Subsections

Thread programming examples

This chapter gives some full code examples of thread programs. These examles are taken from a variety of sources:

- The sun workshop developers web page ***http://www.sun.com/workshop/threads/share-code/*** on threads is an excelleny source
- The web page ***http://www.sun.com/workshop/threads/Berg-Lewis/examples.html*** where example from the ***Threads Primer*** Book by D. Berg anD B. Lewis are also a major resource.

Using `thr_create()` and `thr_join()`

This example exercises the `thr_create()` and `thr_join()` calls. There is not a parent/child relationship between threads as there is for processes. This can easily be seen in this example, because threads are created and joined by many different threads in the process. The example also shows how threads behave when created with different attributes and options.

Threads can be created by any thread and joined by any other.

The main thread: In this example the main thread's sole purpose is to create new threads. Threads A, B, and C are created by the main thread. Notice that thread B is created suspended. After creating the new threads, the main thread exits. Also notice that the main thread exited by calling thr_exit(). If the main thread had used the exit() call, the whole process would have exited. The main thread's exit status and resources are held until it is joined by thread C.

Thread A: The first thing thread A does after it is created is to create thread D. Thread A then simulates some processing and then exits, using `thr_exit()`. Notice that thread A was created with the `THR_DETACHED` flag, so thread A's resources will be immediately reclaimed upon its exit. There is no way for thread A's exit status to be collected by a `thr_join()` call.

Thread B: Thread B was created in a suspended state, so it is not able to run until thread D continues it by making the `thr_continue()` call. After thread B is continued, it simulates some processing and then exits. Thread B's exit status and thread resources are held until joined by thread E.

Thread C: The first thing that thread C does is to create thread F. Thread C then joins the main thread. This action will collect the main thread's exit status and allow the main thread's resources to be reused by another thread. Thread C will block, waiting for the main thread to exit, if the main thread has not yet called `thr_exit()`. After joining the main thread, thread C will simulate some processing and then exit. Again, the exit status and thread resources are held until joined by thread E.

Thread D: Thread D immediately creates thread E. After creating thread E, thread D continues thread B by making the `thr_continue()` call. This call will allow thread B to start its execution. Thread D then tries to join thread E, blocking until thread E has exited. Thread D then simulates some processing and exits. If all went well, thread D should be the last nondaemon thread running. When thread D exits, it should do two things: stop the execution of any daemon threads and stop the execution of the process.

Thread E: Thread E starts by joining two threads, threads B and C. Thread E will block, waiting for each of these thread to exit. Thread E will then simulate some processing and will exit. Thread E's exit status and thread resources are held by the operating system until joined by thread D.

Thread F: Thread F was created as a bound, daemon thread by using the `THR_BOUND` and `THR_DAEMON` flags in the `thr_create()` call. This means that it will run on its own LWP until all the nondaemon threads have exited the process. This type of thread can be used when you want some type of "background" processing to always be running, except when all the "regular" threads have exited the process. If thread F was created as a non-daemon thread, then it would continue to run forever, because a process will continue while there is at least one thread still running. Thread F will exit when all the nondaemon threads have exited. In this case, thread D should be the last nondaemon thread running, so when thread D exits, it will also cause thread F to exit.

This example, however trivial, shows how threads behave differently, based on their creation options. It also shows what happens on the exit of a thread, again based on how it was created. If you understand this example and how it flows, you should have a good understanding of how to use `thr_create()` and `thr_join()` in your own programs. Hopefully you can also see how easy it is to create and join threads.

The source to `multi_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Function prototypes for thread routines */
void *sub_a(void *);
void *sub_b(void *);
void *sub_c(void *);
void *sub_d(void *);
void *sub_e(void *);
void *sub_f(void *);

thread_t thr_a, thr_b, thr_c;

void main()
{
thread_t main_thr;

main_thr = thr_self();
printf("Main thread = %d\n", main_thr);

if (thr_create(NULL, 0, sub_b, NULL, THR_SUSPENDED|THR_NEW_LWP, &thr_b))
        fprintf(stderr,"Can't create thr_b\n"), exit(1);

if (thr_create(NULL, 0, sub_a, (void *)thr_b, THR_NEW_LWP, &thr_a))
        fprintf(stderr,"Can't create thr_a\n"), exit(1);
```

```
if (thr_create(NULL, 0, sub_c, (void *)main_thr, THR_NEW_LWP, &thr_c))
        fprintf(stderr,"Can't create thr_c\n"), exit(1);

printf("Main Created threads A:%d B:%d C:%d\n", thr_a, thr_b, thr_c);
printf("Main Thread exiting...\n");
thr_exit((void *)main_thr);
}

void *sub_a(void *arg)
{
thread_t thr_b = (thread_t) arg;
thread_t thr_d;
int i;

printf("A: In thread A...\n");

if (thr_create(NULL, 0, sub_d, (void *)thr_b, THR_NEW_LWP, &thr_d))
        fprintf(stderr, "Can't create thr_d\n"), exit(1);

printf("A: Created thread D:%d\n", thr_d);

/* process
*/
for (i=0;i<1000000*(int)thr_self();i++);
printf("A: Thread exiting...\n");
thr_exit((void *)77);
}

void * sub_b(void *arg)
{
int i;

printf("B: In thread B...\n");

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("B: Thread exiting...\n");
thr_exit((void *)66);
}

void * sub_c(void *arg)
{
void *status;
int i;
thread_t main_thr, ret_thr;

main_thr = (thread_t)arg;

printf("C: In thread C...\n");

if (thr_create(NULL, 0, sub_f, (void *)0, THR_BOUND|THR_DAEMON, NULL))
        fprintf(stderr, "Can't create thr_f\n"), exit(1);

printf("C: Join main thread\n");

if (thr_join(main_thr,(thread_t *)&ret_thr, &status))
        fprintf(stderr, "thr_join Error\n"), exit(1);

printf("C: Main thread (%d) returned thread (%d) w/status %d\n", main_thr, r

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("C: Thread exiting...\n");
thr_exit((void *)88);
}
```

```
void * sub_d(void *arg)
{
thread_t thr_b = (thread_t) arg;
int i;
thread_t thr_e, ret_thr;
void *status;

printf("D: In thread D...\n");

if (thr_create(NULL, 0, sub_e, NULL, THR_NEW_LWP, &thr_e))
        fprintf(stderr,"Can't create thr_e\n"), exit(1);

printf("D: Created thread E:%d\n", thr_e);
printf("D: Continue B thread = %d\n", thr_b);

thr_continue(thr_b);
printf("D: Join E thread\n");

if(thr_join(thr_e,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("D: E thread (%d) returned thread (%d) w/status %d\n", thr_e,
ret_thr, (int) status);

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("D: Thread exiting...\n");
thr_exit((void *)55);
}


void * sub_e(void *arg)
{
int i;
thread_t ret_thr;
void *status;

printf("E: In thread E...\n");
printf("E: Join A thread\n");

if(thr_join(thr_a,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: A thread (%d) returned thread (%d) w/status %d\n", ret_thr, ret_t
printf("E: Join B thread\n");

if(thr_join(thr_b,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: B thread (%d) returned thread (%d) w/status %d\n", thr_b, ret_th
printf("E: Join C thread\n");

if(thr_join(thr_c,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: C thread (%d) returned thread (%d) w/status %d\n", thr_c, ret_th

for (i=0;i<1000000*(int)thr_self();i++);

printf("E: Thread exiting...\n");
thr_exit((void *)44);
}


void *sub_f(void *arg)
{
int i;

printf("F: In thread F...\n");
while (1) {
        for (i=0;i<10000000;i++);
        printf("F: Thread F is still running...\n");
        }
}
```

Arrays

This example uses a data structure that contains multiple arrays of data. Multiple threads will concurrently vie for access to the arrays. To control this access, a mutex variable is used within the data structure to lock the entire array and serialize the access to the data.

The main thread first initializes the data structure and the mutex variable. It then sets a level of concurrency and creates the worker threads. The main thread then blocks by joining all the threads. When all the threads have exited, the main thread prints the results.

The worker threads modify the shared data structure from within a loop. Each time the threads need to modify the shared data, they lock the mutex variable associated with the shared data. After modifying the data, the threads unlock the mutex, allowing another thread access to the data.

This example may look quite simple, but it shows how important it is to control access to a simple, shared data structure. The results can be quite different if the mutex variable is not used.

The source to `array.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* sample array data structure */
struct {
        mutex_t data_lock[5];
        int     int_val[5];
        float   float_val[5];
        } Data;

/* thread function */
void *Add_to_Value();

main()
{
int i;

/* initialize the mutexes and data */
for (i=0; i<5; i++) {
        mutex_init(&Data.data_lock[i], USYNC_THREAD, 0);
        Data.int_val[i] = 0;
        Data.float_val[i] = 0;
        }

/* set concurrency and create the threads */
thr_setconcurrency(4);

for (i=0; i<5; i++)
    thr_create(NULL, 0, Add_to_Value, (void *)(2*i), 0, NULL);

/* wait till all threads have finished */
for (i=0; i<5; i++)
        thr_join(0,0,0);

/* print the results */
printf("Final Values.....\n");
for (i=0; i<5; i++) {
        printf("integer value[%d] =\t%d\n", i, Data.int_val[i]);
        printf("float value[%d] =\t%.0f\n\n", i, Data.float_val[i]);
        }

return(0);
}


/* Threaded routine */
void *Add_to_Value(void *arg)
{
int inval = (int) arg;
int i;

for (i=0;i<10000;i++){
    mutex_lock(&Data.data_lock[i%5]);
       Data.int_val[i%5] += inval;
       Data.float_val[i%5] += (float) 1.5 * inval;
    mutex_unlock(&Data.data_lock[i%5]);
    }

return((void *)0);
}
```

Deadlock

This example demonstrates how a deadlock can occur in multithreaded programs that use synchronization variables. In this example a thread is created that continually adds a value to a global variable. The thread uses a mutex lock to protect the global data.

The main thread creates the counter thread and then loops, waiting for user input. When the user presses the Return key, the main thread suspends the counter thread and then prints the value of the global variable. The main thread prints the value of the global variable under the protection of a mutex lock.

The problem arises in this example when the main thread suspends the counter thread while the counter thread is holding the mutex lock. After the main thread suspends the counter thread, it tries to lock the mutex variable. Since the mutex variable is already held by the counter thread, which is suspended, the main thread deadlocks.

This example may run fine for a while, as long as the counter thread just happens to be suspended when it is not holding the mutex lock. The example demonstrates how tricky some programming issues can be when you deal with threads.

