95

```
/*
**  NGPT − Next Generation POSIX Threading
**  Copyright (c) 2001 IBM Corporation <babt@us.ibm.com>
**
**  This file is part of NGPT, a non−preemptive thread scheduling
**  library which can be found at http://www.ibm.com/developer.
**
**  This library is free software; you can redistribute it and/or
**  modify it under the terms of the GNU Lesser General Public
**  License as published by the Free Software Foundation; either
**  version 2.1 of the License, or (at your option) any later version.
**
**  This library is distributed in the hope that it will be useful,
**  but WITHOUT ANY WARRANTY; without even the implied warranty of
**  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
**  Lesser General Public License for more details.
**
**  You should have received a copy of the GNU Lesser General Public
**  License along with this library; if not, write to the Free Software
**  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307
**  USA.
**
**  pthreadtypes.h: POSIX Thread ("Pthread") API for NGPT
*/
                            /* ''Only those who attempt the absurd
                                 can achieve the impossible.''
                                                  −− Unknown          */
#if !defined _BITS_TYPES_H && !defined _PTHREAD_H
# error "Never include <bits/pthreadtypes.h> directly; use <sys/types.h> instead."
#endif

#ifndef _BITS_PTHREADTYPES_H
#define _BITS_PTHREADTYPES_H    1

#define __need_schedparam
#undef sched_param
#include <bits/sched.h>



#ifndef _PTHREAD_QLOCK_DEFINED
typedef struct pth_qlock_st pth_qlock_t;
typedef struct pth_qlock_st pthread_lock_t;
struct pth_qlock_st {
    int             lock    __attribute__ ((aligned (16)));
    int             owner;
    int             count;
};
#define _PTHREAD_QLOCK_DEFINED
#endif

struct _pthread_fastlock {
    long int __status;
    int __spinlock;
};

#ifndef _PTHREAD_DESCR_DEFINED
/* Thread descriptors */
typedef struct pthread_st *_pthread_descr;
# define _PTHREAD_DESCR_DEFINED
#endif


#ifdef __USE_XOPEN2K
/* POSIX spinlock data type.  */
typedef volatile int pthread_spinlock_t;

/* POSIX barrier. */
typedef struct {
  struct _pthread_fastlock __ba_lock; /* Lock to guarantee mutual exclusion */
  int __ba_required;                  /* Threads needed for completion */
  int __ba_present;                   /* Threads waiting */
  _pthread_descr __ba_waiting;        /* Queue of waiting threads */
} pthread_barrier_t;

/* barrier attribute */
typedef struct {
  int __pshared;
} pthread_barrierattr_t;

#endif

/*
 * Unprotect namespace, so we can define our own variants now
 */
#undef pthread_t
#undef pthread_attr_t
#undef pthread_key_t
#undef pthread_once_t
#undef pthread_mutex_t
```

```
91  #undef pthread_mutexattr_t
92  #undef pthread_cond_t
93  #undef pthread_condattr_t
94  #undef pthread_rwlock_t
95  #undef pthread_rwlockattr_t
96
97  /*
98   * Forward structure definitions.
99   * These are mostly opaque to the application.
100  */
101 struct pthread_st;
102 struct pthread_attr_st;
103 struct pthread_cond_st {
104     void *  res0;
105     int     res1;
106     int     res2;
107 };
108 struct pthread_mutex_st {
109     void *  res0;
110     int     res1;
111     int     res2;
112     int     res3;
113     int     res4;
114     int     res5;
115 };
116 struct pthread_mutexattr_st;
117 struct pthread_rwlock_st;
118
119 /*
120  * Primitive system data type definitions required by P1003.1c
121  */
122 typedef struct  pthread_st              *pthread_t;
123 typedef struct  pthread_attr_st         *pthread_attr_t;
124 typedef int                              pthread_key_t;
125 typedef int                              pthread_once_t;
126 typedef struct  pthread_mutexattr_st    *pthread_mutexattr_t;
127 typedef struct  pthread_mutex_st         pthread_mutex_t;
128 typedef int                              pthread_condattr_t;
129 typedef struct  pthread_cond_st          pthread_cond_t;
130 typedef int                              pthread_rwlockattr_t;
131 typedef struct  pthread_rwlock_st       *pthread_rwlock_t;
132
133 #endif  /* bits/pthreadtypes.h */
```

```
/*
**  NGPT − Next Generation POSIX Threading
**  Copyright (c) 2001 IBM Corporation <babt@us.ibm.com>
**
**  This file is part of NGPT, a non−preemptive thread scheduling
**  library which can be found at http://www.ibm.com/developer.
**
**  This library is free software; you can redistribute it and/or
**  modify it under the terms of the GNU Lesser General Public
**  License as published by the Free Software Foundation; either
**  version 2.1 of the License, or (at your option) any later version.
**
**  This library is distributed in the hope that it will be useful,
**  but WITHOUT ANY WARRANTY; without even the implied warranty of
**  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
**  Lesser General Public License for more details.
**
**  You should have received a copy of the GNU Lesser General Public
**  License along with this library; if not, write to the Free Software
**  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307
**  USA.
**
**  semaphore.h: POSIX Thread ("Pthread") API for NGPT
*/
                              /* ‘‘Only those who attempt the absurd
                                   can achieve the impossible.’’
                                                 −− Unknown          */
#ifndef _SEMAPHORE_H
#define _SEMAPHORE_H    1

#include <features.h>
#include <sys/types.h>
#ifdef __USE_XOPEN2K
# define __need_timespec
# include <time.h>
#endif

#ifndef _PTHREAD_DESCR_DEFINED
typedef struct pthread_st *_pthread_descr;
# define _PTHREAD_DESCR_DEFINED
#endif

typedef struct _sem_lock_st *_sem_lock_t;
struct _sem_lock_st {
    pthread_lock_t      __sem_lock;
    pthread_mutex_t     __lock;
    pthread_cond_t      nonzero;
};

/* System specific semaphore definition.  */
typedef struct {
    struct _pthread_fastlock    __sem_lock;
    int                         __sem_value;
    _sem_lock_t                 __sem_waiting;
} sem_t;



/* Value returned if ‘sem_open’ failed.  */
#define SEM_FAILED      ((sem_t *) 0)

/* Maximum value the semaphore can have.  */
#define SEM_VALUE_MAX   ((int) ((~0u) >> 1))


__BEGIN_DECLS

/* Initialize semaphore object SEM to VALUE.  If PSHARED then share it
   with other processes.  */
extern int sem_init (sem_t *__sem, int __pshared, unsigned int __value) __THROW;

/* Free resources associated with semaphore object SEM.  */
extern int sem_destroy (sem_t *__sem) __THROW;

/* Open a named semaphore NAME with open flaot OFLAG.  */
extern sem_t *sem_open (__const char *__name, int __oflag, ...) __THROW;

/* Close descriptor for named semaphore SEM.  */
extern int sem_close (sem_t *__sem) __THROW;

/* Remove named semaphore NAME.  */
extern int sem_unlink (__const char *__name) __THROW;

/* Wait for SEM being posted.  */
extern int sem_wait (sem_t *__sem) __THROW;

#ifdef __USE_XOPEN2K
/* Similar to ‘sem_wait’ but wait only until ABSTIME.  */
extern int sem_timedwait (sem_t *__restrict __sem,
                          __const struct timespec *__restrict __abstime)
```

```
91        __THROW;
92  #endif
93
94  /* Test whether SEM is posted.  */
95  extern int sem_trywait (sem_t *__sem) __THROW;
96
97  /* Post SEM.  */
98  extern int sem_post (sem_t *__sem) __THROW;
99
100 /* Get current value of SEM and store it in *SVAL.  */
101 extern int sem_getvalue (sem_t *__restrict __sem, int *__restrict __sval)
102      __THROW;
103
104 __END_DECLS
105
106 #endif  /* semaphore.h */
```

```
/*
 *
 *   Copyright (c) International Business Machines  Corp., 2001
 *
 *   This program is free software;  you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY;  without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
 *   the GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program;  if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */

/*
 *  FILE        : pth_str01.c
 *  DESCRIPTION : create a tree of threads
 *  HISTORY:
 *    04/09/2001 Paul Larson (plars@us.ibm.com)
 *      -Ported
 *
 */

#ifdef GLOBAL
#include <pthread.h>
#else
#define _PTHREAD_PRIVATE
#include "pthread.h"
#endif


#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include "test_str.h"

int     depth = 3;
int     breadth = 4;
int     timeout = 30;                   /* minutes */
int     cdepth;                         /* current depth */
int     debug = 0;

c_info          *child_info;            /* pointer to info array */
int             node_count;             /* number of nodes created so far */
pthread_mutex_t node_mutex;             /* mutex for the node_count */
pthread_cond_t  node_condvar;           /* condition variable for node_count */
pthread_attr_t  attr;                   /* attributes for created threads */

int num_nodes(int, int);
int synchronize_children(c_info *);
int doit(c_info *);

/*
 * parse_args
 *
 * Parse command line arguments.  Any errors cause the program to exit
 * at this point.
