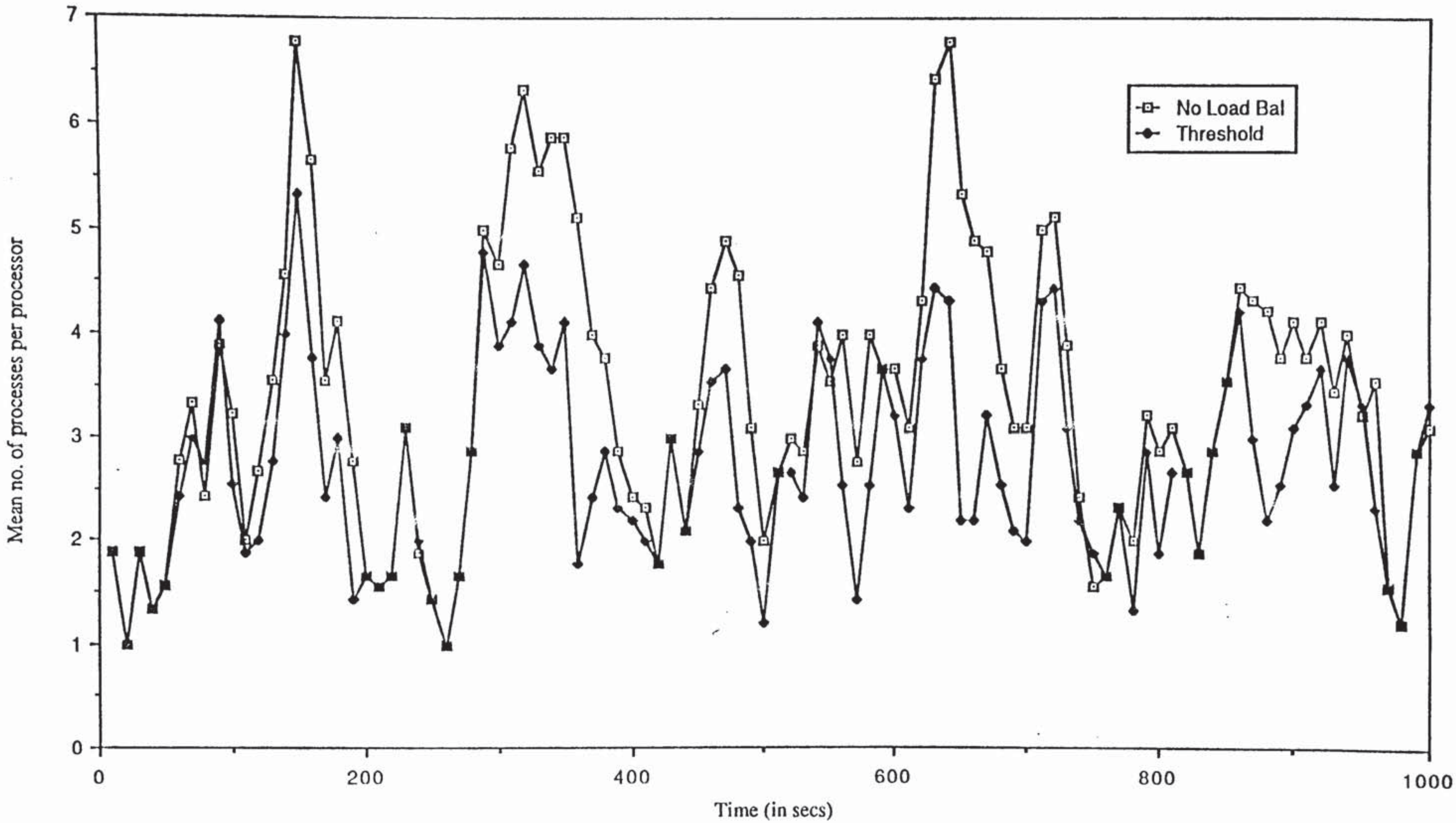

Fig. B.2.1 **Mean Load** - No Load Balancing vs Threshold using the Cooperating Process Group Model (Load Value = 0.5)

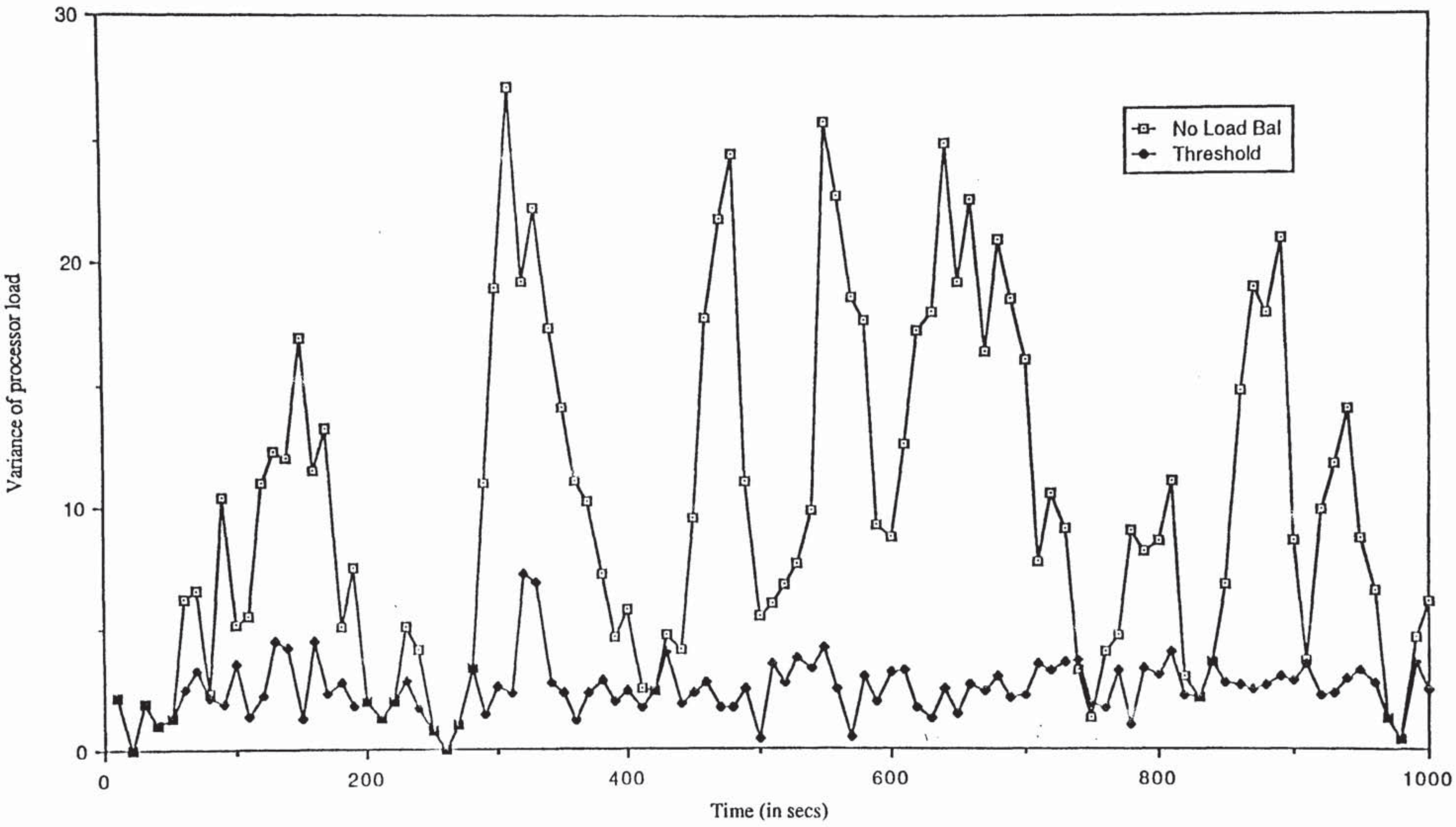

Fig. B.2.2 **Load Variance** - No Load Balancing vs Threshold using the Cooperating Process Group Model (Load Value = 0.5)

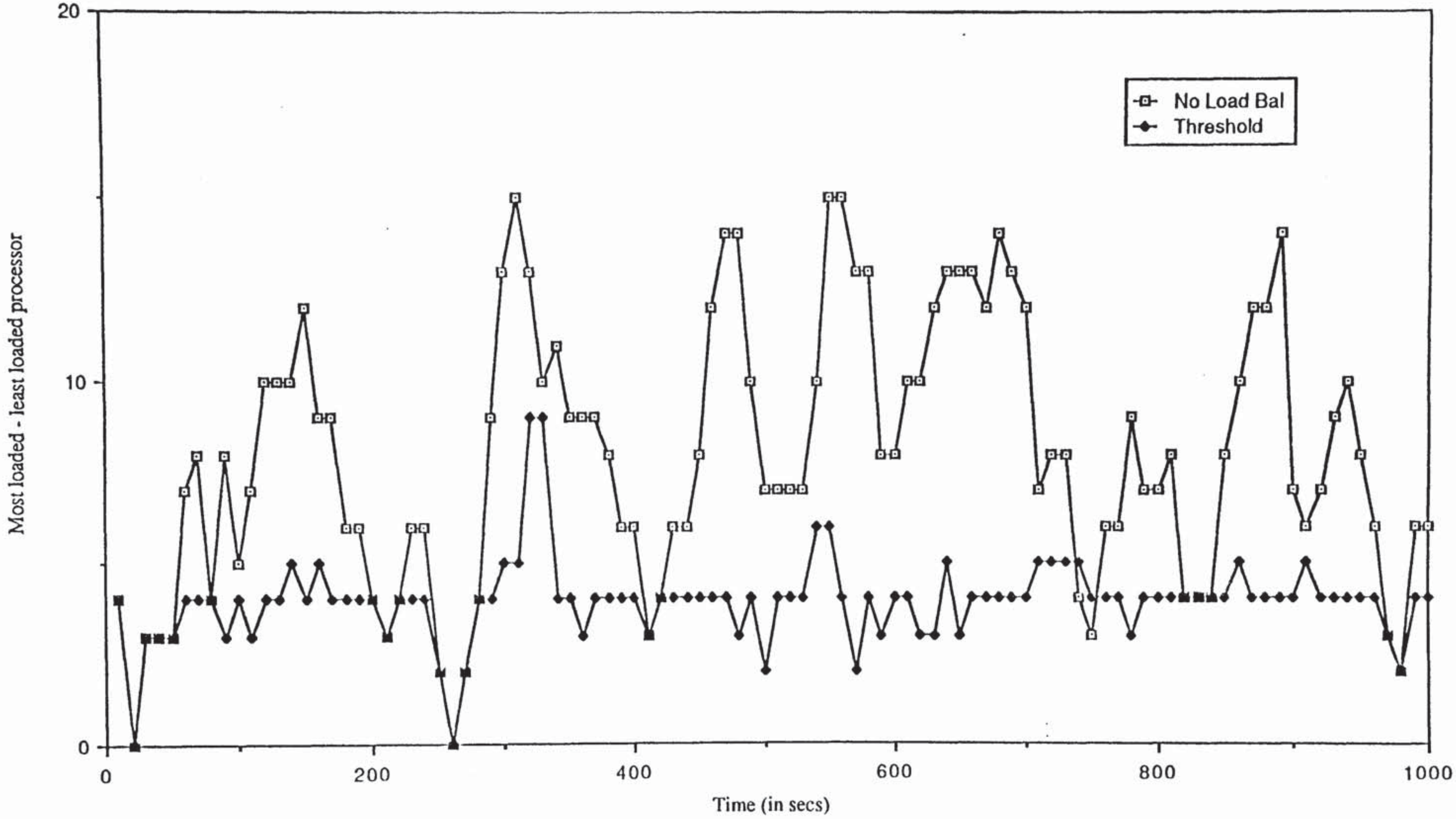

Fig. B.2.3 **Load Difference** - No Load Balancing vs Threshold using the Cooperating Process Group Model (Load Value = 0.5)

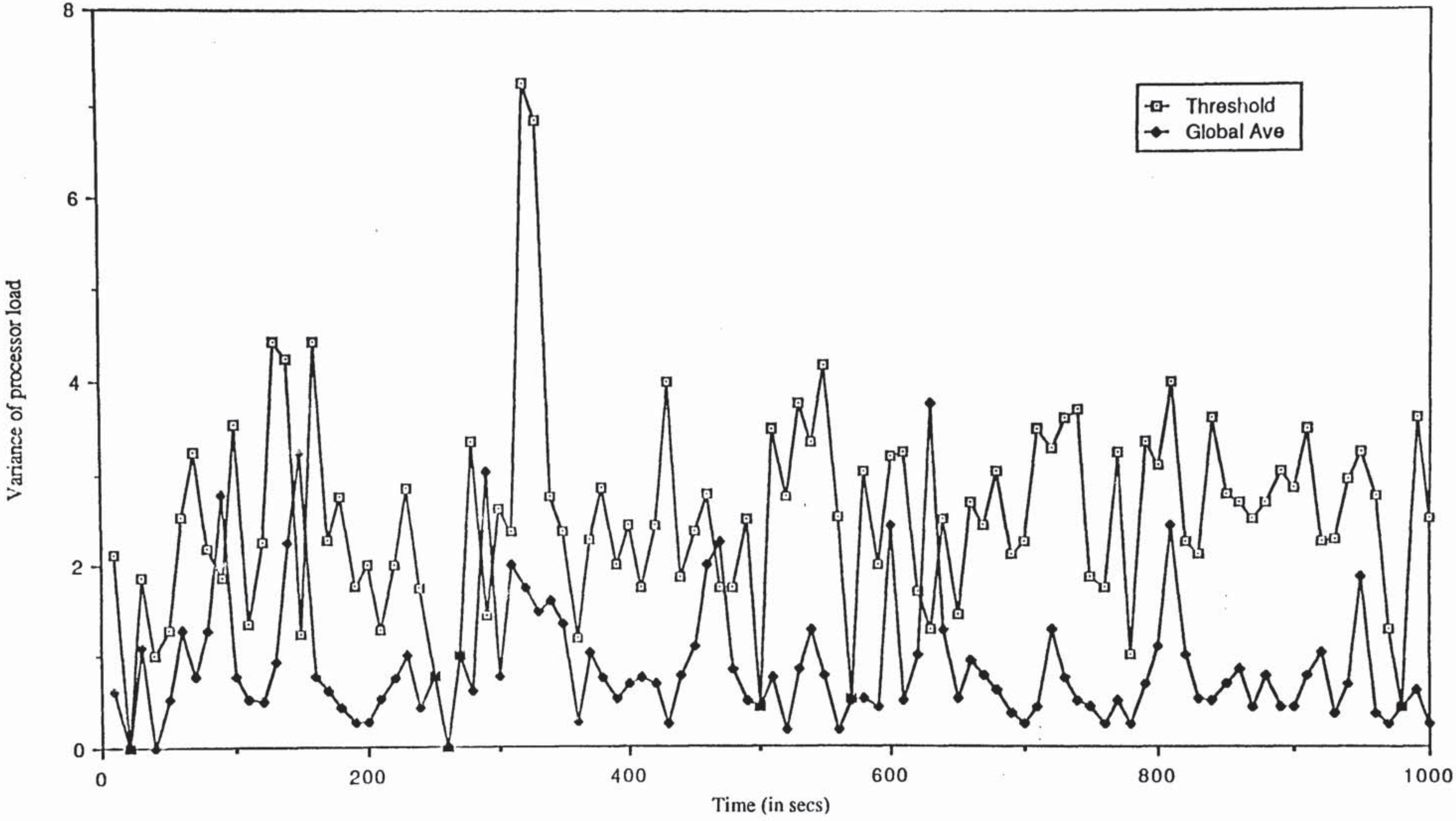

Fig. B.2.4 **Load Variance** - Threshold vs Global Average using the Cooperating Process Group Model (Load Value = 0.5)

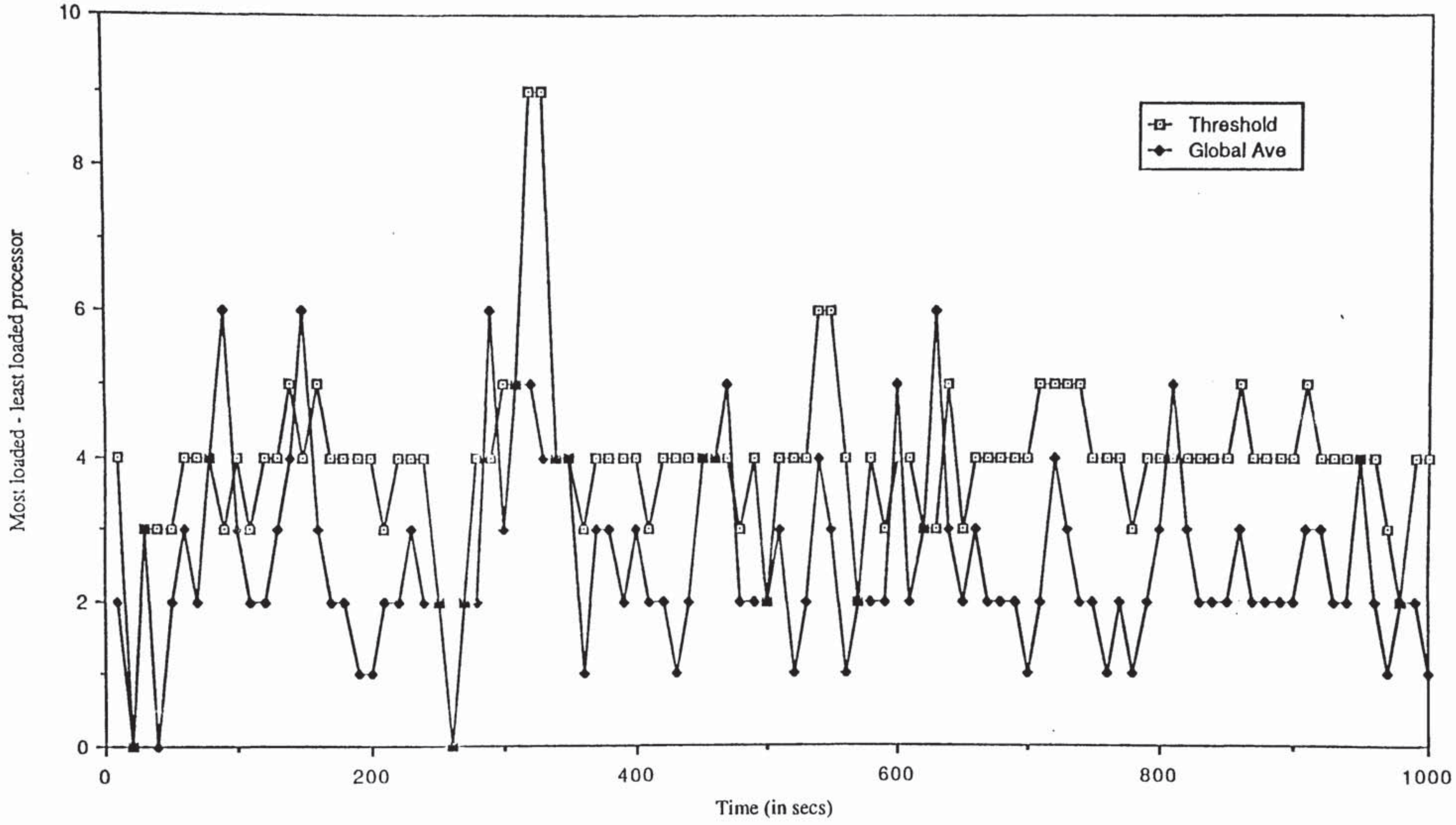

Fig. B.2.5 **Load Difference** - Threshold vs Global Average using the Cooperating Process Group Model (Load Value = 0.5)

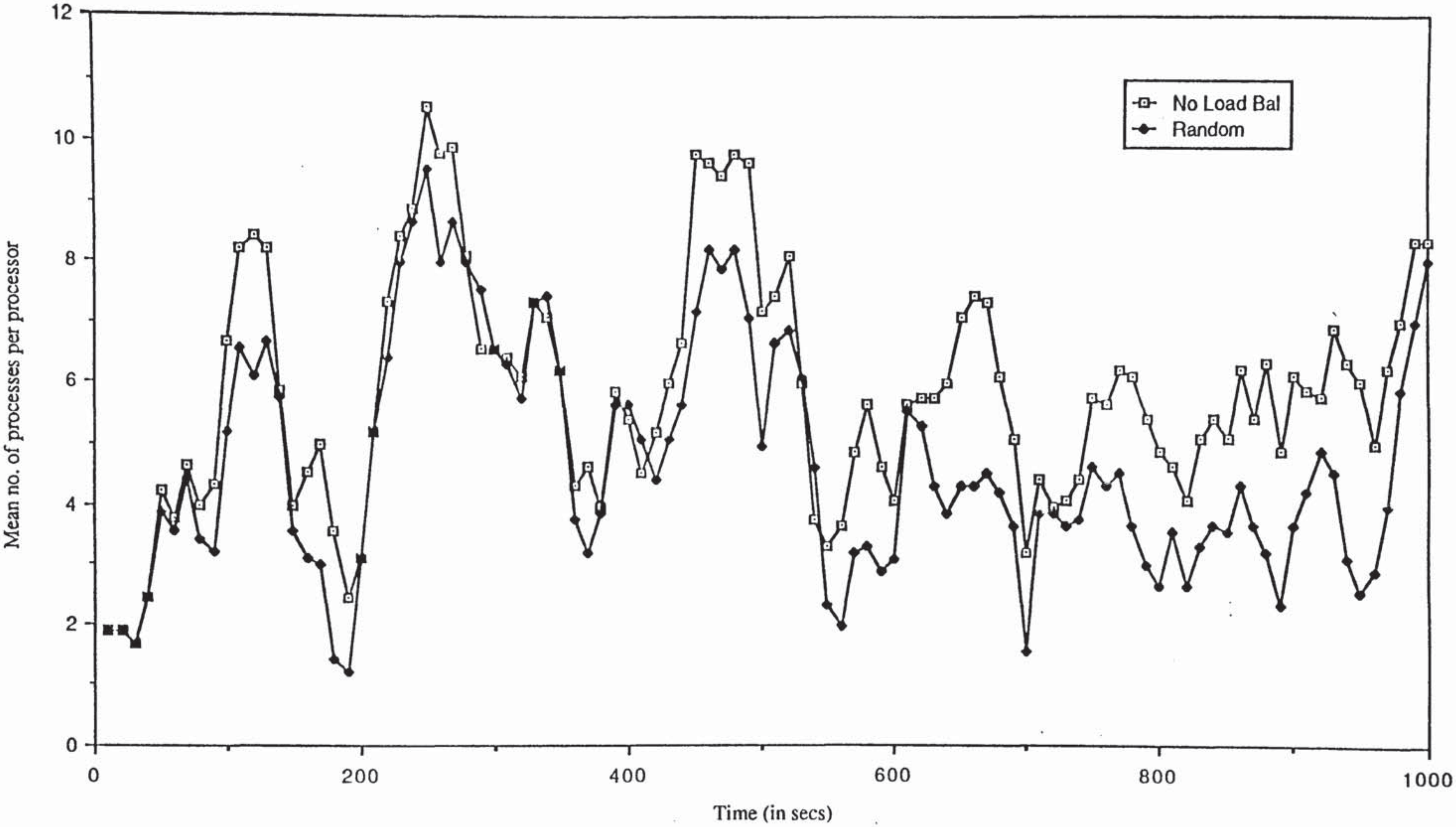

Fig. B.3.1 **Mean Load** - No Load Balancing vs Random
using the Cooperating Process Group Model (Load Value = 0.7)

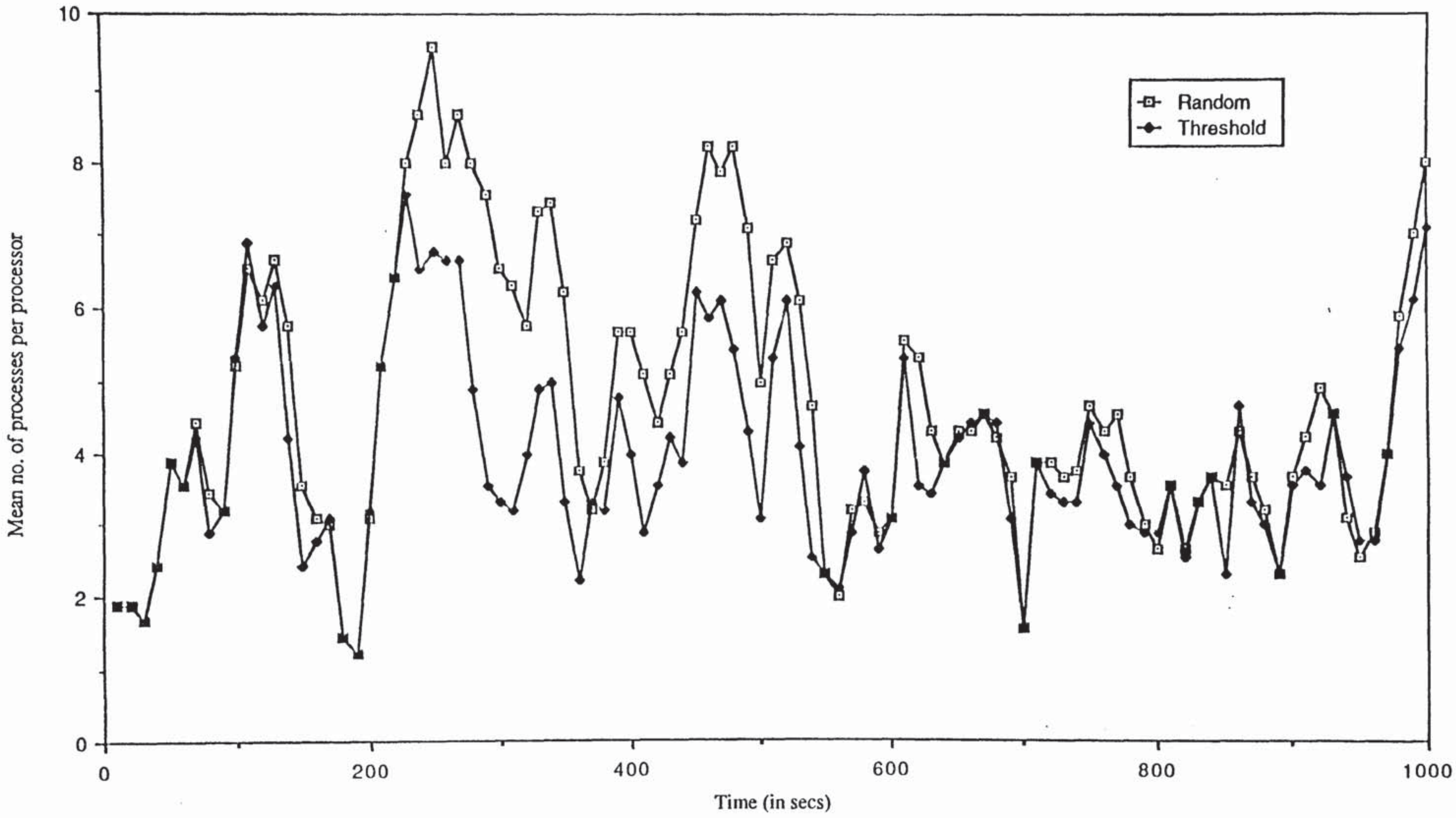

Fig. B.3.2 **Mean Load** - Random vs Threshold
using the Cooperating Process Group Model (Load Value = 0.7)

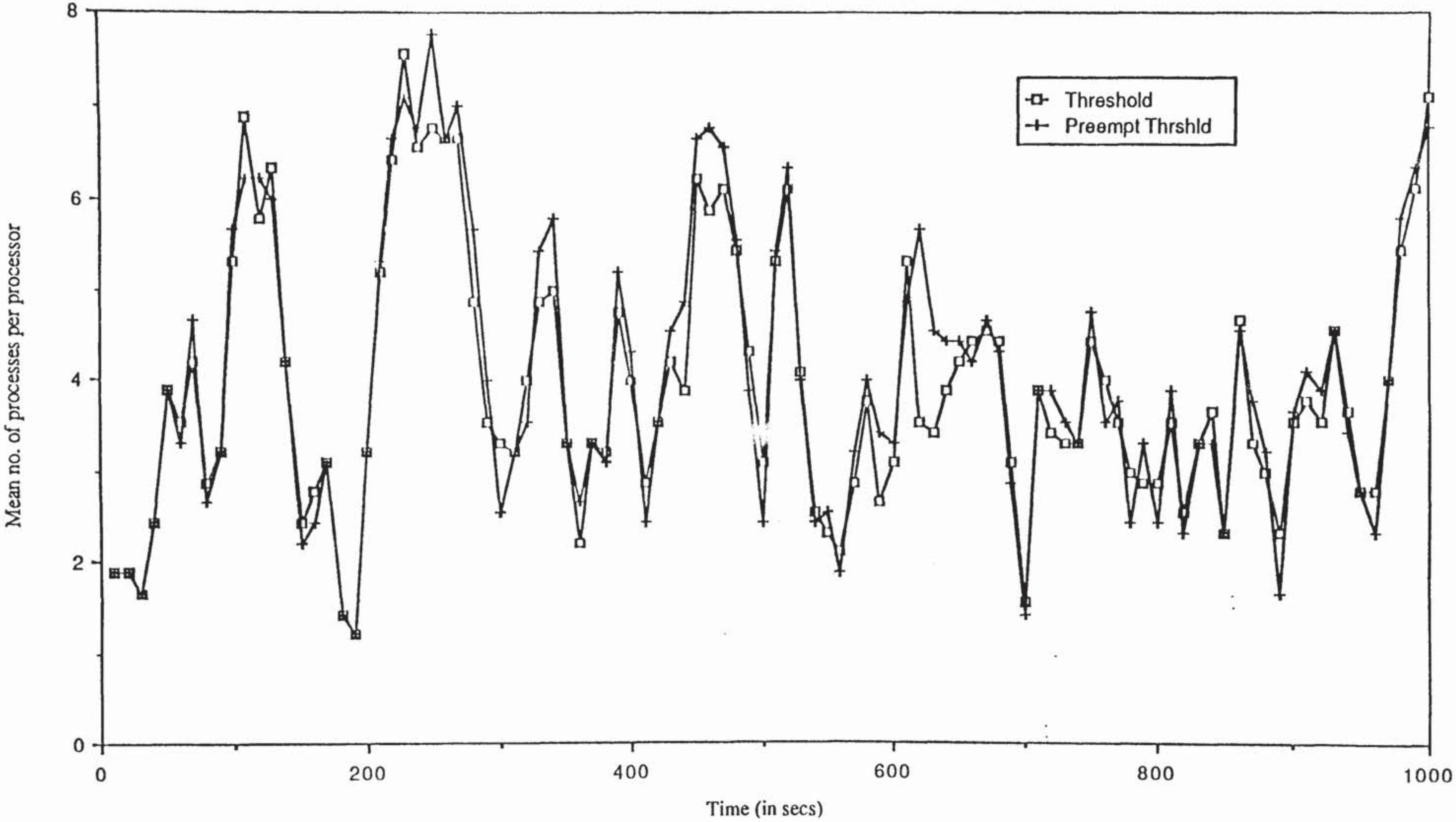

Fig. B.3.3 **Mean Load** - Threshold vs Preemptive Threshold
using the Cooperating Process Group Model (Load Value = 0.7)

