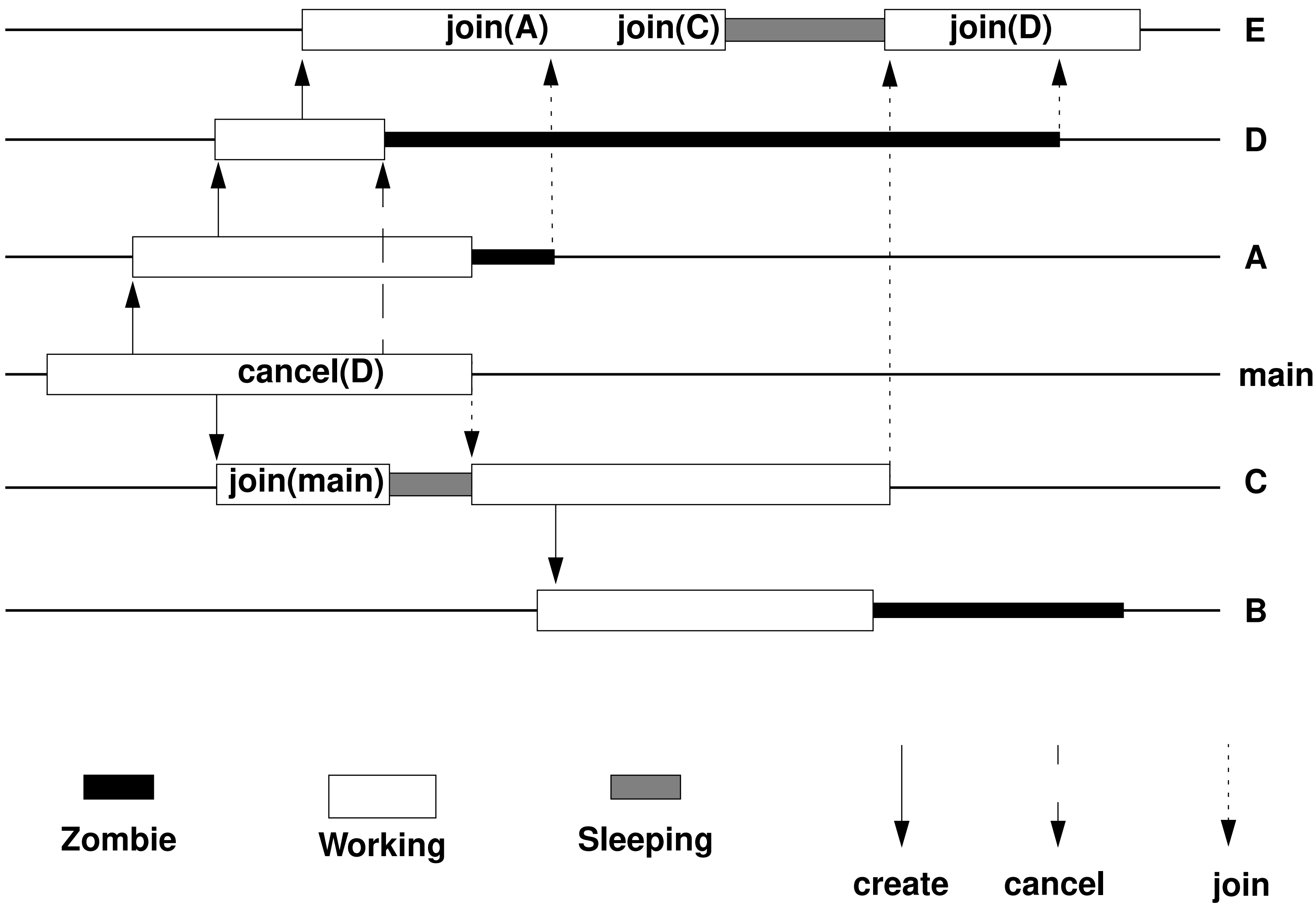
POSIX
Example Code

Table of Contents

POSIX Thread Create and Join



Creating & Joining Threads: multi_thr.c

```
/*                  multi_thr.c                */

/*
  Simple program that just illustrates thread creation, thread exiting,
  waiting for threads, and returning status from threads.
*/

/*
cc -o multi_thr multi_thr.c -L. -R. -g -lpthread -lthread -pthread_extensions -
lposix4
*/

#define _POSIX_C_SOURCE 199506L

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include "thread_extensions.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 *);

pthread_t               tid_a, tid_b, tid_c, tid_d, tid_e, tid_main;
pthread_attr_t          attr;
int                     zero;
float                   f;


int time1()
{return(time(NULL)-zero);}


void *sub_a(void *arg)
{int            err, i;
```

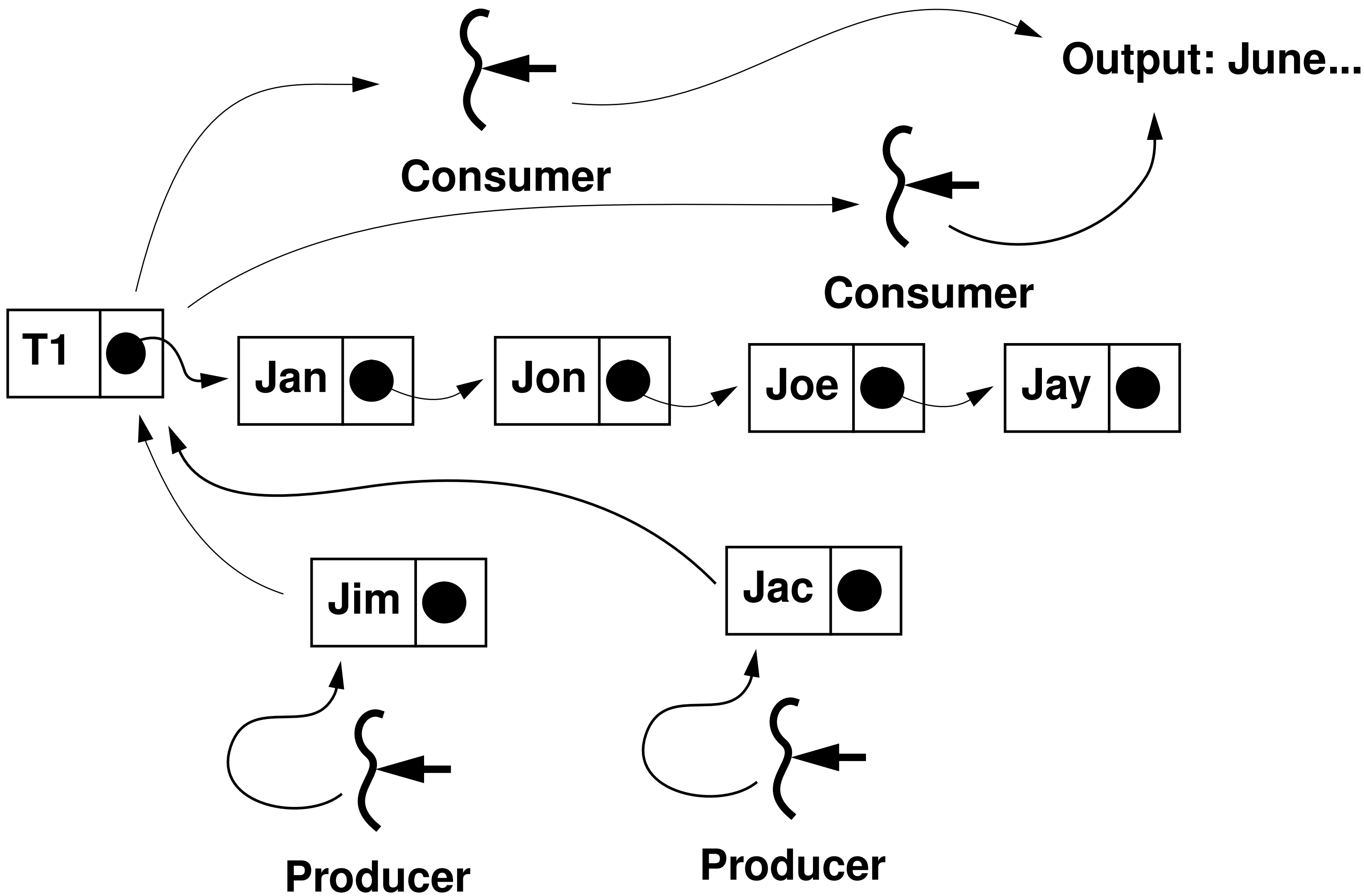
```
 pthread_t      tid = pthread_self();

 printf("[%2d] A: \t In thread A [%s]\n", time1(), thread_name(tid));
 sleep(1);
 PTHREAD_CREATE(&tid_d, &attr, sub_d, NULL);
 printf("[%2d] A: \t Created thread D [%s]\n", time1(), thread_name(tid_d));

 sleep(3);
 printf("[%2d] A: \t Thread exiting...\n", time1());
 return((void *)77);            /* Same as pthread_exit((void *)77) */
}


void *sub_b(void *arg)
{pthread_t      tid = pthread_self();
 printf("[%2d] B: \t In thread B [%s]\n", time1(), thread_name(tid));

 sleep(4);

 printf("[%2d] B: \t Thread exiting...\n", time1());
 pthread_exit(NULL);
}


void *sub_c(void *arg)
{void           *status;
 int            err, i;

 pthread_t tid = pthread_self();
 pthread_t      *ptid = (pthread_t *) arg; /* Passing a POINTER to the TID */
 char           *name = thread_name(tid);

 printf("[%2d] C: \t In thread C [%s]\n", time1(), name);
 sleep(2);

 printf("[%2d] C: \t Joining main thread...\n", time1());
 if (err=pthread_join(*ptid, &status))
   printf("pthread_join Error. %s", strerror(err)), exit(1);
 printf("[%2d] C: \t Main thread [%s] returning status: %d\n",
        time1(), thread_name(*ptid), (int) status);

 sleep(1);

 PTHREAD_CREATE(&tid_b, &attr, sub_b, NULL);
 printf("[%2d] C: \t Created thread B [%s]\n", time1(), thread_name(tid_b));

 sleep(4);
 printf("[%2d] C: \t Thread exiting...\n", time1());
 f = 88.8;
```

```
 pthread_exit((void *) &f);
}

void *cleanup(void *arg)
{pthread_t tid = pthread_self();
 char *name = thread_name(tid);

  printf("[%2d] D: \t %s cancelled! \n", time1(), name);
}



void * sub_d(void *arg)
{int            err, i;
 pthread_t      tid_e;
 void           *status;
 pthread_t      tid = pthread_self();

 printf("[%2d] D: \t In thread D [%s]\n", time1(), thread_name(tid));

 pthread_cleanup_push(cleanup, NULL);
 pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
 pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

 sleep(1);
 PTHREAD_CREATE(&tid_e, &attr, sub_e, NULL);
 printf("[%2d] D: \t Created thread E [%s]\n", time1(), thread_name(tid_e));

 sleep(5);

        /* D should get cancelled before this runs. */
 printf("[%2d] D: \t Thread exiting...\n", time1());
 pthread_cleanup_pop(0);
 return((void *)55);
}


void * sub_e(void *arg)
{int            err, i;
 void           *status;
 pthread_t      tid = pthread_self();

 printf("[%2d] E: \t In thread E [%s]\n", time1(), thread_name(tid));

 sleep(3);
 printf("[%2d] E: \t Joining thread A...\n", time1());
 if (err=pthread_join(tid_a, &status))
   printf("pthread_join Error. %s", strerror(err)), exit(1);
 printf("[%2d] E: \t Thread A [%s] returning status: %d\n",
```

```
        time1(), thread_name(tid_a), (int) status);

 sleep(2);

 printf("[%2d] E: \t Joining thread C...\n", time1());
 if (err=pthread_join(tid_c, &status))
   printf("pthread_join Error. %s", strerror(err)), exit(1);
 printf("[%2d] E: \t Thread C [%s] returning status: %f\n",
        time1(), thread_name(tid_c), *((float *) status));

 sleep(2);
 printf("[%2d] E: \t Joining thread D...\n", time1());
 if (err=pthread_join(tid_d, &status))
   printf("pthread_join Error. %s", strerror(err)), exit(1);
 if ((void*) status == (void*) PTHREAD_CANCELED)
   printf("[%2d] E: \t Thread D [%s] returning status: PTHREAD_CANCELED\n",
          time1(), thread_name(tid_d));
 else
   printf("[%2d] E: \t Thread D [%s] returning status: %d\n",
          time1(), thread_name(tid_d), (int) status);

 sleep(1);
 printf("[%2d] E: \t Thread exiting...\n", time1());
 pthread_exit((void *)44);
}



main()
{int            err;

 zero = time(NULL);
 tid_main = pthread_self();
 printf("Time Thread \t Event\n");
 printf("==== ====== \t =====\n");
 printf("[%2d] Main: \t Started [%s]\n", time1(), thread_name(tid_main));

 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); /* Also default */

 sleep(1);
 PTHREAD_CREATE(&tid_a, &attr, sub_a, NULL);
 printf("[%2d] Main: \t Created thread A [%s]\n", time1(), thread_name(tid_a));

 sleep(1);
 PTHREAD_CREATE(&tid_c, &attr, sub_c, (void *) &tid_main);
 printf("[%2d] Main: \t Created thread C [%s]\n", time1(), thread_name(tid_c));
```

```
 sleep(2);
 printf("[%2d] Main: \t Cancelling thread D [%s]\n",
        time1(), thread_name(tid_d));
 pthread_cancel(tid_d);

 sleep(1);
 printf("[%2d] Main: \t Thread exiting...\n", time1());
 pthread_exit((void *) NULL);
}
```