 */
static void     parse_args( int argc, char *argv[] )
{
        int             opt, errflag = 0;
        int             bflag = 0, dflag = 0, tflag = 0;

        while ( (opt = getopt( argc, argv, "b:d:t:Dh?" )) != EOF ) {
            switch ( opt ) {
                case 'b':
                    if ( bflag )
                        errflag++;
                    else {
                        bflag++;
                        breadth = atoi( optarg );
                        if ( breadth <= 0 )
                            errflag++;
                    }
                    break;
                case 'd':
                    if ( dflag )
                        errflag++;
                    else {
                        dflag++;
                        depth = atoi( optarg );
                        if ( depth <= 0 )
```

```
                                errrflag++;
                        }
                        break;
                    case 't':
                        if ( tflag )
                            errflag++;
                        else {
                            tflag++;
                            timeout = atoi( optarg );
                            if ( timeout <= 0 )
                                errflag++;
                        }
                        break;
                    case 'D':
                        debug = 1;
                        break;
                    case 'h':
                    default:
                        errflag++;
                        break;
                }
            }

            if ( errflag ) {
                    fprintf( stderr, "usage: %s [−b <num>] [−d <num>] [−t <num>] [−D]", argv[0] );
                    fprintf( stderr, " where:\n" );
                    fprintf( stderr, "\t−b <num>\tbreadth of child nodes\n" );
                    fprintf( stderr, "\t−d <num>\tdepth of child nodes\n" );
                    fprintf( stderr, "\t−t <num>\ttimeout for child communication (in minutes)\n" );
                    fprintf( stderr, "\t−D\t\tdebug mode on\n" );
                    exit( 1 );
            }

    }
```

```
126 /*
127  * num_nodes
128  *
129  * Caculate the number of child nodes for a given breadth and depth tree.
130  */
131 int     num_nodes( int b, int d )
132 {
133         int     n, sum = 1, partial_exp = 1;
134
135         /*
136          * The total number of children = sum ( b ** n )
137          *                                n=0->d
138          * Since b ** 0 == 1, and it's hard to compute that kind of value
139          * in this simplistic loop, we start sum at 1 (above) to compensate
140          * and do the computations from 1->d below.
141          */
142         for ( n = 1; n <= d; n++ ) {
143                 partial_exp *= b;
144                 sum += partial_exp;
145         }
146
147         return( sum );
148 }
149
```

```
150 /*
151  * synchronize_children
152  *
153  * Register the child with the parent and then wait for all of the children
154  * at the same level to register also.  Return when everything is synched up.
155  */
156 int     synchronize_children( c_info *parent ) {
157         int             my_index, rc;
158         c_info          *info_p;
159         struct timespec timer;
160
161         if ( debug ) {
162             printf( "trying to lock node_mutex\n" );
163             fflush( stdout );
164         }
165
166         /*
167          * Lock the node_count mutex to we can safely increment it.  We
168          * will unlock it when we broadcast that all of our siblings have
169          * been created or when we block waiting for that broadcast.
170          */
171         pthread_mutex_lock( &node_mutex );
172         my_index = node_count++;
173
174         printf( "thread %d started\n", my_index );
175         fflush( stdout );
176
177         /*
178          * Get a pointer into the array of thread structures which will
179          * be "me".
180          */
181         info_p = &child_info[my_index];
182         info_p->index = my_index;
183
184         if ( debug ) {
185             printf( "thread %d info_p=%x\n", my_index, (unsigned int)info_p );
186             fflush( stdout );
187         }
188
189         /*
190          * Register with parent bumping the parent's child_count variable.
191          * Make sure we have exclusive access to that variable before we
192          * do the increment.
193          */
194         if ( debug ) {
195             printf( "thread %d locking child_mutex %x\n", my_index,
196               (unsigned int)&parent->child_mutex );
197             fflush( stdout );
198         }
199         pthread_mutex_lock( &parent->child_mutex );
200         if ( debug ) {
201             printf( "thread %d bumping child_count (currently %d)\n",
202               my_index, parent->child_count );
203             fflush( stdout );
204         }
205         parent->child_ptrs[parent->child_count++] = info_p;
206         if ( debug ) {
207             printf( "thread %d unlocking child_mutex %x\n", my_index,
208               (unsigned int)&parent->child_mutex );
209             fflush( stdout );
210         }
211         pthread_mutex_unlock( &parent->child_mutex );
212
213         if ( debug ) {
214             printf( "thread %d node_count = %d\n", my_index, node_count );
215             printf( "expecting %d nodes\n", num_nodes(breadth, cdepth) );
216             fflush( stdout );
217         }
218
219         /*
220          * Have all the nodes at our level in the tree been created yet?
221          * If so, then send out a broadcast to wake everyone else up (to let
222          * them know they can now create their children (if they are not
223          * leaf nodes)).  Otherwise, go to sleep waiting for the broadcast.
224          */
225         if ( node_count == num_nodes(breadth, cdepth) ) {
226
227             /*
228              * Increase the current depth variable, as the tree is now
229              * fully one level taller.
230              */
231             if ( debug ) {
232                 printf( "thread %d doing cdepth++ (%d)\n", my_index, cdepth );
233                 fflush( stdout );
234             }
235             cdepth++;
236
237             if ( debug ) {
238                 printf( "thread %d sending child_mutex broadcast\n", my_index );
239                 fflush( stdout );
```

```
240                 }
241                 /*
242                  * Since all of our siblings have been created, this level of
243                  * the tree is now allowed to continue its work, so wake up the
244                  * rest of the siblings.