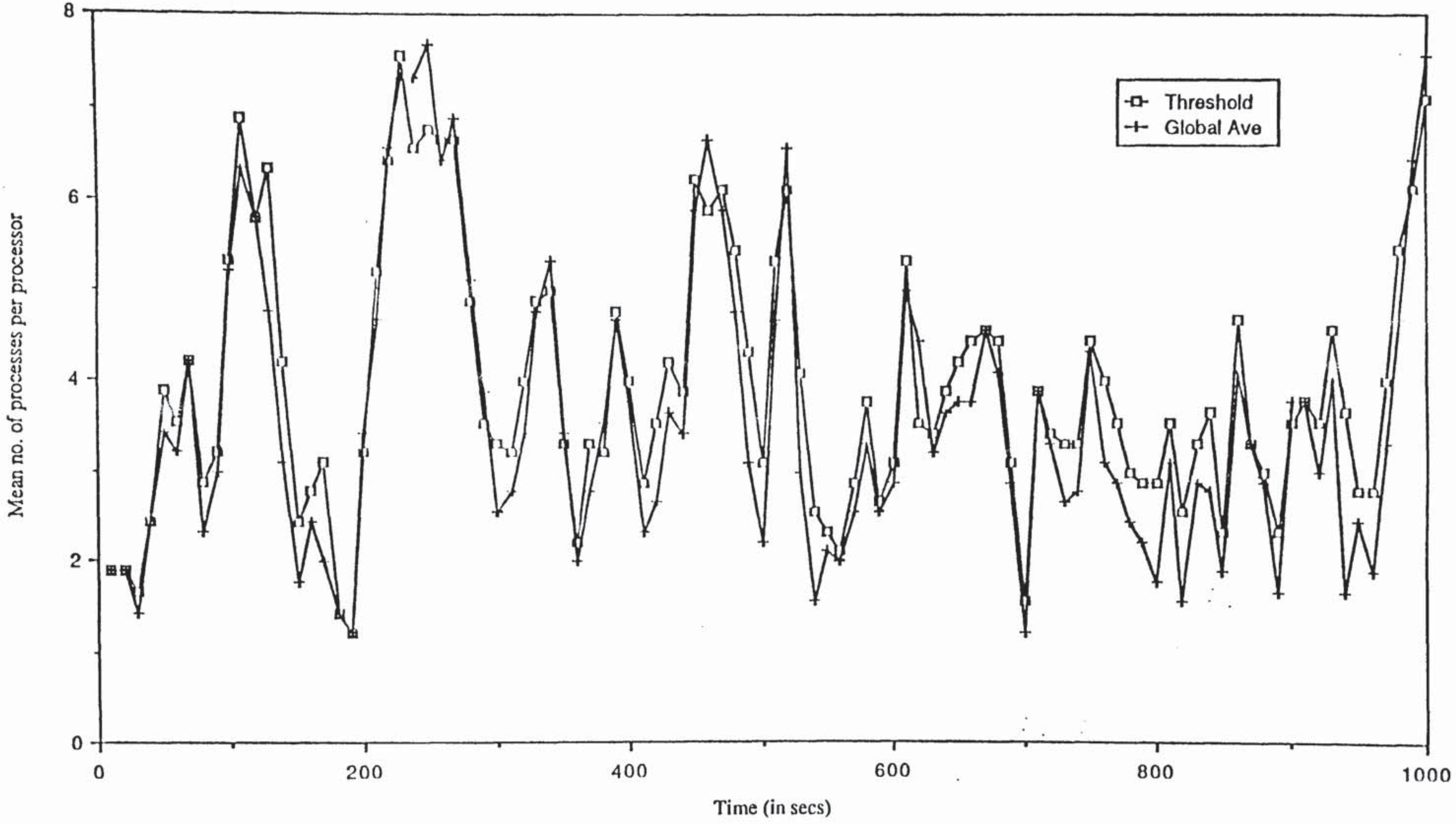

Fig. B.3.4 **Mean Load** - Threshold vs Global Average
using the Cooperating Process Group Model (Load Value = 0.7)

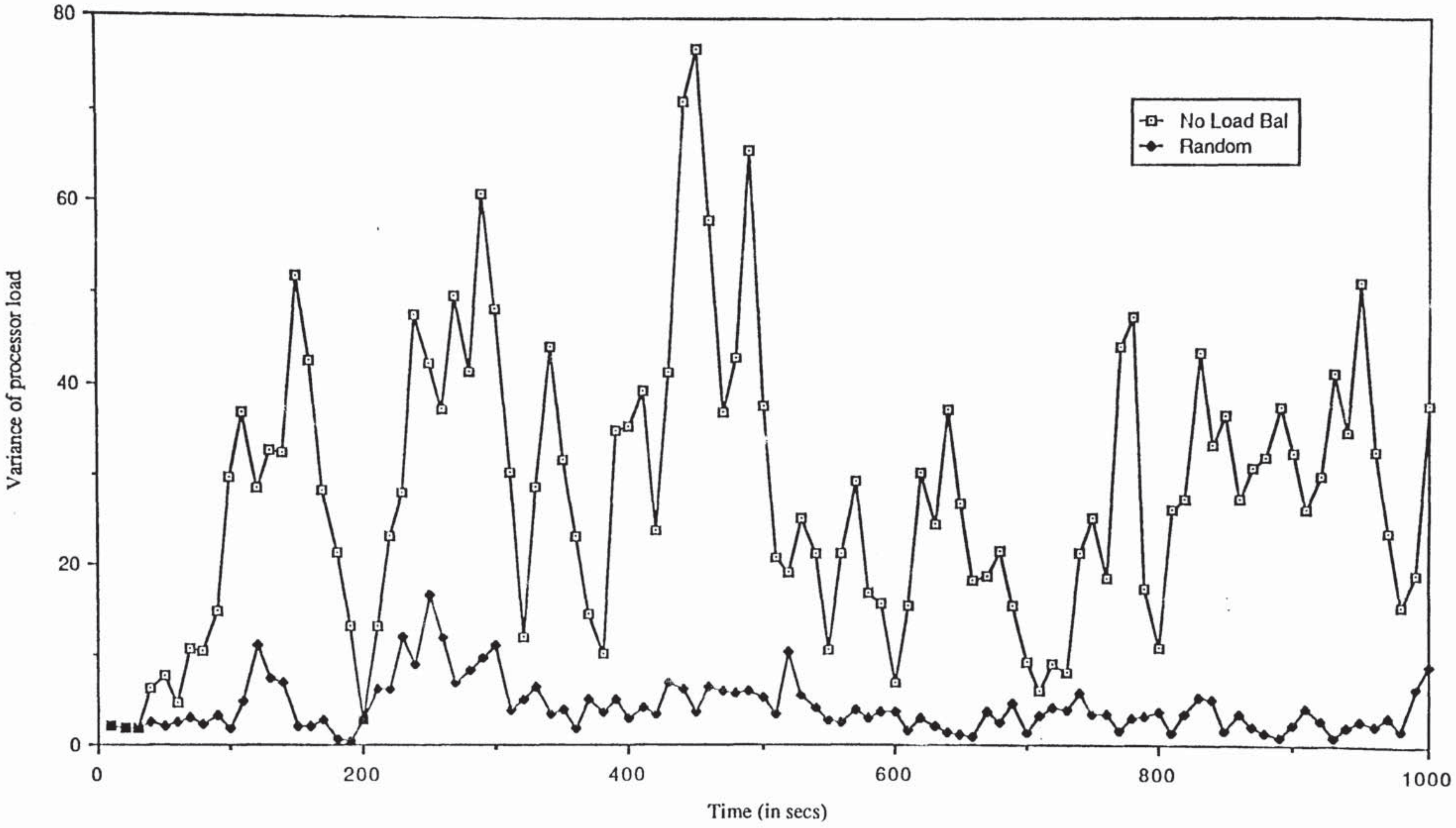

Fig. B.3.5 **Load Variance** - No Load Balancing vs Random
using the Cooperating Process Group Model (Load Value = 0.7)

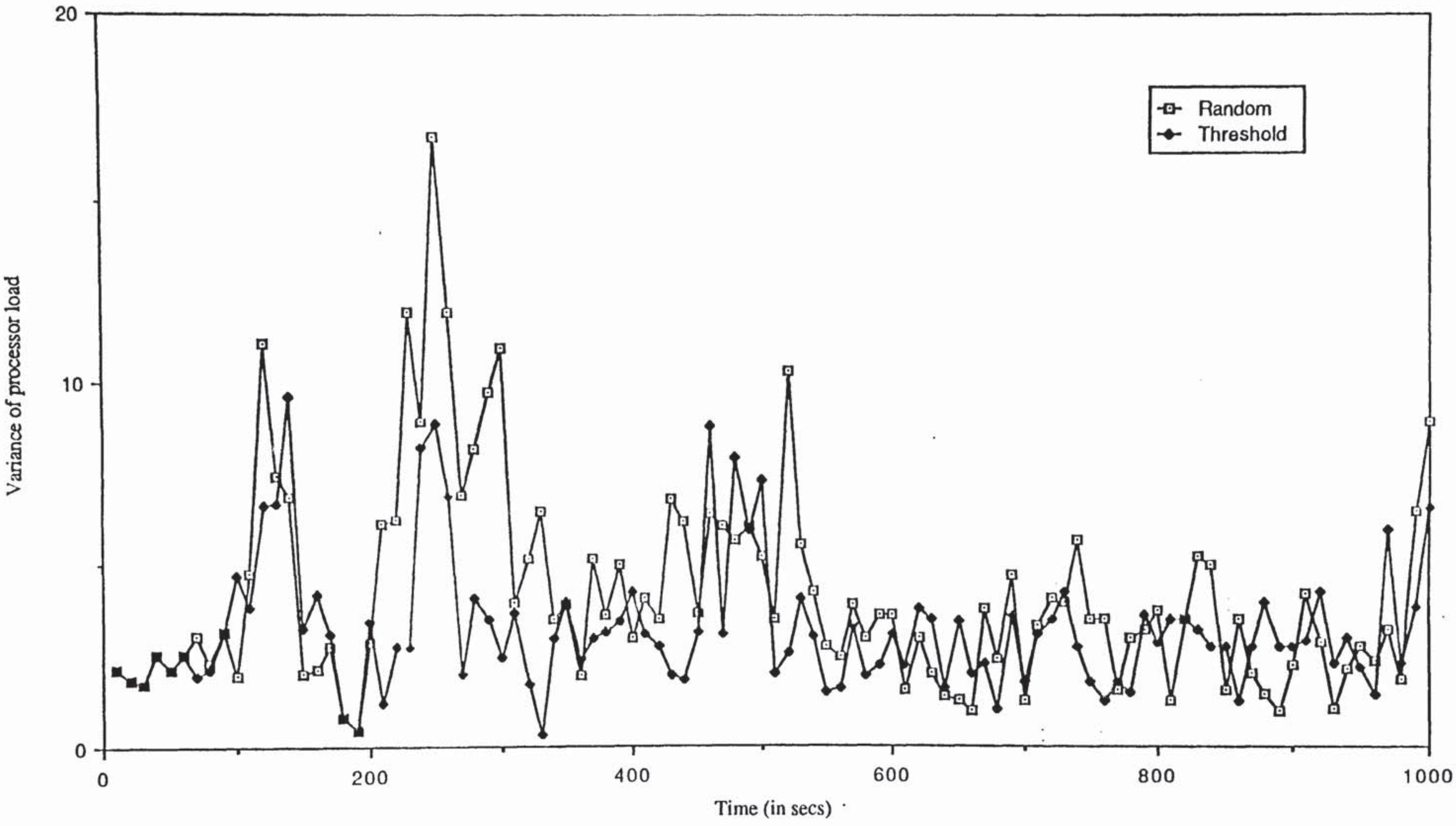

Fig. B.3.6 **Load Variance** - Random vs Threshold using the Cooperating Process Group Model (Load Value = 0.7)

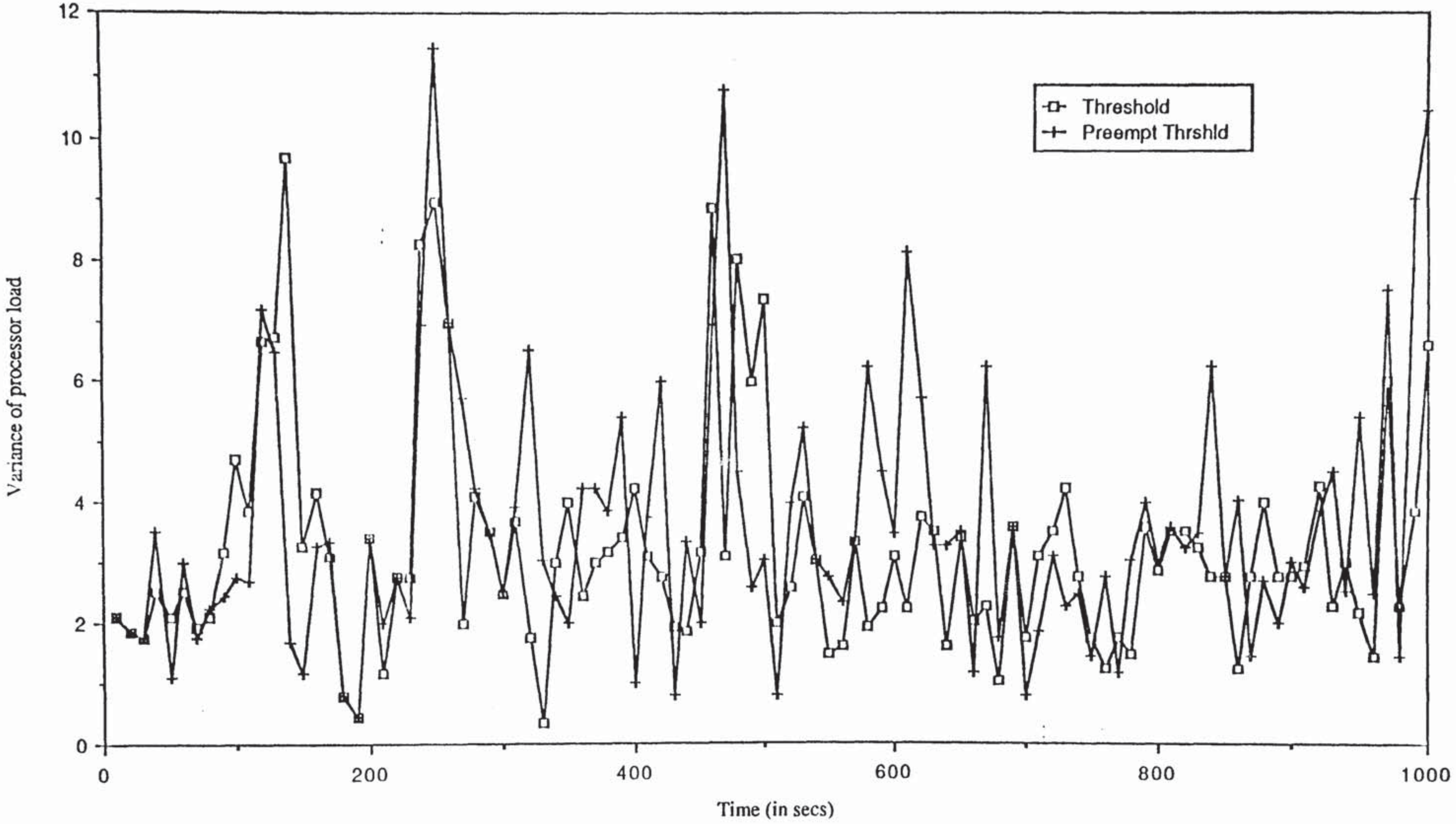

Fig. B.3.7 **Load Variance** - Threshold vs Preemptive Threshold using the Cooperating Process Group Model (Load Value = 0.7)

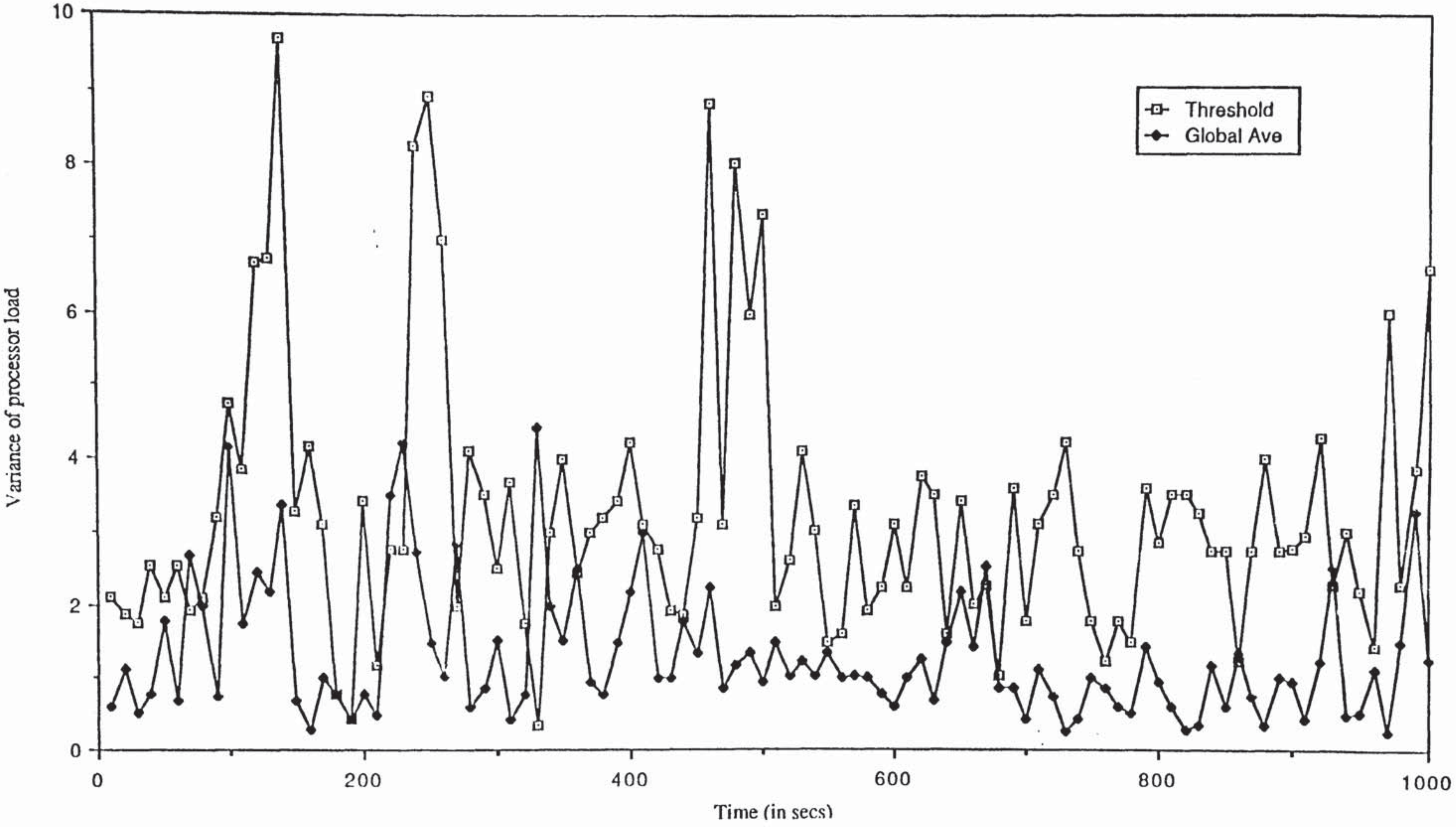

Fig. B.3.8 **Load Variance** - Threshold vs Global Average using the Cooperating Process Group Model (Load Value = 0.7)

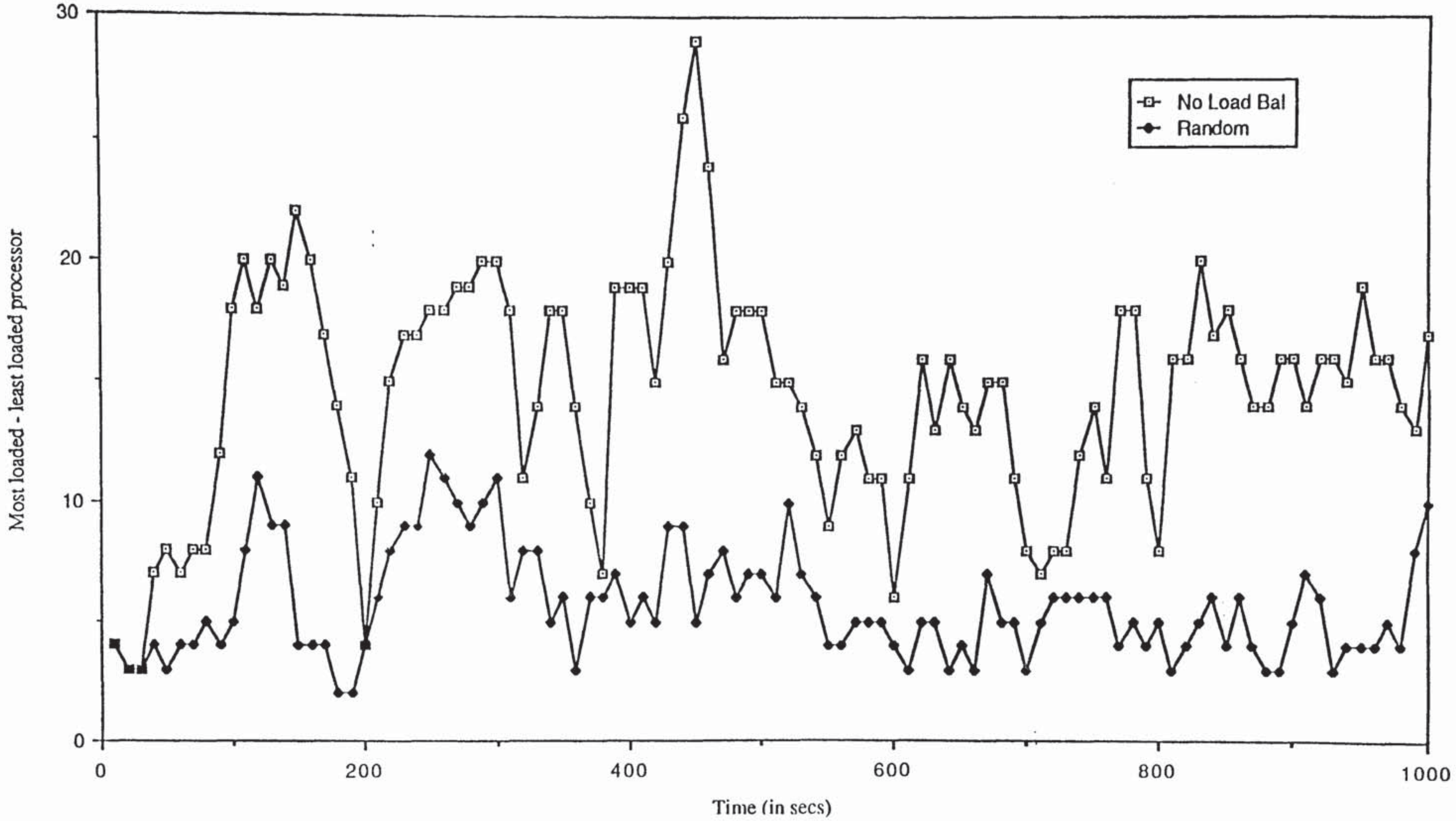

Fig. B.3.9 **Load Difference** - No Load Balancing vs Random using the Cooperating Process Group Model (Load Value = 0.7)

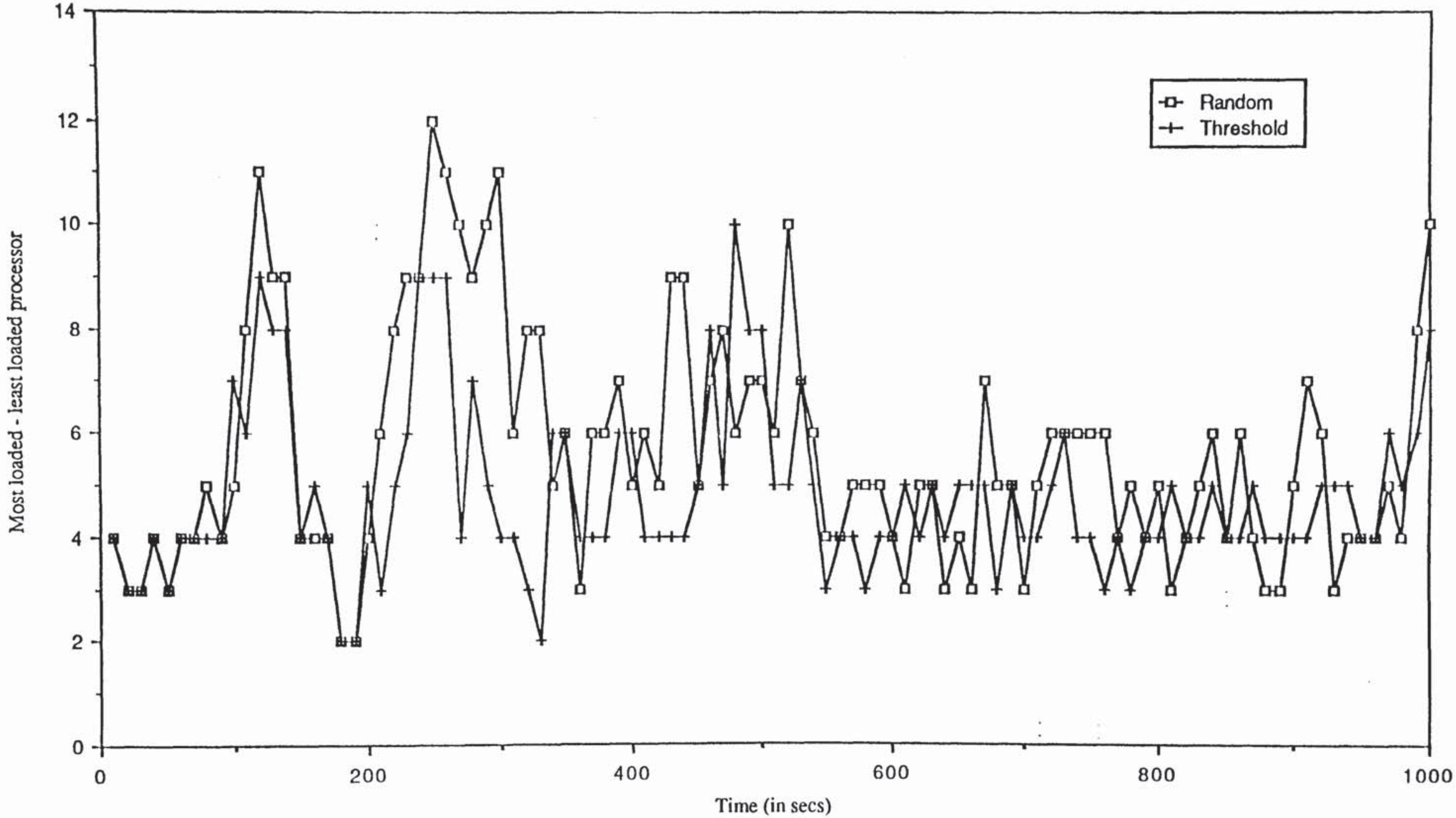

Fig. B.3.10 **Load Difference** - Random vs Threshold
using the Cooperating Process Group Model (Load Value = 0.7)

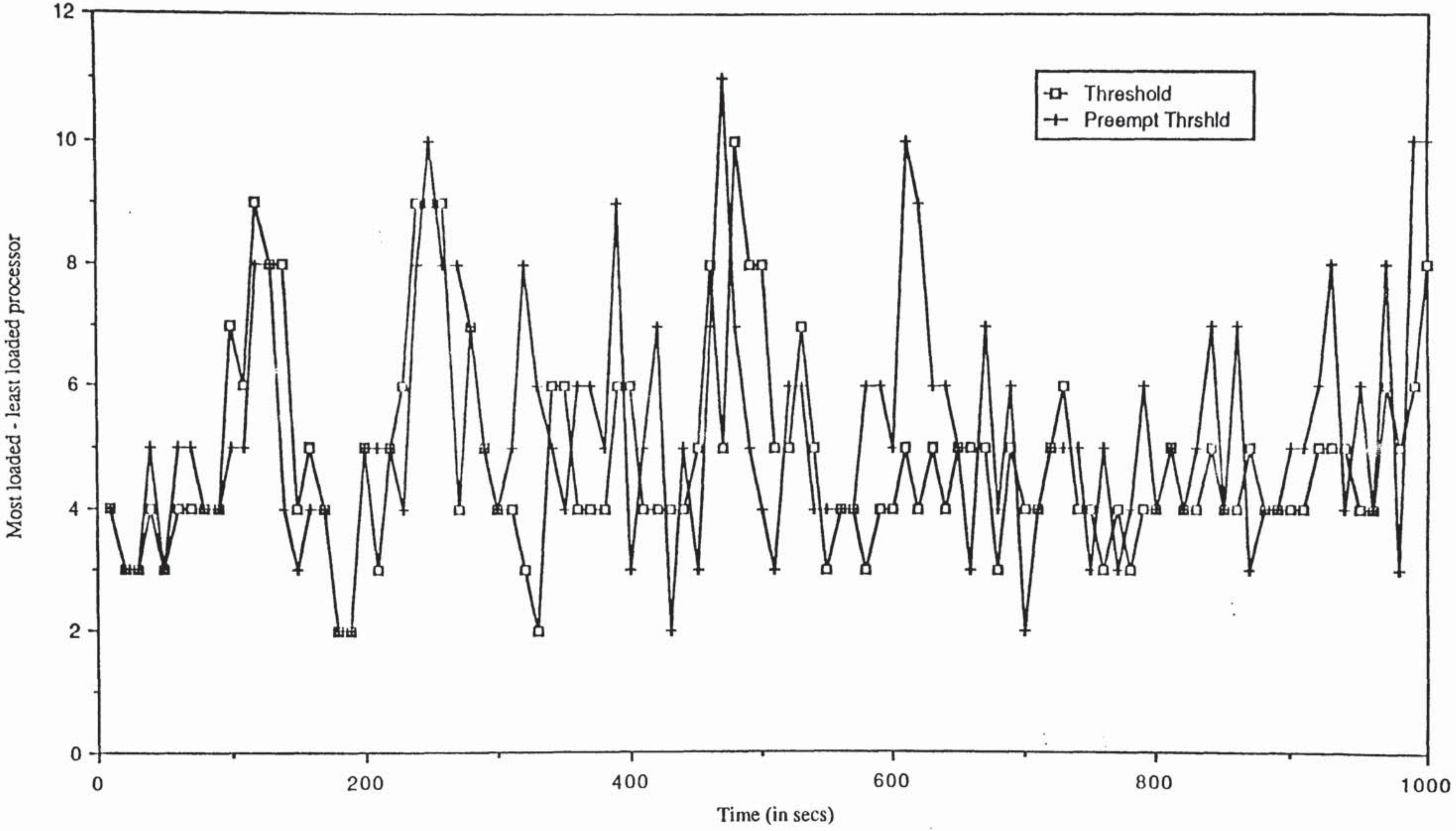

Fig. B.3.11 **Load Difference** - Threshold vs Preemptive Threshold using the Cooperating Process Group Model (Load Value = 0.7)

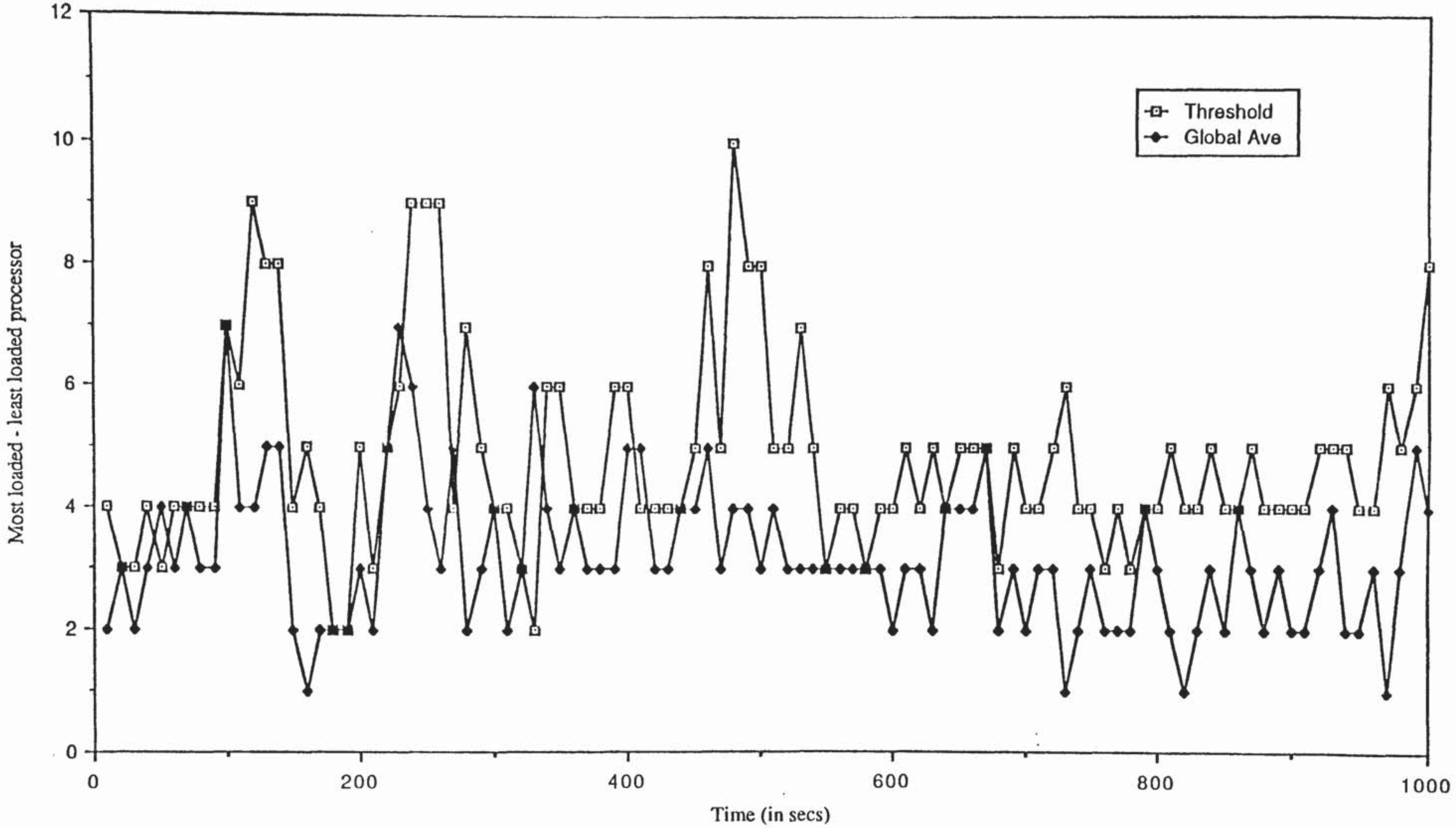

Fig. B.3.12 **Load Difference** - Threshold vs Global Average using the Cooperating Process Group Model (Load Value = 0.7)

APPENDIX C

Program for the Simulated System

C.1 Technical Implementation Notes

The following are a number of details regarding implementation features of the simulation system:

1. All synchronisation between processes (both user processes and the simulated processors) is peformed using UNIX signals. Since the manner in which the system call `sleep()` is implemented, this code includes our own version using `pause() and alarm()` to avoid deadlock under certain sequences of interrupt.