Classic Producer/Consumer



Classic Producer/Consumer: one_queue_problem.c

```
/*                  one_queue_problem.c                */

/*
  This is the prototypical producer/consumer example done by
  building a singly linked list.  Note the use of the semaphore
  to prevent the consumers from running when the list is empty.
  Unfortunately, there's nothing to prevent the list from getting
  infinitely long...

  Delays are inserted to simulate disk I/O (and to make things
  run slower!).
*/


/*
cc -o one_queue_problem one_queue_problem.c -L. -R. -g -lpthread -lthread -
pthread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include "thread_extensions.h"


typedef struct request_struct
{int                    client;
 char                   *buf;
 pthread_t              tid;
 struct request_struct  *next;
} request_t;

request_t               *requests = NULL;
pthread_mutex_t         requests_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_attr_t          attr;
sem_t                   requests_length;
int                        max_length_empirical = 0, sval;
```

```
int                         GET_DELAY = 100, PROCESS_DELAY = 250;
int                         N_PROD = 2, N_CONS = 2;



void print_requests()
{request_t *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->client);
    r = r->next;
   }
}


request_t *get_request()
{static int count = 0;
 static pthread_mutex_t mr  = PTHREAD_MUTEX_INITIALIZER;
 int c;
 request_t *request;
 pthread_t tid = pthread_self();

 delay(GET_DELAY, 0);
 pthread_mutex_lock(&mr);
 c=count++;
 pthread_mutex_unlock(&mr);

 request = (request_t *) malloc(sizeof(request_t));
 request->client = c;                         /* Cannot use count here! */
 request->tid = pthread_self();

 DEBUG(printf("[%s] Creating request  %4d\n", thread_name(tid), c));
 return(request);
}


void process_request(request_t *request)
{pthread_t tid = pthread_self();
  delay(PROCESS_DELAY, 0);
  DEBUG(printf("[%s] Processed request %4d [%s]\n", thread_name(tid),
               request->client, thread_name(request->tid)));
  free(request);
}


request_t *remove_request()
{request_t *request;

 pthread_mutex_lock(&requests_lock);
```

```
 request = requests;
 requests = requests->next;
 pthread_mutex_unlock(&requests_lock);
 return(request);
}


void add_request(request_t *request)
{
 pthread_mutex_lock(&requests_lock);
 request->next = requests;                      /* Insert at head of list */
 requests = request;
 sem_getvalue(&requests_length, &sval);
 if (sval+1 > max_length_empirical)
   max_length_empirical = sval+1;
 pthread_mutex_unlock(&requests_lock);
}


void *producer(void *arg)
{request_t      *request;

 while(TRUE)
   {
     request = get_request();
     add_request(request);
     sem_post(&requests_length);
   }
}


void *consumer(void *arg)
{request_t      *request;

 while(TRUE)
   {
     SEM_WAIT(&requests_length);
     request = remove_request();
     process_request(request);
   }
}


main(int argc, char **argv)
{int i;
pthread_t tid;

 SEM_INIT(&requests_length, NULL,  0);
```

```
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

 if (argc >= 2) GET_DELAY = atoi(argv[1]);
 if (argc >= 3) PROCESS_DELAY = atoi(argv[2]);
 if (argc >= 4) N_PROD        = atoi(argv[3]);
 if (argc >= 5) N_CONS        = atoi(argv[4]);
 printf("GET_DELAY (ms) = %d PROCESS_DELAY (ms) = %d N_PROD = %d N_CONS = %d\n",
        GET_DELAY, PROCESS_DELAY, N_PROD, N_CONS);

 for (i=0; i<N_CONS; i++)
   PTHREAD_CREATE(&tid, &attr, consumer, NULL);
 for (i=0; i<N_PROD; i++)
   PTHREAD_CREATE(&tid, &attr, producer, NULL);

 sleep(20);
 sem_getvalue(&requests_length, &sval);
 printf("Done.\nCurrent List length: %d Max length was: %d\n",
        sval, max_length_empirical);
 exit(0);
}
```

Exercise Objective

Edit the code in `one_queue_problem.c` and fix it so that the length of the list never exceeds some number (say 10).

Hint: Using `sem_getvalue()` will not help!

Don't peek at `one_queue_solution.c` until you have figured it out on your own! (The objective is not to get the answer, the objective is for you to think about the issue, so that when you encounter something similar in real code, you'll be able to deal with it!)

There is really only one simple solution to the problem.

Wild Guess at Time to Completion:

30 minutes think/design time.
2 minutes programming/debugging
My solution changed 4 lines of code.