245                  */
246                 pthread_cond_broadcast( &node_condvar );
247
248             } else {
249
250                 /*
251                  * In this case, not all of our siblings at this level of the
252                  * tree have been created, so go to sleep and wait for the
253                  * broadcast on node_condvar.
254                  */
255                 if ( debug ) {
256                     printf( "thread %d waiting for siblings to register\n",
257                       my_index );
258                     fflush( stdout );
259                 }
260                 time( &timer.tv_sec );
261                 timer.tv_sec += (unsigned long)timeout * 60;
262                 timer.tv_nsec = (unsigned long)0;
263                 if ((rc = pthread_cond_timedwait(&node_condvar, &node_mutex,
264                   &timer))) {
265                     fprintf( stderr, "pthread_cond_timedwait (sync) %d: %s\n",
266                         my_index, sys_errlist[rc] );
267                     exit( 2 );
268                 }
269
270                 if ( debug ) {
271                     printf( "thread %d is now unblocked\n", my_index );
272                     fflush( stdout );
273                 }
274
275             }
276
277             /*
278              * Unlock the node_mutex lock, as this thread is finished
279              * initializing.
280              */
281             if ( debug ) {
282                 printf( "thread %d unlocking node_mutex\n", my_index );
283                 fflush( stdout );
284             }
285             pthread_mutex_unlock( &node_mutex );
286             if ( debug ) {
287                 printf( "thread %d unlocked node_mutex\n", my_index );
288                 fflush( stdout );
289             }
290
291             if ( debug ) {
292                 printf( "synchronize_children returning %d\n", my_index );
293                 fflush( stdout );
294             }
295
296             return( my_index );
297
298     }
299
```

```
300 /*
301  * doit
302  *
303  * Do it.
304  */
305 int     doit( c_info *parent ) {
306         int             rc, child, my_index;
307         void            *status;
308         c_info          *info_p;
309         struct timespec timer;
310
311         if ( debug ) {
312             printf( "parent=%#010x\n", (unsigned int)parent );
313             fflush( stdout );
314         }
315
316         if ( parent != NULL ) {
317             /*
318              * Synchronize with our siblings so that all the children at
319              * a given level have been created before we allow those children
320              * to spawn new ones (or do anything else for that matter).
321              */
322             if ( debug ) {
323                 printf( "non−root child calling synchronize_children\n" );
324                 fflush( stdout );
325             }
326             my_index = synchronize_children( parent );
327             if ( debug ) {
328                 printf( "non−root child has been assigned index %d\n",
329                   my_index );
330                 fflush( stdout );
331             }
332         } else {
333             /*
334              * The first thread has no one with which to synchronize, so
335              * set some initial values for things.
336              */
337             if ( debug ) {
338                 printf( "root child\n" );
339                 fflush( stdout );
340             }
341             cdepth = 1;
342             my_index = 0;
343             node_count = 1;
344         }
345
346         /*
347          * Figure out our place in the pthread array.
348          */
349         info_p = &child_info[my_index];
350
351         if ( debug ) {
352             printf( "thread %d getting to heart of doit.\n", my_index );
353             printf( "info_p=%x, cdepth=%d, depth=%d\n", (unsigned int)info_p, cdepth, depth );
354             fflush( stdout );
355         }
356
357         if ( cdepth <= depth ) {
358
359             /*
360              * Since the tree is not yet complete (it is not yet tall enough),
361              * we need to create another level of children.
362              */
363
364             printf( "thread %d creating kids, cdepth=%d\n", my_index, cdepth );
365             fflush( stdout );
366
367             /*
368              * Create breadth children.
369              */
370             for ( child = 0; child < breadth; child++ ) {
371                 if ( debug ) {
372                     printf( "thread %d making child %d, ptr=%x\n", my_index,
373                       child, (unsigned int)&(info_p−>threads[child]) );
374                     fflush( stdout );
375                 }
376                 if ((rc = pthread_create(&(info_p−>threads[child]), &attr,
377                   (void *)doit, (void *)info_p))) {
378                     fprintf( stderr, "pthread_create (doit): %s\n",
379                       sys_errlist[rc] );
380                     exit( 3 );
381                 } else {
382                     if ( debug ) {
383                         printf( "pthread_create made thread %x\n",
384                           (unsigned int)&(info_p−>threads[child]) );
385                         fflush( stdout );
386                     }
387                 }
388             }
389
```

```
390             if ( debug ) {
391                 printf( "thread %d waits on kids, cdepth=%d\n", my_index,
392                     cdepth );
393                 fflush( stdout );
394             }
395 
396             /*
397              * Wait for our children to finish before we exit ourselves.
398              */
399             for ( child = 0; child < breadth; child++ ) {
400                 if ( debug ) {
401                     printf( "attempting join on thread %x\n",
402                       (unsigned int)&(info_p->threads[child]) );
403                     fflush( stdout );
404                 }
405                 if ((rc = pthread_join((info_p->threads[child]), &status))) {
406                     if ( debug ) {
407                         fprintf( stderr,
408                           "join failed on thread %d, status addr=%x: %s\n",
409                           my_index, (unsigned int)status, sys_errlist[rc] );
410                         fflush( stderr );
411                     }
412                     exit( 4 );
413                 } else {
414                     if ( debug ) {
415                         printf( "thread %d joined child %d ok\n", my_index,
416                           child );
417                         fflush( stdout );
418                     }
419                 }
420             }
421 
422         } else {
423 
424             /*
425              * This is the leaf node case.  These children don't create
426              * any kids; they just talk amongst themselves.
427              */
428             printf( "thread %d is a leaf node, depth=%d\n", my_index, cdepth );
429             fflush( stdout );
430 
431             /*
432              * Talk to siblings (children of the same parent node).
433              */
434             if ( breadth > 1 ) {
435 
436                 for ( child = 0; child < breadth; child++ ) {
437                     /*
438                      * Don't talk to yourself.
439                      */
440                     if ( parent->child_ptrs[child] != info_p ) {
441                         if ( debug ) {
442                             printf( "thread %d locking talk_mutex\n",
443                               my_index );
444                             fflush( stdout );
445                         }
446                         pthread_mutex_lock(
447                           &(parent->child_ptrs[child]->talk_mutex) );
448                         if ( ++parent->child_ptrs[child]->talk_count
449                           == (breadth - 1) ) {
450                             if ( debug ) {
451                                 printf( "thread %d talk siblings\n", my_index );
452                                 fflush( stdout );
453                             }
454                             if ((rc = pthread_cond_broadcast(
455                               &parent->child_ptrs[child]->talk_condvar))) {
456                                 fprintf( stderr, "pthread_cond_broadcast: %s\n",
457                                   sys_errlist[rc] );
458                                 exit( 5 );
459                             }
460                         }
461                         if ( debug ) {
462                             printf( "thread %d unlocking talk_mutex\n",
463                               my_index );
464                             fflush( stdout );
465                         }
466                         pthread_mutex_unlock(
467                           &(parent->child_ptrs[child]->talk_mutex) );
468                     }
469                 }
470 
471                 /*
472                  * Wait until the breadth - 1 siblings have contacted us.