2. Mutual exclusion is assured using the file "lock.f", and assigning aliasses to it via the `link();` this is used especially for access to pipes shared by user processes.

3. The system can be run in the background using `nohup`, to allow the experimenter to log out; his terminal is explicitly opened by the program for writing, and so any error messages will still be displayed on the screen.

4. Configuration File Format :

   Each processor has a configuration file "config", with the extension of its processor identifier. This file must contain the following:

   < number of physical links to/from the processor>

   < identifying integers of neighbour processors on each link>

   < link/distance pairs of integers for each other processor >

5. Message Formats :

   Each external message between processors has a header of type EM_HDR whose fields contain:

   - the identity of the sending processor
   - the name of the process causing message transmission
   - the identity of the destination processor.

The following messages are used for interkernel communications :

EM_CPORT (announcing port creation) :

fields :

- port name
- port's permitted message type

EM_DPORT (announcing port destruction) :

fields :

- port name

EM_LP_REQ (request for remote port link) :

fields :

- "link to" port name
- "link from" port name

EM_LP_ACK (acknowledgement of successful remote port link) :

fields :

- "link to" port name
- "link from" port name

EM_LP_NACK (announcing failure of remote port link) :

fields :

- reason for failure

EM_UP_REQ (request for remote port unlink) :

fields :

- "link to" port name
- "link from" port name

EM_UP_ACK (acknowledgement of successful port unlink) :

fields :

- indication of successful unlink

EM_UP_NACK (announcing failure of remote port unlink) :

fields :

- reason for failure

MSG (user interprocess message) :

fields :

- message header
- message text

where message header has fields :

- source port of message
- destination port of message
- message length
- message type
- blocking/non-blocking send flag

6. Directory Structure :

The system is organised in the following directories :

include	- files containing parameters of the system
kernel	- files containing all kernel routines
physnetwk	- files containing the network simulation
utils	- files containing general utility routines
usrf	- files containing user process interface routines

C.2 Program Listing

DIRECTORY NAME : INCLUDE

```
/* FILE: project/src/include/keywds.h */

/* This file contains useful defines for making the system more readable */

# define  READ                     0
# define  WRITE                    1
# define  FIFO               0010000
# define  OWNACC             0000700
# define  EMPTY                 (-1)
# define  FAIL                  (-1)
# define  MANAGER               (-1)
# define  PENDING                  1
# define  BLOCKING                 1
# define  NON_BLOCKING             2
# define  BLOCKED                  1
# define  UNBLOCKED                0
# define  KCALL_MASK          0x8000
# define  EMSG_MASK           0x4000
# define  KCFAIL_MASK         0x8000
# define  KCSUCC_MASK         0x4000
# define  SUCCESS                  0
# define  FORWARDED                1
# define  SIGKCR             SIGUSR1
# define  SIGKCRACK          SIGUSR2
# define  SIGMSG             SIGUSR2
# define  SIGMSGACK           SIGFPE
# define  SIGMIG              SIGFPE
# define  SIGMIGACK          SIGUSR1
# define  SIGCONT            SIGTERM
# define  SIGSCHED            SIGINT
# define  SIGSETUP           SIGTERM
# define  CHILD                    0
# define  PARENT             default
# define  REMLINK                  2
# define  REMQUERY                 3
# define  REMUNLINK                4
# define  RW                       2
# define  OWN                 static
# define  LOCAL                 auto
# define  REG               register
# define  EXTERN              extern
# define  PARAMS                /**/
# define  FOREVER            for(;;)
# define  TRUE                     1
# define  FALSE                    0

/* Emsg types */
# define CPROC_MSG         0
# define CPORT_MSG         1
# define DPORT_MSG         2
# define LPORT_REQ         3
# define LPORT_ACK         4
# define LPORT_NACK        5
# define UPORT_REQ         6
# define QPORT_REQ         7
# define QPORT_ACK         8
# define PINFO_MSG          9
# define USR_MSG           10
# define UPORT_ACK         11
# define UPORT_NACK        12
# define EXIT_MSG          13
# define PROBE_MSG         14
# define REPLY_PROBE_MSG   15
# define NEGOCIATE_MSG     16
# define CH_AVE_MSG        17
# define LVEC_MSG          18
# define PORT_LOC_MSG      19


/* Kcall types */
# define CPROC_KC     0
# define CPORT_KC     1
# define EPROC_KC     2
# define DPORT_KC     3
# define LPORT_KC     4
# define UPORT_KC     5
# define QPORT_KC     6
# define BRMSG_KC     7
# define NBRMSG_KC    8
# define BSMSG_KC     9
# define NBSMSG_KC   10
# define DOPROC_KC   11

/* Time types */
# define USERtime     1
# define OStime       2


/* Shared memory size */
# define  SHMSIZ    sizeof(PROC_ENTRY)*MAXPROCS +\
                    sizeof(ROUTE)*MAXMCS +\
                    sizeof(double) +\
                    sizeof(int) +\
                    sizeof(QTUM_ENTRY)*NQUANTA +\
                    sizeof(int)

# define  CTRL_SEG_SIZ  sizeof(double) + MAXMCS * sizeof(unsigned short)
```

```
/* FILE: project/src/include/params.h

/* This file contains changeable parameters for the system */

# define  LOCKFNAME  "lock.f"
# define  OPLOCKSIZ   8
# define  KCRLOCKSIZ  8
# define  OPNAMSIZ    14
# define  KCRNAMSIZ   14
# define  MAXMCS      16
# define  MAXLINKS    8
# define  MAXPNAME    12
# define  MAXLFROM    4
# define  MAXLTO      4
# define  MAXMSGS     4
# define  MAXOWNPRT   2
# define  MAXPROCS   50
# define  MAXPORTS   100
# define  MAXFNAME   16
# define  N_SYS_PROCS 1
# define  MSG_SIZ     6
# define  AVE_PROC_GROUP 3
# define  AVE_EXEC_TIME 3.0

/* Defines for timing */

# define  CONTEXT_SWITCH  200
# define  RX_BYTE_TIME    1
# define  TX_BYTE_TIME    1
# define  PROTOCOL_TIME   1000
# define  FAIL_TIME       30
# define  AVE_INST        1
# define  NQUANTA         10
# define  QUANTUM         20000
# define  DUMP_INTERVAL   10000000.0
# define  SYNC_TIME       50000.0

/* Defines for load balancing algorithms */
# define  PROBE_LIMIT     3
# define  THRESHOLD       4
# define  CHANGE_AMOUNT   0.5
# define  ACCEPTABLE_RANGE 1.0
# define  TOO_HIGH        1
# define  ACCEPT          2
# define  TOO_LOW         3
# define  SENDER          1
# define  RECEIVER        2
# define  GOING_UP        4.0
# define  GOING_DOWN      3.0
# define  NAWAITS         10
# define  OVERLOADED      my_average_load > global_average_load\
                           + ACCEPTABLE_RANGE
# define  UNDERLOADED     my_average_load < global_average_load\
                           - ACCEPTABLE_RANGE
# define  TIMEOUT_INTERVAL 200000
# define  VECTOR_SIZE     6
# define  NLOADS           VECTOR_SIZE/2
# define  CONSIDER_INTERVAL  1000000
# define  SEND_INTERVAL      250000
# define  MIN_EXEC_TIME      200000
```

```
/* FILE: project/src/include/sys/errcodes.h */

/* This file contains the system error codes */

# define  PIPE_CREATION  0
# define  FORK_MCS       1
# define  C_HDR_ERR      2
# define  TM_PORTS       3
# define  FORK_FAIL      4
# define  TM_PROCS       5
# define  PIPE_READ      6
# define  PIPE_WRITE     7
# define  LOCK_OPEN      8
# define  PIPE_OPEN      9


/* User error codes */
# define  TM_LTO       1
# define  TM_LFROM     2
# define  UN_LFPORT    8
# define  UN_LTPORT    9
# define  UN_RPORT     10
# define  TM_NBRCVS    11
# define  UN_SPORT     12
# define  UN_DPORT     13
# define  UN_MTYPE     14
# define  DP_LINKED    15
# define  DP_MSGS      16
# define  EX_PORTS     17
```

```
/* FILE: project/src/include/sys/globvars.h */

/* This file contains EXTERNal declarations of all global variables */

EXTERN  PORT_ENTRY  port_table [];
EXTERN  PORT_ENTRY  *nxt_port;
EXTERN  int         nports;
EXTERN  PROC_ENTRY  process_table[];
EXTERN  PROC_ENTRY  *nxt_proc;
EXTERN  int         nprocs;
EXTERN  int         last_proc_creat;
EXTERN  int         own_pipe;
EXTERN  int         kcret_pipe;
EXTERN  ROUTE       route_table[];
EXTERN  int         mcpids [];
EXTERN  int         this_mc;
EXTERN  PLINK       phys_link [];
EXTERN  char        ownp_locks [][OPLOCKSIZ];
EXTERN  char        kcret_locks [][KCRLOCKSIZ];
EXTERN  FILE        *trace;
EXTERN  FILE        *config;
EXTERN  int         nmcs;
EXTERN  int         boot;
EXTERN  int         load_bal_active;
EXTERN  int         synth_workload;
EXTERN  int         got_ackkcrsig;
EXTERN  int         got_msgsigack;
EXTERN  int         got_migack;
EXTERN  int         proc_setup;
EXTERN  int         n_local_procs;
EXTERN  int         n_active_local_procs;
EXTERN  double      sys_real_time;
EXTERN  int         OSoverhead;
EXTERN  QTUM_ENTRY  quanta[];
EXTERN  int         current;
EXTERN  int         shmid;
EXTERN  unsigned short xsubi[];
EXTERN  double      *stop_time;
EXTERN  unsigned short  *reached;
EXTERN  double      cumul_exist_time;
EXTERN  int         n_deaths;
EXTERN  TIME_OUT    too_low;
EXTERN  TIME_OUT    too_high;
EXTERN  AWAIT_TIMEOUT awaiting_process;
EXTERN  float       global_average_load;
EXTERN  int         n_virtual_procs;
EXTERN  PROC_ENTRY  *scheduled_proc;
EXTERN  float       my_average_load;
EXTERN  PROC_LOAD   load_vector[];
EXTERN  int         process_groups[];
EXTERN  unsigned short rnd_job[];
EXTERN  int         nxt_job;
EXTERN  int         n_migrates;
EXTERN  int         n_TXs;
EXTERN  int         n_nbours;
EXTERN  int         neighbours[];
EXTERN  int         policy;
```

```
/* FILE: project/src/include/sys/macros.h */

/* This file contains macro defs to ease writing of system routines */

# include <memory.h>
# define procncpy(a,b)       memcpy((char *)&(a), (char*)&(b), sizeof(PROCN))
# define kcfail(a)         {KCR_HDR kcr_hdr = (a) | KCFAIL_MASK;\
                            kcreturn (caller->upid, (char *)&kcr_hdr,\
                            sizeof(KCR_HDR));\
                            time_update (FAIL_TIME * AVE_INST + CONTEXT_SWITCH *
                            AVE_INST,USERtime,caller);}
# define getp_blk(a)        RX ((char *)&kc_hdr, sizeof(KC_HDR));\
                            RX ((char *)&p_blk, sizeof(a))
# define kcsucc(a,b)          kcreturn (caller->upid, (char *)&(a), sizeof(a));\
                              time_update(CONTEXT_SWITCH * AVE_INST + (b) *
AVE_INST,                                  USERtime, caller)

# define contxt_swtch       time_update (CONTEXT_SWITCH * AVE_INST, USERtime,
                                         caller)
```

```
/* FILE: project/src/include/sys/types.h */

/* This file contains typedefs for all common system types */


/***** MESSAGES *****/
typedef  struct {
                  char  dst_port[MAXPNAME];
                  char  src_port[MAXPNAME];
                  int   msg_length;
                  int   snd_type;
                  int   msg_type;

                } MSG_HDR;

typedef  struct msg {
                      MSG_HDR  msg_hdr;
                      char  msg_txt[MSG_SIZ];
                      struct msg *next;

                    } MSG;



/***** POINTER TO FUNCTION *****/

typedef  int (*PFI)();


/***** PHYSICAL LINK *****/

typedef struct {
                 int  link;
                 int  nbour;

               } PLINK;


/***** ROUTE TABLE ENTRY *****/

typedef struct {
                 int  lnk;
                 int  distance;

               } ROUTE;



/***** PROCESS NAME *****/

typedef struct {
                 int  gmc;
                 int  gnum;
                } PGRP;

typedef struct {
                 PGRP pgroup;
                 char pname [MAXPNAME];

                } PROCN;




/***** PORT TABLE ENTRIES *****/

typedef struct {
                 int  lnkf_length;
                 int  lnkt_length;
                 int  nxt_lf;
                 int  nxt_lt;
                 int  inmq_length;
                 int  nb_pending;
                 int  b_pending;

                } PORT_PROF;

typedef struct port_entry {
                             char  port_name [MAXPNAME];
                             int   residency;
                             PROCN owner_proc;
                             struct port_entry *links_from [MAXLFROM];
                             struct link_to {
                                             struct port_entry *port;
                                             int                nmsgs;
                                             int                tot_msglength;
                                            }links_to [MAXLTO];
                             MSG   *msg_q_head;
                             MSG   *msg_q_tail;
                             MSG   *nb_msg_loc;
                             int   msg_type;
                             PFI   destruct;
                             PFI   rcvfunc;
                             PORT_PROF  profile;

                           } PORT_ENTRY;


/***** PROCESS TIMES *****/
typedef struct {
                 int  exec_time;
                 int  exec_here_time;
                 int  residency_time;
                 int  exist_time;

                } TIMES;



/***** PROCESS TABLE ENTRIES *****/
```

```
typedef struct {
                PROCN  proc_name;
                int    n_probes;
                int    mcs_probed[PROBE_LIMIT];
                unsigned short  schedulable;
                int    upid;
                int    residency;
                int    orig_mc;
                int    migration_siz;
                int    siz;
                int    tot_msg_siz;
                TIMES  times;
                int    blocked;
                int    preferred_mc;
                int    level_of_preference [MAXMCS];
                PORT_ENTRY  *owned_ports [MAXOWNPRT];
                int    ownp_length;
                int    nxt_ownp;
                PROCN  parent;

               } PROC_ENTRY;



/***** HEADERS *****/

typedef int  COMMS_HDR;

typedef struct {
                int  sending_mc;
                PROCN  caller;
                int  dst_mc;

               } EM_HDR;

typedef PROCN  KC_HDR;

typedef int  KCR_HDR;



/***** KCRETURNS *****/

typedef int QRET;


/***** TIME QUANTA *****/

typedef struct {
                int  actual_load;
                int  virtual_load;
                int  OSportion;
                int  used;

               } QTUM_ENTRY;
```

```
/***** TIME_OUTS *****/

typedef struct {
                 unsigned short  set;
                 double          timer;

               } TIME_OUT;

typedef struct {
                 unsigned short  set;
                 double          timer[NAWAITS];

               } AWAIT_TIMEOUT;


/***** LOAD VECTOR *****/

typedef struct {
                 int  processor;
                 float load;

               } PROC_LOAD;
```