Classic Producer/Consumer: one_queue_*solution*.c

```
/*                 one_queue_solution.c                 */

/*
  This is the prototypical producer/consumer example done by
  building a singly linked list.  Note the use of the semaphore
  to prevent the consumers from running when the list is empty.
  The second semaphore prevents the list from getting
  infinitely long.

  Delays are inserted to simulate disk I/O (and to make things
  run slower!).
*/

/*
cc -o one_queue_solution one_queue_solution.c -L. -R. -g -lpthread -lthread -
pthread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include "thread_extensions.h"



struct request_struct
{int                    client;
 char                   *buf;
 pthread_t              tid;
 struct request_struct  *next;
};
typedef struct request_struct request_t;

request_t               *requests = NULL;
pthread_mutex_t         requests_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_attr_t          attr;
sem_t                   requests_slots, requests_length;
int                                         max_length_empirical = 0, sval;
```

```
int                        GET_DELAY = 200, PROCESS_DELAY = 250;
int                        N_PROD = 2, N_CONS = 2;



void print_requests()
{request_t *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->client);
    r = r->next;
  }
}


request_t *get_request()
{static int count = 0;
 static pthread_mutex_t mr  = PTHREAD_MUTEX_INITIALIZER;
 int c;
 request_t *request;
 pthread_t tid = pthread_self();

 delay(GET_DELAY, 0);
 pthread_mutex_lock(&mr);
 c=count++;
 pthread_mutex_unlock(&mr);

 request = (request_t *) malloc(sizeof(request_t));
 request->client = c;                         /* Cannot use count here! */
 request->tid = pthread_self();

 DEBUG(printf("[%s] Creating request  %4d\n", thread_name(tid), c));
 return(request);
}


void process_request(request_t *request)
{pthread_t tid = pthread_self();

  delay(PROCESS_DELAY, 0);
  DEBUG(printf("[%s] Processed request %4d [%s]\n", thread_name(tid),
               request->client, thread_name(request->tid)));
  free(request);
}


request_t *remove_request()
{request_t *request;
```

```
 pthread_mutex_lock(&requests_lock);
 request = requests;
 requests = requests->next;
 pthread_mutex_unlock(&requests_lock);
 return(request);
}


void add_request(request_t *request)
{
 pthread_mutex_lock(&requests_lock);
 request->next = requests;                     /* Insert at head of list */
 requests = request;
 sem_getvalue(&requests_length, &sval);
 if (sval+1 > max_length_empirical)
   max_length_empirical = sval+1;
 pthread_mutex_unlock(&requests_lock);
}



void *producer(void *arg)
{request_t      *request;

 while(TRUE)
   {
     SEM_WAIT(&requests_slots);
     request = get_request();
     add_request(request);
     sem_post(&requests_length);
   }
}


void *consumer(void *arg)
{request_t      *request;

 while(TRUE)
   {
     SEM_WAIT(&requests_length);
     request = remove_request();
     process_request(request);
     sem_post(&requests_slots);
   }
}


main(int argc, char **argv)
{int i;
```

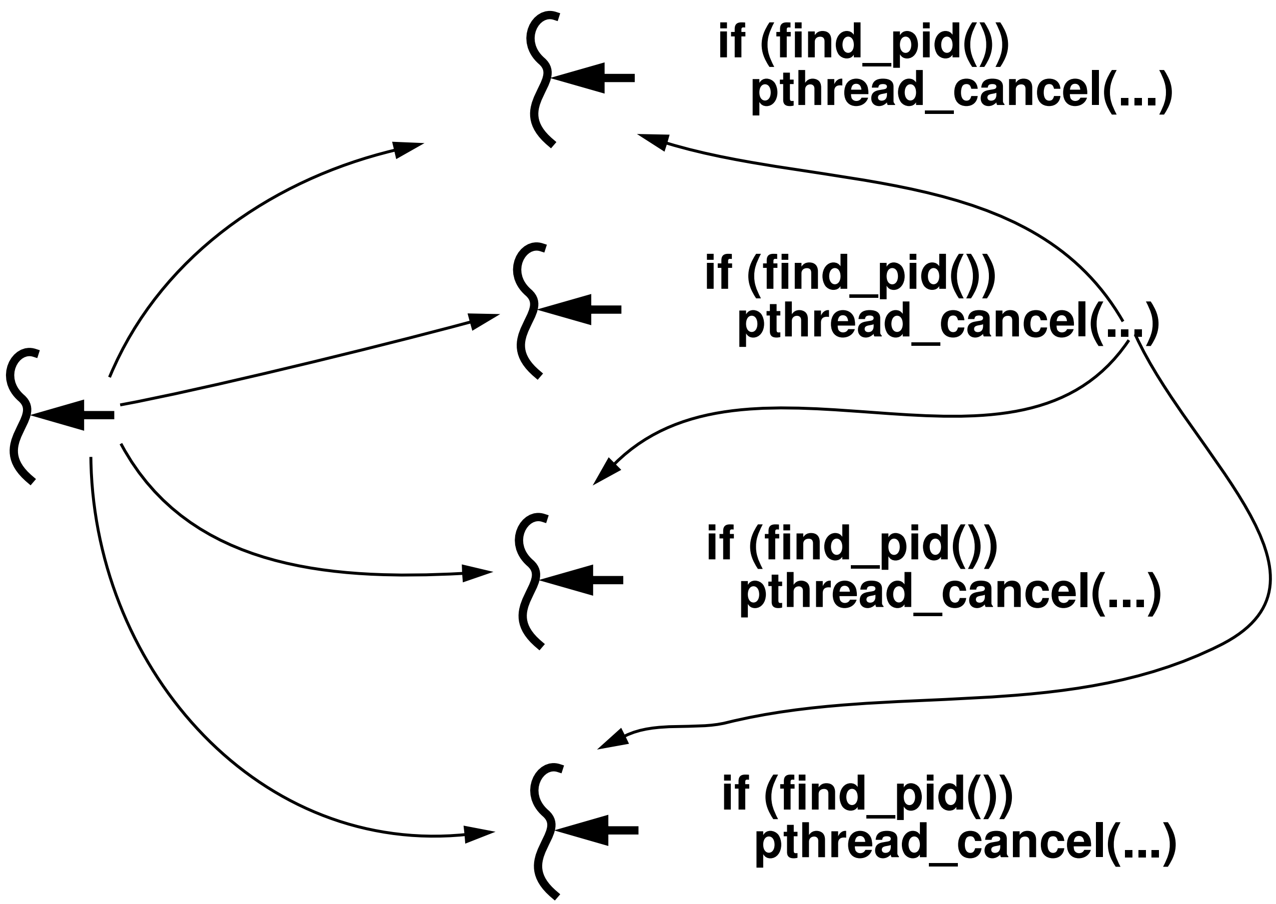
```
pthread_t tid;

SEM_INIT(&requests_length, NULL,  0);
SEM_INIT(&requests_slots, NULL, 10);

PTHREAD_ATTR_INIT(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

if (argc >= 2) GET_DELAY = atoi(argv[1]);
if (argc >= 3) PROCESS_DELAY = atoi(argv[2]);
if (argc >= 4) N_PROD        = atoi(argv[3]);
if (argc >= 5) N_CONS        = atoi(argv[4]);
printf("GET_DELAY (ms) = %d PROCESS_DELAY (ms) = %d N_PROD = %d N_CONS = %d\n",
       GET_DELAY, PROCESS_DELAY, N_PROD, N_CONS);

for (i=0; i<N_CONS; i++)
  PTHREAD_CREATE(&tid, &attr, consumer, NULL);
for (i=0; i<N_PROD; i++)
  PTHREAD_CREATE(&tid, &attr, producer, NULL);

sleep(20);
sem_getvalue(&requests_length, &sval);
printf("Done.\nCurrent List length: %d Max length was: %d\n",
                        sval, max_length_empirical);
exit(0);
}
```

Using a Single Length Counter

```
producer()
{while(TRUE)
 {request = get_request();
  pthread_mutex_lock(&m);
  while (length>10)
    pthread_cond_wait(&producer_cv, &m);
  add(request);
  length++;
  pthread_mutex_unlock(&m);
  pthread_cond_signal(&consumer_cv);
}}
```

```
consumer()
{while(TRUE)
 {pthread_mutex_lock(&m);
  while (length==0)
    pthread_cond_wait(&consumer_cv, &m);
  request = remove();
  length--;
  pthread_mutex_unlock(&m);
  pthread_cond_signal(&producer_cv);
  process(request);
}}
```