473                  */
474                 if ( debug ) {
475                     printf( "thread %d waiting for talk siblings\n",
476                       my_index );
477                     fflush( stdout );
478                 }
479 
```

```
480                     pthread_mutex_lock( &info_p->talk_mutex );
481                     if ( info_p->talk_count < (breadth - 1) ) {
482                         time( &timer.tv_sec );
483                         timer.tv_sec += (unsigned long)timeout * 60;
484                         timer.tv_nsec = (unsigned long)0;
485                         if ((rc = pthread_cond_timedwait(&info_p->talk_condvar,
486                           &info_p->talk_mutex, &timer))) {
487                             fprintf( stderr,
488                               "pthread_cond_timedwait (leaf) %d: %s\n",
489                               my_index, sys_errlist[rc] );
490                             exit( 6 );
491                         }
492                     }
493                     pthread_mutex_unlock( &info_p->talk_mutex );
494 
495                 }
496 
497             }
498 
499             /*
500              * Our work is done.  We're outta here.
501              */
502             printf( "thread %d exiting, depth=%d, status=%d, addr=%x\n", my_index,
503               cdepth, info_p->status, (unsigned int)info_p);
504             fflush( stdout );
505 
506             pthread_exit( 0 );
507 
508             /*NOTREACHED*/
509             return 0;
510 
511 }
512 
```

```
/*
 * main
 */
int     main( int argc, char *argv[] ) {
        int             rc, ind, total;
        pthread_t       root_thread;

        parse_args( argc, argv );

        /*
         * Initialize node mutex.
         */
        if ((rc = pthread_mutex_init(&node_mutex, NULL))) {
                fprintf( stderr, "pthread_mutex_init(node_mutex): %s\n",
                    sys_errlist[rc] );
                exit( 7 );
        }

        /*
         * Initialize node condition variable.
         */
        if ((rc = pthread_cond_init(&node_condvar, NULL))) {
                fprintf( stderr, "pthread_cond_init(node_condvar): %s\n",
                    sys_errlist[rc] );
                exit( 8 );
        }

        /*
         * Allocate pthread info structure array.
         */
        total = num_nodes( breadth, depth );
        printf( "Allocating %d nodes.\n", total );
        fflush( stdout );
        if ( (child_info = (c_info *)malloc( total * sizeof(c_info) ))
            == NULL ) {
                perror( "malloc child_info" );
                exit( 10 );
        }
        bzero( child_info, total * sizeof(c_info) );

        if ( debug ) {
                printf( "Initializing array for %d children\n", total );
                fflush( stdout );
        }

        /*
         * Allocate array of pthreads descriptors and initialize variables.
         */
        for ( ind = 0; ind < total; ind++ ) {

                if ( (child_info[ind].threads =
                    (pthread_t *)malloc( breadth * sizeof(pthread_t) ))
                    == NULL ) {
                        perror( "malloc threads" );
                        exit( 11 );
                }
                bzero( child_info[ind].threads, breadth * sizeof(pthread_t) );

                if ( (child_info[ind].child_ptrs =
                    (c_info **)malloc( breadth * sizeof(c_info *) )) == NULL ) {
                        perror( "malloc child_ptrs" );
                        exit( 12 );
                }
                bzero( child_info[ind].child_ptrs,
                    breadth * sizeof(c_info *) );

                if ((rc = pthread_mutex_init(&child_info[ind].child_mutex,
                    NULL))) {
                        fprintf( stderr, "pthread_mutex_init child_mutex: %s\n",
                            sys_errlist[rc] );
                        exit( 13 );
                }

                if ((rc = pthread_mutex_init(&child_info[ind].talk_mutex,
                    NULL))) {
                        fprintf( stderr, "pthread_mutex_init talk_mutex: %s\n",
                            sys_errlist[rc] );
                        exit( 14 );
                }

                if ((rc = pthread_cond_init(&child_info[ind].child_condvar,
                    NULL))) {
                        fprintf( stderr,
                            "pthread_cond_init child_condvar: %s\n",
                            sys_errlist[rc] );
                        exit( 15 );
                }

                if ((rc = pthread_cond_init(&child_info[ind].talk_condvar,
                    NULL))) {
```

```
603                             fprintf( stderr, "pthread_cond_init talk_condvar: %s\n",
604                                 sys_errlist[rc] );
605                             exit( 16 );
606                     }
607
608                     if ( debug ) {
609                             printf( "Successfully initialized child %d.\n", ind );
610                             fflush( stdout );
611                     }
612
613             }
614
615             printf( "Creating root thread attributes via pthread_attr_init.\n" );
616             fflush( stdout );
617
618             if ((rc = pthread_attr_init(&attr))) {
619                     fprintf( stderr, "pthread_attr_init: %s\n", sys_errlist[rc] );
620                     exit( 17 );
621             }
622
623             /*
624              * Make sure that all the thread children we create have the
625              * PTHREAD_CREATE_JOINABLE attribute.  If they don't, then the
626              * pthread_join call will sometimes fail and cause mass confusion.
627              */
628             if ((rc = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE))
629                ) {
630                     fprintf( stderr, "pthread_attr_setdetachstate: %s\n",
631                         sys_errlist[rc] );
632                     exit( 18 );
633             }
634
635             printf( "Creating root thread via pthread_create.\n" );
636             fflush( stdout );
637
638             if ((rc = pthread_create(&root_thread, &attr, (void *)doit, NULL))) {
639                     fprintf( stderr, "pthread_create: %s\n", sys_errlist[rc] );
640                     exit( 19 );
641             }
642
643             if ( debug ) {
644                     printf( "Doing pthread_join.\n" );
645                     fflush( stdout );
646             }
647
648             /*
649              * Wait for the root child to exit.
650              */
651             if (( rc = pthread_join(root_thread, NULL) )) {
652                     fprintf( stderr, "pthread_join: %s\n", sys_errlist[rc] );
653                     exit( 20 );
654             }
655
656             if ( debug ) {
657                     printf( "About to pthread_exit.\n" );
658                     fflush( stdout );
659             }
660
661             exit( 0 );
662     }
```

```
/*
 *
 *   Copyright (c) International Business Machines  Corp., 2001
 *
 *   This program is free software;  you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY;  without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
 *   the GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program;  if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 */

/*
 *  FILE        : pth_str01.c
 *  DESCRIPTION : Create n threads
 *  HISTORY:
 *    05/16/2001 Paul Larson (plars@us.ibm.com)
 *      −Ported
 *
 */

#ifdef GLOBAL
#include <pthread.h>
#else
#define _PTHREAD_PRIVATE
#include "pthread.h"
#endif

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/errno.h>


/* Defines
 *
 * DEFAULT_NUM_THREADS: Default number of threads to create,
 * user can specifiy with [−n] command line option.