The source to `susp_lock.c`

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Prototype for thread subroutine */
void *counter(void *);

int count;
mutex_t count_lock;

main()
{
char str[80];
thread_t ctid;

/* create the thread counter subroutine */
thr_create(NULL, 0, counter, 0, THR_NEW_LWP|THR_DETACHED, &ctid);
while(1) {
        gets(str);
        thr_suspend(ctid);

        mutex_lock(&count_lock);
        printf("\n\nCOUNT = %d\n\n", count);
        mutex_unlock(&count_lock);

        thr_continue(ctid);
        }

return(0);
}

void *counter(void *arg)
{
int i;

while (1) {
        printf("."); fflush(stdout);

        mutex_lock(&count_lock);
        count++;

        for (i=0;i<50000;i++);

        mutex_unlock(&count_lock);

        for (i=0;i<50000;i++);
        }

return((void *)0);
}
```

Signal Handler

This example shows how easy it is to handle signals in multithreaded programs. In most programs, a different signal handler would be needed to service each type of signal that you wanted to catch. Writing each of the signal handlers can be time consuming and can be a real pain to debug.

This example shows how you can implement a signal handler thread that will service all asynchronous signals that are sent to your process. This is an easy way to deal with signals, because only one thread is needed to handle all the signals. It also makes it easy when you create new threads within the process, because you need not worry about signals in any of the threads.

First, in the main thread, mask out all signals and then create a signal handling thread. Since threads inherit the signal mask from their creator, any new threads created after the new signal mask will also mask all signals. This idea is key, because the only thread that will receive signals is the one thread that does not block all the signals.

The signal handler thread waits for all incoming signals with the sigwait() call. This call unmasks the signals given to it and then blocks until a signal arrives. When a signal arrives, sigwait() masks the signals again and then returns with the signal ID of the incoming signal.

You can extend this example for use in your application code to handle all your signals. Notice also that this signal concept could be added in your existing nonthreaded code as a simpler way to deal with signals.

The source to `thr_sig.c`

```
#define _REENTRANT
```

```
#include <stdio.h>
#include <thread.h>
#include <signal.h>
#include <sys/types.h>

void *signal_hand(void *);

main()
{
sigset_t set;

/* block all signals in main thread.  Any other threads that are
   created after this will also block all signals */

sigfillset(&set);

thr_sigsetmask(SIG_SETMASK, &set, NULL);

/* create a signal handler thread.  This thread will catch all
   signals and decide what to do with them.  This will only
   catch nondirected signals.  (I.e., if a thread causes a SIGFPE
   then that thread will get that signal. */

thr_create(NULL, 0, signal_hand, 0, THR_NEW_LWP|THR_DAEMON|THR_DETACHED, NUI

while (1) {
        /*
        Do your normal processing here....
        */
        }  /* end of while */

return(0);
}

void *signal_hand(void *arg)
{
sigset_t set;
int sig;

sigfillset(&set); /* catch all signals */

while (1) {
        /* wait for a signal to arrive */

        switch (sig=sigwait(&set)) {

          /* here you would add whatever signal you needed to catch */
          case SIGINT : {
                        printf("Interrupted with signal %d, exiting...\n", s
                        exit(0);
                        }

          default : printf("GOT A SIGNAL = %d\n", sig);
          } /* end of switch */
        } /* end of while */

return((void *)0);
} /* end of signal_hand */
```

Another example of a signal handler, `sig_kill.c`:

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
```

```
 *
 *  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
 *  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
 *  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
 *
 *  This file is provided with no support and without any obligation on the
 *  part of SunSoft, Inc. to assist in its use, correction, modification or
 *  enhancement.
 *
 *  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
 *  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
 *  FILE OR ANY PART THEREOF.
 *
 *  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
 *  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
 *  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
 *  DAMAGES.
 *
 *  SunSoft, Inc.
 *  2550 Garcia Avenue
 *  Mountain View, California  94043
 */

/*
 * Rich Schiavi writes:                 Sept 11, 1994
 *
 * I believe the recommended way to kill certain threads is
 * using a signal handler which then will exit that particular
 * thread properly. I'm not sure the exact reason (I can't remember), but
 * if you take out the signal_handler routine in my example, you will see wh
 * you describe, as the main process dies even if you send the
 * thr_kill to the specific thread.

 * I whipped up a real quick simple example which shows this using
 * some sleep()s to get a good simulation.
 */

#include <stdio.h>
#include <thread.h>
#include <signal.h>

static  thread_t        one_tid, two_tid, main_thread;
static  void    *first_thread();
static  void    *second_thread();
void            ExitHandler(int);

static  mutex_t         first_mutex, second_mutex;
int     first_active = 1 ;
int     second_active = 1;

main()

{
  int i;
  struct sigaction act;

  act.sa_handler = ExitHandler;
  (void) sigemptyset(&act.sa_mask);
  (void) sigaction(SIGTERM, &act, NULL);

  mutex_init(&first_mutex, 0 , 0);
  mutex_init(&second_mutex, 0 , 0);
  main_thread = thr_self();

  thr_create(NULL,0,first_thread,0,THR_NEW_LWP,&one_tid);
  thr_create(NULL,0,second_thread,0,THR_NEW_LWP,&two_tid);

  for (i = 0; i < 10; i++){
    fprintf(stderr, "main loop: %d\n", i);
    if (i == 5) {
      thr_kill(one_tid, SIGTERM);
    }
    sleep(3);
  }
  thr_kill(two_tid, SIGTERM);
  sleep(5);
  fprintf(stderr, "main exit\n");
}

static void *first_thread()
{
  int i = 0;

  fprintf(stderr, "first_thread id: %d\n", thr_self());
  while (first_active){
    fprintf(stderr, "first_thread: %d\n", i++);
    sleep(2);
  }
  fprintf(stderr, "first_thread exit\n");
}

static void *second_thread()
{
  int i = 0;

  fprintf(stderr, "second_thread id: %d\n", thr_self());

  while (second_active){
    fprintf(stderr, "second_thread: %d\n", i++);
    sleep(3);
  }
  fprintf(stderr, "second_thread exit\n");
}

void ExitHandler(int sig)
{
  thread_t id;

  id = thr_self();

  fprintf(stderr, "ExitHandler thread id: %d\n", id);
  thr_exit(0);

}
```

Interprocess Synchronization

This example uses some of the synchronization variables available in the threads library to synchronize access to a resource shared between two processes. The synchronization variables used in the threads library are an advantage over standard IPC synchronization mechanisms because of their speed. The synchronization variables in the threads libraries have been tuned to be very lightweight and very fast. This speed can be an advantage when your application is spending time synchronizing between processes.

This example shows how semaphores from the threads library can be used between processes. Note that this program does not use threads; it is just using the lightweight semaphores available from the threads library.

When using synchronization variables between processes, it is important to make sure that only one process initializes the variable. If both processes try to initialize the synchronization variable, then one of the processes will overwrite the state of the variable set by the other process.

The source to `ipc.c`

```
#include <stdio.h>
#include <fcntl.h>
```

```
#include <sys/mman.h>
#include <synch.h>
#include <sys/types.h>
#include <unistd.h>

/* a structure that will be used between processes */
typedef struct {
        sema_t mysema;
        int num;
} buf_t;

main()
{
int     i, j, fd;
buf_t   *buf;

/* open a file to use in a memory mapping */
fd = open("/dev/zero", O_RDWR);

/* create a shared memory map with the open file for the data
   structure that will be shared between processes */
buf=(buf_t *)mmap(NULL, sizeof(buf_t), PROT_READ|PROT_WRITE, MAP_SHARED, fd,

/* initialize the semaphore -- note the USYNC_PROCESS flag; this makes
   the semaphore visible from a process level */
sema_init(&buf->mysema, 0, USYNC_PROCESS, 0);

/* fork a new process */
if (fork() == 0) {
        /* The child will run this section of code */
        for (j=0;j<5;j++)
                {
                /* have the child "wait" for the semaphore */

                printf("Child PID(%d): waiting...\n", getpid());
                sema_wait(&buf->mysema);

                /* the child decremented the semaphore */

                printf("Child PID(%d): decrement semaphore.\n", getpid());
                }
        /* exit the child process */
        printf("Child PID(%d): exiting...\n", getpid());
        exit(0);
        }

/* The parent will run this section of code */
/* give the child a chance to start running */

sleep(2);

for (i=0;i<5;i++)
        {
        /* increment (post) the semaphore */

        printf("Parent PID(%d): posting semaphore.\n", getpid());
        sema_post(&buf->mysema);

        /* wait a second */
        sleep(1);
        }

/* exit the parent process */
printf("Parent PID(%d): exiting...\n", getpid());

return(0);
}
```

The Producer / Consumer Problem

This example will show how condition variables can be used to control access of reads and writes to a buffer. This example can also be thought as a producer/consumer problem, where the producer adds items to the buffer and the consumer removes items from the buffer.

Two condition variables control access to the buffer. One condition variable is used to tell if the buffer is full, and the other is used to tell if the buffer is empty. When the producer wants to add an item to the buffer, it checks to see if the buffer is full; if it is full the producer blocks on the `cond_wait()` call, waiting for an item to be removed from the buffer. When the consumer removes an item from the buffer, the buffer is no longer full, so the producer is awakened from the `cond_wait()` call. The producer is then allowed to add another item to the buffer.

The consumer works, in many ways, the same as the producer. The consumer uses the other condition variable to determine if the buffer is empty. When the consumer wants to remove an item from the buffer, it checks to see if it is empty. If the buffer is empty, the consumer then blocks on the `cond_wait()` call, waiting for an item to be added to the buffer. When the producer adds an item to the buffer, the consumer's condition is satisfied, so it can then remove an item from the buffer.

The example copies a file by reading data into a shared buffer (producer) and then writing data out to the new file (consumer). The Buf data structure is used to hold both the buffered data and the condition variables that control the flow of the data.

The main thread opens both files, initializes the `Buf` data structure, creates the consumer thread, and then assumes the role of the producer. The producer reads data from the input file, then places the data into an open buffer position. If no buffer positions are available, then the producer waits via the `cond_wait()` call. After the producer has read all the data from the input file, it closes the file and waits for (joins) the consumer thread.

The consumer thread reads from a shared buffer and then writes the data to the output file. If no buffers positions are available, then the consumer waits for the producer to fill a buffer position. After the consumer has read all the data, it closes the output file and exits.

If the input file and the output file were residing on different physical disks, then this example could execute the reads and writes in parallel. This parallelism would significantly increase the throughput of the example through the use of threads.

The source to `prod_cons.c`:

```
#define _REEENTRANT
#include <stdio.h>
#include <thread.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>

#define BUFSIZE 512
#define BUFCNT  4

/* this is the data structure that is used between the producer
   and consumer threads */

struct {
        char buffer[BUFCNT][BUFSIZE];
        int byteinbuf[BUFCNT];
        mutex_t buflock;
        mutex_t donelock;
        cond_t adddata;
        cond_t remdata;
        int nextadd, nextrem, occ, done;
```

```
} Buf;

/* function prototype */
void *consumer(void *);

main(int argc, char **argv)
{
int ifd, ofd;
thread_t cons_thr;

/* check the command line arguments */
if (argc != 3)
        printf("Usage: %s <infile> <outfile>\n", argv[0]), exit(0);

/* open the input file for the producer to use */
if ((ifd = open(argv[1], O_RDONLY)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[1]);
        exit(1);
        }

/* open the output file for the consumer to use */
if ((ofd = open(argv[2], O_WRONLY|O_CREAT, 0666)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[2]);
        exit(1);
        }

/* zero the counters */
Buf.nextadd = Buf.nextrem = Buf.occ = Buf.done = 0;

/* set the thread concurrency to 2 so the producer and consumer can
   run concurrently */

thr_setconcurrency(2);

/* create the consumer thread */
thr_create(NULL, 0, consumer, (void *)ofd, NULL, &cons_thr);

/* the producer ! */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        /* check to see if any buffers are empty */
        /* If not then wait for that condition to become true */

        while (Buf.occ == BUFCNT)
                cond_wait(&Buf.remdata, &Buf.buflock);

        /* read from the file and put data into a buffer */
        Buf.byteinbuf[Buf.nextadd] = read(ifd,Buf.buffer[Buf.nextadd],BUFSIZ

        /* check to see if done reading */
        if (Buf.byteinbuf[Buf.nextadd] == 0) {

                /* lock the done lock */
                mutex_lock(&Buf.donelock);

                /* set the done flag and release the mutex lock */
                Buf.done = 1;

                mutex_unlock(&Buf.donelock);

                /* signal the consumer to start consuming */
                cond_signal(&Buf.adddata);

                /* release the buffer mutex */
                mutex_unlock(&Buf.buflock);

                /* leave the while looop */
                break;
                }

        /* set the next buffer to fill */
        Buf.nextadd = ++Buf.nextadd % BUFCNT;

        /* increment the number of buffers that are filled */
        Buf.occ++;

        /* signal the consumer to start consuming */
        cond_signal(&Buf.adddata);

        /* release the mutex */
        mutex_unlock(&Buf.buflock);
        }

close(ifd);

/* wait for the consumer to finish */
thr_join(cons_thr, 0, NULL);

/* exit the program */
return(0);
}


/* The consumer thread */
void *consumer(void *arg)
{
int fd = (int) arg;

/* check to see if any buffers are filled or if the done flag is set */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        if (!Buf.occ && Buf.done) {
           mutex_unlock(&Buf.buflock);
           break;
           }

        /* check to see if any buffers are filled */
        /* if not then wait for the condition to become true */

        while (Buf.occ == 0 && !Buf.done)
                cond_wait(&Buf.adddata, &Buf.buflock);

        /* write the data from the buffer to the file */
        write(fd, Buf.buffer[Buf.nextrem], Buf.byteinbuf[Buf.nextrem]);

        /* set the next buffer to write from */
        Buf.nextrem = ++Buf.nextrem % BUFCNT;

        /* decrement the number of buffers that are full */
        Buf.occ--;

        /* signal the producer that a buffer is empty */
        cond_signal(&Buf.remdata);

        /* release the mutex */
        mutex_unlock(&Buf.buflock);
        }

/* exit the thread */
thr_exit((void *)0);
}
```

A Socket Server

The socket server example uses threads to implement a "standard" socket port server. The example shows how easy it is to use `thr_create()` calls in the place of `fork()` calls in existing programs.