Cancellation



Cancellation: cancellation.c

```
/*                cancellation.c                */

/*
   This program does a heuristic search for an integer, then cancels
   all threads that didn't find it.  The actual heuristic is silly
   (it calls rand_r()), but the technique is valid.

   All of the calls to delay() are there to slow things down and
   make different contention situations more likely.

   A couple of simple cleanup handlers are included.  In a real
   program, these would be even more complex.

   NB: sem_trywait() EBUSY in 2.5 is a bug.
   It *should* be EAGAIN (fixed in 2.6).
*/

/*
cc -o cancellation cancellation.c -L. -R. -g -lpthread -lthread -
pthread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include "thread_extensions.h"

#ifdef __sun2_5                /* This is a bug in Solaris 2.5 */
#define MY_EAGAIN EBUSY
#else
#define MY_EAGAIN EAGAIN       /* Correct errno value from trywait() */
#endif

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

pthread_attr_t  attr;
```

```
pthread_t       threads[NUM_THREADS];
pthread_mutex_t threads_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t wait_lock    = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rand_lock    = PTHREAD_MUTEX_INITIALIZER;
sem_t           death_lock;     /* I'm using it like a lock */
pthread_mutex_t count_lock   = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  wait_cv      = PTHREAD_COND_INITIALIZER;
sem_t           cancel_barrier;
int answer;             /* Protected by death_lock */


void count_tries(int i)         /* Note the encapsulation */
{static int count=0, old_count=0, max_count = 0;
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("\n\nTotal attempt count: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
}


void cleanup_count(void *arg)
{int *ip = (int *) arg;
 int i = *ip;
 pthread_t tid = pthread_self();
 char *name = thread_name(tid);

 count_tries(i);
 printf("%s exited (maybe cancelled) on its %d try.\n", name, i);
 /* Note that you can't tell if the thread exited, or was cancelled. */
}


void cleanup_lock(void *arg)
{pthread_t tid = pthread_self();
 char *name = thread_name(tid);

  printf("Freeing & releasing: %s\n", name);
  free(arg);
  pthread_mutex_unlock(&rand_lock);
        /* This would be hard to do in a real program. */
}


void *search(void *arg)
{char *p;
```

```
unsigned int seed;
int i=0, j, err, guess, target = (int) arg;
pthread_t tid = pthread_self();
char *name = thread_name(tid);

seed = (unsigned int) tid;

pthread_cleanup_push(cleanup_count, (void *) &i); /* Q: Why &i ? */
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
sem_post(&cancel_barrier);     /* Don't start cancelling before here!   8 */

while (1)
  {i++;

       /* Extra stuff to make it more realistic and complex. */
   pthread_mutex_lock(&rand_lock);
   p = (char *) malloc(10);           /* Must free this up! */
       /* Q:  What if you allow cancellation here? */
   pthread_cleanup_push(cleanup_lock, (void *) p);     /*  4 */
   guess = rand_r(&seed);
   delay(0, 10);
   pthread_testcancel();                               /*  5 */
   pthread_cleanup_pop(0);
   /* Q:  What if you allow cancellation here? */      /*  6 */
   free(p);
   pthread_mutex_unlock(&rand_lock);

   delay(0, 10);

   if (target == guess)
     {printf("%s found the number on try %d!\n", name, i); /*  7 */
      /* I could also simply do a sem_wait() and let cancellation work. */
      while (((err = sem_trywait(&death_lock)) == -1) && (errno == EINTR)) ;
      if ((err == -1) && (errno == MY_EAGAIN))
        /*  1 */
        {printf("%s Exiting...\n", name);
          pthread_exit(NULL);                      /*  3 */
        }
      answer = guess;
      delay(5000, 0);  /* Encourage a few more threads to find it. */

      /* Make sure all threads have gotten to #8 before cancelling */
      thread_single_barrier(&cancel_barrier, NUM_THREADS);
      pthread_mutex_lock(&threads_lock);
      for (j=0;j<NUM_THREADS;j++)
        if (!pthread_equal(threads[j], tid))
          if (pthread_cancel(threads[j]) == ESRCH)     /*  2 */
            printf("Missed thread %s\n", thread_name(threads[j]));
```

```
        pthread_mutex_unlock(&threads_lock);
        break;                   /* Cannot release death_lock yet! */
      }
    pthread_testcancel();        /* Insert a known cancellation point */
  }
 pthread_cleanup_pop(1);
 pthread_exit(NULL);
}


start_searches()
{int i, pid, n_cancelled=0, status;
 pthread_t tid;

 pid = getpid();

 while (pid > RAND_MAX)
     pid /= 2;

 printf("\n\nSearching for the number = %d...\n", pid);

 pthread_mutex_lock(&threads_lock);   /* Q: Why do we need threads_lock ? */
 for (i=0;i<NUM_THREADS;i++)
   PTHREAD_CREATE(&threads[i], &attr, search, (void *)pid);
 pthread_mutex_unlock(&threads_lock);

 for (i=0;i<NUM_THREADS;i++)
   {pthread_mutex_lock(&threads_lock);
    tid = threads[i];                       /* Actually a constant now */
    pthread_mutex_unlock(&threads_lock);    /* Q: Why like this? */
    pthread_join(tid, (void **) &status);
    if ((void *)status == (void *)PTHREAD_CANCELED) n_cancelled++;
  }
 sem_post(&death_lock);          /* Cannot release any earlier! */
 count_tries(-1);
 printf("%d of the threads were cancelled.\n", n_cancelled);
 printf("The answer was: %d\n", answer);
}


main()
{int i;

 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);/* Also default */
 SEM_INIT(&death_lock, NULL, 1);
 SEM_INIT(&cancel_barrier, NULL, 0);
```

```
 for (i=0; i<2; i++)
   start_searches();

 printf("Done\n");
 exit(0);
}
```

Cancellation: cancellation_not.c

```
/*                cancellation_not.c                */

/*
   This program does a heuristic search for an integer, then cancels
   all threads that didn't find it.  The actual heuristic is silly
   (it calls rand_r()), but the technique is valid.

   All of the calls to delay() are there to slow things down and
   make different contention situations more likely.

   A couple of simple cleanup handlers are included.  In a real
   program, these would be even more complex.

   NB: sem_trywait() EBUSY in 2.5 is a bug.
   It *should* be EAGAIN (fixed in 2.6).
*/

/*
cc -o cancellation cancellation.c -L. -R. -g -lpthread -lthread -
pthread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include “thread_extensions.h”

#ifdef __sun2_5                 /* This is a bug in Solaris 2.5 */
#define MY_EBUSY EBUSY
#else
#define MY_EBUSY EAGAIN         /* Correct errno value from trywait() */
#endif

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

pthread_attr_t  attr;
pthread_t       threads[NUM_THREADS];
```

```
pthread_mutex_t threads_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t wait_lock    = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rand_lock    = PTHREAD_MUTEX_INITIALIZER;
sem_t           death_lock;     /* I'm using it like a lock */
pthread_mutex_t count_lock   = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  wait_cv      = PTHREAD_COND_INITIALIZER;

pthread_mutex_t cancel_lock   = PTHREAD_MUTEX_INITIALIZER;
int             time_to_die;

int answer;             /* Protected by death_lock */


void my_testcancel()
{int die;

 if (time_to_die)       /* Quick test w/o mutex.  Might be TRUE in error. */
   {pthread_mutex_lock(&cancel_lock);
    die = time_to_die;
    pthread_mutex_unlock(&cancel_lock);
    /* NB: This will run the cleanup handlers. */
    if (die) pthread_exit((void *)PTHREAD_CANCELED);
  }
}

void count_tries(int i)         /* Note the encapsulation */
{static int count=0, old_count=0, max_count = 0;
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("\n\nTotal attempt count: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
}


void cleanup_count(void *arg)
{int *ip = (int *) arg;
 int i = *ip;
 pthread_t tid = pthread_self();
 char *name = thread_name(tid);

 count_tries(i);
 printf("%s exited (maybe cancelled) on its %d try.\n", name, i);
 /* Note that you can't tell if the thread exited, or was cancelled. */
}
```

```
void cleanup_lock(void *arg)
{pthread_t tid = pthread_self();
 char *name = thread_name(tid);

  printf("Freeing & releasing: %s\n", name);
  free(arg);
  pthread_mutex_unlock(&rand_lock);
        /* This would be hard to do in a real program. */
}


void *search(void *arg)
{char *p;
 unsigned int seed;
 int i=0, j, err, guess, target = (int) arg;
 pthread_t tid = pthread_self();
 char *name = thread_name(tid);

 seed = (unsigned int) tid;

 pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
 pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
 pthread_cleanup_push(cleanup_count, (void *) &i); /* Q: Why &i ? */

 while (1)
   {i++;

        /* Extra stuff to make it more realistic and complex. */
    pthread_mutex_lock(&rand_lock);
    p = (char *) malloc(10);            /* Must free this up! */
        /* Q:  What if you allow cancellation here? */
    pthread_cleanup_push(cleanup_lock, (void *) p);     /*   4 */
    guess = rand_r(&seed);
    delay(0, 10);
    my_testcancel();                            /*   5 */
    pthread_cleanup_pop(0);
    /* Q:  What if you allow cancellation here? */      /*   6 */
    free(p);
    pthread_mutex_unlock(&rand_lock);

    delay(0, 10);

    if (target == guess)
      {printf("%s found the number on try %d!\n", name, i); /*   7 */
       /* I could also simply do a sem_wait() and let cancellation work. */
       while (((err = sem_trywait(&death_lock)) == -1) && (errno == EINTR)) ;
       if ((err == -1) && (errno == MY_EBUSY))   /*  1 */
         {printf("%s Exiting...\n", name);
```

```
          pthread_exit(NULL);                    /*   3 */
        }
      delay(5000, 0);  /* Encourage a few more threads to find it. */
      answer = guess;
      pthread_mutex_lock(&cancel_lock);
      time_to_die = TRUE;
      pthread_mutex_unlock(&cancel_lock);
      break;                   /* Cannot release death_lock yet! */
    }
   my_testcancel();    /* Insert a known cancellation point */
  }
 pthread_cleanup_pop(1);
 pthread_exit(NULL);
}


start_searches()
{int i, pid, n_cancelled=0, status;
 pthread_t tid;

 pid = getpid();

 while (pid > RAND_MAX)
     pid /= 2;
 pthread_mutex_lock(&cancel_lock);
 time_to_die = FALSE;
 pthread_mutex_unlock(&cancel_lock);


 printf("\n\nSearching for the number = %d...\n", pid);

 pthread_mutex_lock(&threads_lock);   /* Q: Why do we need threads_lock ? */
 for (i=0;i<NUM_THREADS;i++)
   PTHREAD_CREATE(&threads[i], &attr, search, (void *)pid);
 pthread_mutex_unlock(&threads_lock);

 for (i=0;i<NUM_THREADS;i++)
   {pthread_mutex_lock(&threads_lock);
    tid = threads[i];                           /* Actually a constant now */
    pthread_mutex_unlock(&threads_lock);   /* Q: Why like this? */
    pthread_join(tid, (void **) &status);
    if ((void *)status == (void *)PTHREAD_CANCELED) n_cancelled++;
  }
 sem_post(&death_lock);          /* Cannot release any earlier! */
 count_tries(-1);
 printf("%d of the threads were cancelled.\n", n_cancelled);
 printf("The answer was: %d\n", answer);
}
```