DIRECTORY NAME : KERNEL

```
/* FILE: project/src/kernel/kernel.c */

/* This file contains the main kernel routine */

/* Includes for this file */

# include <sys/types.h>
# include <sys/times.h>
# include <stdio.h>
# include <signal.h>
# include <math.h>
# include <fcntl.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"
# include "/usr/acct/ian/project/src/include/sys/globvars.h"


/* Routines EXTERNal to this file */

EXTERN  int  cproc();
EXTERN  int  cport();
EXTERN  int  exit_proc();
EXTERN  int  dport();
EXTERN  int  lport();
EXTERN  int  uport();
EXTERN  int  qport();
EXTERN  int  b_rmsg();
EXTERN  int  nb_rmsg();
EXTERN  int  b_smsg();
EXTERN  int  nb_smsg();
EXTERN  int  do_processing();

EXTERN  int  cproc_msg();
EXTERN  int  cport_msg();
EXTERN  int  dport_msg();
EXTERN  int  lp_req();
EXTERN  int  lp_ack();
EXTERN  int  lp_nack();
EXTERN  int  up_req();
EXTERN  int  qp_req();
EXTERN  int  qp_ack();
EXTERN  int  pinfo_msg();
EXTERN  int  usr_msg();
EXTERN  int  up_ack();
EXTERN  int  up_nack();
EXTERN  int  exit_msg();
EXTERN  int  probe_msg();
EXTERN  int  probe_reply_msg();
EXTERN  int  negociate_msg();
EXTERN  int  change_ave_msg();
EXTERN  int  receive_load_vector();
EXTERN  int  port_loc_msg();

EXTERN  int  dump_procs();
EXTERN  float calc_ave_load();


/* Kernel call jump vector */

static  PFI  kcvec [] = {
                          cproc,
                          cport,
                          exit_proc,
                          dport,
                          lport,
                          uport,
                          qport,
                          b_rmsg,
                          nb_rmsg,
                          b_smsg,
                          nb_smsg,
                          do_processing

                          };

/* Trace msgs for kcalls */

static  char  *kc_msg [] = {
                             "create process",
                             "create port",
                             "dest process",
                             "dest port",
                             "link port",
                             "unlink port",
                             "query port",
                             "b_rcv msg",
                             "nb_rcv msg",
                             "b_snd msg",
                             "nb_snd msg",
                             "do some processing"

                            };


/* External msg jump vector */

static  PFI  emvec [] = {
                          cproc_msg,
                          cport_msg,
                          dport_msg,
                          lp_req,
                          lp_ack,
                          lp_nack,
                          up_req,
                          qp_req,
                          qp_ack,
                          pinfo_msg,
                          usr_msg,
                          up_ack,
                          up_nack,
                          exit_msg,
                          probe_msg,
                                probe_reply_msg,
                                negociate_msg,
                                change_ave_msg,
                                receive_load_vector,
                                port_loc_msg

                               };

 /* Trace msgs for emsgs */

 static  char *em_msg [] = {
                                 "announce cproc",
                                 "announce cport",
                                 "announce dport",
                                 "request lport",
                                 "ack lport",
                                 "nack lport",
                                 "request uport",
                                 "request qport",
                                 "ack qport",
                                 "get pinfo",
                                 "get usrmsg",
                                 "ack uport",
                                 "nack uport",
                                 "remove zombie",
                                 "get probe",
                                 "get probe reply",
                                 "negociate mig",
                                 "change global average",
                                 "receive load vector",
                                 "update port location"

                                };


EXTERN  double  erand48();



/***** KERNEL *****/

/* This routine is the kernel executed by each mc */

kernel()
{

LOCAL  COMMS_HDR  comms_hdr;
LOCAL  double     last_performance_dump_time = 0.0;
LOCAL  double     nxt_arrival_time;
LOCAL  char       jobs_name[MAXFNAME];
LOCAL  FILE       *jobs;
LOCAL  int        this_read;
LOCAL  int        mig_mc;
LOCAL  int        OK_to_schedule = 1;
LOCAL  double     nxt_send_time = 0;
LOCAL  double     nxt_consider_time = 0;
LOCAL  int        i;
EXTERN int        msgsigack_handler ();
EXTERN int        ackkcrsig_handler ();
```

```
  EXTERN int          migsigack_handler ();
  EXTERN int          sigsetup_handler();
  EXTERN int          alarm_handler();
  EXTERN FILE         *fopen();
  EXTERN float        calc_ave_load();


          /* Set up initial environment */
          init();

  # ifdef DEBUG
  printf("init exit OK\n");
  # endif

          /* Set up sigmsgack handler */
          signal (SIGMSGACK, msgsigack_handler);

          /* Set up sigkcrack handler */
          signal (SIGKCRACK, ackkcrsig_handler);

          /* Set up sigmigack handler */
          signal (SIGMIGACK, migsigack_handler);

          /* Set up sigsetup handler */
          signal (SIGSETUP, sigsetup_handler);

          /* Set up alarm handler */
          signal (SIGALRM, alarm_handler);

          /* Set up RUBOUT to dump process table */
          signal (SIGINT, dump_procs);

          /* Make RNG unique */
          for (i=0; i<3; i++)
              xsubi[i] = this_mc + i;


          /* Open jobs file */
          sprintf (jobs_name, "jobs%d", this_mc);
          jobs = fopen(jobs_name, "r");
          fcntl ((int)(fileno(jobs)), F_SETFD, 1);

          /* Read in first job arrival time */
          fscanf (jobs, "%d", &nxt_job);
          fscanf (jobs, "%lf", &nxt_arrival_time);

          printf ("%d - Entering loop\n", this_mc);

          /* Loop forever reading in comms. */
          FOREVER
          {
             /* Check for work to do */
             if ((this_read = read (own_pipe, (char *)&comms_hdr,
sizeof(COMMS_HDR))) == 0)
             {
                  if (n_active_local_procs <=N_SYS_PROCS) time_update (10000,
OStime, (PROC_ENTRY *)EMPTY);
             }
             else
```

```
           /* Execute either kcall or emsg routine */
           if (comms_hdr & KCALL_MASK)
                OK_to_schedule = kcall (comms_hdr & ~KCALL_MASK);
           else
           if (comms_hdr & EMSG_MASK)
               emsg (comms_hdr & ~EMSG_MASK);
           else
               ERROR (C_HDR_ERR);

           /* Check to see if reached synchronisation point */
           if (sys_real_time >= *stop_time)
           {
               got_ackkcrsig = 0;
               (*reached)++;
               while (!got_ackkcrsig)
               {
                     alarm(3);
                     pause();
                     alarm(0);
               }
           }

           /* Check to see if need to dump performance info */
           if (sys_real_time >= last_performance_dump_time + DUMP_INTERVAL)
           {
               performance_dump();
               last_performance_dump_time = sys_real_time;
           }

           /* Check to see if synth workload gen necessary */
           if (!feof(jobs))
           if (sys_real_time >= nxt_arrival_time)
           {
               synth_workload = 1;
               cproc();
               synth_workload = 0;
               fscanf (jobs, "%d", &nxt_job);
               fscanf (jobs, "%lf", &nxt_arrival_time);
            }

            /* Reschedule a process */
            if (n_active_local_procs > N_SYS_PROCS && OK_to_schedule)
            {
            reschedule();
            OK_to_schedule = 0;
            }

         }

} /* End of KERNEL */



/***** KCALL *****/
```

```
/* This routine deals with a kernel call */

kcall (kctype)

PARAMS  int  kctype;

{

LOCAL  int  result;
LOCAL  long clock = time(0);

        /* Put trace msg to trace file */
#       ifdef DEBUG
        fprintf (trace, "%s - kcall made to %s\n", ctime(&clock), kc_msg[kctype]);
#       endif

        /* Make kernel call */
        result = (*kcvec[kctype])();

#       ifdef DEBUG
        /* Record result of kernel call */
        record_result (result);
#       endif
        return 1;

} /* End of KCALL */




/***** EMSG *****/

/* This routine deals with receipt of an external msg */

emsg (emtype)

PARAMS  int  emtype;

{

LOCAL  int  result;
LOCAL  long clock = time (0);

        /* Put trace msg to trace file */
#ifdef  DEBUG
        fprintf (trace, "%s\n", em_msg[emtype]);
#endif

        /* Make emsg call */
        result = (*emvec[emtype])();

#       ifdef DEBUG
        /* Record result of emsg call */
        record_result (result);
#       endif
} /* End of EMSG */




/***** RECORD_RESULT *****/

/* This routine records the result of a kcall or emsg */

record_result (result)

PARAMS  int  result;

{

        switch (result)
        {
           case  FAIL:           fprintf (trace, " - failed\n");
                                 break;

           case  SUCCESS:        fprintf (trace, " - success\n");
                                 break;

           case  FORWARDED:      fprintf (trace, " - forwarded\n");
                                 break;

           case  REMLINK:        fprintf (trace, " - remote link\n");
                                 break;

           case  REMQUERY:       fprintf (trace, " - remote query\n");
                                 break;

           case  REMUNLINK:      fprintf (trace, " - remote unlink\n");
                                 break;

           default:              fprintf (trace, " - unknown result\n");
                                 break;

        }

} /* End of RECORD_RESULT */




/***** INIT *****/

/* Initial process */

init()
```

```
{


        /* Read in mcpids of other processors */
         read (own_pipe, (char *)mcpids, sizeof(int)*MAXMCS);
        printf ("%d - read pids\n", this_mc);

        /* Set up reached pointer */
        reached += this_mc;

        /* Start shell process */
        boot = 1;
        cproc();
        boot=0;
        printf ("%d - shell forked\n", this_mc);

        /* Start load balancing process */
/*      load_bal_active = 1;
        cproc();
        load_bal_active = 0; */

} /* End of INIT */
```

```
/***** RESCHEDULE *****/

reschedule()

{

OWN  int  active_proc = N_SYS_PROCS - 1;
LOCAL int kval;


           /* Establish next process to schedule */
           do {
                 if (++active_proc >= MAXPROCS)
                     active_proc = N_SYS_PROCS;

              } while (process_table[active_proc].upid == EMPTY ||
process_table[active_proc].schedulable == FALSE ||
process_table[active_proc].blocked == BLOCKED);

            /* Schedule process */
            if (process_table[active_proc].upid != EMPTY)
            {
               if ((kval = kill (process_table[active_proc].upid, SIGSCHED))==-1)

                    printf ("Kill failed\n");


             }
            /* Remember last process scheduled */
            scheduled_proc = process_table + active_proc;

} /* End of RESCHEDULE */
```

```
/* FILE: project/src/kernel/cport.c */

/* This file contains routines to CREATE A PORT */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_CPORT          70 * AVE_INST
# define  TIME_MSG_CPORT      40 * AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY *find_proc();

/* Typedefs used in these routines */

typedef  struct {
                   char port_name[MAXPNAME];
                   int  msg_type;
                   PFI  destruct;
                   PFI  rcvfunc;

                 } KC_CPORT;

typedef  struct {
                   int  msg_type;
                   char port_name[MAXPNAME];

                 } EM_CPORT;



/***** CPORT *****/

/* This routine deals with a kcall made to create a port */

cport()

{

LOCAL  KC_HDR        kc_hdr;
LOCAL  KC_CPORT      p_blk;
LOCAL  PROC_ENTRY    *caller;
LOCAL  int           save_nxt_ownp;
LOCAL  EM_CPORT      cport_msg;
```

```
/* Get parameter block */
getp_blk(KC_CPORT);

/* Establish calling process */
caller = find_proc (kc_hdr);

/* Update time due to context switch */
contxt_swtch;

/* Check for port table overflow */
if (nports++ >= MAXPORTS)
{
    nports --;
    kcfail(TM_PORTS);
    return FAIL;
}

/* Check for overflow in owned ports table */
if (caller -> ownp_length++ >= MAXOWNPRT)
{
    kcfail(TM_PORTS);
    return FAIL;
}

/* Make entry in owned ports list */
caller -> owned_ports[caller -> nxt_ownp] = nxt_port;

/***  ENTER PORT INFO  ***/

   /* Port name */
   strcpy (nxt_port -> port_name, p_blk.port_name);

   /* Residency */
   nxt_port -> residency = this_mc;

   /* Owner process */
   procncpy (nxt_port->owner_proc, caller->proc_name);

   /* Incoming msg type */
   nxt_port -> msg_type = p_blk.msg_type;

   /* Destruction routine */
   nxt_port -> destruct = p_blk.destruct;

   /* Routine for non-blocking rcv */
   nxt_port -> rcvfunc = p_blk.rcvfunc;

/* Set up ext. msg to announce port creation and broadcast it */
cport_msg.msg_type = nxt_port -> msg_type;
strcpy (cport_msg.port_name, nxt_port -> port_name);
broadcast (CPORT_MSG, caller, (char *)&cport_msg, sizeof(EM_CPORT));

/* Establish next available owned ports entry */
save_nxt_ownp = caller -> nxt_ownp;
while (caller -> owned_ports[caller->nxt_ownp] != (PORT_ENTRY *)EMPTY)
       caller -> nxt_ownp++;

/* Establish next available port table entry */
while (nxt_port -> residency != EMPTY)
```

```
                nxt_port++;

        /* Increase migration size of caller */
        caller -> migration_siz += sizeof(PORT_ENTRY)+sizeof(int);

        /* Return result to caller */
        { KCR_HDR kcr_hdr = save_nxt_ownp | KCSUCC_MASK;
          kcsucc(kcr_hdr, TIME_CPORT);
        }

# ifdef DEBUG
dump_procs();
dump_ports();
# endif

        return SUCCESS;

} /* End of CPORT */





/***** CPORT_MSG *****/

/* This routine deals with an emsg announcing the creation of a port */

cport_msg()

{

LOCAL  EM_HDR    em_hdr;
LOCAL  EM_CPORT  msg;

        /* Check if needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_CPORT),
CPORT_MSG) == FORWARDED)
            return FORWARDED;

        /* Check not too many global ports */
        if (nports++ >= MAXPORTS)
            ERROR (TM_PORTS);

        /***  ENTER PORT INFO  ***/

           /* Port name */
           strcpy (nxt_port -> port_name, msg.port_name);

           /* Owner process */
           procncpy (nxt_port -> owner_proc, em_hdr.caller);

           /* Msg type */
           nxt_port -> msg_type = msg.msg_type;

           /* Residency */
           nxt_port -> residency = em_hdr.sending_mc;
        /* Update next available port entry */
        while (nxt_port -> residency != EMPTY)
                nxt_port++;
# ifdef DEBUG
dump_ports();
# endif

        return SUCCESS;

} /* End of CPORT_MSG */
```

```
/* FILE: project/src/kernel/dport.c */

/* This file contains routines to DESTROY A PORT */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_DPORT 40 * AVE_INST
# define  TIME_MSG_DPORT 30 * AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY  *find_proc();
EXTERN  PORT_ENTRY  *find_port();
EXTERN  int          port_initds();

/* Typedefs for these routines */

typedef  struct {
                  int  dp_index;

                } KC_DPORT;

typedef  struct {
                  KCR_HDR  hdr;
                  PFI      destruct;

                } KCR_DPORT;

typedef  struct {
                  char  port_name[MAXPNAME];

                } EM_DPORT;



/***** DPORT *****/

/* This routine deals with a kcall made to destroy a port */

dport()

{

LOCAL  KC_HDR       kc_hdr;
LOCAL  KC_DPORT     p_blk;
LOCAL  PORT_ENTRY   *dstport;
LOCAL  KCR_DPORT    kcr;
LOCAL  EM_DPORT     dport_msg;
LOCAL  PROC_ENTRY   *caller;

        /* Get parameter block */
        getp_blk (KC_DPORT);

        /* Establish calling process */
        caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

        /* Find dstport in port table */
        if ((dstport = caller -> owned_ports[p_blk.dp_index]) == (PORT_ENTRY
*)EMPTY)
        {
             kcfail (UN_DPORT);
             return FAIL;
        }

        /* For simplicity, fail if still linked to other ports */
        if (dstport -> profile.lnkf_length || dstport -> profile.lnkt_length)
        {
            kcfail (DP_LINKED);
            return FAIL;
        }

        /* Fail if msgs still in q */
        if (dstport -> profile.inmq_length)
        {
            kcfail (DP_MSGS);
            return FAIL;
        }

        /* OK to destroy port */

        /* Remove entry in caller's owned ports table */
        caller -> owned_ports[p_blk.dp_index] = (PORT_ENTRY *)EMPTY;

        /* Decrement caller's migration size */
        caller -> migration_siz -= (sizeof(PORT_ENTRY) + sizeof(int));

        /* Update caller's next owned port entry */
        if (p_blk.dp_index < caller -> nxt_ownp)
            caller -> nxt_ownp = p_blk.dp_index;

        /* Decrement caller's owned ports length */
        caller -> ownp_length --;

        /* Decrement no of global ports */
        nports --;

        /* Update next available port */
        if (dstport < nxt_port)
            nxt_port = dstport;

        /* Return destruction address to caller */
        kcr.hdr = KCSUCC_MASK;
        kcr.destruct = dstport -> destruct;
        kcsucc (kcr, TIME_DPORT);

        /* Broadcast news of destruction */
        strcpy (dport_msg.port_name, dstport -> port_name);
        broadcast (DPORT_MSG, caller, (char *)&dport_msg, sizeof(EM_DPORT));

        /* Re-initialise port entry */
        port_initds (dstport);

#ifdef DEBUG
dump_procs();
dump_ports();
#endif

        return SUCCESS;

} /* End of DPORT */



/***** DPORT_MSG *****/

/* This routine deals with an emsg to announce port destruction */

dport_msg()

{

LOCAL  EM_HDR          em_hdr;
LOCAL  EM_DPORT        msg;
LOCAL  PORT_ENTRY      *dstport;

        /* Check if needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_DPORT),
DPORT_MSG) == FORWARDED)
            return FORWARDED;

        /* Find dstport in port table */
        dstport = find_port (em_hdr.caller, msg.port_name);

        /* Re-initialise port table entry */
        port_initds (dstport);

        /* Decrement no of global ports */
        nports --;

        /* Update nxt available port */
        if (dstport < nxt_port)
            nxt_port = dstport;

        /* Update time */
        time_update (TIME_MSG_DPORT, OStime, (PROC_ENTRY *)EMPTY);

        return SUCCESS;

} /* End of DPORT_MSG */
```

```
/* FILE: project/src/kernel/lport.c */

/* This file contains routines to LINK A PORT */

/* Includes for this file */

# include   <stdio.h>
# include   <signal.h>
# include   "/usr/acct/ian/project/src/include/keywds.h"
# include   "/usr/acct/ian/project/src/include/params.h"
# include   "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include   "/usr/acct/ian/project/src/include/sys/types.h"
# include   "/usr/acct/ian/project/src/include/sys/globvars.h"
# include   "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_LOC_LPORT      40 * AVE_INST
# define  TIME_REM_LPORT      40 * AVE_INST
# define  TIME_FAIL_REQ       30 * AVE_INST
# define  TIME_OK_REQ         50 * AVE_INST
# define  TIME_ACK            50 * AVE_INST
# define  TIME_NACK           15 * AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PORT_ENTRY  *find_port();
EXTERN  PROC_ENTRY  *find_proc();

/* Typedefs used in these routines */

typedef   struct {
                    int  lf_index;
                    char lt_name[MAXPNAME];

                  } KC_LPORT;

typedef   struct {
                    char  lt_name[MAXPNAME];
                    char  lf_name[MAXPNAME];

                  } EM_LP_REQ;

typedef   struct {
                    char  lt_name[MAXPNAME];
                    char  lf_name[MAXPNAME];

                  } EM_LP_ACK;

typedef   struct {
                    int  result;

                  } EM_LP_NACK;
```

```
/***** LPORT *****/

/* This routine deals with a kcall made to link a port */

lport()

{

LOCAL  KC_HDR       kc_hdr;
LOCAL  KC_LPORT     p_blk;
LOCAL  PROC_ENTRY   *caller;
LOCAL  PROC_ENTRY   *lt_owner;
LOCAL  PORT_ENTRY   *lfport;
LOCAL  PORT_ENTRY   *ltport;
LOCAL  KCR_HDR      kcr_hdr;


        /* Get parameter block */
        getp_blk (KC_LPORT);

        /* Establish calling process */
        caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

        /* Check existence of ports */

        /* LF port */
        if ((lfport = caller->owned_ports[p_blk.lf_index]) == (PORT_ENTRY *)EMPTY)
        {
             kcfail (UN_LFPORT);
             return FAIL;
        }

        /* LT port */
        if ((ltport = find_port(caller->proc_name, p_blk.lt_name)) == (PORT_ENTRY
*)EMPTY)
        {
             kcfail (UN_LTPORT);
             return FAIL;
        }

        /* Check lfport has not got too many links to other ports */
        if (lfport -> profile.lnkt_length >= MAXLTO)
        {
            kcfail (TM_LTO);
            return FAIL;
        }

        /* Check residency of ltport; local=enter info; remote=send req */
        if (ltport -> residency == this_mc)  /* Local */
        {
            /* Check ltport has not got too many links from other ports */
            if (ltport -> profile.lnkf_length >= MAXLFROM)
            {
                kcfail (TM_LFROM);
                return FAIL;
            }
            /*** ENTER INFO FOR PORTS ***/
              /* lt port */
                 /* Make entry in links_from table */
                 ltport -> links_from[ltport->profile.nxt_lf] = lfport;

                 /* Increment links_from length */
                 ltport -> profile.lnkf_length++;

                 /* Update next avail. links_from entry */
                 while (ltport -> links_from[ltport->profile.nxt_lf] !=
(PORT_ENTRY *)EMPTY)
                        ltport -> profile.nxt_lf++;

                  /* Increment owner's migration size */
                  lt_owner = find_proc(ltport -> owner_proc);
                  lt_owner -> migration_siz += MAXPNAME + sizeof(int);


              /* lf port */
                 /* Make entry in links_to table */
                 lfport -> links_to[lfport->profile.nxt_lt].port = ltport;

                 /* Increment links_to length */
                 lfport -> profile.lnkt_length++;

                 /* Make note of ltports index */
                 kcr_hdr = lfport -> profile.nxt_lt | KCSUCC_MASK;

                 /* Update next avail. links_to entry */
                 while (lfport -> links_to[lfport->profile.nxt_lt].port !=
(PORT_ENTRY *)EMPTY)
                        lfport -> profile.nxt_lt++;

                 /* Increment caller's migration size */
                 caller -> migration_siz += MAXPNAME + sizeof(int);


# ifdef DEBUG
dump_procs();
dump_ports();
# endif
           /* Return result to caller */
           kcsucc (kcr_hdr, TIME_LOC_LPORT);
           return SUCCESS;

        }
        else  /* Remote */
        {
        EM_LP_REQ  lreq_msg;
        EM_HDR     em_hdr;
        COMMS_HDR  comms_hdr = LPORT_REQ | EMSG_MASK;

            /* Mark process as blocked */
            caller -> blocked = BLOCKED;

            /* Decrement active proc count */
            n_active_local_procs --;
            /* Set up req msg */
            em_hdr.sending_mc = this_mc;
            procncpy (em_hdr.caller, caller->proc_name);
            em_hdr.dst_mc = ltport -> residency;
            strcpy (lreq_msg.lt_name, p_blk.lt_name);
            strcpy (lreq_msg.lf_name, lfport->port_name);

            /* Update time */
            time_update (TIME_REM_LPORT, USERtime, caller);

            /* Send req msg */
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&lreq_msg,
sizeof(EM_LP_REQ), USERtime, caller);

            return  REMLINK;

        }

} /* End of LPORT */




/***** LP_REQ *****/

/* This routine deals with an emsg to request a port link */

lp_req()

{

LOCAL  EM_HDR      em_hdr;
LOCAL  EM_LP_REQ   msg;
LOCAL  COMMS_HDR   comms_hdr;
LOCAL  EM_LP_ACK   lack_msg;
LOCAL  EM_LP_NACK  lnack_msg;
LOCAL  PORT_ENTRY  *ltport;
LOCAL  PORT_ENTRY  *lfport;
LOCAL  PROC_ENTRY  *lt_owner;

        /* Check if needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_LP_REQ),
LPORT_REQ) == FORWARDED)
            return FORWARDED;

        /* Find ltport and lfport */
        ltport = find_port (em_hdr.caller, msg.lt_name);
        lfport = find_port (em_hdr.caller, msg.lf_name);

        /* Check if migrated */
        if (ltport -> residency != this_mc)
        {
            comms_hdr = LPORT_REQ | EMSG_MASK;
            em_hdr.dst_mc = ltport -> residency;
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&msg,
sizeof(EM_LP_REQ), OStime, (PROC_ENTRY *)EMPTY);
```

```
            return FORWARDED;
        }

        /* Check ltport has not got too many links from other ports */
        if (ltport -> profile.lnkf_length >= MAXLFROM)
        {
            comms_hdr = LPORT_NACK | EMSG_MASK;
            em_hdr.dst_mc = em_hdr.sending_mc;
            em_hdr.sending_mc = this_mc;
            lnack_msg.result = TM_LFROM;

            /* Update time */
            time_update (TIME_FAIL_REQ, OStime, (PROC_ENTRY *)EMPTY);

            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&lnack_msg,
sizeof(EM_LP_NACK), OStime, (PROC_ENTRY *)EMPTY);

            return FAIL;
        }

        /* Request must be OK so enter ltport info */
            /* Make entry in links_from table */
            ltport -> links_from[ltport->profile.nxt_lf] = lfport;

            /* Increment links_from length */
            ltport -> profile.lnkf_length++;

            /* Update next avail. links_from entry */
            while (ltport -> links_from[ltport->profile.nxt_lf] != (PORT_ENTRY
*)EMPTY)
                    ltport -> profile.nxt_lf++;

            /* Increment owner's migration size */
            lt_owner = find_proc(ltport -> owner_proc);
            lt_owner -> migration_siz += MAXPNAME + sizeof(int);

        /* Send acknowledgement */
        comms_hdr = LPORT_ACK | EMSG_MASK;
        em_hdr.dst_mc = em_hdr.sending_mc;
        em_hdr.sending_mc = this_mc;
        strcpy (lack_msg.lt_name, msg.lt_name);
        strcpy (lack_msg.lf_name, msg.lf_name);
        TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&lack_msg,
sizeof(EM_LP_ACK), OStime, (PROC_ENTRY *)EMPTY);

        /* Update time */
        time_update (TIME_OK_REQ, OStime, (PROC_ENTRY *)EMPTY);

        return SUCCESS;

} /* End of LP_REQ */




/***** LP_ACK *****/
```

```
/* This routine deals with an emsg acknowledging a port link */

lp_ack()

{

LOCAL  EM_HDR         em_hdr;
LOCAL  EM_LP_ACK      msg;
LOCAL  PORT_ENTRY     *ltport;
LOCAL  PORT_ENTRY     *lfport;
LOCAL  KCR_HDR        kcr_hdr;
LOCAL  PROC_ENTRY     *caller;

        /* Check if needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_LP_ACK),
LPORT_ACK) == FORWARDED)
            return FORWARDED;

        /* Find ltport and lfport */
        ltport = find_port (em_hdr.caller, msg.lt_name);
        lfport = find_port (em_hdr.caller, msg.lf_name);

        /*** ENTER INFO FOR LFPORT ***/
           /* Make entry in links_to table */
           lfport -> links_to[lfport->profile.nxt_lt].port = ltport;

           /* Increment links_to length */
           lfport -> profile.lnkt_length++;

           /* Make note of link_to index */
           kcr_hdr = lfport -> profile.nxt_lt | KCSUCC_MASK;

           /* Update nxt avail. links_to entry */
           while (lfport -> links_to[lfport->profile.nxt_lt].port != (PORT_ENTRY
*)EMPTY)
                  lfport -> profile.nxt_lt++;

        /* Establish original caller */
        caller = find_proc (em_hdr.caller);

        /* Increment caller's migration size */
        caller -> migration_siz += MAXPNAME + sizeof(int);

        /* Mark process as unblocked */
        caller -> blocked = UNBLOCKED;

        /* Increment active proc count */
        n_active_local_procs ++;

        /* Return lt index to caller */
        kcsucc (kcr_hdr, TIME_ACK);
        return SUCCESS;

} /* End of LP_ACK */
```

```
/***** LP_NACK *****/

/* This routine deals with an emsg for a failed remote port link */

lp_nack()

{

LOCAL  EM_HDR      em_hdr;
LOCAL  EM_LP_NACK msg;
LOCAL  PROC_ENTRY *caller;

        /* Check if needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_LP_NACK),
LPORT_NACK) == FORWARDED)
            return FORWARDED;

        /* Establish original caller */
        caller = find_proc (em_hdr.caller);

        /* Mark process as unblocked */
        caller -> blocked = UNBLOCKED;

        /* Increment active proc count */
        n_active_local_procs ++;

        /* Update time */
        time_update (TIME_NACK, USERtime, caller);

        /* Return result to caller */
        kcfail (msg.result);
        return SUCCESS;

} /* End of LP_NACK */
```

```
/* FILE project/src/kernel/uport.c */

/* This file contains routines to UNLINK A PORT */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_LOC_UPORT       55 * AVE_INST
# define  TIME_REM_UPORT       60 * AVE_INST
# define  TIME_FAIL_REQ        30 * AVE_INST
# define  TIME_OK_REQ          50 * AVE_INST
# define  TIME_ACK             15 * AVE_INST
# define  TIME_NACK            13 * AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PORT_ENTRY  *find_port();
EXTERN  PROC_ENTRY  *find_proc();

/* Typedefs used in these routines */

typedef  struct {
                   int  lf_index;
                   int  lt_index;

                 } KC_UPORT;

typedef  struct {
                   char  lt_name[MAXPNAME];
                   char  lf_name[MAXPNAME];

                 } EM_UP_REQ;

typedef  struct {
                   int  result;

                 } EM_UP_ACK;

typedef  struct {
                   int  result;

                 } EM_UP_NACK;



/***** UPORT *****/
```

```
/* This routine deals with a kcall made to unlink a port */

uport ()

{

LOCAL  KC_HDR      kc_hdr;
LOCAL  KC_UPORT    p_blk;
LOCAL  PROC_ENTRY *caller;
LOCAL  PROC_ENTRY *lt_owner;
LOCAL  PORT_ENTRY *lfport;
LOCAL  PORT_ENTRY *ltport;
LOCAL  KCR_HDR     kcr_hdr;
LOCAL  int         p;

        /* Get parameter block */
        getp_blk (KC_UPORT);

        /* Establish calling process */
        caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

        /* Find & check lfport */
        if ((lfport = caller -> owned_ports[p_blk.lf_index]) == (PORT_ENTRY
*)EMPTY)
        {
             kcfail (UN_LFPORT);
             return FAIL;
        }

        /* Find & check ltport */
        if ((ltport = lfport -> links_to[p_blk.lt_index].port) == (PORT_ENTRY
*)EMPTY)
        {
             kcfail (UN_LTPORT);
             return FAIL;
        }

        /* Remove ltport from lfport's links to table */
        lfport -> links_to[p_blk.lt_index].port = (PORT_ENTRY *)EMPTY;
        lfport -> links_to[p_blk.lt_index].nmsgs = 0;
        lfport -> links_to[p_blk.lt_index].tot_msglength = 0;

        /* Update next link_to for lfport */
        if (p_blk.lt_index < lfport -> profile.nxt_lt)
            lfport -> profile.nxt_lt = p_blk.lt_index;

        /* Decrement link_to length for lfport */
        lfport -> profile.lnkt_length --;

        /* Decrement caller's migration size */
        caller -> migration_siz -= (MAXPNAME + sizeof(int));

        /* Check if ltport is local */
        if (ltport -> residency == this_mc)    /* Local */
        {
            /* Remove lfport from ltport's links from table */
            p = 0;
            while (ltport -> links_from[p] != lfport)
                   p++;
            ltport -> links_from[p] = (PORT_ENTRY *)EMPTY;

            /* Update next link_from for ltport */
            if (p < ltport -> profile.nxt_lf)
                ltport -> profile.nxt_lf = p;

            /* Decrement link_from length for ltport */
            ltport -> profile.lnkf_length --;

            /* Find owner of ltport */
            lt_owner = find_proc (ltport -> owner_proc);

            /* Decrement owner's migration size */
            lt_owner -> migration_siz -= (MAXPNAME + sizeof(int));

            /* Return to caller */
            kcr_hdr = KCSUCC_MASK;
            kcsucc(kcr_hdr, TIME_LOC_UPORT);
#ifdef DEBUG
dump_procs();
dump_ports();
#endif
            return SUCCESS;

        }
        else  /* Ltport is remote */
        {
            EM_UP_REQ  ureq_msg;
            EM_HDR     em_hdr;
            COMMS_HDR  comms_hdr = UPORT_REQ | EMSG_MASK;

            /* Mark calling process as blocked */
            caller -> blocked = BLOCKED;

            /* Decrement active proc count */
            n_active_local_procs --;

            /* Set up req. msg */
            em_hdr.sending_mc = this_mc;
            procncpy (em_hdr.caller, caller -> proc_name);
            em_hdr.dst_mc = ltport -> residency;
            strcpy (ureq_msg.lt_name, ltport -> port_name);
            strcpy (ureq_msg.lf_name, lfport -> port_name);

            /* Send req msg */
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&ureq_msg,
sizeof(EM_UP_REQ), USERtime, caller);

            /* Update time */
            time_update (TIME_REM_UPORT, USERtime, caller);

#ifdef DEBUG
dump_procs();
dump_ports();
#endif
```

```
                return REMUNLINK;

            }

    } /* End of uport */







/***** UP_REQ *****/

/* This routine deals with an emsg requesting to unlink a port */

up_req ()

{

LOCAL   EM_HDR          em_hdr;
LOCAL   EM_UP_REQ       msg;
LOCAL   COMMS_HDR       comms_hdr;
LOCAL   EM_UP_ACK       uack_msg;
LOCAL   EM_UP_NACK      unack_msg;
LOCAL   PORT_ENTRY      *lfport;
LOCAL   PORT_ENTRY      *ltport;
LOCAL   PROC_ENTRY      *lt_owner;
LOCAL   int             p;

        /* Check if needs to be forwarded */
        if (check_forward ((char *)&em_hdr, (char *)&msg, sizeof(EM_UP_REQ),
UPORT_REQ) == FORWARDED)
            return FORWARDED;

        /* Find ltport in port table */
        if((ltport = find_port (em_hdr.caller, msg.lt_name)) == (PORT_ENTRY
*)EMPTY)
        {
            /* Port must have been destroyed */
            /* Send nack to sending mc */
            comms_hdr = UPORT_NACK | EMSG_MASK;
            em_hdr.dst_mc = em_hdr.sending_mc;
            em_hdr.sending_mc = this_mc;
            unack_msg.result = UN_LTPORT;
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&unack_msg,
sizeof(EM_UP_NACK), OStime, (PROC_ENTRY *)EMPTY);

             /* Update time */
             time_update (TIME_FAIL_REQ, OStime, (PROC_ENTRY *)EMPTY);

             return FAIL;
        }

        /* Check if migrated */
        if (ltport -> residency !=this_mc)
        {
            comms_hdr = UPORT_REQ | EMSG_MASK;
            em_hdr.dst_mc = ltport -> residency;
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&msg,
sizeof(EM_UP_REQ), OStime, (PROC_ENTRY *)EMPTY);
            return FORWARDED;
        }

        /* Find lfport in port table */
        lfport = find_port (em_hdr.caller, msg.lf_name);

        /* Remove links_from entry for ltport */
        p = 0;
        while (ltport -> links_from[p] != lfport)
               p++;
        ltport -> links_from[p] = (PORT_ENTRY *)EMPTY;

        /* Update next link_from for ltport */
        if (p < ltport -> profile.nxt_lf)
            ltport -> profile.nxt_lf = p;

        /* Decrement link_from length for ltport */
        ltport -> profile.lnkf_length --;

        /* Find owner of ltport */
        lt_owner = find_proc (ltport -> owner_proc);

        /* Decrement owner's migration size */
        lt_owner -> migration_siz -= (MAXPNAME + sizeof(int));

        /* Send ack to sending mc */
        comms_hdr = UPORT_ACK | EMSG_MASK;
        em_hdr.dst_mc = em_hdr.sending_mc;
        em_hdr.sending_mc = this_mc;
        uack_msg.result = p;
        TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&uack_msg,
sizeof(EM_UP_ACK), OStime, (PROC_ENTRY *)EMPTY);

#ifdef DEBUG
dump_procs();
dump_ports();
#endif
        return SUCCESS;

} /* End of UPORT_REQ */




/***** UPORT_ACK *****/

/* This routine deals with an emsg acknowledging an unlink port */

up_ack()

{

LOCAL  EM_HDR          em_hdr;
LOCAL  EM_UP_ACK       msg;
LOCAL  KCR_HDR        kcr_hdr;
LOCAL  PROC_ENTRY     *caller;

        /* Check if needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_UP_ACK),
UPORT_ACK) == FORWARDED)
            return FORWARDED;

        /* Find caller */
        caller = find_proc (em_hdr.caller);

        /* Mark caller as unblocked */
        caller -> blocked = UNBLOCKED;

        /* Increment active proc count */
        n_active_local_procs ++;

        /* Send caller result */
        kcr_hdr = msg.result | KCSUCC_MASK;
        kcsucc(kcr_hdr, TIME_ACK);

        return SUCCESS;

} /* End of UP_ACK */




/***** UP_NACK *****/

/* This routine deals with an emsg for a failed remote unlink port */

up_nack()

{

LOCAL  EM_HDR         em_hdr;
LOCAL  EM_UP_NACK     msg;
LOCAL  PROC_ENTRY     *caller;

        /* Check if needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_UP_NACK),
UPORT_NACK) == FORWARDED)
            return FORWARDED;

        /* Find caller */
        caller = find_proc(em_hdr.caller);

        /* Mark caller as unblocked */
        caller -> blocked = UNBLOCKED;

        /* Increment active proc count */
        n_active_local_procs ++;

        /* Update time */
        time_update (TIME_NACK, USERtime, caller);
        /* Send result to caller */
        kcfail (msg.result);

        return SUCCESS;

} /* End of UP_NACK */
```

```
/* FILE: project/src/kernel/cproc.c */

/* This file contains routines for dealing with a kernel call */
/* TO CREATE A PROCESS */

/* Includes for this file */

# include <stdio.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"
# include "/usr/acct/ian/project/src/include/sys/globvars.h"
# include "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_CPROC 150 * AVE_INST


/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY  *find_proc();


/* Types used in this file */

typedef struct {
                  char  pname[MAXPNAME];
                  char  pfile[MAXFNAME];
                  int   idr;

                } KC_CPROC;

typedef struct {
                  PROCN  pid;

                } EM_CPROC;

typedef struct {
                  KCR_HDR  hdr;
                  PROCN  pid;

                } KCR_CPROC;



/***** CPROC *****/

cproc()
{

LOCAL  KC_HDR        kc_hdr;
LOCAL  KC_CPROC      p_blk;
LOCAL  PROC_ENTRY    *caller;
LOCAL  int           upid;
LOCAL  double       rnd_num;
LOCAL  int          mig_mc;
OWN    int          gno;
EXTERN double       erand48();


        /* Check if boot time */
        if (boot)
        {
            strcpy (p_blk.pfile, "shell.x");
            strcpy (p_blk.pname, "shell");
            time_update (CONTEXT_SWITCH, OStime, (PROC_ENTRY *)EMPTY);
        }

        /* Check if load balancing process needs to be created */
        else if (load_bal_active)
             {
                 strcpy (p_blk.pfile, "lb_alg.x");
                 sprintf (p_blk.pname, "l_bal%d", this_mc);
                 time_update (CONTEXT_SWITCH, OStime, (PROC_ENTRY *)EMPTY);
             }

        /* Check for synthetic workload generation */
        else if (synth_workload)
             {
                 strcpy (p_blk.pfile, "parent.x");
                 sprintf (p_blk.pname, "usr%d", gno);
                 p_blk.idr = nxt_job;
                 caller = &process_table[0];
             }

        else
        {


        /* Get parameter block */
        getp_blk(KC_CPROC);

        /* Establish calling process */
        caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

#       ifdef DEBUG
        if (caller == (PROC_ENTRY *)EMPTY)
            printf ("BUG\n");
        else
            printf ("Caller is %d\n", (caller -
process_table)/sizeof(PROC_ENTRY));
#       endif

        /* Increment no. of global processes checking if too many */
        if (++nprocs > MAXPROCS)
        {
            nprocs--;
            /* Return reason for failure to caller */
            kcfail(TM_PROCS);
            return FAIL;
        }


        }
# ifdef DEBUG
printf ("cproc - check 1\n");
# endif

        /* Increment active & local proc counts */
        n_local_procs ++;
        n_active_local_procs ++;


        /* Enter new process info. */

        if (boot)
        {
            nxt_proc -> proc_name.pgroup.gmc = this_mc;
            nxt_proc -> proc_name.pgroup.gnum = 0;
            strcpy (nxt_proc->proc_name.pname, "shell");
            nxt_proc -> residency = this_mc;
            nxt_proc -> orig_mc = this_mc;
            procncpy(nxt_proc->parent, nxt_proc->proc_name);
        }

        /* Check if load balancing process is being created */
        else if (load_bal_active)
             {
                 nxt_proc -> proc_name.pgroup.gmc = 0;
                 nxt_proc -> proc_name.pgroup.gnum = 0;
                 sprintf (nxt_proc -> proc_name.pname, "l_bal%d",this_mc);
                 nxt_proc -> residency = this_mc;
                 procncpy (nxt_proc -> parent, nxt_proc -> proc_name);
             }

        else
        {
           /* Process Name */
           nxt_proc -> proc_name.pgroup.gmc = caller -> proc_name.pgroup.gmc;
#ifdef REMTEST
nxt_proc -> proc_name.pgroup.gmc = 0;
#endif
           if (caller -> proc_name.pgroup.gnum == 0)
           {
               gno++;
               nxt_proc -> proc_name.pgroup.gnum = gno;
           }
           else
           nxt_proc -> proc_name.pgroup.gnum = caller -> proc_name.pgroup.gnum;
           strcpy (nxt_proc->proc_name.pname, p_blk.pname);

           /* Residency */
           nxt_proc -> residency = this_mc;
           nxt_proc -> orig_mc = this_mc;

           /* Parent */
           procncpy(nxt_proc->parent, caller->proc_name);

        }
```

```
# ifdef DEBUG
printf ("cproc - check 2\n");
# endif

        proc_setup = 0;
        alarm(0);


        if (!boot)
        /* Start up new process */
        switch (upid = fork())
        {
             case  FAIL:          ERROR (FORK_FAIL);

             case  CHILD:         {
                                  char mc_id [sizeof(int)+1];
                                  char mypid [sizeof(PROCN)+1];
                                  char kcmk_lock [OPLOCKSIZ];
                                  char kcrt_lock [KCRLOCKSIZ];
                                  char uppid [sizeof(int)+1];
                                  char idrstr[2];


                                  /* Set up args for execl */
                                  sprintf (mc_id, "%d", this_mc);

                                  sprintf (mypid, "%d %d %s",
                                           nxt_proc -> proc_name.pgroup.gmc,
                                           nxt_proc -> proc_name.pgroup.gnum,
                                           nxt_proc -> proc_name.pname);
                                  sprintf (kcmk_lock, "%s", ownp_locks[this_mc]);
                                  sprintf (kcrt_lock, "%s", kcret_locks[this_mc]);
                                  sprintf (uppid, "%d", getppid());
                                  sprintf (idrstr,"%d", p_blk.idr);

                                  close (own_pipe);
                                  close (kcret_pipe);

                                  if (load_bal_active)
                                  {
                                      char  shared_id[sizeof(int)+1];
                                      sprintf (shared_id, "%d", shmid);
                                      execl (p_blk.pfile, p_blk.pname,
                                             mc_id, mypid, kcmk_lock, kcrt_lock,
                                             uppid, shared_id, 0);
                                  }
                                  else
                                  /* Exec new process */
                                  execl (p_blk.pfile, p_blk.pname,
                                         mc_id, mypid, kcmk_lock, kcrt_lock,
                                         uppid,idrstr,0);

                                  printf ("EXECL FAILED\n");

                                 }

        PARENT:                  /* Enter new process Unix pid */
                                 nxt_proc -> upid = upid;
```

```
                               if (!load_bal_active)
                               while (!proc_setup)
                               {
                                      alarm(3);
                                      pause();
                                      alarm(0);
                               }

                               if (!synth_workload && !load_bal_active)
                               {
                                KCR_CPROC  kcr;

                                /* Send result back to caller */
                                kcr.hdr = KCSUCC_MASK;
                                procncpy(kcr.pid, nxt_proc->proc_name);
                                kcsucc(kcr, TIME_CPROC);
                               }

                               /* Store index of last process created */
                               last_proc_creat = ((int)nxt_proc -
                                (int)process_table)/sizeof(PROC_ENTRY);

                               /* Make process schedulable */
                               nxt_proc -> schedulable = TRUE;

                               /* Update nxt_proc */
                               while (nxt_proc -> residency != EMPTY)
                                      nxt_proc++;

                               /* THRESHOLD LOAD BALANCING */
                               /*if (n_local_procs > N_SYS_PROCS + THRESHOLD)
                               {*/
                                  /* Suspend process */
                                 /* (process_table + last_proc_creat) ->
                                     schedulable = FALSE;
                                  n_active_local_procs --; */

                                  /* Probe for alternative processor */
                                 /* send_probe (process_table + last_proc_creat);
                               } */
# ifdef DEBUG
dump_procs();
# endif

                                return SUCCESS;

        }

        else /* Boot time */
        {
           nxt_proc -> upid = 0;
           nxt_proc++;
           return SUCCESS;
        }

} /* End of CPROC */
```

```
/* FILE: project/src/kernel/exit_proc.c */

/* This file contains a routine to deal with process exit */

/* Includes for this file */
# include  <stdio.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Defines for time */
# define  TIME_EXIT  50 * AVE_INST

/* Functions EXTERNal to this file */
EXTERN  PROC_ENTRY  *find_proc();

/* Typedefs for these routines */

typedef  struct {
                 int  retval;
               } KC_EXIT;

typedef  struct {
                 int upid;
               } EM_EXIT;


/***** EXIT_PROC *****/

/* This routine deals with a kcall to exit a process */


exit_proc()

{
LOCAL  KC_EXIT  p_blk;
LOCAL  KC_HDR   kc_hdr;
LOCAL  int      died;
LOCAL  int      status;
LOCAL  PROC_ENTRY  *caller;

        /* Get parameter block */
        getp_blk (KC_EXIT);

        /* Establish caller */
        caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

        /* Check if caller has ports open - if so fail */
        if (caller -> ownp_length)
        {
            kcfail (EX_PORTS);
            return FAIL;
        }

        else  /* Exit is OK so return to caller */
        {
            KCR_HDR kcr_hdr = KCSUCC_MASK;
            kcsucc (kcr_hdr, TIME_EXIT);
        };

        /* Dump process info for performance evaluation */
        exit_dump (caller);

        /* Wait for child to die */
        if (caller->orig_mc != this_mc)  /* Proc originated elsewhere */

        {
            EM_EXIT  ex_msg;
            EM_HDR   em_hdr;
            COMMS_HDR  comms_hdr = EXIT_MSG | EMSG_MASK;

            /* Set up exit msg */
            em_hdr.sending_mc = this_mc;
            procncpy (em_hdr.caller, caller->proc_name);
            em_hdr.dst_mc = caller -> orig_mc;
            ex_msg.upid = caller -> upid;

            /* Send msg */
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&ex_msg,
sizeof(EM_EXIT),OStime,(PROC_ENTRY *)EMPTY);
        }

        else  /* Process originated here */

        {
             died = wait(&status);
             if (died == -1) printf("Int during own wait\n");
             if (status != 0) printf ("SIGNAL DEATH\n");
        }

        /* Remove process entry */
        proc_initds (caller);

        if (caller < nxt_proc)
            nxt_proc = caller;

        /* Update process counts */
        nprocs --;
        n_local_procs --;
        n_active_local_procs --;

        /* Dump performance information */

        return SUCCESS;

} /* End of EXIT_PROC */
```

```
/***** EXIT_MSG *****/

/* This routine removes zombies from the process table */

exit_msg()

{

LOCAL  EM_HDR  em_hdr;
LOCAL  EM_EXIT msg;
LOCAL  int     status;
LOCAL  int     died;

        /* Check if msg needs to be forwarded */
        if (check_forward((char *)&em_hdr, (char *)&msg, sizeof(EM_EXIT),
EXIT_MSG) == FORWARDED)
            return FORWARDED;

        /* Remove zombie from process table */
        died = wait(&status);
        if (died == -1)printf ("Int during wait-msg\n");
        if (status != 0) printf("SIGNAL DEATH\n");

        return SUCCESS;

} /* End of EXIT_MSG */
```

```
/* This file contains a routine to do processing on a process's behalf */

/* Includes for this file */
# include  <stdio.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Routines EXTERNal to this file */
EXTERN  PROC_ENTRY *find_proc();

/***** DO_PROCESSING *****/

/* This routine simulates normal processing */

/* Typedefs for this routine */

typedef  struct {
                  int  ninst;
                } KC_DOPROC;

do_processing()

{

LOCAL  KC_HDR          kc_hdr;
LOCAL  KC_DOPROC       p_blk;
LOCAL  KCR_HDR         kcr_hdr;
LOCAL  PROC_ENTRY      *caller;

        /* Set up parameter block */
        getp_blk (KC_DOPROC);

        /* Establish calling process */
        caller = find_proc (kc_hdr);

#ifdef DEBUG
        fprintf (trace,"Caller %s\n", caller->proc_name.pname);
#endif

        /* Update time due to context switch */
        contxt_swtch;

        /* Return to caller */
        kcr_hdr = KCSUCC_MASK;
        kcsucc (kcr_hdr, p_blk.ninst*AVE_INST);

        return SUCCESS;

} /* End of DO_PROCESSING */
```

```
/* FILE: project/src/kernel/rmsg.c */

/* This file contains routines to receive msgs */


/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_NBR_SETUP              5 * AVE_INST
# define  NBR_TIME_MSG_RDY            50 * AVE_INST
# define  TIME_PENDING_NBR            10 * AVE_INST

# define  BR_TIME_MSG_RDY             40 * AVE_INST
# define  BR_PENDING_TIME             10 * AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY *find_proc();
EXTERN  PORT_ENTRY *find_port();
EXTERN  char        *malloc();

/* Typedefs used in these routines */

typedef  struct {
                    int   port;
                    MSG   *msg_loc;

                  } KC_NBRMSG;

typedef  struct {
                    int  port;

                  } KC_BRMSG;

typedef  struct {
                    KCR_HDR  hdr;
                    MSG      msg;

                  } KCR_BRMSG;

typedef  struct {
                    char  src_port_name[MAXPORTNAME];

                  } EM_NOT_MSG;
```

```
/***** NB_RMSG *****/

/* This routine deals with a kcall made to non-blocking rcv a msg */

nb_rmsg()

{

LOCAL  KC_HDR         kc_hdr;
LOCAL  KC_NBRMSG      p_blk;
LOCAL  PROC_ENTRY     *caller;
LOCAL  PROC_ENTRY     *save_caller;
LOCAL  PORT_ENTRY     *this_port;
LOCAL  PORT_ENTRY     *src_port;
LOCAL  MSG            *save_p;

        /* Get parameter block */
        getp_blk(KC_NBRMSG);

        /* Establish caller */
        caller = save_caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

        /* Check rcv port exists */
        if ((this_port = caller -> owned_ports[p_blk.port]) == (PORT_ENTRY
*)EMPTY)
        {
             kcfail(UN_RPORT);
             return FAIL;
        }

        /* Check there are no more rcvs pending */
        if (this_port -> profile.nb_pending || this_port -> profile.b_pending)
        {
            kcfail(TM_NBRCVS);
            return FAIL;
        }
        else
        {
            KCR_HDR kcr_hdr = KCSUCC_MASK;
            kcsucc(kcr_hdr, TIME_NBR_SETUP);
        }

        /*await_sig (); */   /* Waiting for ack from caller */

        /* Check to see if msg is in q */
        if (this_port -> profile.inmq_length)
        {
            printf ("In mg\n");
            give_msg (p_blk.msg_loc, this_port->msg_q_head, this_port->rcvfunc);

            /* Check if msg rcv'd was sent blocking */
            if (this_port -> msg_q_head -> msg_hdr.snd_type == BLOCKING)
            {
                src_port = find_port (caller->proc_name,
this_port->msg_q_head->msg_hdr.src_port);
```

```
                /* Caller is now msg sender so do a kcreturn for him */
                if (src_port -> residency == this_mc)
                {
                    caller = find_proc (src_port -> owner_proc);
                    KCR_HDR kcr_hdr = KCSUCC_MASK;
                    kcsucc(kcr_hdr, 0);
                }
                else
                {
                    COMMS_HDR comms_hdr = NOT_MSG | EMSG_MASK;
                    EM_HDR     em_hdr;
                    EM_NOT_MSG  notify_msg;

                    em_hdr.sending_mc = this_mc;
                    em_hdr.dst_mc = src_port -> residency;
                    procncpy (em_hdr.caller, caller -> proc_name);
                    strcpy (notify_msg.src_port_name, src_port -> port_name);

                    TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&notify_msg),
                        sizeof (NOT_MSG), OStime, (PROC_ENTRY *)EMPTY);

                }

            }

            /* Update the msg q */
            this_port -> profile.inmq_length --;

            /* Decrement the caller's migration size */
            save_caller -> migration_siz = save_caller -> migration_siz -
sizeof(MSG);
            save_caller -> tot_msg_siz -=
this_port->msg_q_head->msg_hdr.msg_length;

            save_p = this_port -> msg_q_head;

            this_port -> msg_q_head = (this_port->msg_q_head) -> next;

            if (this_port -> msg_q_head == (MSG *)EMPTY)
                this_port -> msg_q_tail = (MSG *)EMPTY;


            /* Tell original caller that msg has arrived */
            printf ("Informing\n");
            got_msgsigack = 0;
            kill (save_caller -> upid, SIGMSG);

            /*await_sig ();*/   /* Waiting for ack of receipt of msg */

            while (!got_msgsigack)
            {
                    alarm(3);
                    pause();
                    alarm(0);
            }
```

```
             /* Update time */
             time_update (NBR_TIME_MSG_RDY, USERtime, save_caller);

             free ((char *)save_p);
        }
        else  /** NO msgs in the q **/
        {
             this_port -> profile.nb_pending = PENDING;
             this_port -> nb_msg_loc = p_blk.msg_loc;

             /* Update time */
             time_update (TIME_PENDING_NBR, USERtime, caller);
        }

#ifdef DEBUG
dump_procs();
dump_ports();
#endif

        return SUCCESS;

} /* End of NB_RMSG */




/***** GIVE_MSG *****/

/* This routine gives a msg to the caller */

give_msg (msg_loc, msg, rcvfunc)

PARAMS  MSG  *msg_loc;
        MSG  *msg;
        PFI  rcvfunc;

{

        /* Send msg location */
        pwrite (kcret_pipe, (char *)&msg_loc, sizeof(MSG *));

        /* Send MSG */
        pwrite (kcret_pipe, (char *)msg, sizeof(MSG));

        /* Send rcvfunc */
        pwrite (kcret_pipe, (char *)&rcvfunc, sizeof(PFI));

}




/***** B_RMSG *****/

/* This routine deals with a kcall made to blocking rcv a msg */
```

```
b_rmsg()

{

LOCAL  KC_HDR         kc_hdr;
LOCAL  KC_BRMSG       p_blk;
LOCAL  PROC_ENTRY     *caller;
LOCAL  PROC_ENTRY     *save_caller;
LOCAL  PORT_ENTRY     *rport;
LOCAL  MSG            *save_p;

        /* Get parameter block */
        getp_blk (KC_BRMSG);

        /* Establish caller */
        caller = save_caller = find_proc (kc_hdr);

        /* Check that rcv port exists */
        if ((rport = caller -> owned_ports[p_blk.port]) == (PORT_ENTRY *)EMPTY)
        {
             kcfail(UN_RPORT);
             return FAIL;
        }

        /* Check if a msg is available in msg_q */
        if (rport -> profile.inmq_length)
        {
            KCR_BRMSG  kcr;

            /* Return the msg to the caller */
            kcr.hdr = KCSUCC_MASK;
            kcr.msg.msg_hdr.msg_length = rport -> msg_q_head ->
msg_hdr.msg_length;
            kcr.msg.msg_hdr.msg_type = rport -> msg_q_head -> msg_hdr.msg_type;
            kcr.msg.msg_hdr.snd_type = rport -> msg_q_head -> msg_hdr.snd_type;
            strcpy (kcr.msg.msg_hdr.dst_port,
rport->msg_q_head->msg_hdr.dst_port);
            strcpy (kcr.msg.msg_hdr.src_port,
rport->msg_q_head->msg_hdr.src_port);
            strcpy (kcr.msg.msg_txt, rport -> msg_q_head -> msg_txt);

            /* Do a kcreturn to caller */
            got_ackkcrsig = 0;
            pwrite (kcret_pipe, (char *)&kcr, sizeof(kcr));
            kill (caller->upid, SIGKCR);
            while (!got_ackkcrsig)
            {
                  alarm(3);
                  pause();
                  alarm(0);
            }

            /* Update time */
            time_update (BR_TIME_MSG_RDY, USERtime, caller);

            /* Check to see if msg rcv'd was sent blocking */
            if (this_port -> msg_q_head -> msg_hdr.snd_type == BLOCKING)
            {
                 src_port = find_port (caller->proc_name,
this_port->msg_q_head->msg_hdr.src_port);



                 /* Caller is now msg sender so do a kcreturn for him */
                 if (src_port -> residency == this_mc)
                 {
                     caller = find_proc (src_port -> owner_proc);
                     KCR_HDR kcr_hdr = KCSUCC_MASK;
                     kcsucc(kcr_hdr, 0);
                 }
                 else
                 {

                     COMMS_HDR comms_hdr = NOT_MSG | EMSG_MASK;
                     EM_HDR     em_hdr;
                     EM_NOT_MSG  notify_msg;

                     em_hdr.sending_mc = this_mc;
                     em_hdr.dst_mc = src_port -> residency;
                     procncpy (em_hdr.caller, caller -> proc_name);
                     strcpy (notify_msg.src_port_name, src_port -> port_name);

                     TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&notify_msg),
                         sizeof (NOT_MSG), OStime, (PROC_ENTRY *)EMPTY);

                 }

              }


             /* Update msg_q */
             rport -> profile.inmq_length --;

             /* Decrement caller's migration size */
             save_caller -> migration_siz = save_caller -> migration_siz -