A standard socket server should listen on a socket port and, when a message arrives, fork a process to service the request. Since a `fork()` system call would be used in a nonthreaded program, any communication between the parent and child would have to be done through some sort of interprocess communication.

We can replace the `fork()` call with a `thr_create()` call. Doing so offers a few advantages: `thr_create()` can create a thread much faster then a `fork()` could create a new process, and any communication between the ***server*** and the new thread can be done with common variables. This technique makes the implementation of the socket server much easier to understand and should also make it respond much faster to incoming requests.

The server program first sets up all the needed socket information. This is the basic setup for most socket servers. The server then enters an endless loop, waiting to service a socket port. When a message is sent to the socket port, the server wakes up and creates a new thread to handle the request. Notice that the server creates the new thread as a detached thread and also passes the socket descriptor as an argument to the new thread.

The newly created thread can then read or write, in any fashion it wants, to the socket descriptor that was passed to it. At this point the server could be creating a new thread or waiting for the next message to arrive. The key is that the server thread does not care what happens to the new thread after it creates it.

In our example, the created thread reads from the socket descriptor and then increments a global variable. This global variable keeps track of the number of requests that were made to the server. Notice that a mutex lock is used to protect access to the shared global variable. The lock is needed because many threads might try to increment the same variable at the same time. The mutex lock provides serial access to the shared variable. See how easy it is to share information among the new threads! If each of the threads were a process, then a significant effort would have to be made to share this information among the processes.

The client piece of the example sends a given number of messages to the server. This client code could also be run from different machines by multiple users, thus increasing the need for concurrency in the server process.

The source code to `soc_server.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/uio.h>
#include <unistd.h>
#include <thread.h>

/* the TCP port that is used for this example */
#define TCP_PORT   6500

/* function prototypes and global variables */
void *do_chld(void *);
mutex_t lock;
int     service_count;

main()
{
        int     sockfd, newsockfd, clilen;
        struct sockaddr_in cli_addr, serv_addr;
        thread_t chld_thr;

        if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                fprintf(stderr,"server: can't open stream socket\n"), exit((

        memset((char *) &serv_addr, 0, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        serv_addr.sin_port = htons(TCP_PORT);

        if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) <
0)
                fprintf(stderr,"server: can't bind local address\n"), exit((

        /* set the level of thread concurrency we desire */
        thr_setconcurrency(5);

        listen(sockfd, 5);

        for(;;){
                clilen = sizeof(cli_addr);
                newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);

                if(newsockfd < 0)
                        fprintf(stderr,"server: accept error\n"), exit(0);

                /* create a new thread to process the incomming request */
                thr_create(NULL, 0, do_chld, (void *) newsockfd, THR_DETACHE
&chld_thr);

                /* the server is now free to accept another socket request *
                }
        return(0);
}

/*
        This is the routine that is executed from a new thread
*/

void *do_chld(void *arg)
{
int     mysocfd = (int) arg;
char    data[100];
int     i;

        printf("Child thread [%d]: Socket number = %d\n", thr_self(), mysocf

        /* read from the given socket */
        read(mysocfd, data, 40);

        printf("Child thread [%d]: My data = %s\n", thr_self(), data);

        /* simulate some processing */
        for (i=0;i<1000000*thr_self();i++);

        printf("Child [%d]: Done Processing...\n", thr_self());

        /* use a mutex to update the global service counter */
        mutex_lock(&lock);

        service_count++;
        mutex_unlock(&lock);

        printf("Child thread [%d]: The total sockets served = %d\n", thr_sel

        /* close the socket and exit this thread */
        close(mysocfd);
        thr_exit((void *)0);
}
```

Using Many Threads

This example that shows how easy it is to create many threads of execution in Solaris. Because of the lightweight nature of threads, it is possible to create literally thousands of threads. Most applications may not need a very large number of threads, but this example shows just how lightweight the threads can be.

We have said before that anything you can do with threads, you can do without them. This may be a case where it would be very hard to do without threads. If you have some spare time (and lots of memory), try implementing this program by using processes, instead of threads. If you try this, you will see why threads can have an advantage over processes.

This program takes as an argument the number of threads to create. Notice that all the threads are created with a user-defined stack size, which limits the amount of memory that the threads will need for execution. The stack size for a given thread can be hard to calculate, so some testing usually needs to be done to see if the chosen stack size will work. You may want to change the stack size in this program and see how much you can lower it before things stop working. The Solaris threads library provides the `thr_min_stack()` call, which returns the minimum allowed stack size. Take care when adjusting the size of a threads stack. A stack overflow can happen quite easily to a thread with a small stack.

After each thread is created, it blocks, waiting on a mutex variable. This mutex variable was locked before any of the threads were created, which prevents the threads from proceeding in their execution. When all of the threads have been created and the user presses Return, the mutex variable is unlocked, allowing all the threads to proceed.

After the main thread has created all the threads, it waits for user input and then tries to join all the threads. Notice that the `thr_join()` call does not care what thread it joins; it is just counting the number of joins it makes.

This example is rather trivial and does not serve any real purpose except to show that it is possible to create a lot of threads in one process. However, there are situations when many threads are needed in an application. An example might be a network port server, where a thread is created each time an incoming or outgoing request is made.

The source to `many_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <stdlib.h>
#include <thread.h>

/* function prototypes and global varaibles */
void *thr_sub(void *);
mutex_t lock;

main(int argc, char **argv)
{
int i, thr_count = 100;
char buf;

/* check to see if user passed an argument
   -- if so, set the number of threads to the value
      passed to the program */

if (argc == 2) thr_count = atoi(argv[1]);

printf("Creating %d threads...\n", thr_count);

/* lock the mutex variable -- this mutex is being used to
   keep all the other threads created from proceeding   */
mutex_lock(&lock);

/* create all the threads -- Note that a specific stack size is
   given.  Since the created threads will not use all of the
   default stack size, we can save memory by reducing the threads'
   stack size */

for (i=0;i<thr_count;i++) {
        thr_create(NULL,2048,thr_sub,0,0,NULL);
        }

printf("%d threads have been created and are running!\n", i);
printf("Press <return> to join all the threads...\n", i);

/* wait till user presses return, then join all the threads */
gets(&buf);

printf("Joining %d threads...\n", thr_count);

/* now unlock the mutex variable, to let all the threads proceed */
mutex_unlock(&lock);

/* join the threads */
for (i=0;i<thr_count;i++)
        thr_join(0,0,0);

printf("All %d threads have been joined, exiting...\n", thr_count);
return(0);
}

/* The routine that is executed by the created threads */

void *thr_sub(void *arg)
{
/* try to lock the mutex variable -- since the main thread has
   locked the mutex before the threads were created, this thread
   will block until the main thread unlock the mutex */

mutex_lock(&lock);

printf("Thread %d is exiting...\n", thr_self());

/* unlock the mutex to allow another thread to proceed */
mutex_unlock(&lock);

/* exit the thread */
return((void *)0);
}
```

Real-time Thread Example

This example uses the Solaris real-time extensions to make a single bound thread within a process run in the real-time scheduling class. Using a thread in the real-time class is more desirable than running a whole process in the real-time class, because of the many problems that can arise with a process in a real-time state. For example, it would not be desirable for a process to perform any I/O or large memory operations while in realtime, because a real-time process has priority over system-related processes; if a real-time process requests a page fault, it can starve, waiting for the system to fault in a new page. We can limit this exposure by using threads to execute only the instructions that need to run in realtime.

Since this book does not cover the concerns that arise with real-time programming, we have included this code only as an example of how to promote a thread into the real-time class. You must be very careful when you use real-time threads in your applications. For more information on real-time programming, see the Solaris documentation.

This example should be safe from the pitfalls of real-time programs because of its simplicity. However, changing this code in any way could have adverse affects on your system.

The example creates a new thread from the main thread. This new thread is then promoted to the real-time class by looking up the real-time class ID and then setting a real-time priority for the thread. After the thread is running in realtime, it simulates some processing. Since a thread in the real-time class can have an infinite time quantum, the process is allowed to stay on a CPU as long as it likes. The time quantum is the amount of time a thread is allowed to stay running on a CPU. For the timesharing class, the time quantum (time-slice) is 1/100th of a second by default.

In this example, we set the time quantum for the real-time thread to infinity. That is, it can stay running as long as it likes; it will not be preempted or scheduled off the CPU. If you run this example on a UP machine, it will have the effect of stopping your system for a few seconds while the thread simulates its processing. The system does not actually stop, it is just working in the real-time thread. When the real-time thread finishes its processing, it exits and the system returns to normal.

Using real-time threads can be quite useful when you need an extremely high priority and response time but can also cause big problems if it not used properly. Also note that this example must be run as root or have root execute permissions.

The source to `rt_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <string.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>

/* thread prototype */
void *rt_thread(void *);

main()
{

/* create the thread that will run in realtime */
thr_create(NULL, 0, rt_thread, 0, THR_DETACHED, 0);

/* loop here forever, this thread is the TS scheduling class */
while (1) {
        printf("MAIN: In time share class... running\n");
        sleep(1);
        }

return(0);
}

/*
        This is the routine that is called by the created thread
*/

void *rt_thread(void *arg)
{
pcinfo_t pcinfo;
pcparms_t pcparms;
int i;

/* let the main thread run for a bit */
sleep(4);

/* get the class ID for the real-time class */
strcpy(pcinfo.pc_clname, "RT");

if (priocntl(0, 0, PC_GETCID, (caddr_t)&pcinfo) == -1)
```

```
        fprintf(stderr, "getting RT class id\n"), exit(1);

/* set up the real-time parameters */
pcparms.pc_cid = pcinfo.pc_cid;
((rtparms_t *)pcparms.pc_clparms)->rt_pri = 10;
((rtparms_t *)pcparms.pc_clparms)->rt_tqnsecs = 0;

/* set an infinite time quantum */
((rtparms_t *)pcparms.pc_clparms)->rt_tqsecs = RT_TQINF;

/* move this thread to the real-time scheduling class */
if (priocntl(P_LWPID, P_MYID, PC_SETPARMS, (caddr_t)&pcparms) == -1)
        fprintf(stderr, "Setting RT mode\n"), exit(1);

/* simulate some processing */
for (i=0;i<100000000;i++);

printf("RT_THREAD: NOW EXITING...\n");
thr_exit((void *)0);
}
```

POSIX Cancellation

This example uses the POSIX thread cancellation capability to kill a thread that is no longer needed. Random termination of a thread can cause problems in threaded applications, because a thread may be holding a critical lock when it is terminated. Since the lock was help before the thread was terminated, another thread may deadlock, waiting for that same lock. The thread cancellation capability enables you to control when a thread can be terminated. The example also demonstrates the capabilities of the POSIX thread library in implementing a program that performs a multithreaded search.

This example simulates a multithreaded search for a given number by taking random guesses at a target number. The intent here is to simulate the same type of search that a database might execute. For example, a database might create threads to start searching for a data item; after some amount of time, one or more threads might return with the target data item.

If a thread guesses the number correctly, there is no need for the other threads to continue their search. This is where thread cancellation can help. The thread that finds the number first should cancel the other threads that are still searching for the item and then return the results of the search.

The threads involved in the search can call a cleanup function that can clean up the threads resources before it exits. In this case, the cleanup function prints the progress of the thread when it was cancelled.

The source to `posix_cancel.c`:

```
 #define _REENTRANT
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

/* defines the number of searching threads */
#define NUM_THREADS 25

/* function prototypes */
void *search(void *);
void print_it(void *);