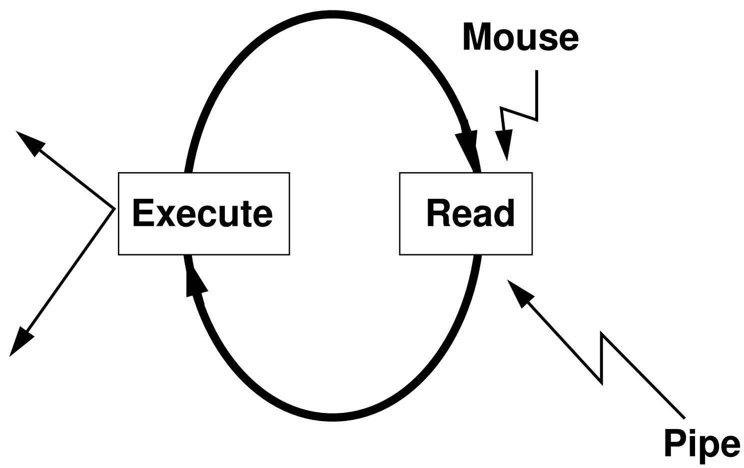
```
main()
{int i;

 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);/* Also default */
 SEM_INIT(&death_lock, NULL, 1);

 for (i=0; i<2; i++)
   start_searches();

 printf("Done\n");
 exit(0);
}
```

Window Toolkits

```
% ThreadWin [MT]
```

```
DoWork()
{sleep(6);
write(Pipe);
}
```

```
ButtonPushed()
{.. invert button ...
 pthread_create(DoWork);}
```




```
WorkComplete()
{... revert button ...}
```

Threads and X11: ThreadWin.c

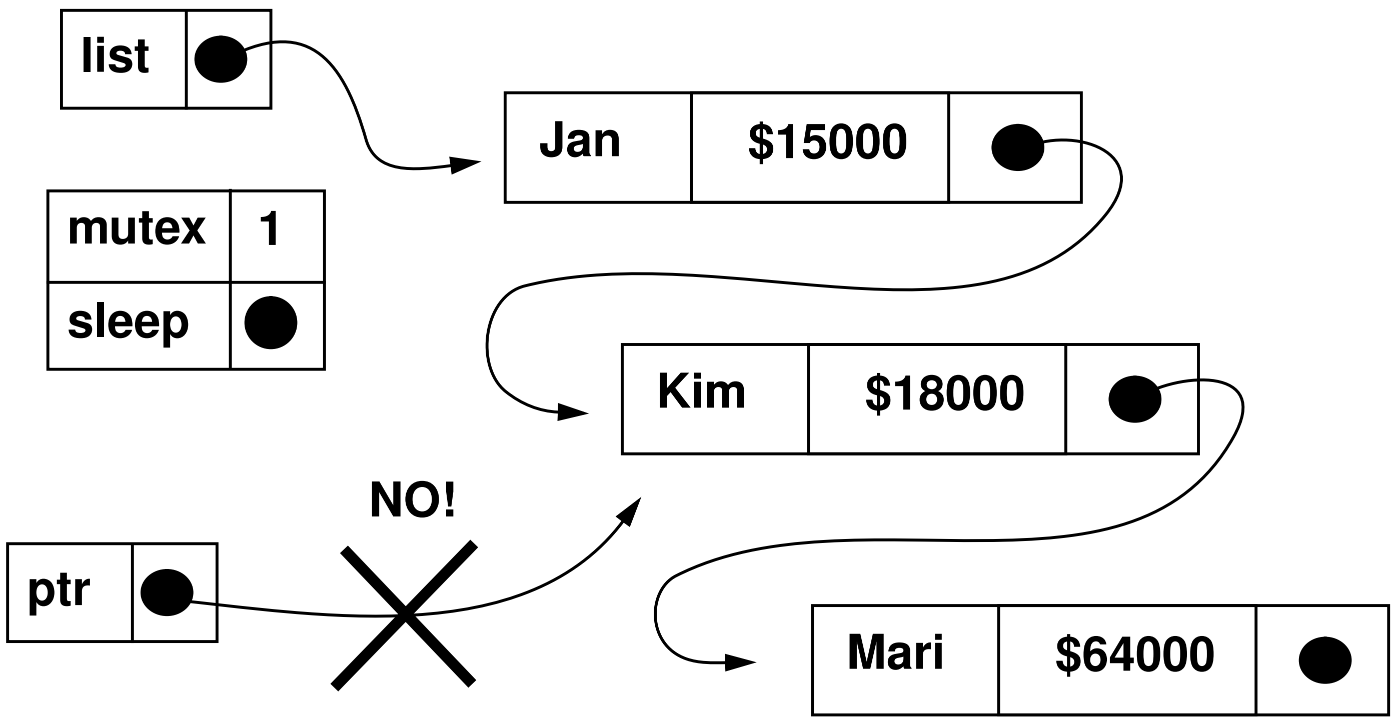
```
 /*                ThreadWin.c                */

/*
  Simple window program that inverts the colour of each button for
  6 seconds.

  In the non-threaded version, it hangs for 6 seconds with the GUI
  inactive (tho events are queued).  In the MT version, it spawns off
  a new thread to do that long “calculation”, hence the GUI remains
  active.  Upon completion, the thread writes to a pipe.  The X
  notifier reads from that pipe and inverts the colour back to the
  original colour.
*/
/*
cc -o ThreadWin ThreadWin.c -L. -R. -g -lpthread -lthread -pthread_extensions -
lposix4 -I/usr/dt/include -I/usr/openwin/include -L/usr/openwin/lib -L/usr/dt/
lib -R/usr/openwin/lib -R/usr/dt/lib -lXm -lXt -lX11 -lm

*/

#define _POSIX_C_SOURCE 199506L
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>
#include <X11/Intrinsic.h>
#include <X11/X.h>
#include <X11/StringDefs.h>
#include <Xm/Xm.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>
#include <pthread.h>
#include “thread_extensions.h”

#define READ 0
#define WRITE 1


/* function prototypes */
int                    ButtonPushed(Widget, XtPointer);
```

```
void                    *DoWork(void *);
XtAppContext            _XtDefaultAppContext();
XtInputCallbackProc     WorkComplete();
static int              message[2];
pthread_attr_t          attr;


main(int argc, char **argv)
{Widget                 toplevel, base, button[200];
 Arg                    wargs[10];
 int                    i, MT=0, N_BUTTONS=25;
 char                   but_label[5];
 XEvent                 event, fake;
 Display                *disp;
 Window                 win;
 XtAppContext           app;
 pthread_t              tid;

 /* check arguments */
 if (argc > 2)
   printf("Usage: %s [MT]\n", argv[0]), exit(0);

 if (argc == 2 && strcmp(argv[1], "MT") == 0) MT = 1;

 if (pipe(message) < 0)
   {fprintf(stderr, "Can't open pipe\n");
    exit(1);
  }
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 PTHREAD_CREATE(&tid, &attr, killer, (void *) 60);

 /* setup our main window */
 toplevel=XtInitialize(argv[0], "ThreadWin", NULL, 0, &argc, argv);

 /* set some arguments for our Row/Column Widget */
 XtSetArg(wargs[0], XmNorientation, XmHORIZONTAL);
 XtSetArg(wargs[1], XmNentryAlignment, XmALIGNMENT_CENTER);
 XtSetArg(wargs[2], XmNisAligned, True);
 XtSetArg(wargs[3], XmNnumColumns, (int)floor(sqrt((double)N_BUTTONS)));
 XtSetArg(wargs[4], XmNpacking, XmPACK_COLUMN);

 /* create the Row/Column Widget */
 base=XtCreateManagedWidget("base", xmRowColumnWidgetClass,
                            toplevel, wargs, 5);


 /* create the button widgets with the button number as it label */
```

```
 for (i=0;i<N_BUTTONS;i++)
   {sprintf(but_label, " %d ", i);
    XtSetArg(wargs[0], XmNlabelString,
             XmStringCreate(but_label, XmSTRING_DEFAULT_CHARSET));
    button[i] = XmCreatePushButton(base, "button", wargs, 1);

    /* tell the button to call ButtonPushed() when pushed */
    XtAddCallback(button[i], XmNactivateCallback,
                   (XtCallbackProc) ButtonPushed, (void *)MT);
  }

 /* manage the buttons and go into the X event loop */
 XtManageChildren(button, N_BUTTONS);
 XtRealizeWidget(toplevel);
 app = _XtDefaultAppContext();
 XtAppAddInput(app, message[READ], (void *)XtInputReadMask,
               (XtInputCallbackProc)WorkComplete, NULL);
 XtMainLoop();
}


XtInputCallbackProc WorkComplete()
{Pixel          fg, bg;
 Arg            warg[2];
 Widget         w;

 if (read(message[READ], &w, sizeof(w)) != sizeof(w))
   {fprintf(stderr, "Read error\n");
   exit(1);
 }

 /* get the buttons foreground and background colors */
 XtSetArg(warg[0], XmNforeground, &fg);
 XtSetArg(warg[1], XmNbackground, &bg);
 XtGetValues(w, warg, 2);

 /* swap the buttons foreground and background colors */
 XtSetArg(warg[0], XmNforeground, bg);
 XtSetArg(warg[1], XmNbackground, fg);
 XtSetValues(w, warg, 2);

 XtSetSensitive(w, True);
 return((XtInputCallbackProc)0);
}

/*
        Button callback routine -- Called when a button is pushed
*/
```

```
int ButtonPushed(Widget w, XtPointer MT)
{int            mt = (int) MT;
 Pixel          fg, bg;
 Arg            warg[2];
 pthread_t      tid;

 XtSetSensitive(w, False);

 /* get the buttons foreground and background colors */
 XtSetArg(warg[0], XmNforeground, &fg);
 XtSetArg(warg[1], XmNbackground, &bg);
 XtGetValues(w, warg, 2);

 /* swap the buttons foreground and background colors */
 XtSetArg(warg[0], XmNforeground, bg);
 XtSetArg(warg[1], XmNbackground, fg);
 XtSetValues(w, warg, 2);

 if (MT)
   PTHREAD_CREATE(&tid, &attr, DoWork, (void *)w);
 else
   DoWork((void *)w);


 return(0);
}


void *DoWork(void *arg)
{Widget                 w = (Widget) arg;
 pthread_t              tid = pthread_self();
 char                   *name = thread_name(tid);

 printf("Thread %s: Start Processing...\n", name);
 sleep(6);
 printf("Thread %s: Processing done...\n", name);

 if (write(message[WRITE], &w, sizeof(w)) != sizeof(w))
   {fprintf(stderr, "Write error\n");
    exit(1);
  }
 return((void *)0);
}
```