 *
 * USAGE: usage statement
 */

#define DEFAULT_NUM_THREADS              100
#define USAGE    "\nUsage: %s [−l | −n num_threads] [−d]\n\n" \
                 "\t−l                    Test as many as threads as possible\n" \
                 "\t−n num_threads    Number of threads to create\n" \
                 "\t−d           Debug option\n\n"

/*
 * Function prototypes
 *
 * sys_error (): System error message function
 * error (): Error message function
 * parse_args (): Parses command line arguments
 */

static void sys_error (const char *, int);
static void error (const char *, int);
static void parse_args (int, char **);
void *thread (void *);

/*
 * Global Variables
 */

int num_threads = DEFAULT_NUM_THREADS;
int test_limit  = 0;
int debug       = 0;


/*−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+
|                              main ()                                   |
| ======================================================================= |
|                                                                         |
| Function:  Main program  (see prolog for more details)                  |
|                                                                         |
+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−*/
int main (int argc, char **argv)
{
        /*
         * Parse command line arguments and print out program header
         */
```

```
91          parse_args (argc, argv);
92
93          if(test_limit) {
94            printf ("\n   Creating as many threads as possible\n\n");
95          } else {
96            printf ("\n   Creating %d threads\n\n", num_threads);
97          }
98          thread (0);
99
100         /*
101          * Program completed successfully...
102          */
103         printf ("\ndone...\n\n");
104         fflush (stdout);
105         exit (0);
106 }
107
108
109 /*--------------------------------------------------------------------------+
110 |                                thread ()                                 |
111 | ================================================================== |
112 |                                                                          |
113 | Function:  Recursively creates threads while num < num_threads           |
114 |                                                                          |
115 +--------------------------------------------------------------------------*/
116 void *thread (void *parm)
117 {
118         int num = (int) parm;
119         pthread_t       th;
120         pthread_attr_t  attr;
121
122         if (debug) {
123                 printf ("\tThread [%d]: new\n", num);
124                 fflush (stdout);
125         }
126
127         /*
128          * Create threads while num < num_threads...
129          */
130         if (test_limit || (num < num_threads)) {
131
132                 if (pthread_attr_init (&attr))
133                         sys_error ("pthread_attr_init failed", __LINE__);
134                 if (pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE))
135                         sys_error ("pthread_attr_setdetachstate failed", __LINE__);
136                 if (pthread_create (&th, &attr, thread, (void *)(num + 1))) {
137                         if (test_limit) {
138                            printf ("Testing pthread limit, %d pthreads created.\n", num);
139                            pthread_exit(0);
140                         }
141                         if (errno == EAGAIN) {
142                             fprintf (stderr, "Thread [%d]: unable to create more threads!\n", num);
143                             return NULL;
144                         }
145                         else
146                             sys_error ("pthread_create failed", __LINE__);
147                 }
148                 if (test_limit && ((num % 1000) == 0))
149                     printf ("Testing pthread limit: %d pthreads created so far...\n", num);
150                 pthread_join (th, (void *) NULL);
151         }
152         if (debug) {
153                 printf ("\tThread [%d]: done\n", num);
154                 fflush (stdout);
155         }
156
157         pthread_exit(0);
158
159         /*NOTREACHED*/
160         return 0;
161 }
162
163
164 /*--------------------------------------------------------------------------+
165 |                              parse_args ()                               |
166 | ================================================================== |
167 |                                                                          |
168 | Function:  Parse the command line arguments & initialize global          |
169 |            variables.                                                    |
170 |                                                                          |
171 +--------------------------------------------------------------------------*/
172 static void parse_args (int argc, char **argv)
173 {
174         int     i;
175         int     errflag = 0;
176         char    *program_name = *argv;
177
178         while ((i = getopt(argc, argv, "dln:?")) != EOF) {
179                 switch (i) {
180                 case 'd':               /* debug option */
```

```
                                debug++;
                                break;
                        case 'l':               /* test pthread limit */
                                test_limit++;
                                break;
                        case 'n':               /* number of threads */
                                num_threads = atoi (optarg);
                                break;
                        case '?':
                                errflag++;
                                break;
                }
        }

        /* If any errors exit program */
        if (errflag) {
                fprintf (stderr, USAGE, program_name);
                exit (2);
        }
}


/*--------------------------------------------------------------------------+
|                              sys_error ()                                |
| ======================================================================== |
|                                                                          |
| Function:  Creates system error message and calls error ()               |
|                                                                          |
+--------------------------------------------------------------------------*/
static void sys_error (const char *msg, int line)
{
        char syserr_msg [256];

        sprintf (syserr_msg, "%s: %s\n", msg, strerror (errno));
        error (syserr_msg, line);
}


/*--------------------------------------------------------------------------+
|                                error ()                                  |
| ======================================================================== |
|                                                                          |
| Function:  Prints out message and exits...                               |
|                                                                          |
+--------------------------------------------------------------------------*/
static void error (const char *msg, int line)
{
        fprintf (stderr, "ERROR [line: %d] %s\n", line, msg);
        exit (-1);
}
```

```
/*
 *
 *   Copyright (c) International Business Machines  Corp., 2001
 *
 *   This program is free software;  you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY;  without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
 *   the GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program;  if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 */

/*
 *  FILE        : pth_str01.c
 *  DESCRIPTION : create a tree of threads does calculations, and
 *                returns result to parent
 *  HISTORY:
 *    05/l6/2001 Paul Larson (plars@us.ibm.com)
 *      −Ported
 *
 */

#ifdef GLOBAL
#include <pthread.h>
#else
#define _PTHREAD_PRIVATE
#include "pthread.h"
#endif

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define MAXTHREADS 50000

/* Type definition */
struct kid_info {
        long              sum;            /* sum of childrens indexes plus our own */
        int               index;          /* our index into the array */
        int               status;         /* return status of this thread */
        int               child_count;    /* Count of children created */
        int               talk_count;     /* Count of siblings that we talked to */
        pthread_t         *threads;       /* dynamic array of thread id of kids */
        pthread_mutex_t   talk_mutex;     /* mutex for the talk_count */
        pthread_mutex_t   child_mutex;    /* mutex for the child_count */
        pthread_cond_t    talk_condvar;   /* condition variable for talk_count */
        pthread_cond_t    child_condvar;  /* condition variable for child_count */
        struct kid_info   **child_ptrs;   /* dynamic array of ptrs to kids */
} ;
typedef struct kid_info c_info;


/* Global variables */
int     depth = 3;
int     breadth = 4;
int     timeout = 30;                   /* minutes */
int     cdepth;                         /* current depth */
int     debug = 0;
int     shared_fd = 0;

c_info          *child_info;            /* pointer to info array */
int             node_count;             /* number of nodes created so far */
pthread_mutex_t node_mutex;             /* mutex for the node_count */
pthread_cond_t  node_condvar;           /* condition variable for node_count */
pthread_attr_t  attr;                   /* attributes for created threads */

int             use_shared = 0;
typedef struct shared_area shared_area_t;
struct shared_area {
    int             initialized;
    int             locked;
    pthread_mutex_t mutex;
};

int num_nodes(int, int);
int synchronize_children(c_info *) ;
void *doit(void *);
```

```
91
92  /*--------------------------------------------------------------------------------*
93   * parse_args
94   *
95   * Parse command line arguments.  Any errors cause the program to exit
96   * at this point.