/* global variables */
pthread_t  threads[NUM_THREADS];
pthread_mutex_t lock;
```

```
int tries;

main()
{
int i;
int pid;

/* create a number to search for */
pid = getpid();

/* initialize the mutex lock */
pthread_mutex_init(&lock, NULL);
printf("Searching for the number = %d...\n", pid);

/* create the searching threads */
for (i=0;i<NUM_THREADS;i++)
        pthread_create(&threads[i], NULL, search, (void *)pid);

/* wait for (join) all the searching threads */
for (i=0;i<NUM_THREADS;i++)
        pthread_join(threads[i], NULL);

printf("It took %d tries to find the number.\n", tries);

/* exit this thread */
pthread_exit((void *)0);
}

/*
        This is the cleanup function that is called when
        the threads are cancelled
*/

void print_it(void *arg)
{
int *try = (int *) arg;
pthread_t tid;

/* get the calling thread's ID */
tid = pthread_self();

/* print where the thread was in its search when it was cancelled */
printf("Thread %d was canceled on its %d try.\n", tid, *try);
}

/*
        This is the search routine that is executed in each thread
*/

void *search(void *arg)
{
int num = (int) arg;
int i=0, j;
pthread_t tid;

/* get the calling thread ID */
tid = pthread_self();

/* use the thread ID to set the seed for the random number generator */
srand(tid);

/* set the cancellation parameters --
   - Enable thread cancellation
   - Defer the action of the cancellation
*/

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

/* push the cleanup routine (print_it) onto the thread
   cleanup stack.  This routine will be called when the
   thread is cancelled.  Also note that the pthread_cleanup_push
   call must have a matching pthread_cleanup_pop call.  The
   push and pop calls MUST be at the same lexical level
   within the code */

/* pass address of `i' since the current value of `i' is not
   the one we want to use in the cleanup function */

pthread_cleanup_push(print_it, (void *)&i);

/* loop forever */
while (1) {
        i++;

        /* does the random number match the target number? */
        if (num == rand()) {

                /* try to lock the mutex lock --
                   if locked, check to see if the thread has been cancelled
                   if not locked then continue */

                while (pthread_mutex_trylock(&lock) == EBUSY)
                             pthread_testcancel();

                /* set the global variable for the number of tries */

                tries = i;

                printf("thread %d found the number!\n", tid);

                /* cancel all the other threads */
                for (j=0;j<NUM_THREADS;j++)
                        if (threads[j] != tid) pthread_cancel(threads[j]);

                /* break out of the while loop */
                break;
                }

        /* every 100 tries check to see if the thread has been cancelled
           if the thread has not been cancelled then yield the thread's
           LWP to another thread that may be able to run */

        if (i%100 == 0) {
                pthread_testcancel();
                sched_yield();
                }
        }

/* The only way we can get here is when the thread breaks out
   of the while loop.  In this case the thread that makes it here
   has found the number we are looking for and does not need to run
   the thread cleanup function.  This is why the pthread_cleanup_pop
   function is called with a 0 argument; this will pop the cleanup
   function off the stack without executing it */

pthread_cleanup_pop(0);
return((void *)0);
}
```

Software Race Condition

This example shows a trivial software race condition. A software race condition occurs when the execution of a program is affected by the order and timing of a threads execution. Most software race conditions can be alleviated by using synchronization variables to control the threads' timing and access of shared resources. If a program depends on order of execution, then threading that program may not be a good solution, because the order in which threads execute is nondeterministic.

In the example, `thr_continue()` and `thr_suspend()` calls continue and suspend a given thread, respectively. Although both of these calls are valid, use caution when implementing them. It is very hard to determine where a thread is in its execution. Because of this, you may not be able to tell where the thread will suspend when the call to `thr_suspend()` is made. This behavior can cause problems in threaded code if not used properly.

The following example uses `thr_continue()` and `thr_suspend()` to try to control when a thread starts and stops. The example looks trivial, but, as you will see, can cause a big problem.

Do you see the problem? If you guessed that the program would eventually suspend itself, you were correct! The example attempts to flip-flop between the main thread and a subroutine thread. Each thread continues the other thread and then suspends itself.

Thread A continues thread B and then suspends thread A; now the continued thread B can continue thread A and then suspend itself. This should continue back and forth all day long, right? Wrong! We can't guarantee that each thread will continue the other thread and then suspend itself in one atomic action, so a software race condition could be created. Calling `thr_continue()` on a running thread and calling `thr_suspend()` on a suspended thread has no effect, so we don't know if a thread is already running or suspended.

If thread A continues thread B and if between the time thread A suspends itself, thread B continues thread A, then both of the threads will call `thr_suspend()`. This is the race condition in this program that will cause the whole process to become suspended.

It is very hard to use these calls, because you never really know the state of a thread. If you don't know exactly where a thread is in its execution, then you don't know what locks it holds and where it will stop when you suspend it.

The source to `sw_race.c`

`Tgrep`: Threadeds version of UNIX `grep`

`Tgrep` is a multi-threaded version of `grep`. `Tgrep` supports all but the -w (word search) options of the normal `grep` command, and a few options that are only avaliable to `Tgrep`. The real change from `grep`, is that `Tgrep` will recurse down through sub-directories and search all files for the target string. `Tgrep` searches files like the following command:

```
find <start path> -name "<file/directory pattern>" -exec \ (Line wrapped)
        grep <options> <target> /dev/null {} \;
```

An example of this would be (run from this `Tgrep` directory)

```
% find . -exec grep thr_create /dev/null {} \;
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work,
%
Running the same command with timex:
real        4.26
user        0.64
sys         2.81
```

The same search run with `Tgrep` would be

```
% {\tt Tgrep} thr_create
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work,
%
```

```
Running the same command with timex:
real        0.79
user        0.62
sys         1.50
```

`Tgrep` gets the results almost four times faster. The numbers above where gathered on a SS20 running 5.5 (build 18) with 4 50MHz CPUs.

You can also filter the files that you want `Tgrep` to search like you can with find. The next two commands do the same thing, just `Tgrep` gets it done faster.

```
find . -name "*.c" -exec grep thr_create /dev/null {} \;
and
{\tt Tgrep} -p '.*\.c$' thr_create
```

The -p option will allow `Tgrep` to search only files that match the "regular expression" file pattern string. This option does NOT use shell expression, so to stop `Tgrep` from seeing a file named foobar.cyou must add the "$" meta character to the pattern and escape the real ``." character.

Some of the other `Tgrep` only options are -r, -C, -P, -e, -B, -S and -Z. The -r option stops `Tgrep` from searching any sub-directories, in other words, search only the local directory, but -l was taken. The -C option will search for and print "continued" lines like you find in Makefile. Note the differences in the results of grep and `Tgrep` run in the current directory.

The `Tgrep` output prints the continued lines that ended with the "character. In the case of grep I would not have seen the three values assigned to SUBDIRS, but `Tgrep` shows them to me (Common, Solaris, Posix).

The -P option I use when I am sending the output of a long search to a file and want to see the "progress" of the search. The -P option will print a "." (dot) on stderr for every file (or groups of files depending on the value of the -P argument) `Tgrep` searches.

The -e option will change the way `Tgrep` uses the target string. `Tgrep` uses two different patter matching systems. The first (with out the -e option) is a literal string match call Boyer-Moore. If the -e option is used, then a MT-Safe PD version of regular expression is used to search for the target string as a regexp with meta characters in it. The regular expression method is slower, but `Tgrep` needed the functionality. The -Z option will print help on the meta characters `Tgrep` uses.

The -B option tells `Tgrep` to use the value of the environment variable called TGLIMIT to limit the number of threads it will use during a search. This option has no affect if TGLIMIT is not set. `Tgrep` can "eat" a system alive, so the -B option was a way to run `Tgrep` on a system with out having other users scream at you.

The last new option is -S. If you want to see how things went while `Tgrep` was searching, you can use this option to print statistic about the number of files, lines, bytes, matches, threads created, etc.

Here is an example of the -S options output. (again run in the current directory)

```
% {\tt Tgrep} -S zimzap

----------------- {\tt Tgrep} Stats. --------------------
Number of directories searched:          7
Number of files searched:                37
Number of lines searched:                9504
Number of matching lines to target:      0
Number of cascade threads created:       7
Number of search threads created:        20
Number of search threads from pool:      17
Search thread pool hit rate:             45.95%
Search pool overall size:                 20
Search pool size limit:                   58
Number of search threads destroyed:       0
Max # of threads running concurrenly:     20
Total run time, in seconds.               1
Work stopped due to no FD's:  (058)       0 Times, 0.00%
Work stopped due to no work on Q:         19 Times, 43.18%
Work stopped due to TGLIMITS: (Unlimited) 0 Times, 0.00%
----------------------------------------------------
%
```

For more information on the usage and options, see the man page `Tgrep`

The `Tgrep.c` source code is:

```
/* Copyright (c) 1993, 1994  Ron Winacott                               */
/* This program may be used, copied, modified, and redistributed freely */
/* for ANY purpose, so long as this notice remains intact.              */

#define _REENTRANT

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include <sys/stat.h>
#ifdef __sparc
#include <note.h> /* warlock/locklint */
#else
#define NOTE(s)
#endif
#include <dirent.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <thread.h>
#include <synch.h>

#include "version.h"
#include "pmatch.h"
#include "debug.h"

#define PATH_MAX                1024 /* max # of characters in a path name
#define HOLD_FDS                6  /* stdin,out,err and a buffer */
#define UNLIMITED               99999 /* The default tglimit */
#define MAXREGEXP               10  /* max number of -e options */

#define FB_BLOCK                0x00001
#define FC_COUNT                0x00002
#define FH_HOLDNAME             0x00004
#define FI_IGNCASE              0x00008
#define FL_NAMEONLY             0x00010
#define FN_NUMBER               0x00020
#define FS_NOERROR              0x00040
#define FV_REVERSE              0x00080
#define FW_WORD                 0x00100
#define FR_RECUR                0x00200
#define FU_UNSORT               0x00400
#define FX_STDIN                0x00800
#define TG_BATCH                0x01000
#define TG_FILEPAT              0x02000
#define FE_REGEXP               0x04000
#define FS_STATS                0x08000
#define FC_LINE                 0x10000
#define TG_PROGRESS             0x20000
```

```
#define FILET                   1
#define DIRT                    2
#define ALPHASIZ                128

/*
 * New data types
 */

typedef struct work_st {
    char                *path;
    int                 tp;
    struct work_st      *next;
} work_t;

typedef struct out_st {
    char                *line;
    int                 line_count;
    long                byte_count;
    struct out_st       *next;
} out_t;

typedef struct bm_pattern {     /* Boyer - Moore pattern                */
        short            p_m;           /* length of pattern string     */
        short            p_r[ALPHASIZ]; /* "r" vector                   */
        short           *p_R;           /* "R" vector                   */
        char            *p_pat;         /* pattern string               */
} BM_PATTERN;


/*
 * Prototypes
 */

/* bmpmatch.c */
extern BM_PATTERN *bm_makepat(char *);
extern char *bm_pmatch(BM_PATTERN *, register char *);
extern void bm_freepat(BM_PATTERN *);
/* pmatch.c */
extern char *pmatch(register PATTERN *, register char *, int *);
extern PATTERN *makepat(char *string, char *);
extern void freepat(register PATTERN *);
extern void printpat(PATTERN *);

#include "proto.h"  /* function prototypes of main.c */

void *SigThread(void *arg);
void sig_print_stats(void);

/*
 * Global data
 */

BM_PATTERN      *bm_pat;  /* the global target read only after main */
NOTE(READ_ONLY_DATA(bm_pat))

PATTERN         *pm_pat[MAXREGEXP];  /* global targets read only for pmatch
NOTE(READ_ONLY_DATA(pm_pat))

mutex_t global_count_lk;
int     global_count = 0;
NOTE(MUTEX_PROTECTS_DATA(global_count_lk, global_count))
NOTE(DATA_READABLE_WITHOUT_LOCK(global_count))  /* see prnt_stats() */

work_t  *work_q = NULL;
cond_t  work_q_cv;
mutex_t work_q_lk;
int     all_done = 0;
int     work_cnt = 0;
int     current_open_files = 0;
int     tglimit = UNLIMITED;    /* if -B limit the number of threads */
NOTE(MUTEX_PROTECTS_DATA(work_q_lk, work_q all_done work_cnt \
                         current_open_files tglimit))
```