Linked Lists: Global Lock



Linked Lists, Global Mutex: list_global_lock.c

```
 /*                 list_global_lock.c                */

/*

  Make a list of people (with names like "Jan_1234") and their salaries.
  Then spawn some threads to run down that list and give raises to all
  the people I like.  Spawn some more threads to delete from the list
  the people you don't like.  (Some people I like might also be people
  you don't like.)

  Insert delays during the raising and deleting functions to simulate disk
  I/O and make the program more interesting.

  Now consider the following parameters:
        o  Raise delay
        o  Liquidation delay
        o  Time for average search

  Which of the various locking designs works fastest for which combination
  of the above?



  This program allocates one global mutex to protect everything.

  GLOBAL_LOCK


         _________________________________
        |     Name      |  salary  |  next |
        |  "Jan_123"    |  100000  |   o---|------>
        |_______________|__________|_______|

*/

/*
cc -o list_global_lock list_global_lock.c -L. -R. -g -lpthread -lthread -
pthread_extensions -lposix4
*/
```

```
#define _POSIX_C_SOURCE 199506L

#include <signal.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include "thread_extensions.h"


typedef struct person_struct
{char *                 name;
 int                    salary;
 struct person_struct * next;
} person_t;




sem_t           barrier;
pthread_mutex_t people_lock = PTHREAD_MUTEX_INITIALIZER;
person_t        people;

int             E_THREADS = 1;         /* # threads running liquidate */
int             F_THREADS = 10;        /* # threads running raise */
int             N_PEOPLE = 1000;       /* # of people in initial list */
int             N_FRIENDS = 100;       /* # of people to give raises */
int             LIQUIDATE_DELAY = 100; /* sleep time during liquidation */
int             RAISE_DELAY = 100;     /* sleep time during raise giving */



void count_conflicts(int i)             /* Note the encapsulation */
{static int             count=0, old_count=0, max_count = 0;
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("Conflicts: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
}

void count_finds(int i)         /* Note the encapsulation */
{static int count=0, old_count=0, max_count = 0;
```

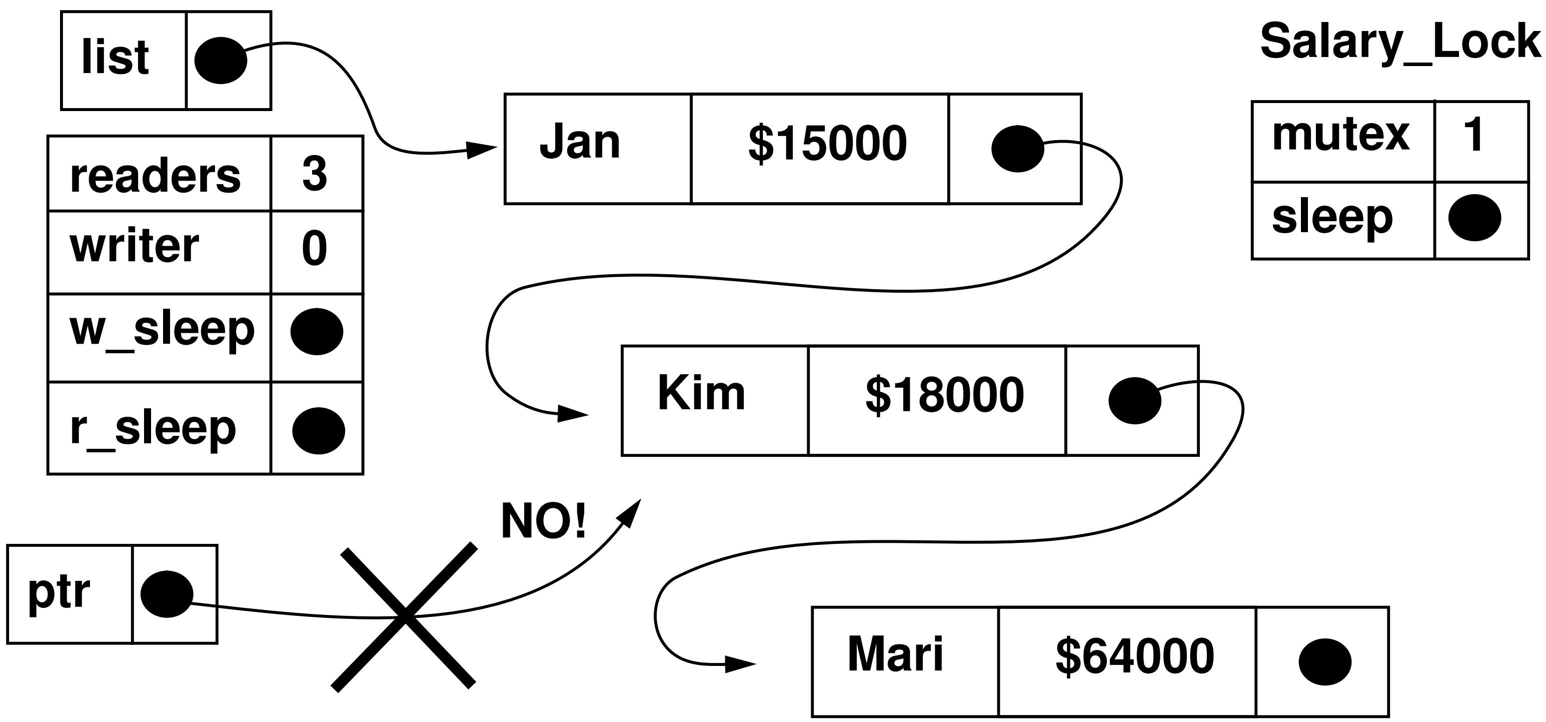
```
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("Total raises + liquidations: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
}


void print_people(person_t *p)  /* Debugging function. Not 100% MT safe */
{int i = 0;

 pthread_mutex_lock(&people_lock);
 while(p != NULL)
   {printf("(%s  %d) ", p->name, p->salary);
    p = p->next;
    i++;
    if ((i % 5) == 0) printf("\n");
  }
 pthread_mutex_unlock(&people_lock);
}


person_t *find_person(char *name)
{person_t *p = &people, *p1, *p2;

 p1 = p->next;
 while(p1 != NULL)
   {if (strcmp(p1->name, name) == 0)
      /* Found! Return previous, Still holding p->lock ! */
      {count_finds(1); return(p);}
    p = p1;
    p1 = p->next;
  }
 return(NULL);                           /* Couldn't find 'em!  Too bad. */
}


void give_raise(person_t *p)
{
  if (p == NULL) return;
  p->next->salary++;
  delay(RAISE_DELAY, 0);
}


pthread_mutex_lock_conflicts(pthread_mutex_t *people_lock)
```

```
{if (pthread_mutex_trylock(people_lock) == EBUSY)
      {count_conflicts(1);
       pthread_mutex_lock(people_lock);
  }
}

void *give_friends_raise(void *arg)
{person_t *p, *friends = (person_t *) arg;

  while(friends != NULL)
    {pthread_mutex_lock_conflicts(&people_lock);
     p = find_person(friends->name);
     give_raise(p);
     pthread_mutex_unlock(&people_lock);
     friends = friends->next;
     delay(0, 0);
   }
  sem_post(&barrier);
}


void liquidate_person(person_t *p)
{person_t  *p1;

 p1 = p->next;
 p->next = p1->next;

 DEBUG(printf("Killing: (%s %d)\n", p1->name, p1->salary));
 free(p1->name);
 free(p1);
 delay(LIQUIDATE_DELAY, 0);
 return;
}


void *liquidate_enemies(void *arg)
{person_t *p, *enemies = (person_t *) arg;

  while(enemies != NULL)
    {pthread_mutex_lock_conflicts(&people_lock);
     p = find_person(enemies->name);
     if (p) liquidate_person(p);
     pthread_mutex_unlock(&people_lock);
     enemies = enemies->next;
     delay(0, 0);
   }
 DEBUG({printf("\nAfter the purge...\n"); print_people(people.next);});
 sem_post(&barrier);
}
```

```
person_t *make_list_of_friends(int seed1)
{int i, r;
 unsigned int seed = (int) seed1;
 char s[20], *s1;
 person_t *p, friends;

 friends.next = NULL;

 for (i=0; i<N_FRIENDS; i++)
   {r = rand_r(&seed) % N_PEOPLE;
    sprintf(s, "Jan_%d", r);
    s1 = (char *) malloc(strlen(s)+1);
    strcpy(s1, s);
    p = (person_t *) malloc(sizeof(person_t));
    p->name = s1;
    p->next = friends.next;
    friends.next = p;
  }
 return(friends.next);
}


person_t *make_list_of_enemies(int s)
{
  return(make_list_of_friends(s));
}


void add_to_people(char *name)
{person_t *p;

 p = (person_t *) malloc(sizeof(person_t));
 p->name = name;
 pthread_mutex_lock(&people_lock);
 p->next = people.next;
 people.next = p;
 pthread_mutex_unlock(&people_lock);
}


void make_list_of_people()
{int i, n;
 char s[20], *s1;

 for (i=0; i<N_PEOPLE; i++)
   {sprintf(s, "Jan_%d", i);
    s1 = (char *) malloc(strlen(s)+1);
```