97   *--------------------------------------------------------------------------------*/
98  static void parse_args( int argc, char *argv[] )
99  {
100         int             opt, errflag = 0;
101         int             bflag = 0, dflag = 0, tflag = 0;
102
103         while ( (opt = getopt( argc, argv, "b:d:t:Ds?" )) != EOF ) {
104             switch ( opt ) {
105                 case 's':
106                     use_shared = 1;
107                     break;
108                 case 'b':
109                     if ( bflag )
110                         errflag++;
111                     else {
112                         bflag++;
113                         breadth = atoi( optarg );
114                         if ( breadth <= 0 )
115                             errflag++;
116                     }
117                     break;
118                 case 'd':
119                     if ( dflag )
120                         errflag++;
121                     else {
122                         dflag++;
123                         depth = atoi( optarg );
124                         if ( depth <= 0 )
125                             errflag++;
126                     }
127                     break;
128                 case 't':
129                     if ( tflag )
130                         errflag++;
131                     else {
132                         tflag++;
133                         timeout = atoi( optarg );
134                         if ( timeout <= 0 )
135                             errflag++;
136                     }
137                     break;
138                 case 'D':
139                     debug = 1;
140                     break;
141                 case '?':
142                 default:
143                     errflag++;
144                     break;
145             }
146         }
147
148         if ( errflag ) {
149                 fprintf( stderr, "usage: %s [-b <num>] [-d <num>] [-t <num>] [-s] [-D]", argv[0] );
150                 fprintf( stderr, "where:\n" );
151                 fprintf( stderr, "\t-b <num>\tbreadth of child nodes\n" );
152                 fprintf( stderr, "\t-d <num>\tdepth of child nodes\n" );
153                 fprintf( stderr, "\t-t <num>\ttimeout for child communication (in minutes)\n" );
154                 fprintf( stderr, "\t-s     \twrap with shared mutex\n" );
155                 fprintf( stderr, "\t-D\tdebug mode\n" );
156                 exit( 1 );
157         }
158
159 }
160
161
162 /*--------------------------------------------------------------------------------*
163  * num_nodes
164  *
165  * Caculate the number of child nodes for a given breadth and depth tree.
166  *--------------------------------------------------------------------------------*/
167 int num_nodes( int b, int d )
168 {
169         int     n, sum = 1, partial_exp = 1;
170
171         /*
172          * The total number of children = sum ( b ** n )
173          *                                n=0->d
174          * Since b ** 0 == 1, and it's hard to compute that kind of value
175          * in this simplistic loop, we start sum at 1 (above) to compensate
176          * and do the computations from 1->d below.
177          */
178         for ( n = 1; n <= d; n++ ) {
179                 partial_exp *= b;
180                 sum += partial_exp;
```

```
181         }
182
183         return( sum );
184 }
185
186
187 /*--------------------------------------------------------------------------*
188  * synchronize_children
189  *
190  * Register the child with the parent and then wait for all of the children
191  * at the same level to register also.  Return when everything is synched up.
192  *--------------------------------------------------------------------------*/
193 int synchronize_children( c_info *parent )
194 {
195         int               my_index, rc;
196         c_info            *info_p;
197         struct timespec   timer;
198
199         if ( debug ) {
200             printf( "trying to lock node_mutex\n" );
201             fflush( stdout );
202         }
203
204         /* Lock the node_count mutex to we can safely increment it.  We
205          * will unlock it when we broadcast that all of our siblings have
206          * been created or when we block waiting for that broadcast.  */
207         pthread_mutex_lock( &node_mutex );
208         my_index = node_count++;
209
210         printf( "thread %d started\n", my_index );
211         fflush( stdout );
212
213         /* Get a pointer into the array of thread structures which will be "me". */
214         info_p = &child_info[my_index];
215         info_p->index = my_index;
216         info_p->sum = (long) my_index;
217
218         if ( debug ) {
219             printf( "thread %d info_p=%x\n", my_index, (unsigned int)info_p );
220             fflush( stdout );
221         }
222
223         /* Register with parent bumping the parent's child_count variable.
224          * Make sure we have exclusive access to that variable before we
225          * do the increment.  */
226         if ( debug ) {
227             printf( "thread %d locking child_mutex %x\n", my_index, (unsigned int)&parent->child_mutex );
228             fflush( stdout );
229         }
230         pthread_mutex_lock( &parent->child_mutex );
231         if ( debug ) {
232             printf( "thread %d bumping child_count (currently %d)\n",
233               my_index, parent->child_count );
234             fflush( stdout );
235         }
236         parent->child_ptrs[parent->child_count++] = info_p;
237         if ( debug ) {
238             printf( "thread %d unlocking child_mutex %x\n", my_index,
239               (unsigned int)&parent->child_mutex );
240             fflush( stdout );
241         }
242         pthread_mutex_unlock( &parent->child_mutex );
243
244         if ( debug ) {
245             printf( "thread %d node_count = %d\n", my_index, node_count );
246             printf( "expecting %d nodes\n", num_nodes(breadth, cdepth) );
247             fflush( stdout );
248         }
249
250         /* Have all the nodes at our level in the tree been created yet?
251          * If so, then send out a broadcast to wake everyone else up (to let
252          * them know they can now create their children (if they are not
253          * leaf nodes)).  Otherwise, go to sleep waiting for the broadcast.  */
254         if ( node_count == num_nodes(breadth, cdepth) ) {
255
256             /* Increase the current depth variable, as the tree is now
257              * fully one level taller.  */
258             if ( debug ) {
259                 printf( "thread %d doing cdepth++ (%d)\n", my_index, cdepth );
260                 fflush( stdout );
261             }
262             cdepth++;
263
264             if ( debug ) {
265                 printf( "thread %d sending child_mutex broadcast\n", my_index );
266                 fflush( stdout );
267             }
268
269             /* Since all of our siblings have been created, this level of
270              * the tree is now allowed to continue its work, so wake up the
```

```
271                  * rest of the siblings.  */
272                 pthread_cond_broadcast( &node_condvar );
273
274             } else {
275
276                 /* In this case, not all of our siblings at this level of the
277                  * tree have been created, so go to sleep and wait for the
278                  * broadcast on node_condvar.  */
279                 if ( debug ) {
280                     printf( "thread %d waiting for siblings to register\n",
281                       my_index );
282                     fflush( stdout );
283                 }
284                 time( &timer.tv_sec );
285                 timer.tv_sec += (unsigned long)timeout * 60;
286                 timer.tv_nsec = (unsigned long)0;
287                 if ((rc = pthread_cond_timedwait(&node_condvar, &node_mutex,
288                   &timer))) {
289                     fprintf( stderr, "pthread_cond_timedwait (sync) %d: %s\n",
290                         my_index, sys_errlist[rc] );
291                     exit( 2 );
292                 }
293
294                 if ( debug ) {
295                     printf( "thread %d is now unblocked\n", my_index );
296                     fflush( stdout );
297                 }
298
299             }
300
301             /* Unlock the node_mutex lock, as this thread is finished initializing.  */
302             if ( debug ) {
303                 printf( "thread %d unlocking node_mutex\n", my_index );
304                 fflush( stdout );
305             }
306             pthread_mutex_unlock( &node_mutex );
307             if ( debug ) {
308                 printf( "thread %d unlocked node_mutex\n", my_index );
309                 fflush( stdout );
310             }
311
312             if ( debug ) {
313                 printf( "synchronize_children returning %d\n", my_index );
314                 fflush( stdout );
315             }
316
317             return( my_index );
318     }
319
320
321     /*--------------------------------------------------------------------------------*
322      * doit
323      *
324      * Do it.