```
work_t  *search_q = NULL;
mutex_t search_q_lk;
cond_t  search_q_cv;
int     search_pool_cnt = 0;    /* the count in the pool now */
int     search_thr_limit = 0;   /* the max in the pool */
NOTE(MUTEX_PROTECTS_DATA(search_q_lk, search_q search_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(search_pool_cnt)) /* see prnt_stats() */
NOTE(READ_ONLY_DATA(search_thr_limit))

work_t  *cascade_q = NULL;
mutex_t cascade_q_lk;
cond_t  cascade_q_cv;
int     cascade_pool_cnt = 0;
int     cascade_thr_limit = 0;
NOTE(MUTEX_PROTECTS_DATA(cascade_q_lk, cascade_q cascade_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(cascade_pool_cnt))  /* see prnt_stats() */
NOTE(READ_ONLY_DATA(cascade_thr_limit))

int     running = 0;
mutex_t running_lk;
NOTE(MUTEX_PROTECTS_DATA(running_lk, running))

sigset_t set, oldset;
NOTE(READ_ONLY_DATA(set oldset))

mutex_t stat_lk;
time_t  st_start = 0;
int     st_dir_search = 0;
int     st_file_search = 0;
int     st_line_search = 0;
int     st_cascade = 0;
int     st_cascade_pool = 0;
int     st_cascade_destroy = 0;
int     st_search = 0;
int     st_pool = 0;
int     st_maxrun = 0;
int     st_worknull = 0;
int     st_workfds = 0;
int     st_worklimit = 0;
int     st_destroy = 0;
NOTE(MUTEX_PROTECTS_DATA(stat_lk, st_start st_dir_search st_file_search \
                         st_line_search st_cascade st_cascade_pool \
                         st_cascade_destroy st_search st_pool st_maxrun \
                         st_worknull st_workfds st_worklimit st_destroy))

int     progress_offset = 1;
NOTE(READ_ONLY_DATA(progress_offset))

mutex_t output_print_lk;
/* output_print_lk used to print multi-line output only */
int     progress = 0;
NOTE(MUTEX_PROTECTS_DATA(output_print_lk, progress))

unsigned int    flags = 0;
int     regexp_cnt = 0;
char    *string[MAXREGEXP];
int     debug = 0;
int     use_pmatch = 0;
char    file_pat[255];  /* file patten match */
PATTERN *pm_file_pat; /* compiled file target string (pmatch()) */
NOTE(READ_ONLY_DATA(flags regexp_cnt string debug use_pmatch \
                    file_pat pm_file_pat))


/*
 * Locking ording.
 */
NOTE(LOCK_ORDER(output_print_lk stat_lk))

/*
 * Main: This is where the fun starts
 */

int
main(int argc, char **argv)
{
    int         c,out_thr_flags;
    long        max_open_files = 0l, ncpus = 0l;
    extern int  optind;
    extern char *optarg;
    NOTE(READ_ONLY_DATA(optind optarg))
    int         prio = 0;
    struct stat sbuf;
    thread_t    tid,dtid;
    void        *status;
    char        *e = NULL, *d = NULL; /* for debug flags */
    int         debug_file = 0;
    int         err = 0, i = 0, pm_file_len = 0;
    work_t      *work;
    int         restart_cnt = 10;

    flags = FR_RECUR;  /* the default */

    thr_setprio(thr_self(),127);  /* set me up HIGH */
    while ((c = getopt(argc, argv, "d:e:bchilnsvwruf:p:BCSZzHP:")) != EOF) 
        switch (c) {
#ifdef DEBUG
        case 'd':
            debug = atoi(optarg);
            if (debug == 0)
                debug_usage();

            d = optarg;
            fprintf(stderr,"tgrep: Debug on at level(s) ");
            while (*d) {
                for (i=0; i<9; i++)
                    if (debug_set[i].level == *d) {
                        debug_levels |= debug_set[i].flag;
                        fprintf(stderr,"%c ",debug_set[i].level);
                        break;
                    }
                d++;
            }
            fprintf(stderr,"\n");
            break;
        case 'f':
            debug_file = atoi(optarg);
            break;
#endif      /* DEBUG */
        case 'B':
            flags |= TG_BATCH;
            if ((e = getenv("TGLIMIT"))) {
                tglimit = atoi(e);
            }
            else {
                if (!(flags & FS_NOERROR))  /* order dependent! */
                    fprintf(stderr,"env TGLIMIT not set, overriding -B\n");
                flags &= ~TG_BATCH;
            }
            break;
        case 'p':
            flags |= TG_FILEPAT;
            strcpy(file_pat,optarg);
            pm_file_pat = makepat(file_pat,NULL);
            break;
        case 'P':
            flags |= TG_PROGRESS;
            progress_offset = atoi(optarg);
            break;
        case 'S': flags |= FS_STATS;    break;
        case 'b': flags |= FB_BLOCK;    break;
        case 'c': flags |= FC_COUNT;    break;
        case 'h': flags |= FH_HOLDNAME; break;
```

```
            case 'i': flags |= FI_IGNCASE;  break;
            case 'l': flags |= FL_NAMEONLY; break;
            case 'n': flags |= FN_NUMBER;   break;
            case 's': flags |= FS_NOERROR;  break;
            case 'v': flags |= FV_REVERSE;  break;
            case 'w': flags |= FW_WORD;     break;
            case 'r': flags &= ~FR_RECUR;   break;
            case 'C': flags |= FC_LINE;     break;
            case 'e':
                if (regexp_cnt == MAXREGEXP) {
                    fprintf(stderr,"Max number of regexp's (%d) exceeded!\n",
                           MAXREGEXP);
                    exit(1);
                }
                flags |= FE_REGEXP;
                if ((string[regexp_cnt] =(char *)malloc(strlen(optarg)+1))==NULI
                    fprintf(stderr,"tgrep: No space for search string(s)\n");
                    exit(1);
                }
                memset(string[regexp_cnt],0,strlen(optarg)+1);
                strcpy(string[regexp_cnt],optarg);
                regexp_cnt++;
                break;
            case 'z':
            case 'Z': regexp_usage();
                break;
            case 'H':
            case '?':
            default : usage();
            }
        }

        if (!(flags & FE_REGEXP)) {
           if (argc - optind < 1) {
                fprintf(stderr,"tgrep: Must supply a search string(s) "
                        "and file list or directory\n");
                usage();
            }
            if ((string[0]=(char *)malloc(strlen(argv[optind])+1))==NULL){
                fprintf(stderr,"tgrep: No space for search string(s)\n");
                exit(1);
            }
            memset(string[0],0,strlen(argv[optind])+1);
            strcpy(string[0],argv[optind]);
            regexp_cnt=1;
            optind++;
        }

        if (flags & FI_IGNCASE)
            for (i=0; i<regexp_cnt; i++)
                uncase(string[i]);

    #ifdef __lock_lint
        /*
        ** This is NOT somthing you really want to do. This
        ** function calls are here ONLY for warlock/locklint !!!
        */
        pm_pat[i] = makepat(string[i],NULL);
        bm_pat = bm_makepat(string[0]);
        bm_freepat(bm_pat);  /* stop it from becomming a root */
    #else
        if (flags & FE_REGEXP) {
            for (i=0; i<regexp_cnt; i++)
                pm_pat[i] = makepat(string[i],NULL);
            use_pmatch = 1;
        }
        else {
            bm_pat = bm_makepat(string[0]); /* only one allowed */
        }
    #endif

        flags |= FX_STDIN;
        max_open_files = sysconf(_SC_OPEN_MAX);
        ncpus = sysconf(_SC_NPROCESSORS_ONLN);
        if ((max_open_files - HOLD_FDS - debug_file) < 1) {
            fprintf(stderr,"tgrep: You MUST have at lest ONE fd "
                    "that can be used, check limit (>10)\n");
            exit(1);
        }
        search_thr_limit = max_open_files - HOLD_FDS - debug_file;
        cascade_thr_limit = search_thr_limit / 2;
        /* the number of files that can by open */
        current_open_files = search_thr_limit;

        mutex_init(&stat_lk,USYNC_THREAD,"stat");
        mutex_init(&global_count_lk,USYNC_THREAD,"global_cnt");
        mutex_init(&output_print_lk,USYNC_THREAD,"output_print");
        mutex_init(&work_q_lk,USYNC_THREAD,"work_q");
        mutex_init(&running_lk,USYNC_THREAD,"running");
        cond_init(&work_q_cv,USYNC_THREAD,"work_q");
        mutex_init(&search_q_lk,USYNC_THREAD,"search_q");
        cond_init(&search_q_cv,USYNC_THREAD,"search_q");
        mutex_init(&cascade_q_lk,USYNC_THREAD,"cascade_q");
        cond_init(&cascade_q_cv,USYNC_THREAD,"cascade_q");

        if ((argc == optind) && ((flags & TG_FILEPAT) || (flags & FR_RECUR))) {
            add_work(".",DIRT);
            flags = (flags & ~FX_STDIN);
        }
        for ( ; optind < argc; optind++) {
            restart_cnt = 10;
            flags = (flags & ~FX_STDIN);
          STAT_AGAIN:
            if (stat(argv[optind], &sbuf)) {
                if (errno == EINTR) { /* try again !, restart */
                    if (--restart_cnt)
                        goto STAT_AGAIN;
                }
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can't stat file/dir %s, %s\n",
                            argv[optind], strerror(errno));
                continue;
            }
            switch (sbuf.st_mode & S_IFMT) {
            case S_IFREG :
                if (flags & TG_FILEPAT) {
                    if (pmatch(pm_file_pat, argv[optind], &pm_file_len))
                        add_work(argv[optind],FILET);
                }
                else {
                    add_work(argv[optind],FILET);
                }
                break;
            case S_IFDIR :
                if (flags & FR_RECUR) {
                    add_work(argv[optind],DIRT);
                }
                else {
                    if (!(flags & FS_NOERROR))
                        fprintf(stderr,"tgrep: Can't search directory %s, "
                                "-r option is on. Directory ignored.\n",
                                argv[optind]);
                }
                break;
            }
        }

        NOTE(COMPETING_THREADS_NOW)  /* we are goinf threaded */

        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            st_start = time(NULL);
            mutex_unlock(&stat_lk);
```

```
#ifdef SIGNAL_HAND
        /*
        ** setup the signal thread so the first call to SIGINT will
        ** only print stats, the second will interupt.
        */
        sigfillset(&set);
        thr_sigsetmask(SIG_SETMASK, &set, &oldset);
        if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
            thr_sigsetmask(SIG_SETMASK,&oldset,NULL);
            fprintf(stderr,"SIGINT for stats NOT setup\n");
        }
        thr_yield(); /* give the other thread time */
#endif /* SIGNAL_HAND */
    }

    thr_setconcurrency(3);

    if (flags & FX_STDIN) {
        fprintf(stderr,"tgrep: stdin option is not coded at this time\n");
        exit(0);                        /* XXX Need to fix this SOON */
        search_thr(NULL);  /* NULL is not understood in search_thr() */
        if (flags & FC_COUNT) {
            mutex_lock(&global_count_lk);
            printf("%d\n",global_count);
            mutex_unlock(&global_count_lk);
        }
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            prnt_stats();
            mutex_unlock(&stat_lk);
        }
        exit(0);
    }

    mutex_lock(&work_q_lk);
    if (!work_q) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: No files to search.\n");
        exit(0);
    }
    mutex_unlock(&work_q_lk);

    DP(DLEVEL1,("Starting to loop through the work_q for work\n"));