sizeof(MSG);
             save_caller -> tot_msg_siz -= rport->msg_q_head->msg_hdr.msg_length;

             save_p = rport -> msg_q_head;

             rport -> msg_q_head = rport -> msg_q_head -> next;

             if (rport -> msg_q_head == (MSG *)EMPTY)
                 rport -> msg_q_tail = (MSG *)EMPTY;

             free ((char *)save_p);

         }

         else  /* No MSG in the q */
         {
               rport -> profile.b_pending = PENDING;

              /* Mark process as blocked */
              caller -> blocked = BLOCKED;

              /* Decrement active proc count */
```

```
                n_active_local_procs --;

                /* Update time */
                time_update (BR_PENDING_TIME, USERtime, caller);
        }

#ifdef DEBUG
dump_procs();
dump_ports();
#endif

        return SUCCESS;

} /* End of B_RMSG */
```

```
/* FILE: project/src/kernel/smsg.c */

/* This file contains routines to  non-blocking send a msg */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_NBS_SETUP                5 * AVE_INST
# define  TIME_NBR_WAITER              50 * AVE_INST
# define  TIME_BR_WAITER               40 * AVE_INST
# define  TIME_APP_MSGQ                20 * AVE_INST
# define  TIME_REMOTE                  50 * AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY *find_proc();
EXTERN  PORT_ENTRY *find_port();
EXTERN  char       *malloc();

/* Typedefs used in these routines */

typedef  struct {
                   int   sport;
                   int   dport;
                   int   msg_length;
                   int   msg_type;
                   char  msg_txt[MSG_SIZ];

                 } KC_NBSMSG;

typedef  struct {
                   KCR_HDR  hdr;
                   MSG      msg;

                 } KCR_BRMSG;

typedef  struct {
                   MSG  *umsg;

                 } EM_UMSG;

typedef  struct {
                   char  src_port_name;

                 }  EM_NOT_MSG;
```

```
/***** NB_SMSG *****/

/* This routine deals with a kcall made to non-blocking snd a msg */

nb_smsg()

{

LOCAL  KC_HDR           kc_hdr;
LOCAL  KC_NBSMSG        p_blk;
LOCAL  PROC_ENTRY       *caller;
LOCAL  PROC_ENTRY       *rcv_caller;
LOCAL  PROC_ENTRY       *powner;
LOCAL  PORT_ENTRY       *src_port;
LOCAL  PORT_ENTRY       *dst_port;
LOCAL  MSG              *msg;

        /* Get parameter block */
        getp_blk(KC_NBSMSG);

        /* Allocate space for MSG */
        msg = (MSG *)malloc(sizeof(MSG));

        /* Establish caller */
        caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

        /* Establish source port */
        if ((src_port = caller->owned_ports[p_blk.sport]) == (PORT_ENTRY *)EMPTY)
        {
             kcfail (UN_SPORT);
             free ((char *)msg);
             return FAIL;
        }

        /* Establish destination port */
        if ((dst_port = src_port->links_to[p_blk.dport].port) == (PORT_ENTRY
*)EMPTY)
        {
             kcfail (UN_DPORT);
             free ((char *)msg);
             return FAIL;
        }

        /* Check msg_type of dst port is correct */
        if (dst_port -> msg_type != p_blk.msg_type)
        {
            kcfail (UN_MTYPE);
            free ((char *)msg);
            return FAIL;
        }

        else      /* Return success to caller */
        {
            KCR_HDR kcr_hdr = KCSUCC_MASK;
```

```
     src_port -> links_to[p_blk.dport].nmsgs ++;
     src_port -> links_to[p_blk.dport].tot_msglength += p_blk.msg_length;
     kcsucc(kcr_hdr, TIME_NBS_SETUP);
}

/* Set up rest of MSG */
msg -> msg_hdr.msg_length = p_blk.msg_length;
msg -> msg_hdr.snd_type = NON_BLOCKING;
msg -> msg_hdr.msg_type = p_blk.msg_type;
strcpy (msg->msg_hdr.dst_port, dst_port->port_name);
strcpy (msg->msg_hdr.src_port, src_port->port_name);
strcpy (msg->msg_txt, p_blk.msg_txt);
msg -> next = (MSG *)EMPTY;

/* Check if dst port is local or remote */
if (dst_port -> residency == this_mc)
{
      /* See if there is an nb_rcv pending */
      if (dst_port -> profile.nb_pending)
      {
           dst_port -> profile.nb_pending = !PENDING;

           /* Give the msg to the waiter!(sic!) */
           give_msg (dst_port->nb_msg_loc, msg, dst_port->rcvfunc);
           free ((char *)msg);
           rcv_caller = find_proc(dst_port->owner_proc);
           got_msgsigack = 0;
           kill (rcv_caller->upid, SIGMSG);

           /* Wait for acknowledgement */
           while (!got_msgsigack)
           {
                    alarm(3);
                    pause();
                    alarm(0);
           }

           /* Update time */
           time_update (TIME_NBR_WAITER, USERtime, rcv_caller);

       }

       else
           /* See if there is a blocking rcv pending */
           if (dst_port -> profile.b_pending)
           {
                KCR_BRMSG  kcr;

                /* Unset b_pending */
                dst_port -> profile.b_pending = !PENDING;

                /* Establish rcv caller */
                rcv_caller = find_proc (dst_port -> owner_proc);

                /* Mark process as unblocked */
                rcv_caller -> blocked = UNBLOCKED;

                /* Increment active proc count */
                n_active_local_procs ++;
```

```
              /* Do a kcreturn for blocked rcv'er */
              kcr.hdr = KCSUCC_MASK;
              kcr.msg.msg_hdr.msg_length = msg -> msg_hdr.msg_length;
              kcr.msg.msg_hdr.msg_type = msg -> msg_hdr.msg_type;
              kcr.msg.msg_hdr.snd_type = msg -> msg_hdr.snd_type;
              strcpy (kcr.msg.msg_hdr.dst_port, msg->msg_hdr.dst_port);
              strcpy (kcr.msg.msg_hdr.src_port, msg->msg_hdr.src_port);
              strcpy (kcr.msg.msg_txt, msg->msg_txt);

              /* Send msg to blocked rcv'er */
              got_ackkcrsig = 0;
              pwrite (kcret_pipe, (char *)&kcr, sizeof(kcr));
              free ((char *)msg);
              kill (rcv_caller->upid, SIGKCR);
              while (!got_ackkcrsig)
              {
                 alarm(3);
                 pause();
                 alarm(0);
              }

              time_update (TIME_BR_WAITER, USERtime, rcv_caller);

            }

        else   /* No waiters so append to msg_q */
        {
            if (dst_port -> profile.inmq_length == 0)
                dst_port -> msg_q_head = msg;
            else
               (dst_port->msg_q_tail) -> next = msg;

            dst_port -> msg_q_tail = msg;
            dst_port -> profile.inmq_length ++;

            /* Increment owner's migration size */
            powner = find_proc (dst_port -> owner_proc);
            powner -> migration_siz += sizeof(MSG);

            /* Increment owner's size */
            powner -> tot_msg_siz += msg->msg_hdr.msg_length;

            /* Update time */
            time_update (TIME_APP_MSGQ, USERtime, caller);

         }

    }

    else  /* dst port is remote */
    {  COMMS_HDR  comms_hdr = USR_MSG | EMSG_MASK;
       EM_HDR     em_hdr;

       /* Set up em_hdr */
       em_hdr.sending_mc = this_mc;
       em_hdr.dst_mc = dst_port -> residency;
       procncpy(em_hdr.caller, caller->proc_name);
```

```
            /* Send MSG */
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)msg, sizeof(MSG),
USERtime, caller);
            free ((char *)msg);

            /* Update time */
            time_update (TIME_REMOTE, USERtime, caller);

        }

#ifdef DEBUG
dump_procs();
dump_ports();
#endif

        return SUCCESS;

} /* End of NB_SMSG */




/***** USR_MSG *****/

/* This routine deals with the arrival of an external usr msg */

usr_msg()

{

LOCAL  EM_HDR       em_hdr;
LOCAL  EM_UMSG      msg;
LOCAL  PORT_ENTRY   *dst_port;
LOCAL  PROC_ENTRY   *rcv_caller;
LOCAL  PROC_ENTRY   *powner;
LOCAL  COMMS_HDR    comms_hdr;

        /* Allocate space for MSG */
        msg.umsg = (MSG *)malloc(sizeof(MSG));
        if (msg.umsg == (MSG *)EMPTY) fprintf(trace, "Malloc failed in umsg\n");

        if (check_forward ((char *)&em_hdr, (char *)(msg.umsg), sizeof(MSG),
USR_MSG) == FORWARDED)
        {
            free ((char *)(msg.umsg));
            return FORWARDED;
        }

        /* Establish destination port */
        dst_port = find_port (em_hdr.caller, msg.umsg->msg_hdr.dst_port);
        if (dst_port == (PORT_ENTRY *)EMPTY)fprintf(trace,"BUG 1\n");

        /* Check if migrated */
        if (dst_port -> residency != this_mc)
        {
            comms_hdr = USR_MSG | EMSG_MASK;
            em_hdr.dst_mc = dst_port -> residency;
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)(msg.umsg),
sizeof(MSG), OStime, (PROC_ENTRY *)EMPTY);
            free ((char *)(msg.umsg));
            return FORWARDED;
        }

        /* See if there is an nb_rcv pending on dst port */
        if (dst_port -> profile.nb_pending)
        {
            dst_port -> profile.nb_pending = !PENDING;

            /* Give the msg to the waiter */
            give_msg (dst_port->nb_msg_loc, msg.umsg, dst_port->rcvfunc);
            free ((char*)(msg.umsg));
            rcv_caller = find_proc (dst_port->owner_proc);
            got_msgsigack = 0;
            kill (rcv_caller->upid, SIGMSG);

            /* Wait for acknowledgement */
            while (!got_msgsigack)
            {
                    alarm(3);
                    pause();
                    alarm(0);
            }
            /* Check if msg rcv'd was sent blocking */
            if (dst_port -> msg_q_head -> msg_hdr.snd_type == BLOCKING)
            {
                PROC_ENTRY *save_caller = caller;
                src_port = find_port (caller->proc_name,
dst_port->msg_q_head->msg_hdr.src_port);


                /* Caller is now msg sender so do a kcreturn for him */
                if (src_port -> residency == this_mc)
                {
                    caller = find_proc (src_port -> owner_proc);
                    KCR_HDR kcr_hdr = KCSUCC_MASK;
                    kcsucc(kcr_hdr, 0);
                }
                else
                {
                    COMMS_HDR comms_hdr = NOT_MSG | EMSG_MASK;
                    EM_HDR     em_hdr;
                    EM_NOT_MSG  notify_msg;

                    em_hdr.sending_mc = this_mc;
                    em_hdr.dst_mc = src_port -> residency;
                    procncpy (em_hdr.caller, caller -> proc_name);
                    strcpy (notify_msg.src_port_name, src_port -> port_name);

                    TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&notify_msg),
                        sizeof (NOT_MSG), OStime, (PROC_ENTRY *)EMPTY);

                }
                 caller = save_caller;
            }
           /* Update time */
           time_update (TIME_NBR_WAITER, USERtime, rcv_caller);
```

```
}

else
    /* See if there is a blocking rcv pending */
    if (dst_port -> profile.b_pending)
    {
        KCR_BRMSG  kcr;

        /* Unset b_pending */
        dst_port -> profile.b_pending = !PENDING;

        /* Establish original rcv'er */
        rcv_caller = find_proc (dst_port -> owner_proc);

        /* Mark process as unblocked */
        rcv_caller -> blocked = UNBLOCKED;

        /* Increment active proc count */
        n_active_local_procs ++;

        /* Do a kcreturn for blocked rcv'er */
        kcr.hdr = KCSUCC_MASK;
        kcr.msg.msg_hdr.msg_length = msg.umsg->msg_hdr.msg_length;
        kcr.msg.msg_hdr.msg_type = msg.umsg->msg_hdr.msg_type;
        kcr.msg.msg_hdr.snd_type = msg.umsg->msg_hdr.snd_type;
        strcpy (kcr.msg.msg_hdr.dst_port, msg.umsg->msg_hdr.dst_port);
        strcpy (kcr.msg.msg_hdr.src_port, msg.umsg->msg_hdr.src_port);
        strcpy (kcr.msg.msg_txt, msg.umsg->msg_txt);
        free ((char *)(msg.umsg));

        /* Send msg to blocked rcv'er */
        got_ackkcrsig = 0;
        kill (rcv_caller->upid, SIGKCR);
        pwrite (kcret_pipe, (char *)&kcr, sizeof(kcr));
        while (!got_ackkcrsig)
        {
               alarm(3);
               pause();
               alarm(0);
        }

        /* Update time */
        time_update (TIME_BR_WAITER, USERtime, rcv_caller);

        /* Check if msg rcv'd was sent blocking */
    if (dst_port -> msg_q_head -> msg_hdr.snd_type == BLOCKING)
    {
        PRCC_ENTRY *save_caller = caller;
        src_port = find_port (caller->proc_name,
                              dst_port->msg_q_head->msg_hdr.src_port);



        /* Caller is now msg sender so do a kcreturn for him */
        if (src_port -> residency == this_mc)
        {
            caller = find_proc (src_port -> owner_proc);
            KCR_HDR kcr_hdr = KCSUCC_MASK;
```

```
                    kcsucc(kcr_hdr, 0);
                }
                else
                {
                    COMMS_HDR comms_hdr = NOT_MSG | EMSG_MASK;
                    EM_HDR    em_hdr;
                    EM_NOT_MSG  notify_msg;

                    em_hdr.sending_mc = this_mc;
                    em_hdr.dst_mc = src_port -> residency;
                    procncpy (em_hdr.caller, caller -> proc_name);
                    strcpy (notify_msg.src_port_name, src_port -> port_name);

                    TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&notify_msg),
                        sizeof (NOT_MSG), OStime, (PROC_ENTRY *)EMPTY);

                }
                 caller = save_caller;
             }
        }

        else   /* No waiters so append to msg_q */
        {
             if (dst_port -> profile.inmq_length == 0)
                 dst_port -> msg_q_head = msg.umsg;
             else
                 dst_port->msg_q_tail->next = msg.umsg;

             dst_port -> msg_q_tail = msg.umsg;
             dst_port -> profile.inmq_length ++;

             /* Increment owner's migration size */
             powner = find_proc (dst_port -> owner_proc);
             powner -> migration_siz += sizeof(MSG);

             /* Increment process's size */
             powner -> tot_msg_siz += msg.umsg->msg_hdr.msg_length;

             /* Update time */
             time_update (TIME_APP_MSGQ, OStime, (PROC_ENTRY *)EMPTY);

        }

#ifdef DEBUG
dump_procs();
dump_ports();
#endif

        return SUCCESS;

} /* End of USR_MSG */
```

```
/* FILE: project/src/kernel/b_smsg.c */

/* This file contains routines to blocking send a msg */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_BS_SETUP              5 * AVE_INST
# define  TIME_NBR_WAITER           50 * AVE_INST
# define  TIME_BR_WAITER            40 * AVE_INST
# define  TIME_APP_MSGQ             20 * AVE_INST
# define  TIME_REMOTE               50 * AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY *find_proc();
EXTERN  PORT_ENTRY *find_port();
EXTERN  char       *malloc();

/* Typedefs used in these routines */

typedef  struct {
                  int   sport;
                  int   dport;
                  int   msg_length;
                  int   msg_type;
                  char  msg_txt[MSG_SIZ];

                } KC_BSMSG;

typedef  struct {
                  KCR_HDR  hdr;
                  MSG      msg;

                } KCR_BRMSG;

typedef  struct {
                  MSG  *umsg;

                } EM_UMSG;

typedef  struct {
                  char  src_port_name;

                }  EM_NOT_MSG;



/***** NB_SMSG *****/
```

```
/* This routine deals with a kcall made to blocking snd a msg */

b_smsg()

{

LOCAL  KC_HDR          kc_hdr;
LOCAL  KC_BSMSG        p_blk;
LOCAL  PROC_ENTRY      *caller;
LOCAL  PROC_ENTRY      *rcv_caller;
LOCAL  PROC_ENTRY      *powner;
LOCAL  PORT_ENTRY      *src_port;
LOCAL  PORT_ENTRY      *dst_port;
LOCAL  MSG             *msg;

        /* Get parameter block */
        getp_blk(KC_BSMSG);

        /* Allocate space for MSG */
        msg = (MSG *)malloc(sizeof(MSG));

        /* Establish caller */
        caller = find_proc (kc_hdr);

        /* Update time due to context switch */
        contxt_swtch;

        /* Establish source port */
        if ((src_port = caller->owned_ports[p_blk.sport]) == (PORT_ENTRY *)EMPTY)
        {
             kcfail (UN_SPORT);
             free ((char *)msg);
             return FAIL;
        }

        /* Establish destination port */
        if ((dst_port = src_port->links_to[p_blk.dport].port) == (PORT_ENTRY
*)EMPTY)
        {
             kcfail (UN_DPORT);
             free ((char *)msg);
             return FAIL;
        }

        /* Check msg_type of dst port is correct */
        if (dst_port -> msg_type != p_blk.msg_type)
        {
            kcfail (UN_MTYPE);
            free ((char *)msg);
            return FAIL;
        }



        /* Set up rest of MSG */
        msg -> msg_hdr.msg_length = p_blk.msg_length;
        msg -> msg_hdr.snd_type = BLOCKING;
        msg -> msg_hdr.msg_type = p_blk.msg_type;
        strcpy (msg->msg_hdr.dst_port, dst_port->port_name);
        strcpy (msg->msg_hdr.src_port, src_port->port_name);
        strcpy (msg->msg_txt, p_blk.msg_txt);
        msg -> next = (MSG *)EMPTY;

        /* Check if dst port is local or remote */
        if (dst_port -> residency == this_mc)
        {
            /* See if there is an nb_rcv pending */
            if (dst_port -> profile.nb_pending)
            {
                dst_port -> profile.nb_pending = !PENDING;

                /* Give the msg to the waiter!(sic!) */
                give_msg (dst_port->nb_msg_loc, msg, dst_port->rcvfunc);
                free ((char *)msg);
                rcv_caller = find_proc(dst_port->owner_proc);
                got_msgsigack = 0;
                kill (rcv_caller->upid, SIGMSG);

                /* Wait for acknowledgement */
                while (!got_msgsigack)
                {
                        alarm(3);
                        pause();
                        alarm(0);
                }

                /* Update time */
                time_update (TIME_NBR_WAITER, USERtime, rcv_caller);

                KCR_HDR kcr_hdr = KCSUCC_MASK;
                src_port -> links_to[p_blk.dport].nmsgs ++;
                src_port -> links_to[p_blk.dport].tot_msglength +=
p_blk.msg_length;
                kcsucc(kcr_hdr, TIME_BS_SETUP);

            }

            else
                /* See if there is a blocking rcv pending */
                if (dst_port -> profile.b_pending)
                {
                    KCR_BRMSG  kcr;

                    /* Unset b_pending */
                    dst_port -> profile.b_pending = !PENDING;

                    /* Establish rcv caller */
                    rcv_caller = find_proc (dst_port -> owner_proc);

                    /* Mark process as unblocked */
                    rcv_caller -> blocked = UNBLOCKED;

                    /* Increment active proc count */
                    n_active_local_procs ++;

                    /* Do a kcreturn for blocked rcv'er */
```

```
                  kcr.hdr = KCSUCC_MASK;
                  kcr.msg.msg_hdr.msg_length = msg -> msg_hdr.msg_length;
                  kcr.msg.msg_hdr.msg_type = msg -> msg_hdr.msg_type;
                  kcr.msg.msg_hdr.snd_type = msg -> msg_hdr.snd_type;
                  strcpy (kcr.msg.msg_hdr.dst_port, msg->msg_hdr.dst_port);
                  strcpy (kcr.msg.msg_hdr.src_port, msg->msg_hdr.src_port);
                  strcpy (kcr.msg.msg_txt, msg->msg_txt);

                  /* Send msg to blocked rcv'er */
                  got_ackkcrsig = 0;
                  pwrite (kcret_pipe, (char *)&kcr, sizeof(kcr));
                  free ((char *)msg);
                  kill (rcv_caller->upid, SIGKCR);
                  while (!got_ackkcrsig)
                  {
                     alarm(3);
                     pause();
                     alarm(0);
                  }

                  time_update (TIME_BR_WAITER, USERtime, rcv_caller);

                  KCR_HDR kcr_hdr = KCSUCC_MASK;
                  src_port -> links_to[p_blk.dport].nmsgs ++;
                  src_port -> links_to[p_blk.dport].tot_msglength +=
p_blk.msg_length;
                  kcsucc(kcr_hdr, TIME_BS_SETUP);

               }

            else   /* No waiters so append to msg_q */
            {
                if (dst_port -> profile.inmq_length == 0)
                    dst_port -> msg_q_head = msg;
                else
                   (dst_port->msg_q_tail) -> next = msg;

                dst_port -> msg_q_tail = msg;
                dst_port -> profile.inmq_length ++;

                /* Increment owner's migration size */
                powner = find_proc (dst_port -> owner_proc);
                powner -> migration_siz += sizeof(MSG);

                /* Increment owner's size */
                powner -> tot_msg_siz += msg->msg_hdr.msg_length;

                /* Update time */
                time_update (TIME_APP_MSGQ, USERtime, caller);

             }

        }

        else  /* dst port is remote */
        {  COMMS_HDR  comms_hdr = USR_MSG | EMSG_MASK;
           EM_HDR      em_hdr;

           /* Set up em_hdr */
```

```
            em_hdr.sending_mc = this_mc;
            em_hdr.dst_mc = dst_port -> residency;
            procncpy(em_hdr.caller, caller->proc_name);

            /* Send MSG */
            TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)msg, sizeof(MSG),
USERtime, caller);
            free ((char *)msg);

            /* Update time */
            time_update (TIME_REMOTE, USERtime, caller);

         }

#ifdef DEBUG
dump_procs();
dump_ports();
#endif

         return SUCCESS;

} /* End of B_SMSG */
```

```
/* FILE : project/src/kernel/port_loc.c */

/* This file contains routines to UPDATE PORT LOCATION */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Routines EXTERNal to this file */

EXTERN  PORT_ENTRY  *find_port();

/* Typedefs used in these routines */

typedef  struct {
                    int      residency;
                    int      n_entries;
                    PROCN    owner;
                    char     port_name [MAXOWNPRT][MAXPNAME];

                  } EM_PORT_LOC;




/***** PORT_LOC_MSG *****/

/* This routine deals with a msg to update a port location */

port_loc_msg ()

{

LOCAL  EM_HDR             em_hdr;
LOCAL  EM_PORT_LOC        msg;
LOCAL  int                i;
LOCAL  PORT_ENTRY         *port;

        /* Check if needs to be forwarded */
        if (check_forward ((char *)&em_hdr, (char *)&msg, sizeof(EM_PORT_LOC),
PORT_LOC_MSG) == FORWARDED)
            return  FORWARDED;

        /* Update moved port entries */
        if (msg.residency != this_mc)
        for (i=0; i<msg.n_entries; i++)
        {
            port = find_port(msg.owner, msg.port_name[i]);
            if (port != (PORT_ENTRY *)EMPTY)
                if (port -> residency != this_mc)
                port -> residency = msg.residency;
        }

        return SUCCESS;

} /* End of PORT_LOC_MSG */
```

```
/* FILE: project/src/kernel/pinfo.c */

/* This file contains routines to RECEIVE A MIGRATING PROCESS */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Time defines */
# define  TIME_RCV_PROC  200*AVE_INST

/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY  *find_proc();
EXTERN  PORT_ENTRY  *find_port();
EXTERN  char        *malloc();

/* Typedefs used in these routines */

typedef  int  EM_PINFO;

/***** PINFO_MSG *****/

/* This routine rcvs a migrating process */

pinfo_msg()

{

LOCAL  EM_HDR        em_hdr;
LOCAL  EM_PINFO      msg;
LOCAL  PROC_ENTRY    mig_proc;
LOCAL  int           op;
LOCAL  int           op_index;
LOCAL  PORT_ENTRY    mig_port;
LOCAL  int           lnk;
LOCAL  int           lnk_index;
LOCAL  char          pname[MAXPNAME];
LOCAL  int           m;
LOCAL  PROC_ENTRY    *this_proc;
LOCAL  int           tot_siz;
LOCAL  int           siz;
LOCAL  int           distance;

        if (check_forward ((char *)&em_hdr, (char *)&msg, sizeof(EM_PINFO),
PINFO_MSG) == FORWARDED)
                return FORWARDED;

        /* Cancel first pending awaiting process timeout */
        if (awaiting_process.set)
        {
/*          fprintf (trace, "Proc arrived - waited for\n"); */
            remove_await_timeout();
          /*   fprintf (trace, "Virtual load %d\n", n_virtual_procs); */
        }

        /* Get the actual size of the process */
        RX((char *)&tot_siz, sizeof(int));

        /* Get the process table entry */
        RX ((char *)&mig_proc, sizeof(PROC_ENTRY));

        /* Get the list (possibly empty) of owned ports */
        for (op=0; cp < mig_proc.ownp_length; op++)
        {
             /* Get the port's index and port table entry */
             RX ((char *)&op_index, sizeof(int));
             RX ((char *)&mig_port, sizeof(PORT_ENTRY));

             /* Get the (possibly empty) list of link_to ports and their indexes
*/
             for (lnk=0; lnk < mig_port.profile.lnkt_length; lnk++)
             {
                  RX ((char *)&lnk_index, sizeof(int));
                  RX (pname, MAXPNAME);
                  mig_port.links_to[lnk_index].port = find_port
(mig_proc.proc_name, pname);
             }

             /* Get the (possibly empty) list of link_from ports and their indexes
*/
             for (lnk=0; lnk < mig_port.profile.lnkf_length; lnk++)
             {
                  RX ((char *)&lnk_index, sizeof(int));
                  RX (pname, MAXPNAME);
                  mig_port.links_from[lnk_index] = find_port (mig_proc.proc_name,
pname);
             }

             /* Get the (poss empty) msg_q */
             mig_port.msg_q_head = mig_port.msg_q_tail = (MSG *)EMPTY;
             for (m=0; m < mig_port.profile.inmq_length; m++)
                  if (mig_port.msg_q_head == (MSG *)EMPTY)
                  {
                      mig_port.msg_q_head = (MSG *)malloc(sizeof(MSG));
                      RX ((char *)(mig_port.msg_q_head), sizeof(MSG));
                      mig_port.msg_q_tail = mig_port.msg_q_head;
                      mig_port.msg_q_tail -> next = (MSG *)EMPTY;
                  }
                  else
                  {
                      mig_port.msg_q_tail = mig_port.msg_q_tail -> next = (MSG
*)malloc(sizeof(MSG));
                      RX ((char *)(mig_port.msg_q_tail), sizeof(MSG));
                      mig_port.msg_q_tail -> next = (MSG *)EMPTY;
                  }
            /* Put pointer to mig_port in mig_proc's owned ports list */
            mig_proc.owned_ports[op_index] = find_port (mig_proc.proc_name,
mig_port.port_name);

            /* Enter mig_port info in port table */
            memcpy ((char *)(mig_proc.owned_ports[op_index]), (char *)&mig_port,
sizeof(PORT_ENTRY));


        }


        /* Update time for comms.*/
        siz = msg + tot_siz;
        time_update (siz*RX_BYTE_TIME, OStime, (PROC_ENTRY *)EMPTY);
/*      fprintf (trace, "Pinfo1: time %d\n", siz*RX_BYTE_TIME); */

        /* Enter mig_proc info in process table */
        memcpy ((char *)nxt_proc, (char *)&mig_proc, sizeof(PROC_ENTRY));
        /* fprintf (trace, "Pinfo: rcved %d\n", nxt_proc->upid); */

        /* Set times for process N.B. allow for migration time in exist_time */
        nxt_proc->times.exec_here_time = 0;
        nxt_proc->times.residency_time = 0;
        distance = route_table[em_hdr.sending_mc].distance;
        /* fprintf (trace, "Dist to %d is %d\n", em_hdr.sending_mc, distance);*/
        nxt_proc -> times.exist_time += distance*(2*PROTOCOL_TIME*AVE_INST +
                                                  siz*RX_BYTE_TIME +
                                                  siz*TX_BYTE_TIME);
        /* fprintf (trace, "Pinfo2: time %d\n", distance
*(2*PROTOCOL_TIME*AVE_INST + siz*RX_BYTE_TIME + siz*TX_BYTE_TIME)); */

        /* Update time */
        time_update (TIME_RCV_PROC, OStime, (PROC_ENTRY *)EMPTY);

        /* Increment local proc count */
        n_local_procs ++;

        /* If process is active increment active proc count */
        if (! nxt_proc -> blocked)
            n_active_local_procs ++;

        /* Continue suspended process */
        kill (nxt_proc->upid, SIGCONT);

        /* Update next process pointer */
        while (nxt_proc -> residency != EMPTY)
               nxt_proc++;

#ifdef DEBUG
dump_ports();
dump_procs();
#endif


        return SUCCESS;

}
```

```
/* FILE : /usr/acct/ian/project/src/kernel/negociate.c */

/* This file contains routines to NEGOCIATE process migration */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Routines EXTERNal to this file */

EXTERN  PROC_ENTRY *choose_proc_to_migrate();

/* Typedefs for these routines */

typedef  struct {
                   unsigned short  negociation_type;
                   unsigned short  forward_it;

                } EM_NEGOCIATION;

typedef  struct {
                   float  new_average;

                } EM_CHANGE_AVERAGE;



/***** NEGOCIATE_MSG *****/

/* This routine deals with receipt of a TOO_HIGH or ACCEPT msg */

negociate_msg()

{

LOCAL  EM_HDR                em_hdr;
LOCAL  COMMS_HDR             comms_hdr;
LOCAL  EM_NEGOCIATION        msg;
LOCAL  PROC_ENTRY            *mig_proc;
LOCAL  int                   i;
EXTERN PROC_ENTRY            *choose_proc_to_migrate();

        /* Check if needs to be forwarded */
        if (check_forward ((char *)&em_hdr, (char *)&msg, sizeof(EM_NEGOCIATION),
NEGOCIATE_MSG) == FORWARDED)
            return FORWARDED;

        switch (msg.negociation_type)
        {
```

```
        case TOO_HIGH :   if (UNDERLOADED)
                              {
                                /* Cancel too low timeout */
                                too_low.set = FALSE;

                                /* Set awaiting process timeout */
                                add_await_timeout();

                                /* Send accept msg */
                                msg.negociation_type = ACCEPT;
                                comms_hdr = NEGOCIATE_MSG | EMSG_MASK;
                                em_hdr.dst_mc = em_hdr.sending_mc;
                                em_hdr.sending_mc = this_mc;
                                TX ((char *)&comms_hdr, (char *)&em_hdr, (char
*)&msg, sizeof(EM_NEGOCIATION), OStime, (PROC_ENTRY *)EMPTY);

                              }
                              else
                              {
                                 if (msg.forward_it)
                                 {
                                     msg.forward_it --;
                                     for (i=0; i<n_nbours; i++)
                                         if (neighbours[i]!=em_hdr.sending_mc)
                                         {
                                             comms_hdr = NEGOCIATE_MSG | EMSG_MASK;
                                             em_hdr.dst_mc = neighbours[i];
                                             TX((char *)&comms_hdr, (char
*)&em_hdr, (char *)&msg, sizeof(EM_NEGOCIATION), OStime, (PROC_ENTRY *)EMPTY);
                                          }
                                  }
                              }
                              break;

        case ACCEPT   :       /* Cancel too high timeout */
                              too_high.set = FALSE;

                              /* Migrate process if still overloaded */
                              if (OVERLOADED && n_active_local_procs > N_SYS_PROCS
+ 1)
                              {
                                  if ((mig_proc = choose_proc_to_migrate())!=
(PROC_ENTRY *)EMPTY)
                                  migrate (mig_proc, em_hdr.sending_mc);
                              }
                              break;

        }

        return SUCCESS;

} /* End of NEGOCIATE_MSG */
```

```
/***** CHANGE_AVE_MSG *****/

/* This routine modifies the system wide average load value */

change_ave_msg ()

{

LOCAL  EM_HDR                    em_hdr;
LOCAL  EM_CHANGE_AVERAGE         msg;

        /* Check if needs to be forwarded */
        if (check_forward ((char *)&em_hdr, (char *)&msg,
sizeof(EM_CHANGE_AVERAGE), CH_AVE_MSG) == FORWARDED)
            return FORWARDED;

        /* Update average */
        global_average_load = msg.new_average;

        /* Cancel timeouts */
        too_low.set = FALSE;
        too_high.set = FALSE;

        return SUCCESS;

} /* End of CHANGE_AVE_MSG */



/***** ADD_AWAIT_TIMEOUT *****/

/* This routine adds an await process to the queue */

OWN  int  await_length = 0;

add_await_timeout()

{

        if (await_length == 0) awaiting_process.set = TRUE;

        /* Add a time to the queue of timeouts */
        awaiting_process.timer[await_length++] = sys_real_time + TIMEOUT_INTERVAL;

        /* Increase virtual load */
        n_virtual_procs ++;

} /* End of ADD_AWAIT_TIMEOUT */



/***** REMOVE_AWAIT_TIMEOUT *****/

/* This routine removes an await process timeout from the queue */

remove_await_timeout()
```

```
{

LOCAL  int  i;

        for (i=1; i<await_length; i++)
             awaiting_process.timer[i-1] = awaiting_process.timer[i];

        await_length --;
        if (await_length == 0) awaiting_process.set = FALSE;

        /* Decrease virtual load */
        n_virtual_procs --;

} /* End of REMOVE_AWAIT_TIMEOUT */
```

```
/* FILE: /usr/acct/ian/project/src/kernel/probe.c */

/* This file contains routines to DEAL WITH PROBING */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  <math.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/globvars.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Routines EXTERNal to this file */
EXTERN  double  erand48();

/* Typedefs for these routines */

typedef  struct {
                  PROC_ENTRY  *proc;

                } EM_PROBE;

typedef  struct {
                  PROC_ENTRY  *proc;
                  unsigned short  above_threshold;

                } EM_REPLY_PROBE;


/***** PROBE_MSG *****/

/* This routine deals with receipt of a probe msg */

probe_msg()

{

LOCAL  COMMS_HDR           comms_hdr;
LOCAL  EM_HDR              em_hdr;
LOCAL  EM_PROBE            msg;
LOCAL  EM_REPLY_PROBE      reply;

        /* Check to see if needs to be forwarded */
        if (check_forward ((char *)&em_hdr, (char *)&msg, sizeof(EM_PROBE),
PROBE_MSG) == FORWARDED)
            return FORWARDED;

        /* Test whether would be above threshold if process comes here */
        if (n_local_procs + 1 > N_SYS_PROCS + THRESHOLD)
            reply.above_threshold = TRUE;
        else
            reply.above_threshold = FALSE;
        /* Send reply */
        comms_hdr = REPLY_PROBE_MSG | EMSG_MASK;
        em_hdr.dst_mc = em_hdr.sending_mc;
        em_hdr.sending_mc = this_mc;
        reply.proc = msg.proc;
        TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&reply,
sizeof(EM_REPLY_PROBE), OStime, (PROC_ENTRY *)EMPTY);

        return SUCCESS;

} /* End of PROBE_MSG */




/***** PROBE_REPLY_MSG *****/

/* This routine deals with a reply to an earlier probe */

probe_reply_msg()

{

LOCAL  EM_HDR          em_hdr;
LOCAL  EM_REPLY_PROBE  msg;
LOCAL  int             i;

        /* Check if needs to be forwarded */
        if (check_forward ((char *)&em_hdr, (char *)&msg, sizeof(EM_REPLY_PROBE),
REPLY_PROBE_MSG) == FORWARDED)
            return FORWARDED;

        /* If probed mc is above threshold send another probe */
        /* otherwise migrate process */
        if (msg.above_threshold == TRUE)
            send_probe (msg.proc);
        else
           {
             for (i=0; i<msg.proc->n_probes; i++)
                  (msg.proc->mcs_probed)[i] = EMPTY;
             msg.proc -> n_probes = 0;
             msg.proc -> schedulable = TRUE;
             migrate (msg.proc, em_hdr.sending_mc);
           }

        return SUCCESS;

} /* End of PROBE_REPLY_MSG */




/***** SEND_PROBE *****/
```

```
send_probe(p)

PARAMS  PROC_ENTRY  *p;

{

LOCAL  COMMS_HDR   comms_hdr;
LOCAL  EM_HDR      em_hdr;
LOCAL  int         mc;
LOCAL  EM_PROBE    msg;
LOCAL  double      rnd_num;
EXTERN double      erand48();

        /* Check if probe limit exceeded */
        if (p -> n_probes < PROBE_LIMIT)
        {

           /* Generate random mc no. */
           do {
                rnd_num = erand48(xsubi) * nmcs;
                mc = (int)rnd_num;
              } while ((probed (p, mc)) || (mc == this_mc));

           /* Update probe info */
           p -> mcs_probed[p->n_probes] = mc;
           p -> n_probes ++;

           /* Send probe */
           comms_hdr = PROBE_MSG | EMSG_MASK;
           em_hdr.dst_mc = mc;
           em_hdr.sending_mc = this_mc;
           msg.proc = p;
           TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&msg,
sizeof(EM_PROBE), CStime, (PROC_ENTRY *)EMPTY);

        }

        else  /* Probe limit has been exceeded - must process locally */
        {
             n_active_local_procs ++;
             p -> schedulable = TRUE;
        }

} /* End of SEND_PROBE */




/***** PROBED *****/

probed (p, mc)

PARAMS   PROC_ENTRY  *p;
         int         mc;

{
```

```
LOCAL  int  i;

        /* Check to see if mc has already been probed */
        if (p -> n_probes > 0)
        {
            i = 0;
            while (i <= PROBE_LIMIT - 1)
                   if (p -> mcs_probed[i++] == mc)
                       return 1;
        }

        return 0;

 } /* End of PROBED */
```

```
/* FILE: project/src/kernel/utils.c */

/* This file contains routines commonly used by kernel routines */

/* Includes for this file */
# include <stdio.h>
# include <signal.h>
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/shm.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"
# include "/usr/acct/ian/project/src/include/sys/globvars.h"
# include "/usr/acct/ian/project/src/include/sys/macros.h"


/* Routines EXTERNal to this file */

EXTERN  char  *malloc();



/***** CHECK_FORWARD *****/

/* This routine checks if an emsg needs to be forwarded */

check_forward (em_hdr, msg, msize, em_type)

PARAMS  char *em_hdr;
        char *msg;
        int   msize;
        int   em_type;

{

LOCAL  char *var_msg;

        /* Receive ext. msg. hdr */
        RX (em_hdr, sizeof(EM_HDR));

        /* Receive ext. msg. itself */
        RX (msg, msize);

        /* Update elapsed time */
        if (em_type != EXIT_MSG)
        time_update (PROTOCOL_TIME * AVE_INST +
                     (sizeof(COMMS_HDR)+sizeof(EM_HDR)+msize) * RX_BYTE_TIME,
                     OStime,
                     (PROC_ENTRY *)EMPTY);

        /* If msg not for this mc then forward it */
        if (((EM_HDR *)em_hdr) -> dst_mc != this_mc)
        {
             /* Set up comms hdr */
```

```
            COMMS_HDR  comms_hdr = em_type | EMSG_MASK;

            /* Test if var length msg */
            if (em_type == PINFO_MSG)
            {
                var_msg = malloc (*((int *)msg)+2*sizeof(int));
                memcpy (var_msg, msg, sizeof(int));
                RX (var_msg+sizeof(int), sizeof(int));
                RX (var_msg+2*sizeof(int), *((int *)msg));
                time_update ((*((int *)msg)+*((int *)(var_msg+2*sizeof(int)))) *
RX_BYTE_TIME, OStime, (PROC_ENTRY *)EMPTY);
                TX ((char *)&comms_hdr, em_hdr, var_msg, *((int
*)msg)+2*sizeof(int), OStime, (PROC_ENTRY *)EMPTY);
                free(var_msg);
            }

            else

            {
            if (em_type == USR_MSG)
                time_update ((((MSG *)msg)->msg_hdr.msg_length)*RX_BYTE_TIME,
OStime, (PROC_ENTRY *)EMPTY);


            /* Transmit, i.e. forward, msg */
            TX ((char *)&comms_hdr, em_hdr, msg, msize, OStime, (PROC_ENTRY
*)EMPTY);

            }


            return FORWARDED;
        }

        else return !FORWARDED;

} /* End of CHECK_FORWARD */








/***** KCRETURN *****/

/* This routine returns the result of a kernel call to the caller */

kcreturn (upid, result, rsize)

PARAMS  int  upid;
        char *result;
        int  rsize;

{

        got_ackkcrsig = 0;
        /* Wake up caller */
        kill (upid, SIGKCR);

        /* Send back results */
        pwrite (kcret_pipe, result, rsize);

        while (!got_ackkcrsig)
        {
           alarm(3);
           pause();
           alarm(0);
        }

} /* End of KCRETURN */




/***** RX *****/

/* This routine receives a comms. msg */

RX (buffer, n_bytes)

PARAMS  char  *buffer;
        int   n_bytes;

{

        pread (buffer, n_bytes, own_pipe);

} /* End of RX */





/***** TX *****/

/* This routine transmits an ext. msg */

TX (comms_hdr, em_hdr, msg, msize, time_type, calling_proc)

PARAMS  char  *comms_hdr;
        char  *em_hdr;
        char  *msg;
        int   msize;
        int   time_type;
        PROC_ENTRY *calling_proc;

{
```

```
LOCAL  int  mclink = phys_link[route_table[((EM_HDR *)em_hdr)->dst_mc].lnk].link;
LOCAL  int  nbour_mc = phys_link[route_table[((EM_HDR
*)em_hdr)->dst_mc].lnk].nbour;
LOCAL  COMMS_HDR  chdr;

        /* Increment count of TXs */
        n_TXs ++;

        if (((EM_HDR *)em_hdr)->dst_mc == this_mc)
        {
              mclink = phys_link[0].link;
              nbour_mc = phys_link[0].nbour;
        }

        /* Seize neighbour's comms. lock */
        seize (ownp_locks[nbour_mc]);

        /* Send comms hdr */
        pwrite (mclink, comms_hdr, sizeof(COMMS_HDR));

        /* Send em hdr */
        pwrite (mclink, em_hdr, sizeof(EM_HDR));

        /* Send msg */
        pwrite (mclink, msg, msize);

        chdr = *((COMMS_HDR *)comms_hdr);

        /* Update elapsed time */
        if ((chdr & ~EMSG_MASK) != EXIT_MSG)
        time_update ( PROTOCOL_TIME * AVE_INST +
                      (sizeof(COMMS_HDR)+sizeof(EM_HDR)+msize) * TX_BYTE_TIME,
                      time_type,
                      calling_proc);
        if ((chdr & ~EMSG_MASK) == USR_MSG)
             time_update ((((MSG *)msg)->msg_hdr.msg_length)*TX_BYTE_TIME,
time_type, calling_proc);
        else if ((chdr & ~EMSG_MASK) == PINFO_MSG)
             {
                  LOCAL  char *mig_ptr = msg;
                  LOCAL  int  siz;
                         mig_ptr += sizeof(int);
                         siz = *((int *)mig_ptr);
                         time_update (siz*TX_BYTE_TIME, time_type, calling_proc);
                      /*   fprintf (trace, "TX: time %d\n", siz*TX_BYTE_TIME);*/
             }

        /* Release comms. lock */
        release (ownp_locks[nbour_mc]);

} /* End of TX */




/***** BROADCAST *****/
```

```
/* This routine sends an ext. msg to all mcs */

broadcast (em_type, caller, msg, msize)

PARAMS  int  em_type;
        PROC_ENTRY *caller;
        char *msg;
        int  msize;

{

LOCAL  COMMS_HDR  comms_hdr = em_type | EMSG_MASK;
LOCAL  EM_HDR  em_hdr;
LOCAL  int  mc;

        /* Set up em hdr */
        em_hdr.sending_mc = this_mc;
        if (caller != (PROC_ENTRY *)EMPTY)
            procncpy (em_hdr.caller, caller -> proc_name);

        /* TX to all mcs */
        for (mc=0; mc<nmcs; mc++)
             if (mc != this_mc)
             {
                 em_hdr.dst_mc = mc;
                 TX ((char *)&comms_hdr, (char *)&em_hdr, msg, msize, OStime,
(PROC_ENTRY *)EMPTY);
             }

} /* End of BROADCAST */
```

```
/***** ACKKCRSIG_HANDLER *****/

ackkcrsig_handler ()

{

EXTERN  int  ackkcrsig_handler ();

        signal (SIGKCRACK, ackkcrsig_handler);
#       ifdef DEBUG
        printf ("%d - ackhandler\n", getpid());
#       endif

        got_ackkcrsig ++;

} /* End of ACKKCRSIG_HANDLER */
```

```
/***** MSGSIGACK_HANDLER *****/

msgsigack_handler ()

{

EXTERN  int  msgsigack_handler ();

        signal (SIGMSGACK, msgsigack_handler);
        printf ("Got msgsigack\n");

        got_msgsigack ++;

} /* End of MSGSIGACK_HANDLER */




/***** MIGSIGACK_HANDLER *****/

migsigack_handler ()

{

EXTERN  int  migsigack_handler();

        signal (SIGMIGACK, migsigack_handler);
/*      printf ("Got migsigack\n"); */

        got_migack ++;

} /* End of MIGSIGACK_HANDLER */




/***** SIGSETUP_HANDLER *****/

sigsetup_handler()

{

EXTERN  int  sigsetup_handler();

        signal(SIGSETUP, sigsetup_handler);

        proc_setup ++;

} /* End of SIGSETUP_HANDLER */




/***** ALARM_HANDLER *****/
```

```
alarm_handler()

{

EXTERN  int  alarm_handler();

        signal(SIGALRM, alarm_handler);

} /* End of ALARM_HANDLER */
```

```
# define  TIME_MIG_PROC  200*AVE_INST

/***** MIGRATE *****/

/* This routine migrates a process to another processor */

typedef  struct {
                   int      residency;
                   int      n_entries;
                   PROCN    owner;
                   char     port_name [MAXOWNPRT][MAXPNAME];

                } EM_PORT_LOC;

migrate (proc, mc)

PARAMS  PROC_ENTRY  *proc;
        int          mc;

{

LOCAL  int             opno;
LOCAL  int             mno;
LOCAL  int             lnk;
LOCAL  int             np_sent;
LOCAL  int             nl_sent;
LOCAL  char            *mig_info;
LOCAL  char            *mig_ptr;
LOCAL  PORT_ENTRY      *port;
LOCAL  MSG             *msg_p;
LOCAL  COMMS_HDR       comms_hdr;
LOCAL  EM_HDR          em_hdr;
LOCAL  int             tot_siz;
LOCAL  EM_PORT_LOC    loc_port_msg;

/*      fprintf (trace,"Migrating Proc. %d %d %s to mc %d\n",
proc->proc_name.pgroup.gmc, proc->proc_name.pgroup.gnum, proc->proc_name.pname,
mc);*/
        /* Update time */
        time_update (TIME_MIG_PROC, OStime, (PROC_ENTRY *)EMPTY);
        /* Increment count of migrates */
        n_migrates ++;

        /* Allocate space for migration msg */
        mig_info = mig_ptr = malloc (proc -> migration_siz+2*sizeof(int));

        /* Put Info for process in mig_info */
        proc -> residency = mc;
        memcpy (mig_ptr, (char *)&(proc->migration_siz), sizeof(int));
        mig_ptr += sizeof(int);
        tot_siz = proc->siz + proc->tot_msg_siz;
        memcpy (mig_ptr, (char *)&tot_siz, sizeof(int));
        mig_ptr += sizeof(int);
        memcpy (mig_ptr, (char *)proc, sizeof(PROC_ENTRY));
        mig_ptr += sizeof(PROC_ENTRY);

        /* Put Info for each owned port in mig_info */
        np_sent = 0;
        loc_port_msg.residency = mc;
        loc_port_msg.n_entries = 0;
        procncpy (loc_port_msg.owner, proc -> proc_name);
        for (opno = 0; np_sent < proc->ownp_length; opno++)
             if ((port = proc->owned_ports[opno]) != (PORT_ENTRY *)EMPTY)
             {
                  loc_port_msg.n_entries ++;
                  strcpy (loc_port_msg.port_name[np_sent], port->port_name);
                  memcpy (mig_ptr, (char *)&opno, sizeof(int));
                  mig_ptr += sizeof(int);
                  port -> residency = mc;
                  memcpy (mig_ptr, (char *)port, sizeof(PORT_ENTRY));
                  mig_ptr += sizeof(PORT_ENTRY);

                  /* Put Link_to Info in mig_info */
                  nl_sent = 0;
                  for (lnk=0; nl_sent < port->profile.lnkt_length; lnk++)
                       if ((port->links_to[lnk].port) != (PORT_ENTRY *)EMPTY)
                       {
                            memcpy (mig_ptr, (char *)&lnk, sizeof(int));
                            mig_ptr += sizeof(int);
                            memcpy (mig_ptr,
(port->links_to[lnk].port)->port_name, MAXPNAME);
                            mig_ptr += MAXPNAME;
                            nl_sent++;
                       }

                  /* Put Link_from Info in mig_info */
                  nl_sent = 0;
                  for (lnk=0; nl_sent < port->profile.lnkf_length; lnk++)
                       if ((port->links_from[lnk]) != (PORT_ENTRY *)EMPTY)
                       {
                            memcpy (mig_ptr, (char *)&lnk, sizeof(int));
                            mig_ptr += sizeof(int);
                            memcpy (mig_ptr, (port->links_from[lnk])->port_name,
MAXPNAME);
                            mig_ptr += MAXPNAME;
                            nl_sent++;
                       }

                  /* Put msg_q Info in mig_info */
```

```
                   msg_p = port -> msg_q_head;
                   for (mno=0; mno < port->profile.inmq_length; mno++)
                   {
                        memcpy (mig_ptr, (char *)msg_p, sizeof(MSG));
                        mig_ptr += sizeof(MSG);
                        msg_p = msg_p -> next;
                   }

                   np_sent ++;

              }

        /* Interrupt process and send it new info */

        got_migack = 0;

        kill (proc->upid, SIGMIG);

        pwrite (kcret_pipe, (char *)&mc, sizeof(int));
        pwrite (kcret_pipe, (char *)&mcpids[mc], sizeof(int));

        while (!got_migack)
        {
               alarm(3);
               pause();
               alarm(0);
        }

        /* Transmit the process info to new mc */
        comms_hdr = PINFO_MSG | EMSG_MASK;
        em_hdr.sending_mc = this_mc;
        em_hdr.dst_mc = mc;
        TX ((char *)&comms_hdr, (char *)&em_hdr, mig_info,
proc->migration_siz+2*sizeof(int), OStime, (PROC_ENTRY *)EMPTY);

        /* Broadcast loc of owned ports */
        if (proc->ownp_length > 0)
        broadcast (PORT_LOC_MSG, (PROC_ENTRY *)EMPTY, (char *)&loc_port_msg,
sizeof(EM_PORT_LOC));

        /* Free alloc'ed space */
        free (mig_info);

        /* Decrement no. of local procs */
        n_local_procs --;
        n_active_local_procs --;

        /* Re-initialise proc table entry */
        proc_initds(proc);

        /* Establish new next process pointer */
        if (proc < nxt_proc)
            nxt_proc = proc;

        /* NEED TO BROADCAST NEW LOC OF PORTS HERE */

} /* End of MIGRATE */
```

```
/***** TIME_UPDATE *****/

/* This function notes the passage of time in the system */

time_update (elapsed_time, time_type, call_proc)

PARAMS  int        elapsed_time;
        int        time_type;
        PROC_ENTRY *call_proc;

{

LOCAL  PROC_ENTRY  *p = process_table;
LOCAL  int         np = 0;

        /* Update system elapsed real time */
        sys_real_time += elapsed_time;

#       ifdef DEBUG
        fprintf (trace, "Sr_time is %f\n", sys_real_time);
#       endif

        /* If time is User add to process's exec time */
        if (time_type == USERtime)
        {
            call_proc -> times.exec_time += elapsed_time;
            call_proc -> times.exec_here_time += elapsed_time;
        }

        /* Add time to residency and exist time for all local processes */
        while (np < n_local_procs)
        {
               if (p -> upid != EMPTY)
               {
                   p -> times.residency_time += elapsed_time;
                   p -> times.exist_time += elapsed_time;
                   np++;
               }
               p++;
        }

        /* Update quanta for averaging of load over a period */
        quanta_update (elapsed_time, time_type);

} /* End of TIME_UPDATE */




/***** QUANTA_UPDATE *****/

/* This function updates the array of time quanta given an amount of elapsed
```

```
time */

quanta_update (elapsed_time, time_type)

PARAMS   int  elapsed_time;
         int  time_type;

{

LOCAL  int  added_time;

        /* Loop adding elapsed time to quanta*/
        /* Must loop cos time may be > quantum */
        while (elapsed_time > 0)
        {
                added_time = elapsed_time;

                /* Check if time will fill rest of quantum */
                if (elapsed_time + quanta[current].used >= QUANTUM)
                {
                    added_time = QUANTUM - quanta[current].used;
                    add_to_quantum (added_time, time_type);

                    /* Move on to next quantum */
                    next_quantum();
                }
                else
                    add_to_quantum (added_time, time_type);

                elapsed_time -= added_time;

        }

} /* End of QUANTA_UPDATE */




/***** ADD_TO_QUANTUM *****/

/* This function adds an amount of time to a quantum */

add_to_quantum (add_time, time_type)

PARAMS  int  time_type;
        int  add_time;

{

        /* If time is OS add it to OStime counts */
        if (time_type == OStime)
        {
            quanta[current].OSportion += add_time;
            OSoverhead += add_time;
        }
        /* Add time to part of the quantum used */
        quanta[current].used += add_time;

} /* End of ADD_TO_QUANTUM */




/***** NEXT_QUANTUM *****/

/* This function moves on to the next quantum in the period */

next_quantum()
{

        current++;

        /* Check if need to 'wrap' current to start of array */
        if (current >=  NQUANTA)
            current = 0;

        /* Initialise new quantum */
        quanta[current].used = 0;
        OSoverhead -= quanta[current].OSportion;
        quanta[current].OSportion = 0;
        quanta[current].actual_load = n_active_local_procs;
        quanta[current].virtual_load = n_virtual_procs;

} /* End of NEXT_QUANTUM */




/***** INIT_SHARED_GLOBALS *****/

/* This routine initialises shared global variables */

init_shared_globals()
{

LOCAL  PROC_ENTRY   *proc;
LOCAL  ROUTE        *rte;
LOCAL  QTUM_ENTRY   *qta;

        /* Process table */
        for (proc=process_table; proc<process_table+MAXPROCS; proc++)
             proc_initds (proc);

        nxt_proc = process_table;
        scheduled_proc = process_table;

        /* Routing table */
        for (rte=route_table; rte<route_table+MAXMCS; rte++)
        {
             rte -> lnk = EMPTY;
             rte -> distance = EMPTY;
        }

        /* Time info */
        OSoverhead = 0;
        sys_real_time = 0;

        for(qta=quanta; qta<quanta+NQUANTA; qta++)
        {
            qta -> actual_load = N_SYS_PROCS;
            qta -> virtual_load = 0;
            qta -> OSportion = 0;
            qta -> used = 0;
        }

        current = 0;

        /* Process counts */
}




/***** PERFORMANCE_DUMP *****/

/* This routine dumps performance info to trace file */

performance_dump()

{
#       ifndef FILE_INPUT
          printf ("Mc %d - Time is %f\n", this_mc, sys_real_time);
#       endif

        fprintf (trace, "Performance dump at %f\n", sys_real_time);

        fprintf (trace, "No. of procs %d\n\n", n_local_procs);

        if (n_deaths != 0)
        fprintf (trace, "Ave RT = %f\n", cumul_exist_time/(double)n_deaths);
        else
        fprintf (trace, "Ave RT = 0.0\n");

        fprintf (trace, "No. migrates = %d\n", n_migrates);
        n_migrates = 0;

        fprintf (trace, "No. TXs = %d\n", n_TXs);
        n_TXs = 0;

} /* End of PERFORMANCE_DUMP */
```

```
/***** CALC_AVE_LOAD *****/

/* This routine averages the local load over NQUANTA time quanta */

float calc_ave_load()

{

LOCAL  int  total_load = 0;
LOCAL  int  i;

        for (i=0; i<NQUANTA; i++)
             total_load += quanta[i].actual_load + quanta[i].virtual_load;

        return ((float)total_load/NQUANTA);

} /* End of CALC_AVE_LOAD */


typedef  struct {
                   unsigned short negociation_type;
                   unsigned short forward_it;

                } EM_NEGOCIATION;

typedef  struct {
                   float new_average;

                } EM_CHANGE_AVERAGE;



/***** OVERLOAD_CHECK *****/

/* This routine checks if mc is overloaded and if so broadcasts for help */

overload_check ()

{

LOCAL  EM_NEGOCIATION  msg;
LOCAL  COMMS_HDR       comms_hdr;
LOCAL  EM_HDR          em_hdr;
LOCAL  int             i;

        if (OVERLOADED && !too_high.set)
        {

           /* Broadcast too high msg */
           msg.negociation_type = TOO_HIGH;
           msg.forward_it = 1;
           comms_hdr = NEGOCIATE_MSG | EMSG_MASK;
           em_hdr.sending_mc = this_mc;

           for (i=0; i<n_nbours; i++)
           {
                em_hdr.dst_mc = neighbours[i];
                TX ((char *)&comms_hdr, (char *)&em_hdr, (char *)&msg,
sizeof(EM_NEGOCIATION), OStime, (PROC_ENTRY *)EMPTY);
            }


            /* Set timeout */
            too_high.set = TRUE;
            too_high.timer = sys_real_time + TIMEOUT_INTERVAL;

        }

        return;

} /* End of OVERLOAD_CHECK */





/***** UNDERLOAD_CHECK *****/

/* This routine sets a too low timeout if mc is underloaded */

underload_check ()

{

        if (UNDERLOADED && !too_low.set)
        {

            /* Set too low timeout */
            too_low.set = TRUE;
            too_low.timer = sys_real_time + TIMEOUT_INTERVAL;

        }

} /* End of UNDERLOAD_CHECK */





/***** EXP_HIGH_CHECK *****/

/* This routine checks to see if a too high timeout has expired */

exp_high_check ()

{

LOCAL  EM_CHANGE_AVERAGE  msg;

        if (too_high.set && sys_real_time >= too_high.timer)
        {

            /* If still overloaded average load must change */
            if (OVERLOADED)
            {
                global_average_load += CHANGE_AMOUNT;

                /* Broadcast new average */
                msg.new_average = global_average_load;
                broadcast (CH_AVE_MSG, (PROC_ENTRY *)EMPTY, (char *)&msg,
sizeof(EM_CHANGE_AVERAGE));

            }

            /* Cancel too high timeout */
            too_high.set = FALSE;

        }

        return;

} /* End of EXP_HIGH_CHECK */




/***** EXP_LOW_CHECK *****/

/* This routine checks if a too low timeout has expired */

exp_low_check ()

{

LOCAL  EM_CHANGE_AVERAGE  msg;

        if (too_low.set && sys_real_time >= too_low.timer)
        {

            /* If still underloaded average load must change */
            if (UNDERLOADED)
            {

                global_average_load -= CHANGE_AMOUNT;

                /* broadcast new value */
                msg.new_average = global_average_load;
                broadcast (CH_AVE_MSG, (PROC_ENTRY *)EMPTY, (char *)&msg,
sizeof(EM_CHANGE_AVERAGE));

            }

            /* Cancel too low timeout */
            too_low.set = FALSE;

        }

        return;

} /* End of EXP_LOW_CHECK */
```

```
/***** EXP_AWAIT_CHECK *****/

/* This routine checks for an awaiting process timeout */

exp_await_check()

{

        if (awaiting_process.set && sys_real_time >= awaiting_process.timer[0])
        {
            remove_await_timeout ();
        }

} /* End of EXP_AWAIT_CHECK */
```

```
/***** CHOOSE_PROC_TO_MIGRATE *****/

/* This routine picks a non-executing process to migrate */

PROC_ENTRY *choose_proc_to_migrate ()

{

OWN PROC_ENTRY *mig_p = process_table + N_SYS_PROCS - 1;
LOCAL int  p_count = N_SYS_PROCS;

        if (n_active_local_procs <= N_SYS_PROCS + 1)
            return (PROC_ENTRY *)EMPTY;

        /* Pick process */
        do {
              if (++mig_p >= process_table + MAXPROCS)
                  mig_p = process_table + N_SYS_PROCS;
              if (!mig_p -> blocked && mig_p -> upid != EMPTY)
                  p_count++;

           } while ((mig_p -> upid == EMPTY ||
                     mig_p == scheduled_proc ||
                     mig_p -> blocked == BLOCKED)
                    && p_count <= n_active_local_procs);


        if (p_count > n_active_local_procs)
            return  (PROC_ENTRY *)EMPTY;
        else
            return  mig_p;

} /* End of CHOOSE_PROC_TO_MIGRATE */
```

```
/***** CONSIDER_MIGRATION *****/

consider_migration()

{

LOCAL  int              l;
LOCAL  int              lowest;
LOCAL  PROC_ENTRY       *p = (PROC_ENTRY *)EMPTY;
EXTERN PROC_ENTRY       *choose_proc_to_migrate();

        if ((p=choose_proc_to_migrate()) != (PROC_ENTRY *)EMPTY)
        {

            /* Consider if process would run better elsewhere */
            lowest = 1;
            for (l=1; l<VECTOR_SIZE; l++)
                 if (load_vector[l].load < load_vector[lowest].load
                     && load_vector[l].processor != EMPTY)
                     lowest = l;

            if (load_vector[lowest].load < n_active_local_procs -1
                && load_vector[lowest].processor != this_mc)
                migrate (p, load_vector[lowest].processor);

        }

} /* End of CONSIDER_MIGRATION */
```