325      *--------------------------------------------------------------------------------*/
326     void *doit( void *param )
327     {
328             c_info *parent  = (c_info *) param;
329             int             rc, child, my_index;
330             void            *status;
331             c_info          *info_p;
332             struct timespec timer;
333
334             if ( debug ) {
335                 printf( "parent=%#010x\n", (unsigned int)parent );
336                 fflush( stdout );
337             }
338
339             if ( parent != NULL ) {
340                 /* Synchronize with our siblings so that all the children at
341                  * a given level have been created before we allow those children
342                  * to spawn new ones (or do anything else for that matter).  */
343                 if ( debug ) {
344                     printf( "non-root child calling synchronize_children\n" );
345                     fflush( stdout );
346                 }
347                 my_index = synchronize_children( parent );
348                 if ( debug ) {
349                     printf( "non-root child has been assigned index %d\n",
350                       my_index );
351                     fflush( stdout );
352                 }
353             } else {
354                 /* The first thread has no one with which to synchronize, so
355                  * set some initial values for things.  */
356                 if ( debug ) {
357                     printf( "root child\n" );
358                     fflush( stdout );
359                 }
360                 cdepth = 1;
```

```
361             my_index = 0;
362             node_count = 1;
363         }
364
365         /* Figure out our place in the pthread array.  */
366         info_p = &child_info[my_index];
367
368         if ( debug ) {
369             printf( "thread %d getting to heart of doit.\n", my_index );
370             printf( "info_p=%x, cdepth=%d, depth=%d\n", (unsigned int)info_p, cdepth, depth );
371             fflush( stdout );
372         }
373
374         if ( cdepth <= depth )
375         {
376             /* Since the tree is not yet complete (it is not yet tall enough),
377              * we need to create another level of children.  */
378
379             printf( "thread %d creating kids, cdepth=%d\n", my_index, cdepth );
380             fflush( stdout );
381
382             /* Create breadth children.  */
383             for ( child = 0; child < breadth; child++ ) {
384                 if ( debug ) {
385                     printf( "thread %d making child %d, ptr=%x\n", my_index,
386                       child, (unsigned int)&(info_p->threads[child]) );
387                     fflush( stdout );
388                 }
389                 if ((rc = pthread_create(&(info_p->threads[child]), &attr, doit, (void *) info_p))) {
390                     fprintf( stderr, "pthread_create (doit): %s\n",
391                       sys_errlist[rc] );
392                     exit( 3 );
393                 } else {
394                     if ( debug ) {
395                         printf( "pthread_create made thread %x\n",
396                           (unsigned int)&(info_p->threads[child]) );
397                         fflush( stdout );
398                     }
399                 }
400             }
401
402             if ( debug ) {
403                 printf( "thread %d waits on kids, cdepth=%d\n", my_index,
404                     cdepth );
405                 fflush( stdout );
406             }
407
408             /* Wait for our children to finish before we exit ourselves.  */
409             for ( child = 0; child < breadth; child++ ) {
410                 if ( debug ) {
411                     printf( "attempting join on thread %x\n",
412                       (unsigned int)&(info_p->threads[child]) );
413                     fflush( stdout );
414                 }
415                 if ((rc = pthread_join((info_p->threads[child]), &status))) {
416                     if ( debug ) {
417                         fprintf( stderr,
418                           "join failed on thread %d, status addr=%x: %s\n",
419                           my_index, (unsigned int)status, sys_errlist[rc] );
420                         fflush( stderr );
421                     }
422                     exit( 4 );
423                 } else {
424                     if ( debug ) {
425                         printf( "thread %d joined child %d ok\n", my_index, child );
426                         fflush( stdout );
427                     }
428
429                     /* Add all childrens indexes to your index value */
430                     info_p->sum += info_p->child_ptrs[child]->sum;
431                     printf("thread %d adding child thread %d to sum = %ld\n",
432                            my_index, info_p->child_ptrs[child]->index, (long int)info_p->sum);
433                 }
434             }
435
436         } else {
437
438             /* This is the leaf node case.  These children don't create
439              * any kids; they just talk amongst themselves.  */
440             printf( "thread %d is a leaf node, depth=%d\n", my_index, cdepth );
441             fflush( stdout );
442
443             /* Talk to siblings (children of the same parent node).  */
444             if ( breadth > 1 ) {
445
446                 for ( child = 0; child < breadth; child++ ) {
447                     /* Don't talk to yourself.  */
448                     if ( parent->child_ptrs[child] != info_p ) {
449                         if ( debug ) {
450                             printf( "thread %d locking talk_mutex\n",
```

```
                                my_index );
                            fflush( stdout );
                        }
                        pthread_mutex_lock(
                          &(parent->child_ptrs[child]->talk_mutex) );
                        if ( ++parent->child_ptrs[child]->talk_count == (breadth - 1) ) {
                            if ( debug ) {
                                printf( "thread %d talk siblings\n", my_index );
                                fflush( stdout );
                            }
                            if ((rc = pthread_cond_broadcast(
                              &parent->child_ptrs[child]->talk_condvar))) {
                                fprintf( stderr, "pthread_cond_broadcast: %s\n",
                                  sys_errlist[rc] );
                                exit( 5 );
                            }
                        }
                        if ( debug ) {
                            printf( "thread %d unlocking talk_mutex\n",
                              my_index );
                            fflush( stdout );
                        }
                        pthread_mutex_unlock(
                          &(parent->child_ptrs[child]->talk_mutex) );
                    }
                }

                /* Wait until the breadth - 1 siblings have contacted us.  */
                if ( debug ) {
                    printf( "thread %d waiting for talk siblings\n", my_index );
                    fflush( stdout );
                }

                pthread_mutex_lock( &info_p->talk_mutex );
                if ( info_p->talk_count < (breadth - 1) ) {
                    time( &timer.tv_sec );
                    timer.tv_sec += (unsigned long)timeout * 60;
                    timer.tv_nsec = (unsigned long)0;
                    if ((rc = pthread_cond_timedwait(&info_p->talk_condvar,
                      &info_p->talk_mutex, &timer))) {
                        fprintf( stderr,
                          "pthread_cond_timedwait (leaf) %d: %s\n",
                          my_index, sys_errlist[rc] );
                        exit( 6 );
                    }
                }
                pthread_mutex_unlock( &info_p->talk_mutex );

            }

        }

        /* Our work is done.  We're outta here. */
        printf( "thread %d exiting, depth=%d, status=%d, addr=%x\n", my_index,
          cdepth, info_p->status, (unsigned int)info_p);
        fflush( stdout );

        pthread_exit( 0 );

        /*NOTREACHED*/
        return 0;
}


/*--------------------------------------------------------------------------*
 * main