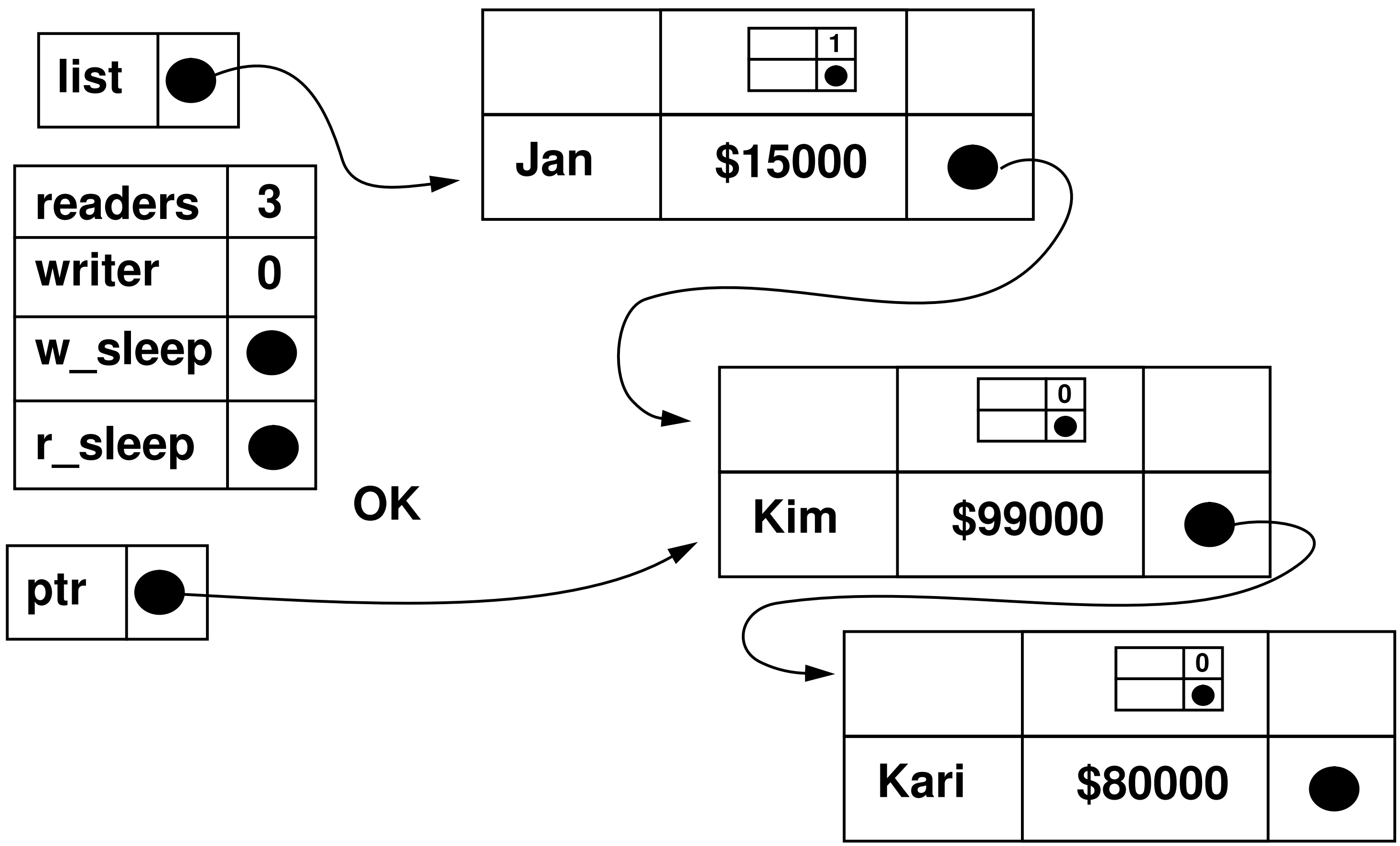
```
    strcpy(s1, s);
    add_to_people(s1);
  }
}


main(int argc, char *argv[])
{int i;
 pthread_attr_t attr;
  pthread_t tid;

 SEM_INIT(&barrier, NULL, 0);
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 people.next = NULL;
 people.name = (char *) -1;                    /* SEGV if you touch this ! */

 if (argc >= 2) LIQUIDATE_DELAY = atoi(argv[1]);
 if (argc >= 3) RAISE_DELAY     = atoi(argv[2]);
 if (argc >= 4) N_PEOPLE        = atoi(argv[3]);
 if (argc >= 5) N_FRIENDS       = atoi(argv[4]);
 if (argc >= 6) E_THREADS       = atoi(argv[5]);
 if (argc >= 7) F_THREADS       = atoi(argv[6]);

 printf(
 "list_global_lock(LIQUIDATE: %d RAISE: %d N_PEOPLE: %d\n N_FRIENDS: %d
E_THREADS: %d F_THREADS: %d)\n",
 LIQUIDATE_DELAY, RAISE_DELAY, N_PEOPLE, N_FRIENDS, E_THREADS, F_THREADS );

 make_list_of_people();
 iobench_start();
 for (i=0; i<F_THREADS; i++)
   PTHREAD_CREATE(&tid, &attr, give_friends_raise, make_list_of_friends(i*10));
 for (i=0; i<E_THREADS; i++)
   PTHREAD_CREATE(&tid, &attr, liquidate_enemies, make_list_of_enemies(i));

 thread_single_barrier(&barrier, F_THREADS + E_THREADS);/* Instead of join()! */
 iobench_end();
 count_finds(-1);
 count_conflicts(-1);
 iobench_report();
 printf("Done\n");
 exit(0);
}
```

Linked Lists: Global RWLock



With one RW Lock protecting the *entire* list & one mutex for salary
No outside pointers allowed without holding the RW Lock.
Could you search down the list with a read lock and then ask for a write lock?

Linked Lists: Global Lock + local



With one mutex protecting the list and a local salary lock.
Outside pointers must hold the *appropriate* mutex.
(cf: Reference counting)