DIRECTORY NAME : PHYSNETWK

```
/* FILE : project/src/physnetwk/main.c */

/* This is the main function for the whole simulation */

/* Includes for this file */

# include <stdio.h>
# include <signal.h>
# include <fcntl.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"



/* GLOBAL variables definitions */

PORT_ENTRY  port_table [MAXPORTS];              /* Global port table */
PORT_ENTRY  *nxt_port;                          /* Ptr to nxt avail. port */
int         nports;                             /* No. of entries in port tab */
PROC_ENTRY  process_table [MAXPROCS];           /* Global process table */
PROC_ENTRY  *nxt_proc;                          /* Ptr to nxt avail. process */
int         nprocs;                             /* No. of entries in process tab */
int         last_proc_creat;                    /* Index of last proc. created */
int         own_pipe;                           /* Processors comms. pipe */
int         kcret_pipe;                         /* Kernel call return pipes */
ROUTE       route_table [MAXMCS];               /* Network routing table */
int         mcpids [MAXMCS];                    /* UNIX pids of simulated mcs */
int         this_mc;                            /* Id of this processor */
PLINK       phys_link [MAXLINKS];               /* Table of links to other mcs */
char        ownp_locks [MAXMCS][OPLOCKSIZ];     /* For exc. access to comms. */
char        kcret_locks [MAXMCS][KCRLOCKSIZ];/* For exc. access to kcrets */
FILE        *trace;                             /* Trace file for monitoring */
FILE        *config;                            /* Configuration file */
int         nmcs;                               /* No. of processors in network */
int         boot;                               /* Is this boot time? */
int         load_bal_active;                    /* Is load balancing active? */
int         synth_workload;                     /* Is workload synthetic? */
int         got_ackkcrsig;                      /* Has an ackkcr signal arrived? */
int         got_msgsigack;                      /* Has an ackmsg signal arrived? */
int         got_migack;                         /* Has a migack signal arrived? */
int         proc_setup;                         /* Has proc. setup OK? */
int         n_local_procs;                      /* No of local processes */
int         n_active_local_procs;               /* No of active local processes */
double      sys_real_time;                      /* Elapsed system time */
int         OSoverhead;                         /* Unavailable time over quanta */
QTUM_ENTRY  quanta [NQUANTA];                   /* Time quanta */
int         current;                            /* Current time quantum */
int         shmid;                              /* Shm seg id */
unsigned short xsubi[3]={300,400,500};          /* Seed for R.N.G. */
double      *stop_time;                         /* Sync. point for all mc's */
unsigned short *reached;                        /* Have mc's reached stop_time? */
double      cumul_exist_time;                   /* Cumulative proc. exist times */
int         n_deaths;                           /* No. of processes thru system */
TIME_OUT    too_low;                            /* Too low time_out */
TIME_OUT    too_high;                        /* Too high time_out */
AWAIT_TIMEOUT awaiting_process;              /* Awaiting process timeout */
float       global_average_load;             /* System_wide average load per mc */
float       my_average_load;                 /* Local load averaged over NQUANTA
time quanta */
int         n_virtual_procs;                 /* No. of procs. in transit to here
*/
PROC_ENTRY  *scheduled_proc;                 /* Process last scheduled */
PROC_LOAD   load_vector[VECTOR_SIZE];        /* Processor load vector */
int         process_groups[] = { 2,3,4};     /* 2,3,4 process groups */
unsigned short rnd_job[] = {7, 8, 9};        /* Seed to choose rnd proc grp */
int         nxt_job;                         /* Proc grp to be created */
int         n_migrates;                      /* No. of migrations */
int         n_TXs;                           /* No. of msg transmissions */
int         n_nbours;                        /* No. of direct neighbours */
int         neighbours[MAXLINKS];            /* Mc id of neighbours */
int         policy = SENDER;                 /* Rcver or sender policy? */



/* Functions EXTERNal to this file */

EXTERN  int  init_ds();
EXTERN  int  start_up();
EXTERN  int  alarm_handler();
EXTERN  int  ERROR();
EXTERN  int  pipes_create();
EXTERN  int  mcs_create();
EXTERN  int  wait_for_children();




main (argc, argv)

PARAMS  int argc;
        char *argv[];
{

LOCAL  int  lockfd;
LOCAL  int  mc;
LOCAL  int  outp;

        /* Redirect stdout to terminal (nohup has redirected it to nohup.out */
        outp = open (argv[1], WRITE);
        close(WRITE);
        dup(outp);
        close(outp);
        printf ("Stdout redirected to terminal\n");

        /* Open trace for debugging */
#       ifdef DEBUG
        trace = fopen("main.trc","w");
        fcntl ((int)(fileno(trace)), F_SETFD, 1);
#       endif
#       ifdef DEBUG
        fprintf (trace, "%d\n", port_table);
#       endif

        /* Create control seg for synchronising mc's */
        create_control_segment();

        /* Initialise all global variables */
        init_ds();

        /* Set up SIGINT handler for starting network simulation */
        signal (SIGINT, start_up);

        /* Set up alarm_handler */
        signal (SIGALRM, alarm_handler);

        /* Open lock file used for mutual exclusion */
        if ((lockfd = open (LOCKFNAME, O_RDONLY|O_CREAT)) < 0)
             ERROR (LOCK_OPEN);
        fcntl (lockfd, F_SETFD, 1);

#       ifdef FILE_INPUT
          config = fopen ("config.main", "r");
#       endif

        /* Establish no. of processors in network */
        nmc_enter();

        /* Create job streams */
        job_creation();

        /* Create pipes for communication (interprocessor and kcalls) */
        pipes_create();

        /* Fork simulated processors */
        mcs_create();

#       ifdef FILE_INPUT
          fclose (config);
#       endif

        /* Send pids of mcs */
        if (this_mc == MANAGER)
        for (mc=0; mc<nmcs; mc++)
        {
             char opname[OPNAMSIZ];
             sprintf (opname, "own_p%d", mc);
             if ((own_pipe = open (opname, O_WRONLY, 0)) == FAIL)
                  ERROR (PIPE_OPEN);
             write (own_pipe, (char *)mcpids, sizeof(int)*MAXMCS);
             close (own_pipe);
        }

        /* Pause waiting for SIGINT (rubout key) */
        pause();

        /* Wait for processors to terminate */
        /* Executed by initial process only */
        wait_for_children();

} /* END OF MAIN */
```

```
/* FILE: project/src/physnetwk/setup.c */

/* This file contains routines to set up the simulated network */

/* Includes for this file */

# include <stdio.h>
# include <signal.h>
# include <fcntl.h>
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/shm.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"
# include "/usr/acct/ian/project/src/include/sys/globvars.h"


/* Functions EXTERNal to this file */

EXTERN  int  dump_ports();
EXTERN  int  dump_procs();
EXTERN  int  port_initds();
EXTERN  int  proc_initds();
EXTERN  int  seize();
EXTERN  int  release();
EXTERN  int  ERROR();
EXTERN  int  cls();
EXTERN  double  log();
EXTERN  double  erand48();

unsigned short exp_xsubi[] = {55213, 10232, 2721};
unsigned short rnd_xsubi[] = {3, 4, 5};



/***** INIT_DS() *****/

/* This routine initialises all global vars */

init_ds()
{

LOCAL PORT_ENTRY  *port;
LOCAL int          i;



        /* Initialise Port Table */
        for (port=port_table; port<port_table+MAXPORTS; port++)
             port_initds (port);

        nxt_port = port_table;
        /* Initialise other tables */
        for (i=0; i<MAXMCS; i++)
        {
             sprintf (ownp_locks[i], "op%d", i);
             sprintf (kcret_locks[i], "kcr%d", i);
             mcpids[i] = EMPTY;
        }

        for (i=0; i<MAXLINKS; i++)
        {
             phys_link[i].link = EMPTY;
             phys_link[i].nbour = EMPTY;
        }

        /* Initialise pipes fds */
        own_pipe = EMPTY;
        kcret_pipe = EMPTY;

        this_mc = MANAGER;

        /* Times */
        *stop_time = 0.0;
        reset_reached();
        cumul_exist_time = 0.0;
        n_deaths = 0;

        /* Time outs */
        too_low.set = FALSE;
        too_low.timer = (double)0.0;
        too_high.set = FALSE;
        too_high.timer = (double)0.0;
        awaiting_process.set = FALSE;
        for (i=0; i<NAWAITS; i++) awaiting_process.timer[i] = (double)0.0;

        /* Load values */
        global_average_load = (float)N_SYS_PROCS;
        my_average_load = (float)N_SYS_PROCS;

        /* Process counts */
        n_local_procs = 0;
        n_active_local_procs = 0;
        n_virtual_procs = 0;

        /* Performance data */
        n_migrates = 0;
        n_TXs = 0;

} /* End of INIT_DS */




/***** PIPES_CREATE *****/

/* This routine creates pipes for communication */
```

```
pipes_create()
{

LOCAL   int  p;
LOCAL   int  p_result;
LOCAL   char opname[OPNAMSIZ];
LOCAL   char kcrname[KCRNAMSIZ];

        for (p=0; p<nmcs; p++)
        {
             /* Create own pipe */
             sprintf (opname, "own_p%d", p);
             if ((p_result = mknod (opname, FIFO|OWNACC, 0)) == FAIL)
                  ERROR (PIPE_CREATION);

             /* Create kcall return pipe */
             sprintf (kcrname, "kcr_p%d", p);
             if ((p_result = mknod (kcrname, FIFO|OWNACC, 0)) == FAIL)
                  ERROR (PIPE_CREATION);
        }

} /* End of PIPES_CREATE */
```

```
/***** NMC_ENTER *****/

/* This routine enters no. of processors in network */

nmc_enter()
{

        do{
#           ifdef FILE_INPUT
               fscanf (config, "%d", &nmcs);
#           else
               printf ("Enter no. of processors");
               scanf ("%d", &nmcs);
#           endif

          }while (nmcs<1 || nmcs>MAXMCS);

} /* End of NMC_ENTER */
```

```
/***** MCS_CREATE *****/

/* This routine creates nmcs processors */
```

```
mcs_create()
{

LOCAL   int  mc;
LOCAL   char opname[OPNAMSIZ];
LOCAL   char kcrname[KCRNAMSIZ];
LOCAL   int  bodge;
LOCAL   char    trce_file[MAXFNAME];
#ifdef FILE_INPUT
LOCAL   char    config_file[MAXFNAME];
#endif

        /* FORK nmcs processors */
        for (mc=0; mc<nmcs; mc++)
             switch (mcpids[mc] = fork())
             {

                  case CHILD:     /* Open trace file */
                                  this_mc = mc;
                                  sprintf (trce_file, "trace%d.trc", this_mc);
                                  trace = fopen (trce_file, "w");
                                  setbuf (trace, NULL);
                                  fcntl ((int)(fileno(trace)), F_SETFD, 1);

                                  /* Open own pipe for reading|no_delay */
                                  sprintf (opname, "own_p%d", this_mc);
                                  if ((own_pipe =
open(opname,O_RDONLY|O_NDELAY,0)) == FAIL)
                                       ERROR (PIPE_OPEN);
                                  fcntl (own_pipe, F_SETFD, 1);

                                  /* Open kcall return pipe for writing */
                                  sprintf (kcrname, "kcr_p%d", this_mc);
                                  bodge = open (kcrname, O_RDONLY|O_NDELAY, 0);
                                  fcntl (bodge, F_SETFD, 1);
                                  if ((kcret_pipe = open (kcrname, O_WRONLY, 0))
== FAIL)

                                       ERROR (PIPE_OPEN);
                                  fcntl (kcret_pipe, F_SETFD, 1);

                                  /* Create shared mem seg */
                                  /* shm_creat(); */

                                  /* Initialise shared globals */
                                  init_shared_globals();

#                                 ifdef FILE_INPUT
                                     sprintf (config_file, "config.%d", this_mc);
                                     config = fopen (config_file, "r");
#                                 endif

                                  /* Get more info on "up" processors */
                                  if (this_mc < nmcs)
                                  {
                                      get_link_info();
                                  }

                                  fclose (config);
```

```
                                        set_neighbours ();

                                        printf ("%d - configured\n", this_mc);

                                        return;


                        case FAIL:      ERROR (FORK_MCS);


                        PARENT:         break;

                }

} /* End of MCS_CREATE */







/***** GET_LINK_INFO *****/

/* This routine asks for info regarding physical links */

get_link_info()
{

LOCAL   int  mc;
LOCAL int  l;
LOCAL int  nlinks;

#       ifndef FILE_INPUT

        seize ("tty");

        cls();

        printf ("Entering Info for Processor %d\n",this_mc);
        printf ("---------------------------------\n\n");

#       endif

        /* Get no. of links for this processor */
        nlnk_enter (&nlinks);

        /* Establish direct neighbour on each link */
#       ifndef FILE_INPUT
        printf ("Enter processor no. of direct neighbour on following links\n");
#       endif

        for (l=0; l<nlinks; l++)
             get_neighbour (l);

        /* Establish routing to all processors */
        for (mc=0; mc<nmcs; mc++)
             get_routing (mc, nlinks);
```

```
#        ifndef FILE_INPUT
         release ("tty");
#        endif

} /* End of GET_LINK_INFO */





/***** NLNK_ENTER *****/

/*  This routine enters the no. of links for a processor */

nlnk_enter (nlinks)

PARAMS int  *nlinks;
{

        do{
#               ifdef FILE_INPUT
                   fscanf (config, "%d", nlinks);
#               else
                   printf ("How many links to/from this processor?");
                   scanf ("%d", nlinks);
#               endif

            }while (*nlinks<1 || *nlinks>MAXLINKS);

} /* End of NLNK_ENTER */





/***** GET_NEIGHBOUR *****/

/* This routine enters dir. neighbour on each link */

get_neighbour (lnk)

PARAMS int  lnk;

{

LOCAL int  neighbour;
LOCAL char pname[OPNAMSIZ];

        /* Get id of direct neighbour on this link */
        do{
#               ifdef FILE_INPUT
                  fscanf (config, "%d", &neighbour);
#               else
                  printf ("Link %d ", lnk);
                  scanf ("%d", &neighbour);
#               endif

          }while (neighbour<0 || neighbour>nmcs);

        /* Enter appropriate pipefd into physical link table */
        sprintf (pname, "own_p%d", neighbour);
        if((phys_link[lnk].link = open (pname, O_WRONLY, 0)) == FAIL)
            ERROR (PIPE_OPEN);
        fcntl (phys_link[lnk].link, F_SETFD, 1);
        phys_link[lnk].nbour = neighbour;

} /* End of GET_NEIGHBOUR */




/***** GET_ROUTING *****/

/* This routine enters routing info for each processor */

get_routing (mc, nlinks)

PARAMS int  mc;
       int  nlinks;

{

        if (mc != this_mc)
            do{
#                ifdef FILE_INPUT
                  fscanf (config, "%d", &(route_table[mc].lnk));
                  fscanf (config, "%d", &(route_table[mc].distance));
#                else
                  printf ("Processor %d on link ", mc);
                  scanf ("%d", &(route_table[mc].lnk));
                  printf ("Distance ");
                  scanf ("%d", &(route_table[mc].distance));
#                endif

              }while (route_table[mc].lnk<0 || route_table[mc].lnk>nlinks ||
route_table[mc].distance<0);

} /* End of GET_ROUTING */




/***** START_UP *****/

/* This is a handler for SIGINT. It starts up each processor */

start_up()
{
```

```
EXTERN int start_up();

         signal (SIGINT, start_up);

         /* Decide if processor should be up */
         if (this_mc<nmcs && this_mc>=0)
         {
             seize ("tty");
             printf ("Processor %d up\n", this_mc);
             release ("tty");

             /* Run the kernel */
             kernel();
         }
         else if (this_mc == MANAGER)
              { master();
              }

         else pause();

} /* End of START_UP */
```

```
/***** WAIT_FOR_CHILDREN *****/

/* This routine waits for all forked processors */

wait_for_children()
{

LOCAL   int  mc;
LOCAL int  died;
LOCAL int  status;

         for (mc=0; mc<nmcs; mc++)
         {
              died = wait (&status);
              printf ("Process no. %d dead\n", died);
              if (status!=0)
                 printf ("Signal %d\n",status);
         }

} /* End of WAIT_FOR_CHILDREN */
```

```
/****************************************************************************/

/***** JOB_CREATION *****/
```

```
job_creation()

{

LOCAL  int       i;
LOCAL  int       j;
LOCAL  FILE      *jobs[MAXMCS];
LOCAL  char      job_name[MAXFNAME];
LOCAL  double    rnd_num;
LOCAL  double    rnd_grp;
LOCAL  double    new_rand;
LOCAL  double    t = 0.0;
LOCAL  float     lamda;
LOCAL  int       njobs;
EXTERN double    log();
EXTERN double    erand48();

       printf ("CREATING job streams ...\n");

       /* Open job files */
       for (i=0; i<nmcs; i++)
       {
            sprintf (job_name, "jobs%d", i);
            jobs[i] = fopen (job_name, "w");
       }

#      ifdef FILE_INPUT
         fscanf (config, "%d", &njobs);
         fscanf (config, "%f", &lamda);
#      else
         printf ("Enter no. of jobs ");
         scanf ("%d", &njobs);
         printf ("Enter system load value ");
         scanf ("%f", &lamda);
#      endif

       /* Create job streams */
       for (j=0; j<njobs; j++)
       {
            /* Generate exp. random no. */
            rnd_num = erand48(exp_xsubi);
            new_rand = -log(rnd_num)/((double)(lamda*nmcs));

            /* Establish time of creation */
            t += new_rand * 1000000 * AVE_PROC_GROUP * AVE_EXEC_TIME;

            /* Give job to random mc */
            rnd_grp = erand48(rnd_job);
            rnd_grp *= 3;
            rnd_num = erand48(rnd_xsubi);
            rnd_num *= nmcs;
            fprintf (jobs[(int)rnd_num], "%d ", process_groups[(int)rnd_grp]);
            fprintf (jobs[(int)rnd_num], "%f\n", t);

       }

       /* Close job files */
       for (i=0; i<nmcs; i++)
            fclose (jobs[i]);
```

```
        printf ("Job Creation COMPLETE\n\n");

} /* End of JOB_CREATION */




/****************************************************************************/

/***** MASTER *****/

master()

{

LOCAL  int  n_reached;
LOCAL  int  i;
EXTERN int  wait_for_children();

        nice(39);
        signal (SIGINT, wait_for_children);

        /* Loop checking to see if mcs have reached stop time */
        FOREVER
        {
            /* Count no of mcs who have reached stop time */
            n_reached = 0;
            while (n_reached < nmcs)
            {
                    n_reached = 0;
                    for (i=0; i<nmcs; i++)
                         if (reached[i])
                             n_reached++;
                         else
                             break;
            }

            /* Reset reached array to zeros */
            reset_reached();

            /* Establish next stop time */
            *stop_time += SYNC_TIME;

            /* Allow mcs to continue */
            restart_mcs();


        }

} /* End of MASTER */
```

```
/*************************************************************************/

/***** RESET_REACHED *****/

reset_reached()

{

LOCAL  int  i;

        for (i=0; i<nmcs; i++)
             reached[i] = 0;

} /* End of RESET_REACHED */




/*************************************************************************/

/***** RESTART_MCS *****/

restart_mcs()

{

LOCAL  int i;

        for (i=0; i<nmcs; i++)
             kill (mcpids[i], SIGKCRACK);


} /* End of RESTART_MCS */




/*************************************************************************/

/***** CREATE_CONTROL_SEG *****/

create_control_segment()

{

LOCAL  int  ctrl_shmid;
LOCAL  char *shmloc;
LOCAL  char *shmptr;
EXTERN char *shmat();

        /* Create control segment */
        ctrl_shmid = shmget (IPC_PRIVATE, CTRL_SEG_SIZ, IPC_CREAT|SHM_R|SHM_W);

        /* Attach it */
        shmloc = shmat (ctrl_shmid, 0, 0);
        /* Set up pointers to control seg */
        shmptr = shmloc;
        stop_time = (double *)shmptr;
        shmptr += sizeof(double);
        reached = (unsigned short *)shmptr;

} /* End of CREATE_CONTROL_SEGMENT */



/***** SET_NEIGHBOURS *****/

set_neighbours ()

{

LOCAL  int  i;

        n_nbours = 0;

        for (i=0; i<nmcs; i++)
             if (i!=this_mc && route_table[i].distance==1)
             {
                 neighbours[n_nbours] = i;
                 n_nbours ++;
                 printf ("Mc %d nbour %d\n",this_mc,i);
             }
        printf ("Mc %d  %d nbours\n",this_mc,n_nbours);

} /* End of SET_NEIGHBOURS */
```