    /* OTHER THREADS ARE RUNNING */
    while (1) {
        mutex_lock(&work_q_lk);
        while ((work_q == NULL || current_open_files == 0 || tglimit <= 0) &
               all_done == 0) {
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                if (work_q == NULL)
                    st_worknull++;
                if (current_open_files == 0)
                    st_workfds++;
                if (tglimit <= 0)
                    st_worklimit++;
                mutex_unlock(&stat_lk);
            }
            cond_wait(&work_q_cv,&work_q_lk);
        }
        if (all_done != 0) {
            mutex_unlock(&work_q_lk);
            DP(DLEVEL1,("All_done was set to TRUE\n"));
            goto OUT;
        }
        work = work_q;
        work_q = work->next;  /* maybe NULL */
        work->next = NULL;
        current_open_files--;
        mutex_unlock(&work_q_lk);
        tid = 0;
        switch (work->tp) {
        case DIRT:
            mutex_lock(&cascade_q_lk);
            if (cascade_pool_cnt) {
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_cascade_pool++;
                    mutex_unlock(&stat_lk);
                }
                work->next = cascade_q;
                cascade_q = work;
                cond_signal(&cascade_q_cv);
                mutex_unlock(&cascade_q_lk);
                DP(DLEVEL2,("Sent work to cascade pool thread\n"));
            }
            else {
                mutex_unlock(&cascade_q_lk);
                err = thr_create(NULL,0,cascade,(void *)work,
                                 THR_DETACHED|THR_DAEMON|THR_NEW_LWP
                                 ,&tid);
                DP(DLEVEL2,("Sent work to new cascade thread\n"));
                thr_setprio(tid,64);  /* set cascade to middle */
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_cascade++;
                    mutex_unlock(&stat_lk);
                }
            }
            break;
        case FILET:
            mutex_lock(&search_q_lk);
            if (search_pool_cnt) {
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_pool++;
                    mutex_unlock(&stat_lk);
                }
                work->next = search_q;  /* could be null */
                search_q = work;
                cond_signal(&search_q_cv);
                mutex_unlock(&search_q_lk);
                DP(DLEVEL2,("Sent work to search pool thread\n"));
            }
            else {
                mutex_unlock(&search_q_lk);
                err = thr_create(NULL,0,search_thr,(void *)work,
                                 THR_DETACHED|THR_DAEMON|THR_NEW_LWP
                                 ,&tid);
                thr_setprio(tid,0);  /* set search to low */
                DP(DLEVEL2,("Sent work to new search thread\n"));
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_search++;
                    mutex_unlock(&stat_lk);
                }
            }
            break;
        default:
            fprintf(stderr,"tgrep: Internal error, work_t->tp no valid\n");
            exit(1);
        }
        if (err) {  /* NEED TO FIX THIS CODE. Exiting is just wrong */
            fprintf(stderr,"Cound not create new thread!\n");
            exit(1);
        }
    }

OUT:
   if (flags & TG_PROGRESS) {
       if (progress)
           fprintf(stderr,".\n");
        else
            fprintf(stderr,"\n");
    }
    /* we are done, print the stuff. All other threads ar parked */
    if (flags & FC_COUNT) {
        mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    return(0); /* should have a return from main */
}


/*
 * Add_Work: Called from the main thread, and cascade threads to add file
 * and directory names to the work Q.
 */
int
add_work(char *path,int tp)
{
    work_t      *wt,*ww,*wp;

    if ((wt = (work_t *)malloc(sizeof(work_t))) == NULL)
        goto ERROR;
    if ((wt->path = (char *)malloc(strlen(path)+1)) == NULL)
        goto ERROR;

    strcpy(wt->path,path);
    wt->tp = tp;
    wt->next = NULL;
    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        if (wt->tp == DIRT)
            st_dir_search++;
        else
            st_file_search++;
        mutex_unlock(&stat_lk);
    }
    mutex_lock(&work_q_lk);
    work_cnt++;
    wt->next = work_q;
    work_q = wt;
    cond_signal(&work_q_cv);
    mutex_unlock(&work_q_lk);
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Could not add %s to work queue. Ignored\n",
                path);
    return(-1);
}

/*
 * Search thread: Started by the main thread when a file name is found
 * on the work Q to be serached. If all the needed resources are ready
 * a new search thread will be created.
 */
void *
search_thr(void *arg) /* work_t *arg */
{
    FILE        *fin;
    char        fin_buf[(BUFSIZ*4)];  /* 4 Kbytes */
    work_t      *wt,std;
    int         line_count;
    char        rline[128];
    char        cline[128];
    char        *line;
    register char *p,*pp;
    int            pm_len;
    int         len = 0;
```

```
long        byte_count;
long        next_line;
int         show_line;  /* for the -v option */
register int slen,plen,i;
out_t       *out = NULL;    /* this threads output list */

thr_setprio(thr_self(),0);  /* set search to low */
thr_yield();
wt = (work_t *)arg; /* first pass, wt is passed to use. */

/* len = strlen(string);*/  /* only set on first pass */

while (1) {  /* reuse the search threads */
    /* init all back to zero */
    line_count = 0;
    byte_count = 0l;
    next_line = 0l;
    show_line = 0;

    mutex_lock(&running_lk);
    running++;
    mutex_unlock(&running_lk);
    mutex_lock(&work_q_lk);
    tglimit--;
    mutex_unlock(&work_q_lk);
    DP(DLEVEL5,("searching file (STDIO) %s\n",wt->path));

    if ((fin = fopen(wt->path,"r")) == NULL) {
        if (!(flags & FS_NOERROR)) {
            fprintf(stderr,"tgrep: %s. File \"%s\" not searched.\n",
                    strerror(errno),wt->path);
        }
        goto ERROR;
    }
    setvbuf(fin,fin_buf,_IOFBF,(BUFSIZ*4));  /* XXX */
    DP(DLEVEL5,("Search thread has opened file %s\n",wt->path));
    while ((fgets(rline,127,fin)) != NULL) {
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            st_line_search++;
            mutex_unlock(&stat_lk);
        }
        slen = strlen(rline);
        next_line += slen;
        line_count++;
        if (rline[slen-1] == '\n')
            rline[slen-1] = '\0';
        /*
        ** If the uncase flag is set, copy the read in line (rline)
        ** To the uncase line (cline) Set the line pointer to point at
        ** cline.
        ** If the case flag is NOT set, then point line at rline.
        ** line is what is compared, rline is waht is printed on a
        ** match.
        */
        if (flags & FI_IGNCASE) {
            strcpy(cline,rline);
            uncase(cline);
            line = cline;
        }
        else {
            line = rline;
        }
        show_line = 1;  /* assume no match, if -v set */
        /* The old code removed */
        if (use_pmatch) {
            for (i=0; i<regexp_cnt; i++) {
                if (pmatch(pm_pat[i], line, &pm_len)) {
                    if (!(flags & FV_REVERSE)) {
                        add_output_local(&out,wt,line_count,
                                         byte_count,rline);
                        continue_line(rline,fin,out,wt,
                                      &line_count,&byte_count);
                    }
                    else {
                        show_line = 0;
                    } /* end of if -v flag if / else block */
                    /*
                    ** if we get here on ANY of the regexp targets
                    ** jump out of the loop, we found a single
                    ** match so, do not keep looking!
                    ** If name only, do not keep searcthing the same
                    ** file, we found a single match, so close the file,
                    ** print the file name and move on to the next file.
                    */
                    if (flags & FL_NAMEONLY)
                        goto OUT_OF_LOOP;
                    else
                        goto OUT_AND_DONE;
                } /* end found a match if block */
            } /* end of the for pat[s] loop */
        }
        else {
            if (bm_pmatch( bm_pat, line)) {
                if (!(flags & FV_REVERSE)) {
                    add_output_local(&out,wt,line_count,byte_count,rline
                    continue_line(rline,fin,out,wt,
                                  &line_count,&byte_count);
                }
                else {
                    show_line = 0;
                }
                if (flags & FL_NAMEONLY)
                    goto OUT_OF_LOOP;
            }
        }
      OUT_AND_DONE:
        if ((flags & FV_REVERSE) && show_line) {
            add_output_local(&out,wt,line_count,byte_count,rline);
            show_line = 0;
        }
        byte_count = next_line;
    }
  OUT_OF_LOOP:
    fclose(fin);
    /*
    ** The search part is done, but before we give back the FD,
    ** and park this thread in the search thread pool, print the
    ** local output we have gathered.
    */
    print_local_output(out,wt);  /* this also frees out nodes */
    out = NULL; /* for the next time around, if there is one */
ERROR:
    DP(DLEVEL5,("Search done for %s\n",wt->path));
    free(wt->path);
    free(wt);

    notrun();
    mutex_lock(&search_q_lk);
    if (search_pool_cnt > search_thr_limit) {
        mutex_unlock(&search_q_lk);
        DP(DLEVEL5,("Search thread exiting\n"));
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            st_destroy++;
            mutex_unlock(&stat_lk);
        }
        return(0);
    }
    else {
        search_pool_cnt++;
        while (!search_q)
            cond_wait(&search_q_cv,&search_q_lk);
        search_pool_cnt--;
        wt = search_q;  /* we have work to do! */
        if (search_q->next)
            search_q = search_q->next;
        else
            search_q = NULL;
        mutex_unlock(&search_q_lk);
    }
}
/*NOTREACHED*/
}

/*
* Continue line: Speacial case search with the -C flag set. If you are
* searching files like Makefiles, some lines may have escape char's to
* contine the line on the next line. So the target string can be found, but
* no data is displayed. This function continues to print the escaped line
* until there are no more "\" chars found.
*/
int
continue_line(char *rline, FILE *fin, out_t *out, work_t *wt,
          int *lc, long *bc)
{
int len;
int cnt = 0;
char *line;
char nline[128];

if (!(flags & FC_LINE))
    return(0);

line = rline;
AGAIN:
len = strlen(line);
if (line[len-1] == '\\') {
    if ((fgets(nline,127,fin)) == NULL) {
        return(cnt);
    }
    line = nline;
    len = strlen(line);
    if (line[len-1] == '\n')
        line[len-1] = '\0';
    *bc = *bc + len;
    *lc++;
    add_output_local(&out,wt,*lc,*bc,line);
    cnt++;
    goto AGAIN;
}
return(cnt);
}

/*
* cascade: This thread is started by the main thread when directory names
* are found on the work Q. The thread reads all the new file, and directory
* names from the directory it was started when and adds the names to the
* work Q. (it finds more work!)
*/
void *
cascade(void *arg)  /* work_t *arg */
{
char        fullpath[1025];
int         restart_cnt = 10;
DIR         *dp;

char        dir_buf[sizeof(struct dirent) + PATH_MAX];
struct dirent *dent = (struct dirent *)dir_buf;
struct stat   sbuf;
char        *fpath;
work_t      *wt;
int         fl = 0, dl = 0;
int         pm_file_len = 0;

thr_setprio(thr_self(),64);  /* set search to middle */
thr_yield();  /* try toi give control back to main thread */
wt = (work_t *)arg;

while(1) {
    fl = 0;
    dl = 0;
    restart_cnt = 10;
    pm_file_len = 0;

    mutex_lock(&running_lk);
    running++;
    mutex_unlock(&running_lk);
    mutex_lock(&work_q_lk);
    tglimit--;
    mutex_unlock(&work_q_lk);

    if (!wt) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Bad work node passed to cascade\n");
        goto DONE;
    }
    fpath = (char *)wt->path;
    if (!fpath) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Bad path name passed to cascade\n");
        goto DONE;
    }
    DP(DLEVEL3,("Cascading on %s\n",fpath));
    if (( dp = opendir(fpath)) == NULL) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Can't open dir %s, %s. Ignored.\n",
                    fpath,strerror(errno));
        goto DONE;
    }
    while ((readdir_r(dp,dent)) != NULL) {
        restart_cnt = 10;  /* only try to restart the interupted 10 X *

        if (dent->d_name[0] == '.') {
            if (dent->d_name[1] == '.' && dent->d_name[2] == '\0')
                continue;
            if (dent->d_name[1] == '\0')
                continue;
        }

        fl = strlen(fpath);
        dl = strlen(dent->d_name);
        if ((fl + 1 + dl) > 1024) {
            fprintf(stderr,"tgrep: Path %s/%s is too long. "
                    "MaxPath = 1024\n",
                    fpath, dent->d_name);
            continue;  /* try the next name in this directory */
        }
        strcpy(fullpath,fpath);
        strcat(fullpath,"/");
        strcat(fullpath,dent->d_name);