 *--------------------------------------------------------------------------*/
int main( int argc, char *argv[] )
{
        int             rc, ind, total, i_initialized = 0;
        pthread_mutexattr_t myattr;
        pthread_t       root_thread;
        shared_area_t   *sa = NULL;

        parse_args( argc, argv );

        if (use_shared == 1) {

            if (shared_fd == 0) {
                shared_fd = open("test_str03m", O_RDWR, (S_IRWXU | S_IRWXG | S_IRWXO));
                if (shared_fd < 0) {
                    fprintf(stderr, "unable to open shared memory\n");
                    exit(1);
                }
                sa = (shared_area_t *)mmap(NULL, (sizeof(struct shared_area)),
                                           PROT_READ | PROT_WRITE | PROT_EXEC, MAP_SHARED,
                                           shared_fd, 0);
                if (sa == MAP_FAILED) {
                    fprintf(stderr, "unable to map shared memory\n");
```

```
                    exit(1);
                }
            }


            /* Initialize shared mutex... */
            if (sa->initialized != 1) {
                fprintf(stderr,"shared memory was not initialized.\n");
                pthread_mutexattr_init(&myattr);
                pthread_mutexattr_setpshared(&myattr, PTHREAD_PROCESS_SHARED);
                if ((rc = pthread_mutex_init(&(sa->mutex), &myattr))) {
                    fprintf( stderr, "pthread_mutex_init(test_mutex): %s\n",
                            sys_errlist[rc] );
                    exit( 7 );
                }
                fprintf(stderr,"mutex was initialized.\n");
                sa->initialized = 1;
                i_initialized = 1;
                msync(sa, sizeof(struct shared_area), MS_INVALIDATE | MS_SYNC);
            } else
                fprintf(stderr,"shared memory was initialized.\n");

            if (i_initialized == 1) {
                fprintf(stderr,"I did initialize, spinning...\n");
                fprintf(stderr,"waiting for partner to lock.\n");
                while (sa->locked != 1) {
                    sleep(1);
                    continue;
                }
                fprintf(stderr,"partner is locked.\n");
            }

            if ((rc = pthread_mutex_lock(&(sa->mutex)))) {
                fprintf(stderr, "unable to grab test_mutex, rc = %d\n", rc);
                exit( 7);
            }
            fprintf(stderr,"I am locked.\n");
            sa->locked = 1;
            msync(sa, sizeof(struct shared_area), MS_INVALIDATE | MS_SYNC);

        }


        /* Initialize node mutex.  */
        if ((rc = pthread_mutex_init(&node_mutex, NULL))) {
                fprintf( stderr, "pthread_mutex_init(node_mutex): %s\n",
                    sys_errlist[rc] );
                exit( 7 );
        }

        /* Initialize node condition variable.  */
        if ((rc = pthread_cond_init(&node_condvar, NULL))) {
                fprintf( stderr, "pthread_cond_init(node_condvar): %s\n",
                    sys_errlist[rc] );
                exit( 8 );
        }

        /* Allocate pthread info structure array.  */
        if ( (total = num_nodes( breadth, depth )) > MAXTHREADS ) {
                fprintf( stderr, "Can't create %d threads; max is %d.\n",
                    total, MAXTHREADS );
                exit( 9 );
        }
        printf( "Allocating %d nodes.\n", total );
        fflush( stdout );
        if ( (child_info = (c_info *)malloc( total * sizeof(c_info) )) == NULL ) {
                perror( "malloc child_info" );
                exit( 10 );
        }
        bzero( child_info, total * sizeof(c_info) );

        if ( debug ) {
                printf( "Initializing array for %d children\n", total );
                fflush( stdout );
        }

        /* Allocate array of pthreads descriptors and initialize variables.  */
        for ( ind = 0; ind < total; ind++ ) {

                if ( (child_info[ind].threads =
                    (pthread_t *)malloc( breadth * sizeof(pthread_t) )) == NULL ) {
                        perror( "malloc threads" );
                        exit( 11 );
                }
                bzero( child_info[ind].threads, breadth * sizeof(pthread_t) );

                if ( (child_info[ind].child_ptrs =
                    (c_info **)malloc( breadth * sizeof(c_info *) )) == NULL ) {
                        perror( "malloc child_ptrs" );
                        exit( 12 );
```

```
631                     }
632                     bzero( child_info[ind].child_ptrs,
633                         breadth * sizeof(c_info *) );
634
635                     if ((rc = pthread_mutex_init(&child_info[ind].child_mutex, NULL))) {
636                             fprintf( stderr, "pthread_mutex_init child_mutex: %s\n",
637                                 sys_errlist[rc] );
638                             exit( 13 );
639                     }
640
641                     if ((rc = pthread_mutex_init(&child_info[ind].talk_mutex, NULL))) {
642                             fprintf( stderr, "pthread_mutex_init talk_mutex: %s\n",
643                                 sys_errlist[rc] );
644                             exit( 14 );
645                     }
646
647                     if ((rc = pthread_cond_init(&child_info[ind].child_condvar, NULL))) {
648                             fprintf( stderr,
649                                 "pthread_cond_init child_condvar: %s\n",
650                                 sys_errlist[rc] );
651                             exit( 15 );
652                     }
653
654                     if ((rc = pthread_cond_init(&child_info[ind].talk_condvar, NULL))) {
655                             fprintf( stderr, "pthread_cond_init talk_condvar: %s\n",
656                                 sys_errlist[rc] );
657                             exit( 16 );
658                     }
659
660                     if ( debug ) {
661                             printf( "Successfully initialized child %d.\n", ind );
662                             fflush( stdout );
663                     }
664
665             }
666
667             printf( "Creating root thread attributes via pthread_attr_init.\n" );
668             fflush( stdout );
669
670             if ((rc = pthread_attr_init(&attr))) {
671                     fprintf( stderr, "pthread_attr_init: %s\n", sys_errlist[rc] );
672                     exit( 17 );
673             }
674
675             /* Make sure that all the thread children we create have the
676              * PTHREAD_CREATE_JOINABLE attribute.  If they don't, then the
677              * pthread_join call will sometimes fail and cause mass confusion.  */
678             if ((rc = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE))) {
679                     fprintf( stderr, "pthread_attr_setdetachstate: %s\n",
680                         sys_errlist[rc] );
681                     exit( 18 );
682             }
683
684             printf( "Creating root thread via pthread_create.\n" );
685             fflush( stdout );
686
687             if ((rc = pthread_create(&root_thread, &attr, doit, NULL))) {
688                     fprintf( stderr, "pthread_create: %s\n", sys_errlist[rc] );
689                     exit( 19 );
690             }
691
692             if ( debug ) {
693                     printf( "Doing pthread_join.\n" );
694                     fflush( stdout );
695             }
696
697             /* Wait for the root child to exit.  */
698             if ((rc = pthread_join(root_thread, NULL))) {
699                     fprintf( stderr, "pthread_join: %s\n", sys_errlist[rc] );
700                     exit( 20 );
701             }
702
703             if ( debug ) {
704                     printf( "About to pthread_exit.\n" );
705                     fflush( stdout );
706             }
707
708             printf("\n\nThe sum of tree (breadth %d, depth %d) is %ld\n",
709                     breadth, depth, child_info[0].sum);
710
711             if (use_shared == 1) {
712                 pthread_mutex_unlock(&(sa->mutex));
713                 if (i_initialized == 1) {
714                     pthread_mutex_destroy(&(sa->mutex));
715                     sa->locked = 0;
716                     sa->initialized = 0;
717                 }
718             }
719             exit( 0 );
720     }
```

Table of Content

End of Table of Contents