DIRECTORY NAME : UTILS

```
/* FILE: project/src/utils/ERROR.c */

/* This file contains a common error routine */

/* Includes for this file */

# include <stdio.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"
# include "/usr/acct/ian/project/src/include/sys/globvars.h"



/***** ERROR *****/

/* This routine deals with system errors */

ERROR (errcode)

PARAMS  int  errcode;

{

OWN  char *sys_errs[] = { "Failed to create pipes\n",       /* PIPE_CREATION */
                          "Failed to fork mcs\n",           /* FORK_MCS */
                          "Comms. hdr. error\n",            /* C_HDR_ERR */
                          "Too many global ports\n",        /* TM_PORTS */
                          "Failed to fork usr process\n",   /* FORK_FAIL */
                          "Too many global processes\n",    /* TM_PROCS */
                          "Error during pipe read\n",       /* PIPE_READ */
                          "Error during pipe write\n",      /* PIPE_WRITE */
                          "Failed to open lock file\n",     /* LOCK_OPEN */
                          "Failed to open pipe\n"           /* PIPE_OPEN */
                        };

        fprintf (trace, "Fatal system error on mc %d", this_mc);
        fprintf (trace, "%s\n", sys_errs[errcode]);

        exit (-1);

} /* End of ERROR */
```

```
/* FILE: project/src/utils/tabutils.c */

/* This file contains a number of routines common to */
/* many modules for manipulating the process and port */
/* tables*/

/* Includes for this file */

# include <stdio.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"
# include "/usr/acct/ian/project/src/include/sys/globvars.h"


#ifdef RMSGDBUG
EXTERN char *malloc();
#endif



/***** FIND_PORT *****/

/* This routine finds a port given its name */

PORT_ENTRY *find_port (pid, port_name)

PARAMS PROCN  pid;
       char   port_name[];

{

LOCAL PORT_ENTRY  *p = port_table;

        while (p -> owner_proc.pgroup.gmc != pid.pgroup.gmc ||
               p -> owner_proc.pgroup.gnum != pid.pgroup.gnum ||
              (strcmp(p->port_name,port_name) != 0))

               if (p++ > port_table+MAXPORTS)
                   return (PORT_ENTRY*) EMPTY;

        return p;

} /* End of FIND_PORT */





/***** FIND_PROC *****/

/* This routine finds a process given its name */
```

```
PROC_ENTRY *find_proc (pid)

PARAMS PROCN  pid;

{

LOCAL PROC_ENTRY  *p = process_table;

        while (p -> proc_name.pgroup.gmc != pid.pgroup.gmc ||
               p -> proc_name.pgroup.gnum != pid.pgroup.gnum ||
              (strcmp (p->proc_name.pname, pid.pname) != 0))

               if (p++ > process_table+MAXPROCS)
                   return (PROC_ENTRY*) EMPTY;

        return p;

} /* End of FIND_PROC */






/***** DUMP_PROCS *****/

/* This routine dumps the process table for debugging */

dump_procs()
{

LOCAL long  clock = time (0);
LOCAL   int   i;
LOCAL   int   j;

        fprintf (trace, "Dump at %s\n", ctime(&clock));

        fprintf (trace, "Active %d\n", n_active_local_procs);
        fprintf (trace, "Local %d\n", n_local_procs);
        fprintf (trace, "Ackkcr %d\n", got_ackkcrsig);
        fprintf (trace, "Migack %d\n", got_migack);
        fprintf (trace, "Process Table\n\n\n");


        for (i=0; i<MAXPROCS; i++)
        {

             fprintf (trace,
                     "Entry no. %d\n\
                      --------------\n\n\
                      Name %d %d %s\n\
                      Schedulable %d\n\
                      N_probes %d\n\
                      Mc_p[0] %d\n\
                      Mc_p[1] %d\n\
                      Mc_p[2] %d\n\
                      Upid %d\n\
                      Residency %d\
                      Exec_time %d\
                      Res time %d\
                      Exist time %d\
                      Mig Size %d\n",

                      i,
                      process_table[i].proc_name.pgroup.gmc,
                      process_table[i].proc_name.pgroup.gnum,
                      process_table[i].proc_name.pname,
                      process_table[i].schedulable,
                      process_table[i].n_probes,
                      process_table[i].mcs_probed[0],
                      process_table[i].mcs_probed[1],
                      process_table[i].mcs_probed[2],
                      process_table[i].upid,
                      process_table[i].residency,
                      process_table[i].times.exec_time,
                      process_table[i].times.residency_time,
                      process_table[i].times.exist_time,
                      process_table[i].migration_siz
                     );


             fprintf (trace, "Owned Ports\n");

             for (j=0; j<MAXOWNPRT; j++)
                  if ((process_table[i].owned_ports[j])== (PORT_ENTRY*)EMPTY)
                       fprintf(trace, "EMPTY\n");
                  else
                  fprintf (trace,
                           "%s\n",
                           (process_table[i].owned_ports[j]) -> port_name
                          );



             fprintf (trace,
                      "\n\nOwnp_length %d\n\
                      Nxt_ownp %d\n\n\n",

                      process_table[i].ownp_length,
                      process_table[i].nxt_ownp
                     );

        }

} /* End of DUMP_PROCS */






/***** DUMP_PORTS *****/

/* This routine dumps the port table for debugging */

dump_ports()
```

```
{

LOCAL long  clock = time (0);
LOCAL   int   i, j;

        fprintf (trace, "Dump at %s\n", ctime (&clock));
        fprintf (trace, "Port Table\n\n\n");

        for (i=0; i<MAXPORTS; i++)
        {
             fprintf (trace,
                      "Entry no. %d\n\
                       --------------\n\n\
                       Name %s\n\
                       Residency %d\n\
                       Msg Type %d\n\
                       Lnkf_len %d\n\
                       Lnkt_len %d\n\
                       Nxt_lf %d\n\
                       Nxt_lt %d\n\
                       Inmq_len %d\n\
                       nb_pending %d\n\
                       b_pending %d\n\n\n",

                       i,
                       port_table[i].port_name,
                       port_table[i].residency,
                       port_table[i].msg_type,
                       port_table[i].profile.lnkf_length,
                       port_table[i].profile.lnkt_length,
                       port_table[i].profile.nxt_lf,
                       port_table[i].profile.nxt_lt,
                       port_table[i].profile.inmq_length,
                       port_table[i].profile.nb_pending,
                       port_table[i].profile.b_pending
                     );

             fprintf (trace, "Links_to\n\n");

             for (j=0; j<port_table[i].profile.lnkt_length; j++)
                  fprintf (trace,
                           "Link_to %d\n\
                            Nmsgs %d\n\
                            Tot_l %d\n\n",

                            j,
                            port_table[i].links_to[j].nmsgs,
                            port_table[i].links_to[j].tot_msglength
                          );

              fprintf (trace, "\n\n\n");

        }

} /* End of DUMP_PORTS */
```

```
/***** PORT_INITDS *****/

/* This routine initialises a port table entry */

port_initds (port)

PARAMS PORT_ENTRY  *port;

{

LOCAL  int  i;



        for (i=0; i<MAXPNAME; i++)
             port -> port_name[i] = ' ';

        port -> residency = EMPTY;
        port -> owner_proc.pgroup.gmc = EMPTY;
        port -> owner_proc.pgroup.gnum = EMPTY;

        for (i=0; i<MAXPNAME; i++)
             port -> owner_proc.pname[i] = ' ';

        for (i=0; i<MAXLFROM; i++)
             port -> links_from[i] = (PORT_ENTRY*) EMPTY;

        for (i=0; i<MAXLTO; i++)
        {
             port -> links_to[i].nmsgs = 0;
             port -> links_to[i].tot_msglength = 0;
             port -> links_to[i].port = (PORT_ENTRY*) EMPTY;
        }

        port -> msg_q_head = (MSG *)EMPTY;
        port -> msg_q_tail = (MSG *)EMPTY;
#ifdef RMSGDBUG
port -> msg_q_head = port -> msg_q_tail = (MSG *)malloc(sizeof(MSG));
port -> msg_q_head -> msg_txt = malloc (100);
strcpy (port->msg_q_head->msg_txt, "TEST MESSAGE");
port -> msg_q_head ->msg_hdr.msg_length = 100;
#endif
        port -> rcvfunc = port -> destruct = (PFI) EMPTY;
        port -> msg_type = EMPTY;
        port -> profile.lnkf_length = 0;
        port -> profile.lnkt_length = 0;
        port -> profile.nxt_lf = 0;
        port -> profile.nxt_lt = 0;
        port -> profile.inmq_length = 0;
#ifdef RMSGDBUG
port -> profile.inmq_length = 1;
#endif
        port -> profile.nb_pending = 0;
        port -> profile.b_pending = 0;

} /* End of PORT_INITDS */
```

```
/***** PROC_INITDS *****/

/* This routine initialises a process table entry */

proc_initds (process)

PARAMS PROC_ENTRY *process;

{

LOCAL  int  i;

        process -> proc_name.pgroup.gmc = EMPTY;
        process -> proc_name.pgroup.gnum = EMPTY;

        for (i=0; i<MAXPNAME; i++)
             process -> proc_name.pname[i] = ' ';

        for (i=0; i<PROBE_LIMIT; i++)
             (process -> mcs_probed)[i] = EMPTY;

        process -> n_probes = 0;
        process -> schedulable = FALSE;
        process -> upid = EMPTY;
        process -> residency = EMPTY;
        process -> orig_mc = EMPTY;
        process -> times.exec_time = 0;
        process -> times.exec_here_time = 0;
        process -> times.residency_time = 0;
        process -> times.exist_time = 0;
        process -> migration_siz = sizeof(PROC_ENTRY);
        process -> siz = 10000;    /* TEMPORARY */
        process -> tot_msg_siz = 0;
        process -> blocked = UNBLOCKED;
        process -> preferred_mc = this_mc;

        for (i=0; i<MAXMCS; i++)
             process -> level_of_preference[i] = 0;

        for (i=0; i<MAXOWNPRT; i++)
             process -> owned_ports[i] = (PORT_ENTRY*) EMPTY;

        process -> ownp_length = 0;
        process -> nxt_ownp = 0;
        process -> parent.pgroup.gmc = EMPTY;
        process -> parent.pgroup.gnum = EMPTY;

        for (i=0; i<MAXPNAME; i++)
             process -> parent.pname[i] = ' ';

} /* End of PROC_INITDS */
```

```
/***** EXIT_DUMP *****/

/* This routine dumps a process's info when it exits */

exit_dump (proc)

PARAMS  PROC_ENTRY  *proc;

{


        cumul_exist_time += proc -> times.exist_time;
        n_deaths ++;
# ifdef DEBUG
        fprintf (trace, "\t\t\t\t\t\t\t\t\t\tAve RT = %f\n",
cumul_exist_time/(double)n_deaths);
        fprintf (trace, "Process died at %f \n", sys_real_time);
        fprintf (trace, "-------------------------------------\n\n");

        fprintf (trace, "Name  %d %d %s \n",
                        proc -> proc_name.pgroup.gmc,
                        proc -> proc_name.pgroup.gnum,
                        proc -> proc_name.pname);
        fprintf (trace, "Exec %d  Exist %d\n",
                         proc -> times.exec_time,
                         proc -> times.exist_time);

        fprintf (trace, "Response Ratio = %f\n\n\n",
                       (float)(proc -> times.exec_time) / (float)(proc ->
times.exist_time));
# endif

} /* End of EXIT_DUMP */
```

```
/* FILE: project/src/utils/utils.c */

/* This file contains a number of routines common to */
/* many modules */

/* Includes for this file */

# include <stdio.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"




/***** SEIZE *****/

/* This routine provides exclusive access to an object */

seize (resource)

PARAMS char resource[];

{

        while (link(LOCKFNAME, resource) < 0)
        {
                alarm(3);
                pause();
                alarm(0);
        }

} /* End of SEIZE */




/***** RELEASE *****/

/* This routine releases a seized object */

release (resource)

PARAMS char resource[];

{

        unlink (resource);

} /* End of RELEASE */
```

```
/***** CLS *****/

/* This routine clears the screen */

cls()
{

        printf ("\033[H\033[2J");

} /* End of CLS */




/***** PREAD *****/

/* This routine reads from a given pipe into a given buffer */

pread (buffer, n_bytes, pfd)

PARAMS  char  *buffer;
        int    n_bytes;
        int    pfd;

{

LOCAL  int  this_read = 0;
LOCAL  int  n_read = 0;

        while (n_read < n_bytes)
        {
                this_read = read (pfd, buffer+n_read, n_bytes-n_read);
                if (this_read == FAIL) ERROR (PIPE_READ);
                n_read += this_read;
        }

} /* End of PREAD */




/***** PWRITE *****/

/* This routine writes from a given buffer to a given pipe */
```

```
pwrite (pfd, buffer, n_bytes)

PARAMS  int  pfd;
        char *buffer;
        int  n_bytes;

{

LOCAL  int  this_write = 0;
LOCAL  int  n_written = 0;

        while (n_written < n_bytes)
        {
                this_write = write (pfd, buffer+n_written, n_bytes-n_written);
                if (this_write == FAIL) ERROR (PIPE_WRITE);
                n_written += this_write;
        }

} /* End of PWRITE */
```