      RESTART_STAT:
        if (stat(fullpath,&sbuf)) {
            if (errno == EINTR) {
                if (--restart_cnt)
                    goto RESTART_STAT;
            }
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't stat file/dir %s, %s. "
                        "Ignored.\n",
                        fullpath,strerror(errno));
            goto ERROR;
        }

        switch (sbuf.st_mode & S_IFMT) {
        case S_IFREG :
            if (flags & TG_FILEPAT) {
```

```
                    if (pmatch(pm_file_pat, dent->d_name, &pm_file_len)) {
                        DP(DLEVEL3,("file pat match (cascade) %s\n",
                                    dent->d_name));
                        add_work(fullpath,FILET);
                    }
                }
                else {
                    add_work(fullpath,FILET);
                    DP(DLEVEL3,("cascade added file (MATCH) %s to Work Q\n",
                                fullpath));
                }
                break;
            case S_IFDIR :
                DP(DLEVEL3,("cascade added dir %s to Work Q\n",fullpath));
                add_work(fullpath,DIRT);
                break;
            }
        }

      ERROR:
        closedir(dp);
      DONE:
        free(wt->path);
        free(wt);
        notrun();
        mutex_lock(&cascade_q_lk);
        if (cascade_pool_cnt > cascade_thr_limit) {
            mutex_unlock(&cascade_q_lk);
            DP(DLEVEL5,("Cascade thread exiting\n"));
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_cascade_destroy++;
                mutex_unlock(&stat_lk);
            }
            return(0); /* thr_exit */
        }
        else {
            DP(DLEVEL5,("Cascade thread waiting in pool\n"));
            cascade_pool_cnt++;
            while (!cascade_q)
                cond_wait(&cascade_q_cv,&cascade_q_lk);
            cascade_pool_cnt--;
            wt = cascade_q;  /* we have work to do! */
            if (cascade_q->next)
                cascade_q = cascade_q->next;
            else
                cascade_q = NULL;
            mutex_unlock(&cascade_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Print Local Output: Called by the search thread after it is done searchin
 * a single file. If any oputput was saved (matching lines), the lines are
 * displayed as a group on stdout.
 */
int
print_local_output(out_t *out, work_t *wt)
{
    out_t       *pp, *op;
    int         out_count = 0;
    int         printed = 0;
    int         print_name = 1;

    pp = out;
    mutex_lock(&output_print_lk);
    if (pp && (flags & TG_PROGRESS)) {
        progress++;
        if (progress >= progress_offset) {
            progress = 0;
```

```
                fprintf(stderr,".");
            }
        }
        while (pp) {
            out_count++;
            if (!(flags & FC_COUNT)) {
                if (flags & FL_NAMEONLY) {  /* Pint name ONLY ! */
                    if (!printed) {
                        printed = 1;
                        printf("%s\n",wt->path);
                    }
                }
                else {  /* We are printing more then just the name */
                    if (!(flags & FH_HOLDNAME))  /* do not print name ? */
                        printf("%s :",wt->path);
                    if (flags & FB_BLOCK)
                        printf("%ld:",pp->byte_count/512+1);
                    if (flags & FN_NUMBER)
                        printf("%d:",pp->line_count);
                    printf("%s\n",pp->line);
                }
            }
            op = pp;
            pp = pp->next;
            /* free the nodes as we go down the list */
            free(op->line);
            free(op);
        }
        mutex_unlock(&output_print_lk);
        mutex_lock(&global_count_lk);
        global_count += out_count;
        mutex_unlock(&global_count_lk);
        return(0);
    }

    /*
     * add output local: is called by a search thread as it finds matching lines
     * the matching line, it's byte offset, line count, etc are stored until the
     * search thread is done searching the file, then the lines are printed as
     * a group. This way the lines from more then a single file are not mixed
     * together.
     */
    int
    add_output_local(out_t **out, work_t *wt,int lc, long bc, char *line)
    {
        out_t       *ot,*oo, *op;

        if (( ot = (out_t *)malloc(sizeof(out_t))) == NULL)
            goto ERROR;
        if (( ot->line = (char *)malloc(strlen(line)+1)) == NULL)
            goto ERROR;

        strcpy(ot->line,line);
        ot->line_count = lc;
        ot->byte_count = bc;

        if (!*out) {
            *out = ot;
            ot->next = NULL;
            return(0);
        }
        /* append to the END of the list, keep things sorted! */
        op = oo = *out;
        while(oo) {
            op = oo;
            oo = oo->next;
        }
        op->next = ot;
        ot->next = NULL;
        return(0);
     ERROR:
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Output lost. No space. "
                    "[%s: line %d byte %d match : %s\n",
                    wt->path,lc,bc,line);
        return(1);
    }

    /*
     * print stats: If the -S flag is set, after ALL files have been searched,
     * main thread calls this function to print the stats it keeps on how the
     * search went.
     */
    void
    prnt_stats(void)
    {
        float a,b,c;
        float t = 0.0;
        time_t  st_end = 0;
        char    tl[80];

        st_end = time(NULL); /* stop the clock */
        fprintf(stderr,"\n----------------- Tgrep Stats. --------------------\n'
        fprintf(stderr,"Number of directories searched:           %d\n",
                st_dir_search);
        fprintf(stderr,"Number of files searched:                 %d\n",
                st_file_search);
        c = (float)(st_dir_search + st_file_search) / (float)(st_end - st_start)
        fprintf(stderr,"Dir/files per second:                     %3.2f\n",
                c);
        fprintf(stderr,"Number of lines searched:                 %d\n",
                st_line_search);
        fprintf(stderr,"Number of matching lines to target:       %d\n",
                global_count);

        fprintf(stderr,"Number of cascade threads created:        %d\n",
                st_cascade);
        fprintf(stderr,"Number of cascade threads from pool:      %d\n",
                st_cascade_pool);
        a = st_cascade_pool; b = st_dir_search;
        fprintf(stderr,"Cascade thread pool hit rate:             %3.2f%%\n",
                ((a/b)*100));
        fprintf(stderr,"Cascade pool overall size:                %d\n",
                cascade_pool_cnt);
        fprintf(stderr,"Cascade pool size limit:                  %d\n",
                cascade_thr_limit);
        fprintf(stderr,"Number of cascade threads destroyed:      %d\n",
                st_cascade_destroy);

        fprintf(stderr,"Number of search threads created:         %d\n",
                st_search);
        fprintf(stderr,"Number of search threads from pool:       %d\n",
                st_pool);
        a = st_pool; b = st_file_search;
        fprintf(stderr,"Search thread pool hit rate:              %3.2f%%\n",
                ((a/b)*100));
        fprintf(stderr,"Search pool overall size:                 %d\n",
                search_pool_cnt);
        fprintf(stderr,"Search pool size limit:                   %d\n",
                search_thr_limit);
        fprintf(stderr,"Number of search threads destroyed:       %d\n",
                st_destroy);

        fprintf(stderr,"Max # of threads running concurrenly:     %d\n",
                st_maxrun);
        fprintf(stderr,"Total run time, in seconds.               %d\n",
               (st_end - st_start));

        /* Why did we wait ? */
        a = st_workfds; b = st_dir_search+st_file_search;
        c = (a/b)*100; t += c;
        fprintf(stderr,"Work stopped due to no FD's:  (%.3d)       %d Times, %3.
               search_thr_limit,st_workfds,c);
        a = st_worknull; b = st_dir_search+st_file_search;
        c = (a/b)*100; t += c;
        fprintf(stderr,"Work stopped due to no work on Q:         %d Times, %3.2
               st_worknull,c);
    #ifndef __lock_lint  /* it is OK to read HERE with out the lock ! */
        if (tglimit == UNLIMITED)
            strcpy(tl,"Unlimited");
        else
            sprintf(tl,"   %.3d   ",tglimit);
    #endif
        a = st_worklimit; b = st_dir_search+st_file_search;
        c = (a/b)*100; t += c;
        fprintf(stderr,"Work stopped due to TGLIMIT:  (%.9s) %d Times, %3.2f%%\n
               tl,st_worklimit,c);
        fprintf(stderr,"Work continued to be handed out:          %3.2f%%\n",
                100.00-t);
        fprintf(stderr,"----------------------------------------------------\n")
    }

    /*
     * not running: A glue function to track if any search threads or cascade
     * threads are running. When the count is zero, and the work Q is NULL,
     * we can safly say, WE ARE DONE.
     */
    void
    notrun (void)
    {
        mutex_lock(&work_q_lk);
        work_cnt--;
        tglimit++;
        current_open_files++;
        mutex_lock(&running_lk);
        if (flags & FS_STATS) {
            mutex_lock(&stat_lk);
            if (running > st_maxrun) {
                st_maxrun = running;
                DP(DLEVEL6,("Max Running has increased to %d\n",st_maxrun));
            }
            mutex_unlock(&stat_lk);
        }
        running--;
        if (work_cnt == 0 && running == 0) {
            all_done = 1;
            DP(DLEVEL6,("Setting ALL_DONE flag to TRUE.\n"));
        }
        mutex_unlock(&running_lk);
        cond_signal(&work_q_cv);
        mutex_unlock(&work_q_lk);
    }

    /*
     * uncase: A glue function. If the -i (case insensitive) flag is set, the
     * target strng and the read in line is converted to lower case before
     * comparing them.
     */
    void
    uncase(char *s)
    {
        char        *p;

        for (p = s; *p != NULL; p++)
            *p = (char)tolower(*p);
    }


    /*
     * SigThread: if the -S option is set, the first ^C set to tgrep will
     * print the stats on the fly, the second will kill the process.
     */

    void *
    SigThread(void *arg)
    {
        int sig;
        int stats_printed = 0;

        while (1) {
            sig = sigwait(&set);
            DP(DLEVEL7,("Signal %d caught\n",sig));
            switch (sig) {
            case -1:
                fprintf(stderr,"Signal error\n");
                break;
            case SIGINT:
                if (stats_printed)
                    exit(1);
                stats_printed = 1;
                sig_print_stats();
                break;
            case SIGHUP:
                sig_print_stats();
                break;
            default:
                DP(DLEVEL7,("Default action taken (exit) for signal %d\n",sig));
                exit(1);  /* default action */
            }
        }
    }

    void
    sig_print_stats(void)
    {
        /*
        ** Get the output lock first
        ** Then get the stat lock.
        */
        mutex_lock(&output_print_lk);
        mutex_lock(&stat_lk);
        prnt_stats();
        mutex_unlock(&stat_lk);
        mutex_unlock(&output_print_lk);
        return;
    }