DIRECTORY NAME : USRF

```
/* FILE: project/src/usrf/ERROR.c */

# include <stdio.h>
# include "/usr/acct/ian/project/src/include/keywds.h"
# include "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include "/usr/acct/ian/project/src/include/params.h"
# include "/usr/acct/ian/project/src/include/sys/types.h"
# include <errno.h>

EXTERN  PROCN mypid;
EXTERN  int mc_id;
EXTERN  int scheduled;
EXTERN  int got_contsig;
EXTERN  int got_kcrsig;


ERROR (errcode)

PARAMS  int errcode;

{

        if (errcode == PIPE_READ)
        {
            printf ("ERROR in pipe read\n");
            printf ("errno %d\n", errno);
            printf ("orig mc %d\n", mypid.pgroup.gmc);
            printf ("now on %d\n", mc_id);
            printf ("sched %d\n", scheduled);
            printf ("Cont %d\n", got_contsig);
            printf ("Kcrsig %d\n", got_kcrsig);
            exit (-1);
        }
        else
            if (errcode == PIPE_WRITE)
            {
                printf ("ERROR in pipe write\n");
                exit (-1);
            }
        else
            if (errcode == PIPE_OPEN)
            {
                printf ("ERROR on pipe open\n");
                exit (-1);
            }
        else
           {
            printf ("Unknown ERROR\n");
            exit (-1);
           }

}
```

```
/* FILE: project/src/usrf/kcalls.c */

/* This file contains the user kernel call routines */

/* Includes for this file */

# include   <stdio.h>
# include   <signal.h>
# include   "/usr/acct/ian/project/src/include/keywds.h"
# include   "/usr/acct/ian/project/src/include/params.h"
# include   "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include   "/usr/acct/ian/project/src/include/sys/types.h"
# include   "/usr/acct/ian/project/src/include/sys/macros.h"

/* Routines EXTERNal to this file */

EXTERN  char *malloc();


/* Global error no and got signal flag */

int  err;
int  got_kcrsig = 0;
int  got_contsig = 0;
int  scheduled = 0;



/***************************************************************************/


/***** CPROC *****/

/* Typedefs for this routine */

typedef struct {
                  char  pname[MAXPNAME];
                  char  pfile[MAXFNAME];
                  int   idr;

                } KC_CPROC;


/* This is the routine for creating a process */

PROCN *cproc (pname, pfile,idr)

PARAMS  char  *pname;
        char  *pfile;
        int   idr;

{

LOCAL  KC_CPROC  p_blk;
LOCAL  char  *uproc_name;
LOCAL  int    result;
```

```
        /* Set up parameter block */
        strcpy (p_blk.pname, pname);
        strcpy (p_blk.pfile, pfile);
        p_blk.idr = idr;

        /* Make the call */
        make_call (CPROC_KC, sizeof(KC_CPROC), (char *)&p_blk);

        /* Pause waiting for kcall return */
/*      await_sig (); */

        /* Get result of kcall */
        uproc_name = malloc (sizeof(PROCN));
        result = get_kcret (CPROC_KC, uproc_name);

        /* Return result to user */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return  (PROCN *)FAIL;
        }
        else
            return (PROCN *)uproc_name;

} /* End of CPROC */




/****************************************************************************/

/***** CPORT *****/

/* Typedefs for this routine */

typedef  struct {
                    char  port_name[MAXPNAME];
                    int   msg_type;
                    PFI   destruct;
                    PFI   rcvfunc;

                  } KC_CPORT;


cport (port_name, msg_type, destruct, rcvfunc)

PARAMS  char  port_name[];
        int   msg_type;
        PFI   destruct;
        PFI   rcvfunc;

{

LOCAL  KC_CPORT  p_blk;
LOCAL  int       result;
```

```
        /* Set up parameter block */
        strcpy (p_blk.port_name, port_name);
        p_blk.msg_type = msg_type;
        p_blk.destruct = destruct;
        p_blk.rcvfunc = rcvfunc;

        /* Make the call */
        make_call (CPORT_KC, sizeof(KC_CPORT), (char *)&p_blk);

        /* Pause waiting for kcall return */
        /*await_sig (); */

        /* Get result of kcall */
        result = get_kcret (CPORT_KC, NULL);

        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return FAIL;
        }
        else return (result & ~KCSUCC_MASK);

} /* End of CPORT */






/****************************************************************************/

/***** LPORT *****/

/* This routine makes a kcall to link a port */

/* Typedefs for this routine */

typedef  struct {
                  int     lf_index;
                  char    lt_name[MAXPNAME];

                } KC_LPORT;



lport (lf_index, lt_name)

PARAMS  int  lf_index;
        char lt_name[];

{

LOCAL  KC_LPORT  p_blk;
LOCAL  int       result;
```

```
        /* Set up parameter block */
        p_blk.lf_index = lf_index;
        strcpy (p_blk.lt_name, lt_name);

        /* Make the call */
        make_call (LPORT_KC, sizeof(KC_LPORT), (char *)&p_blk);

        /* Pause waiting for kcall return */
        /*await_sig (); */

        /* Get result of kcall */
        result = get_kcret (LPORT_KC, NULL);

        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return FAIL;
        }
        else return (result & ~KCSUCC_MASK);

} /* End of LPORT */




/****************************************************************************/

/***** NB_RMSG *****/

/* This routine makes a kcall to non-blocking rcv a msg */

/* Typedefs for this routine */

typedef  struct {
                  int  port;
                  MSG  *msg_loc;

                } KC_NBRMSG;


nb_rmsg (port, msg_loc)

PARAMS  int  port;
        MSG  **msg_loc;

{

LOCAL  KC_NBRMSG  p_blk;
LOCAL  int        result;
EXTERN int        sigmsg_handler();

        /* Allocate space for MSG */
        *msg_loc = (MSG *)malloc(sizeof(MSG));

        /* Set up parameter block */
        p_blk.port = port;
        p_blk.msg_loc = *msg_loc;

        /* Set up sigmsg handler routine */
        signal (SIGMSG, sigmsg_handler);

        /* Make the call */
        make_call (NBRMSG_KC, sizeof(KC_NBRMSG), (char *)&p_blk);

        /* Pause waiting for kcall return */
        /*await_sig (); */


        /* Get result of kcall */
        result = get_kcret (NBRMSG_KC, NULL);


        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return FAIL;
        }
        else
        {
            return SUCCESS;
        }

} /* End of NB_RMSG */





/****************************************************************************/

/***** B_RMSG *****/

/* This routine makes a kcall to blocking rcv a msg */

/* Typedefs for this routine */

typedef  struct {
                   int  port;

                 } KC_BRMSG;


MSG *b_rmsg (port)

PARAMS  int  port;

{

LOCAL  KC_BRMSG  p_blk;
LOCAL  int       result;
LOCAL  MSG        *msg;

        /* Allocate space for MSG */
        if ((msg = (MSG *)malloc (sizeof(MSG))) == NULL)
             printf ("Malloc failed\n");;

        /* Set up parameter block */
        p_blk.port = port;

        /* Make the call */
        make_call (BRMSG_KC, sizeof(KC_BRMSG), (char *)&p_blk);

        /* Get result of kcall */
        result = get_kcret (BRMSG_KC, (char *)msg);

        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            free ((char *)msg);
            return  (MSG *)FAIL;
        }
        else
            return msg;

} /* End of B_RMSG */




/**************************************************************************/

/***** NB_SMSG *****/

/* This routine makes a kcall to non-blocking snd a msg */

/* Typedefs for this routine */

typedef  struct {
                 int   sport;
                 int   dport;
                 int   msg_length;
                 int   msg_type;
                 char  msg_txt[MSG_SIZ];

               } KC_NBSMSG;


nb_smsg (sport, dport, msg_length, msg_type, msg_txt)

PARAMS  int   sport;
        int   dport;
        int   msg_length;
        int   msg_type;
        char  msg_txt[];
```

```
{

LOCAL  KC_NBSMSG  p_blk;
LOCAL  int        result;

        /* Set up parameter block */
        p_blk.sport = sport;
        p_blk.dport = dport;
        p_blk.msg_length = msg_length;
        p_blk.msg_type = msg_type;
        strcpy (p_blk.msg_txt, msg_txt);

        /* Make the call */
        make_call (NBSMSG_KC, sizeof(KC_NBSMSG), (char *)&p_blk);

        /* Get result of call */
        result = get_kcret (NBSMSG_KC, NULL);

        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return FAIL;
        }
        else
            return SUCCESS;

} /* End of NB_SMSG */




/*************************************************************************/

/***** DPORT *****/

/* This routine makes a kcall to destroy a port */

/* Typedefs for this routine */

typedef  struct  {
                      int  dp_index;

                 } KC_DPORT;


dport (dp)

PARAMS   int  dp;

{

LOCAL  KC_DPORT  p_blk;
LOCAL  int       result;
LOCAL  PFI       destruct;
```

```
        /* Set up parameter block */
        p_blk.dp_index = dp;

        /* Make the call */
        make_call (DPORT_KC, sizeof(KC_DPORT), (char *)&p_blk);

        /* Get result of kcall */
        result = get_kcret (DPORT_KC, (char *)&destruct);

        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return  FAIL;
        }
        else if (destruct != (PFI)EMPTY)
             {
                 (*destruct)();
                 return SUCCESS;
             }
        else
             {
                 return  SUCCESS;
             }

} /* End of DPORT */




/***************************************************************************/

/***** UPORT *****/

/* This routine makes a kcall to unlink a port */

/* Typedefs for this routine */

typedef  struct  {
                      int  lf_index;
                      int  lt_index;

                 } KC_UPORT;

uport (lf, lt)

PARAMS  int  lf;
        int  lt;

{

LOCAL  KC_UPORT  p_blk;
LOCAL  int       result;

        /* Set up parameter block */
        p_blk.lf_index = lf;
        p_blk.lt_index = lt;
        /* Make the call */
        make_call (UPORT_KC, sizeof(KC_UPORT), (char *)&p_blk);

        /* Get result of kcall */
        result = get_kcret (UPORT_KC, NULL);

        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return  FAIL;
        }
        else  return  SUCCESS;

} /* End of UPORT */




/*****************************************************************************/

/***** EXIT_PROC *****/

/* This routine makes a kcall to exit a process */

/* Typedefs for this routine */

typedef  struct {
                  int  retval;
                } KC_EXIT;

exit_proc(exit_val)

PARAMS  int  exit_val;

{

LOCAL  KC_EXIT  p_blk;
LOCAL  int      result;

        /* Set up parameter block */
        p_blk.retval = exit_val;

        /* Make the call */
        make_call (EPROC_KC, sizeof(KC_EXIT), (char *)&p_blk);

        /* Get result of kcall */
        result = get_kcret(EPROC_KC, NULL);

        /* Return result to caller */
        if (result & KCFAIL_MASK)
        {
            err = result & ~KCFAIL_MASK;
            return FAIL;
        }
        else exit(0);

} /* End of EXIT_PROC */




/*****************************************************************************/

/***** DO_PROCESSING *****/

/* This routine simulates an amount of processing */

/* Typedefs for this routine */

typedef  struct {
                  int  ninst;
                } KC_DOPROC;

do_processing (ninst)

PARAMS  int  ninst;

{

LOCAL  KC_DOPROC  p_blk;
LOCAL  int        result;

        /* Set up parameter block */
        p_blk.ninst = ninst;

        /* Make the call */
        make_call (DOPROC_KC, sizeof(KC_DOPROC), (char *)&p_blk);

        /* Get result of call */
        result = get_kcret (DOPROC_KC, NULL);

        /* Return to caller */
        return SUCCESS;

} /* End of DO_PROCESSING */




/*****************************************************************************/
```

```
/* FILE: project/src/usrf/interface.c */

/* This file contains routines for interfacing with kernel call routines */

/* Includes for this file */

# include  <stdio.h>
# include  <signal.h>
# include  <errno.h>
# include  <fcntl.h>
# include  "/usr/acct/ian/project/src/include/keywds.h"
# include  "/usr/acct/ian/project/src/include/params.h"
# include  "/usr/acct/ian/project/src/include/sys/errcodes.h"
# include  "/usr/acct/ian/project/src/include/sys/types.h"
# include  "/usr/acct/ian/project/src/include/sys/macros.h"

/* Routines EXTERNal to this file */

EXTERN  char  *malloc();

/* Got signal flags */

EXTERN int  got_kcrsig;
EXTERN int  got_contsig;
EXTERN int  scheduled;


/* Hidden static variables */

static  int  kcmk_pfd;
static  int  kcrt_pfd;
int  mc_id;
PROCN  mypid;
static  char kcmk_lock [OPLOCKSIZ];
static  char kcrt_lock [KCRLOCKSIZ];
static  int  uppid;
static  char opname[OPNAMSIZ];
static  char kcrname[KCRNAMSIZ];



/***** SET_UP *****/

/* This routine sets up a usr process's environment */

set_up (args)

PARAMS  char  *args[];

{

EXTERN  int  kcrsig_handler();
EXTERN  int  sigmig_handler();
EXTERN  int  sigcont_handler();
EXTERN  int  sigsched_handler();
EXTERN  int  alarm_handler();

        /* Get info from argv concerning making kcalls */
        sscanf (args[1], "%d", &mc_id);
        sscanf (args[2], "%d %d %s",
                         &(mypid.pgroup.gmc),
                         &(mypid.pgroup.gnum),
                           mypid.pname);
        sscanf (args[3], "%s", kcmk_lock);
        sscanf (args[4], "%s", kcrt_lock);
        sscanf (args[5], "%d", &uppid);

        /* Set up mc's own pipe name */
        sprintf (opname, "own_p%d", mc_id);

        /* Set up kcall return pipe name */
        sprintf (kcrname, "kcr_p%d", mc_id);

        /* Set up interrupt handlers */
        signal (SIGKCR, kcrsig_handler);
        signal (SIGMIG, sigmig_handler);
        signal (SIGCONT, sigcont_handler);
        signal (SIGSCHED, sigsched_handler);
        signal (SIGALRM, alarm_handler);

        /* Allow kernel to continue */
        kill (uppid, SIGSETUP);
#ifdef DEBUG
return mypid.pgroup.gmc;
#endif

} /* End of SET_UP */






/***** MAKE_CALL *****/

/* This routine sends a kcall to the kernel */

# define  snd_k(a,b)  pwrite(kcmk_pfd,a,b)

typedef  struct {
                  int   sport;
                  int   dport;
                  int   msg_length;
                  int   msg_type;
                  char  *msg_txt;

                } KC_NBSMSG;


make_call (kctype, kcsize, kcall)

PARAMS  int  kctype;
        int  kcsize;
        char *kcall;

{

LOCAL  COMMS_HDR  comms_hdr;

        /* Set up comms hdr */
        comms_hdr = kctype | KCALL_MASK;

        /* Wait to be scheduled */
        while (!scheduled)
        {
              alarm(3);
              pause();
              alarm(0);
        }
        scheduled = 0;

        got_kcrsig = 0;

        /* Seize "make kcall lock" */
         seize (kcmk_lock);

        /* Open mc's own pipe */
        if ((kcmk_pfd = open(opname, O_WRONLY, 0)) == FAIL)
             ERROR (PIPE_OPEN);

        /* Make the call by sending it to kernel */
        snd_k ((char *)&comms_hdr, sizeof(COMMS_HDR));
        snd_k ((char *)&mypid, sizeof(PROCN));
        snd_k (kcall, kcsize);

        /* Close pipe */
        close (kcmk_pfd);

        /* Release "make call" lock */
         release (kcmk_lock);

        /* Wait for return */
        while (!got_kcrsig)
        {
              alarm(3);
              pause();
              alarm(0);
        }

#       ifdef DEBUG
        printf ("%d - got kcrsig\n", getpid());
#       endif
} /* End of MAKE_CALL */




/***** GET_KCRET *****/
```

```
/* This routine gets the return from a kcall */

# define rcv_k(a,b) pread(a,b,kcrt_pfd)

KCR_HDR  get_kcret (kctype, res_loc)

PARAMS  int  kctype;
        char *res_loc;

{

LOCAL  KCR_HDR  kcr_hdr;

        /* Seize kcreturn lock for this mc */
/*      seize (kcrt_lock); */

        /* Open kcall return pipe */
        if ((kcrt_pfd = open (kcrname, O_RDONLY, 0)) == FAIL)
             ERROR (PIPE_OPEN);

        /* Get the header of the kcreturn */
        rcv_k((char *)&kcr_hdr, sizeof(KCR_HDR));

        /* If successful kcall, may need more returns */
        if (!(kcr_hdr & KCFAIL_MASK))
             switch (kctype)
             {
                 case  QPORT_KC:  rcv_k(res_loc, sizeof(QRET));
                                  break;

                 case  CPROC_KC:  rcv_k(res_loc, sizeof(PROCN));
                                  break;

                 case  BRMSG_KC:  rcv_k(res_loc, sizeof(MSG));
                                  break;

                 case  DPORT_KC:  rcv_k(res_loc, sizeof(PFI));
                                  break;


                 default:         break;

               }

        /* Close pipe */
        close (kcrt_pfd);

        /* Release kcreturn lock */
/*      release (kcrt_lock); */

#       ifdef DEBUG
        printf ("%d sending ack to %d\n", getpid(), uppid);
#       endif
        if((kill (uppid, SIGKCRACK)) == FAIL) {printf ("Sig failed\n");
                                          printf("errno %d\n", errno);}


        return kcr_hdr;

} /* End of GET_KCRET */
```

```
/***** KCRSIG_HANDLER *****/

kcrsig_handler ()

{

EXTERN  int  kcrsig_handler();

        signal (SIGKCR, kcrsig_handler);
#       ifdef DEBUG
        printf ("%d - kcrsighandler\n", getpid());
#       endif
        got_kcrsig ++;

}




/***** SIGMSG_HANDLER *****/

/* This is the interrupt routine for dealing with msg arrival */

sigmsg_handler()

{

EXTERN  int  sigmsg_handler ();
LOCAL   MSG  *msg_loc;
LOCAL   PFI  rcvfunc;

        /* Reset signal */
        signal (SIGMSG, sigmsg_handler);

        /* Cancel alarm */
        alarm(0);
        signal (SIGALRM, alarm_handler);

        /* Open kcall return pipe */
        if ((kcrt_pfd = open(kcrname, O_RDONLY, 0)) == FAIL)
             ERROR (PIPE_OPEN);


        /* Seize kcall return lock */
/*      seize (kcrt_lock);*/

        /* Get location of msg */
        rcv_k ((char *)&msg_loc, sizeof (MSG *));

        /* Get MSG */
        rcv_k ((char *)msg_loc, sizeof (MSG));

        /* Get rcvfunc */
        rcv_k ((char *)&rcvfunc, sizeof (PFI));

        /* Release kcall return lock */
/*      release (kcrt_lock); */

        /* Close pipe */
        close (kcrt_pfd);

        /* Interrupt kernel to continue */
        kill (uppid, SIGMSGACK);

        /* Call rcvfunc if non_NULL */
        if (rcvfunc != (PFI)EMPTY)
           (*rcvfunc)();

}




/***** SIGMIG_HANDLER *****/

sigmig_handler()
{

LOCAL   int  save_uppid;
EXTERN  int  sigmig_handler();
EXTERN  int  alarm_handler();

        signal (SIGMIG, sigmig_handler);

        /* save old uppid for acknowledgement */
        save_uppid = uppid;

        /* Cancel alarm and ensure that SIGALRM is caught */
        alarm(0);
        signal (SIGALRM, alarm_handler);

        /* Open kcall return pipe */
        if ((kcrt_pfd = open(kcrname, O_RDONLY, 0)) == FAIL)
            ERROR (PIPE_OPEN);

        /* Get new process info */
        pread ((char *)&mc_id, sizeof(int), kcrt_pfd);
        pread ((char *)&uppid, sizeof(int), kcrt_pfd);

        /* Close old mc's pipes */
        close (kcrt_pfd);

        /* Setup new kcall and return pipe names */
        sprintf (opname, "own_p%d", mc_id);
        sprintf (kcrname, "kcr_p%d", mc_id);
        sprintf (kcmk_lock, "op%d", mc_id);

        got_contsig = 0;
        /* Acknowledge process migration */
        kill (save_uppid, SIGMIGACK);

        /* Suspend until kernel ready to continue */
        while (!got_contsig)
        {
                alarm(3);
                pause();
                alarm(0);
        }

} /* End of SIGMIG_HANDLER */




/***** SIGCONT_HANDLER *****/

sigcont_handler()

{

EXTERN  int  sigcont_handler();

        signal (SIGCONT, sigcont_handler);

        got_contsig ++;

} /* End of SIGCONT_HANDLER */




/***** SIGSCHED_HANDLER *****/

sigsched_handler()

{

EXTERN  int  sigsched_handler();

        signal (SIGSCHED, sigsched_handler);

        scheduled ++;

} /* End of SIGSCHED_HANDLER */




/***** ALARM_HANDLER *****/

alarm_handler()

{
```

```
EXTERN  int  alarm_handler();

        signal (SIGALRM, alarm_handler);

} /* End of ALARM_HANDLER */
```