    /*
     * usage: Have to have one of these.
     */
    void
    usage(void)
    {
        fprintf(stderr,"usage: tgrep <options> pattern <{file,dir}>...\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"Where:\n");
    #ifdef DEBUG
        fprintf(stderr,"Debug     -d = debug level -d <levels> (-d0 for usage)\r
        fprintf(stderr,"Debug     -f = block fd's from use (-f #)\n");
    #endif
        fprintf(stderr,"          -b = show block count (512 byte block)\n");
        fprintf(stderr,"          -c = print only a line count\n");
        fprintf(stderr,"          -h = do not print file names\n");
        fprintf(stderr,"          -i = case insensitive\n");
        fprintf(stderr,"          -l = print file name only\n");
        fprintf(stderr,"          -n = print the line number with the line\n");
        fprintf(stderr,"          -s = Suppress error messages\n");
        fprintf(stderr,"          -v = print all but matching lines\n");
    #ifdef NOT_IMP
        fprintf(stderr,"          -w = search for a \"word\"\n");
    #endif
        fprintf(stderr,"          -r = Do not search for files in all "
                                    "sub-directories\n");
        fprintf(stderr,"          -C = show continued lines (\"\\\")\n");
        fprintf(stderr,"          -p = File name regexp pattern. (Quote it)\n");
        fprintf(stderr,"          -P = show progress. -P 1 prints a DOT on stder
                       "              for each file it finds, -P 10 prints a DO
                       "              on stderr for each 10 files it finds, etc
        fprintf(stderr,"          -e = expression search.(regexp) More then one\
        fprintf(stderr,"          -B = limit the number of threads to TGLIMIT\n'
        fprintf(stderr,"          -S = Print thread stats when done.\n");
        fprintf(stderr,"          -Z = Print help on the regexp used.\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"Notes:\n");
        fprintf(stderr,"      If you start tgrep with only a directory name\n");
        fprintf(stderr,"      and no file names, you must not have the -r optior
        fprintf(stderr,"      set or you will get no output.\n");
        fprintf(stderr,"      To search stdin (piped input), you must set -r\n")
        fprintf(stderr,"      Tgrep will search ALL files in ALL \n");
        fprintf(stderr,"      sub-directories. (like */* */*/* */*/*/* etc..)\n'
        fprintf(stderr,"      if you supply a directory name.\n");
        fprintf(stderr,"      If you do not supply a file, or directory name,\n'
        fprintf(stderr,"      and the -r option is not set, the current \n");
        fprintf(stderr,"      directory \".\" will be used.\n");
        fprintf(stderr,"      All the other options should work \"like\" grep\n'
        fprintf(stderr,"      The -p patten is regexp, tgrep will search only\n'
        fprintf(stderr,"      the file names that match the patten\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"      Tgrep Version %s\n",Tgrep_Version);
        fprintf(stderr,"\n");
        fprintf(stderr,"      Copy Right By Ron Winacott, 1993-1995.\n");
        fprintf(stderr,"\n");
        exit(0);
    }

    /*
     * regexp usage: Tell the world about tgrep custom (THREAD SAFE) regexp!
     */
    int
    regexp_usage (void)
    {
        fprintf(stderr,"usage: tgrep <options> -e \"pattern\" <-e ...> "
                "<{file,dir}>...\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"metachars:\n");
        fprintf(stderr,"    . - match any character\n");
        fprintf(stderr,"    * - match 0 or more occurrences of pervious char\n")
        fprintf(stderr,"    + - match 1 or more occurrences of pervious char.\n'
        fprintf(stderr,"    ^ - match at begining of string\n");
        fprintf(stderr,"    $ - match end of string\n");
        fprintf(stderr,"    [ - start of character class\n");
        fprintf(stderr,"    ] - end of character class\n");
        fprintf(stderr,"    ( - start of a new pattern\n");
        fprintf(stderr,"    ) - end of a new pattern\n");
        fprintf(stderr,"    @(n)c - match <c> at column <n>\n");
        fprintf(stderr,"    | - match either pattern\n");
        fprintf(stderr,"    \\ - escape any special characters\n");
        fprintf(stderr,"    \\c - escape any special characters\n");
        fprintf(stderr,"    \\o - turn on any special characters\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"To match two diffrerent patterns in the same command\n")
        fprintf(stderr,"Use the or function. \n"
                "ie: tgrep -e \"(pat1)|(pat2)\" file\n"
                "This will match any line with \"pat1\" or \"pat2\" in it.\n");
        fprintf(stderr,"You can also use up to %d -e expresions\n",MAXREGEXP);
        fprintf(stderr,"RegExp Pattern matching brought to you by Marc Staveley\
        exit(0);
    }

    /*
     * debug usage: If compiled with -DDEBUG, turn it on, and tell the world
     * how to get tgrep to print debug info on different threads.
     */
    #ifdef DEBUG
    void
    debug_usage(void)
    {
        int i = 0;

        fprintf(stderr,"DEBUG usage and levels:\n");
        fprintf(stderr,"----------------------------------------------------\n");
        fprintf(stderr,"Level                       code\n");
        fprintf(stderr,"----------------------------------------------------\n");
        fprintf(stderr,"0                 This message.\n");
        for (i=0; i<9; i++) {
            fprintf(stderr,"%d                 %s\n",i+1,debug_set[i].name);
        }
        fprintf(stderr,"----------------------------------------------------\n");
        fprintf(stderr,"You can or the levels together like -d134 for levels\n")
        fprintf(stderr,"1 and 3 and 4.\n");
        fprintf(stderr,"\n");
        exit(0);
    }
    #endif
```

Multithreaded Quicksort

The following example `tquick.c`implements the quicksort algorithm using threads.

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
*
*  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
*  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
*  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
*
*  This file is provided with no support and without any obligation on the
*  part of SunSoft, Inc. to assist in its use, correction, modification or
*  enhancement.
*
*  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
*  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
*  FILE OR ANY PART THEREOF.
*
*  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
*  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
*  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
*  DAMAGES.
*
*  SunSoft, Inc.
*  2550 Garcia Avenue
*  Mountain View, California  94043
*/

/*
 * multiple-thread quick-sort.  See man page for qsort(3c) for info.
 * Works fine on uniprocessor machines as well.
 *
 *  Written by:  Richard Pettit (Richard.Pettit@West.Sun.COM)
 */

#include <unistd.h>
#include <stdlib.h>
#include <thread.h>

/* don't create more threads for less than this */
#define SLICE_THRESH   4096

/* how many threads per lwp */
#define THR_PER_LWP       4
```

```
/* cast the void to a one byte quanitity and compute the offset */
#define SUB(a, n)      ((void *) (((unsigned char *) (a)) + ((n) * width)))

typedef struct {
  void    *sa_base;
  int      sa_nel;
  size_t   sa_width;
  int    (*sa_compar)(const void *, const void *);
} sort_args_t;

/* for all instances of quicksort */
static int threads_avail;

#define SWAP(a, i, j, width) \
{ \
  int n; \
  unsigned char uc; \
  unsigned short us; \
  unsigned long ul; \
  unsigned long long ull; \
 \
  if (SUB(a, i) == pivot) \
    pivot = SUB(a, j); \
  else if (SUB(a, j) == pivot) \
    pivot = SUB(a, i); \
 \
  /* one of the more convoluted swaps I've done */ \
  switch(width) { \
  case 1: \
    uc = *((unsigned char *) SUB(a, i)); \
    *((unsigned char *) SUB(a, i)) = *((unsigned char *) SUB(a, j)); \
    *((unsigned char *) SUB(a, j)) = uc; \
    break; \
  case 2: \
    us = *((unsigned short *) SUB(a, i)); \
    *((unsigned short *) SUB(a, i)) = *((unsigned short *) SUB(a, j)); \
    *((unsigned short *) SUB(a, j)) = us; \
    break; \
  case 4: \
    ul = *((unsigned long *) SUB(a, i)); \
    *((unsigned long *) SUB(a, i)) = *((unsigned long *) SUB(a, j)); \
    *((unsigned long *) SUB(a, j)) = ul; \
    break; \
  case 8: \
    ull = *((unsigned long long *) SUB(a, i)); \
    *((unsigned long long *) SUB(a,i)) = *((unsigned long long *) SUB(a,j));
    *((unsigned long long *) SUB(a, j)) = ull; \
    break; \
  default: \
    for(n=0; n<width; n++) { \
      uc = ((unsigned char *) SUB(a, i))[n]; \
      ((unsigned char *) SUB(a, i))[n] = ((unsigned char *) SUB(a, j))[n]; \
      ((unsigned char *) SUB(a, j))[n] = uc; \
    } \
    break; \
  } \
}

static void *
_quicksort(void *arg)
{
  sort_args_t *sargs = (sort_args_t *) arg;
  register void *a = sargs->sa_base;
  int n = sargs->sa_nel;
  int width = sargs->sa_width;
  int (*compar)(const void *, const void *) = sargs->sa_compar;
  register int i;
  register int j;
  int z;
  int thread_count = 0;
  void *t;
  void *b[3];
  void *pivot = 0;
  sort_args_t sort_args[2];
  thread_t tid;

  /* find the pivot point */
  switch(n) {
  case 0:
  case 1:
    return 0;
  case 2:
    if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
      SWAP(a, 0, 1, width);
    }
    return 0;
  case 3:
    /* three sort */
    if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
      SWAP(a, 0, 1, width);
    }
    /* the first two are now ordered, now order the second two */
    if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
      SWAP(a, 2, 1, width);
    }
    /* should the second be moved to the first? */
    if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
      SWAP(a, 1, 0, width);
    }
    return 0;
  default:
    if (n > 3) {
      b[0] = SUB(a, 0);
      b[1] = SUB(a, n / 2);
      b[2] = SUB(a, n - 1);
      /* three sort */
      if ((*compar)(b[0], b[1]) > 0) {
        t = b[0];
        b[0] = b[1];
        b[1] = t;
      }
      /* the first two are now ordered, now order the second two */
      if ((*compar)(b[2], b[1]) < 0) {
        t = b[1];
        b[1] = b[2];
        b[2] = t;
      }
      /* should the second be moved to the first? */
      if ((*compar)(b[1], b[0]) < 0) {
        t = b[0];
        b[0] = b[1];
        b[1] = t;
      }
      if ((*compar)(b[0], b[2]) != 0)
        if ((*compar)(b[0], b[1]) < 0)
          pivot = b[1];
        else
          pivot = b[2];
    }
    break;
  }
  if (pivot == 0)
    for(i=1; i<n; i++) {
      if (z = (*compar)(SUB(a, 0), SUB(a, i))) {
        pivot = (z > 0) ? SUB(a, 0) : SUB(a, i);
        break;
      }
    }
  if (pivot == 0)
    return;

  /* sort */
  i = 0;
  j = n - 1;
```

```
while(i <= j) {
  while((*compar)(SUB(a, i), pivot) < 0)
    ++i;
  while((*compar)(SUB(a, j), pivot) >= 0)
    --j;
  if (i < j) {
    SWAP(a, i, j, width);
    ++i;
    --j;
  }
}

/* sort the sides judiciously */
switch(i) {
case 0:
case 1:
  break;
case 2:
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  break;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
    SWAP(a, 2, 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
    SWAP(a, 1, 0, width);
  }
  break;
default:
  sort_args[0].sa_base          = a;
  sort_args[0].sa_nel           = i;
  sort_args[0].sa_width         = width;
  sort_args[0].sa_compar        = compar;
  if ((threads_avail > 0) && (i > SLICE_THRESH)) {
    threads_avail--;
    thr_create(0, 0, _quicksort, &sort_args[0], 0, &tid);
    thread_count = 1;
  } else
    _quicksort(&sort_args[0]);
  break;
}
j = n - i;
switch(j) {
case 1:
  break;
case 2:
  if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
    SWAP(a, i, i + 1, width);
  }
  break;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
    SWAP(a, i, i + 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, i + 2), SUB(a, i + 1)) < 0) {
    SWAP(a, i + 2, i + 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, i + 1), SUB(a, i)) < 0) {
    SWAP(a, i + 1, i, width);
  }
  break;
  default:
    sort_args[1].sa_base          = SUB(a, i);
    sort_args[1].sa_nel           = j;
    sort_args[1].sa_width         = width;
    sort_args[1].sa_compar        = compar;
    if ((thread_count == 0) && (threads_avail > 0) && (i > SLICE_THRESH)) {
      threads_avail--;
      thr_create(0, 0, _quicksort, &sort_args[1], 0, &tid);
      thread_count = 1;
    } else
      _quicksort(&sort_args[1]);
    break;
  }
  if (thread_count) {
    thr_join(tid, 0, 0);
    threads_avail++;
  }
  return 0;
}

void
quicksort(void *a, size_t n, size_t width,
          int (*compar)(const void *, const void *))
{
  static int ncpus = -1;
  sort_args_t sort_args;

  if (ncpus == -1) {
    ncpus = sysconf( _SC_NPROCESSORS_ONLN);

    /* lwp for each cpu */
    if ((ncpus > 1) && (thr_getconcurrency() < ncpus))
      thr_setconcurrency(ncpus);

    /* thread count not to exceed THR_PER_LWP per lwp */
    threads_avail = (ncpus == 1) ? 0 : (ncpus * THR_PER_LWP);
  }
  sort_args.sa_base = a;
  sort_args.sa_nel = n;
  sort_args.sa_width = width;
  sort_args.sa_compar = compar;
  (void) _quicksort(&sort_args);
}
```

Dave Marshall
1/5/1999