Linked Lists, Global RWLock+: list_global_rw2.c

```
/*                list_global_rw2.c                */

/*

  Make a list of people (with names like "Jan_1234") and their salaries.
  Then spawn some threads to run down that list and give raises to all
  the people I like.  Spawn some more threads to delete from the list
  the people you don't like.  (Some people I like might also be people
  you don't like.)

  Insert delays during the raising and deleting functions to simulate disk
  I/O and make the program more interesting.

  Now consider the following parameters:
        o  Raise delay
        o  Liquidation delay
        o  Time for average search

  Which of the various locking designs works fastest for which combination
  of the above?



  This program allocates one global rwlock to protect the list itself, and
  one mutex to protect each salary.  It is *vital* that you know exactly
  what each lock is going to protect.

  GLOBAL_RWLOCK

         __________________________________
        |              |  salary  |        |
        |              |  lock    |        |
        |--------------|----------|--------|
        |    Name      |  salary  |  next  |
        |  "Jan_123"   |  100000  |   o---|------>
        |______________|__________|________|


*/
```

```
/*
cc -o list_global_rw2 list_global_rw2.c -L. -R. -g -lpthread -lthread -
pthread_extensions -lposix4
*/


#define _POSIX_C_SOURCE 199506L

#include <signal.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include <stdlib.h>
#include "thread_extensions.h"


typedef struct person_struct
{char *                  name;
 int                     salary;
 pthread_mutex_t         salary_lock;
 struct person_struct * next;
} person_t;


sem_t           barrier;
thread_rwlock_t people_lock;
person_t        people;

int             E_THREADS = 1;         /* # threads running liquidate */
int             F_THREADS = 10;        /* # threads running raise */
int             N_PEOPLE = 1000;       /* # of people in initial list */
int             N_FRIENDS = 100;       /* # of people to give raises */
int             LIQUIDATE_DELAY = 100; /* sleep time during liquidation */
int             RAISE_DELAY = 100;     /* sleep time during raise giving */



void count_conflicts(int i)             /* Note the encapsulation */
{static int count=0, old_count=0, max_count = 0;
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("Conflicts: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
```

```
}

void count_finds(int i)          /* Note the encapsulation */
{static int count=0, old_count=0, max_count = 0;
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("Total raises + liquidations: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
}


void print_people(person_t *p)  /* Debugging function. Not 100% MT safe */
{int i = 0;

 printf("(");
 thread_rw_rdlock(&people_lock);
 while(p != NULL)
   {pthread_mutex_lock(&(p->salary_lock));
    printf("(%s  %d) ", p->name, p->salary);
    pthread_mutex_unlock(&(p->salary_lock));
    p = p->next;
    i++;
    if ((i % 5) == 0) printf("\n ");
  }
 thread_rw_unlock(&people_lock);
 printf(")\n");
}



person_t *find_person(char *name)
{person_t *p = &people, *p1, *p2;

 p1 = p->next;
 while(p1 != NULL)
   {if (strcmp(p1->name, name) == 0)
      /* Found! Return previous */
      {count_finds(1); return(p);}
    p = p1;
    p1 = p->next;
  }
 return(NULL);                    /* Couldn't find `em!  Too bad. */
}
```

```
void give_raise(person_t *p)
{person_t *p1;

  if (p != NULL)
    {p1 = p->next;
     DEBUG(printf("Giving raise to: (%s %d)\n", p1->name, p1->salary))
     pthread_mutex_lock(&(p1->salary_lock));
     thread_rw_unlock(&people_lock);            /* Unlock here */
     p1->salary++;
     delay(RAISE_DELAY, 0);      /* If the delay is outside the lock? */
     pthread_mutex_unlock(&(p1->salary_lock));
   }
 else
     thread_rw_unlock(&people_lock);            /* Unlock here */
}

thread_rw_rdlock_conflicts(thread_rwlock_t *people_lock)
{if (thread_rw_tryrdlock(people_lock) == EBUSY)
   {count_conflicts(1);
    thread_rw_rdlock(people_lock);
  }
}


thread_rw_wrlock_conflicts(thread_rwlock_t *people_lock)
{if (thread_rw_trywrlock(people_lock) == EBUSY)
   {count_conflicts(1);
    thread_rw_wrlock(people_lock);
  }
}

void *give_friends_raise(void *arg)
{person_t *p, *friends = (person_t *) arg;

  while(friends != NULL)
    {thread_rw_rdlock_conflicts(&people_lock);
     p = find_person(friends->name);
     give_raise(p);
     friends = friends->next;
   }
  sem_post(&barrier);
}



void liquidate_person(person_t *p)
{person_t  *p1;

 if (p)
```

```
   {p1 = p->next;
    p->next = p1->next;
    thread_rw_unlock(&people_lock);

    pthread_mutex_lock(&(p1->salary_lock));
    pthread_mutex_unlock( &(p1->salary_lock));
    pthread_mutex_destroy(&(p1->salary_lock));
    DEBUG(printf("Killing: (%s %d)\n", p1->name, p1->salary))
    free(p1->name);
    free(p1);
    delay(LIQUIDATE_DELAY, 0);
  }
 else
   thread_rw_unlock(&people_lock);

 return;
}



void *liquidate_enemies(void *arg)
{person_t *p, *enemies = (person_t *) arg;

  while(enemies != NULL)
    {thread_rw_wrlock_conflicts(&people_lock);
     p = find_person(enemies->name);
     liquidate_person(p);
     enemies = enemies->next;
   }
 DEBUG({printf("\nAfter the purge...\n"); print_people(people.next);})
 sem_post(&barrier);
}


person_t *make_list_of_friends(int seed1)
{int i, r;
 unsigned int seed = (int) seed1;
 char s[20], *s1;
 person_t *p, friends;

 friends.next = NULL;

 for (i=0; i<N_FRIENDS; i++)
   {r = rand_r(&seed) % N_PEOPLE;
    sprintf(s, "Jan_%d", r);
    s1 = (char *) malloc(strlen(s)+1);
    strcpy(s1, s);
    p = (person_t *) malloc(sizeof(person_t));
    p->name = s1;
```

```
    p->next = friends.next;
    friends.next = p;
  }
 return(friends.next);
}


person_t *make_list_of_enemies(int s)
{
  return(make_list_of_friends(s));
}


void add_to_people(char *name)
{person_t *p;

 p = (person_t *) malloc(sizeof(person_t));
 p->name = name;
 pthread_mutex_init(&(p->salary_lock), NULL);
 thread_rw_wrlock(&people_lock);
 p->next = people.next;
 people.next = p;
 thread_rw_unlock(&people_lock);
}


void make_list_of_people()
{int i, n;
 char s[20], *s1;

 for (i=0; i<N_PEOPLE; i++)
   {sprintf(s, "Jan_%d", i);
    s1 = (char *) malloc(strlen(s)+1);
    strcpy(s1, s);
    add_to_people(s1);
  }
}


main(int argc, char *argv[])
{int i;
 pthread_attr_t attr;
  pthread_t tid;

 SEM_INIT(&barrier, NULL, 0);
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 thread_rwlock_init(&people_lock, NULL);
```

```
 people.next = NULL;
 people.name = (char *) -1;                    /* SEGV if you touch this ! */

 if (argc >= 2) LIQUIDATE_DELAY = atoi(argv[1]);
 if (argc >= 3) RAISE_DELAY     = atoi(argv[2]);
 if (argc >= 4) N_PEOPLE        = atoi(argv[3]);
 if (argc >= 5) N_FRIENDS       = atoi(argv[4]);
 if (argc >= 6) E_THREADS       = atoi(argv[5]);
 if (argc >= 7) F_THREADS       = atoi(argv[6]);

 printf(
 "list_global_rw2(LIQUIDATE: %d RAISE: %d N_PEOPLE: %d\n N_FRIENDS: %d
E_THREADS: %d F_THREADS: %d)\n",
 LIQUIDATE_DELAY, RAISE_DELAY, N_PEOPLE, N_FRIENDS, E_THREADS, F_THREADS );

 make_list_of_people();
 iobench_start();
 for (i=0; i<F_THREADS; i++)
   PTHREAD_CREATE(&tid, &attr, give_friends_raise, make_list_of_friends(i*10));
 for (i=0; i<E_THREADS; i++)
   PTHREAD_CREATE(&tid, &attr, liquidate_enemies, make_list_of_enemies(i));

 thread_single_barrier(&barrier, F_THREADS + E_THREADS);
 iobench_end();
 count_finds(-1);
 count_conflicts(-1);
 iobench_report();
 printf("Done\n");
 exit(0);

}
```

Exercise Objective

Examine the code in each of the different list implementations and understand it. Estimate which design should be faster with different delay times. Test. Fill out the table below.

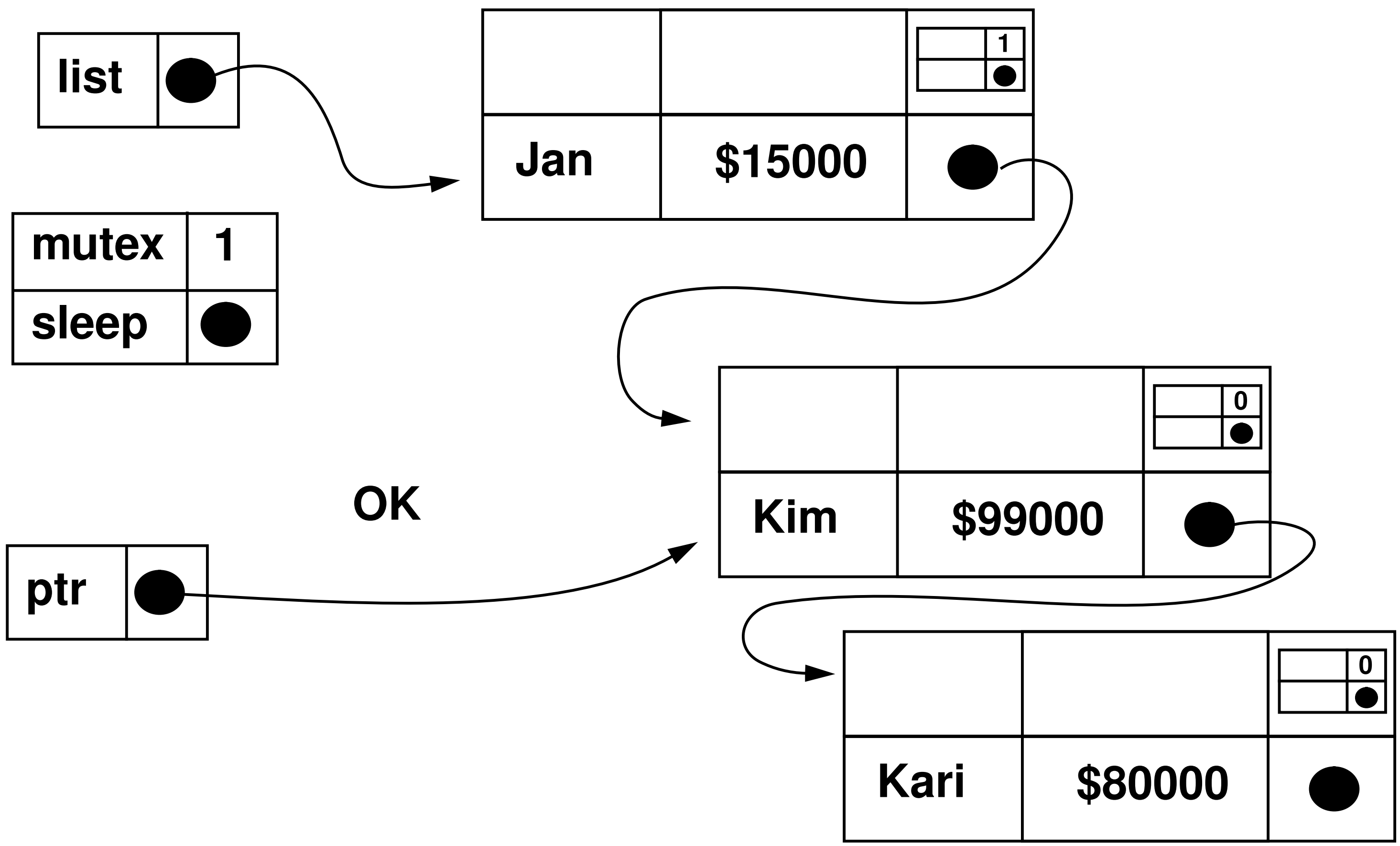
Now figure out why you get some odd looking number of conflicts. What should the effect be of replacing simple mutexes with FIFO mutexes?

What is the time of execution for each of these designs? Figure out the appropriate numbers to fill in for each of the three parameters which will give you the most useful data.

Hint: Start by figuring out how long it takes to compare each member of the list with no delays. And how long it takes to lock local locks. Then you should be able to fill in the data for the rest of the entries from theory only.

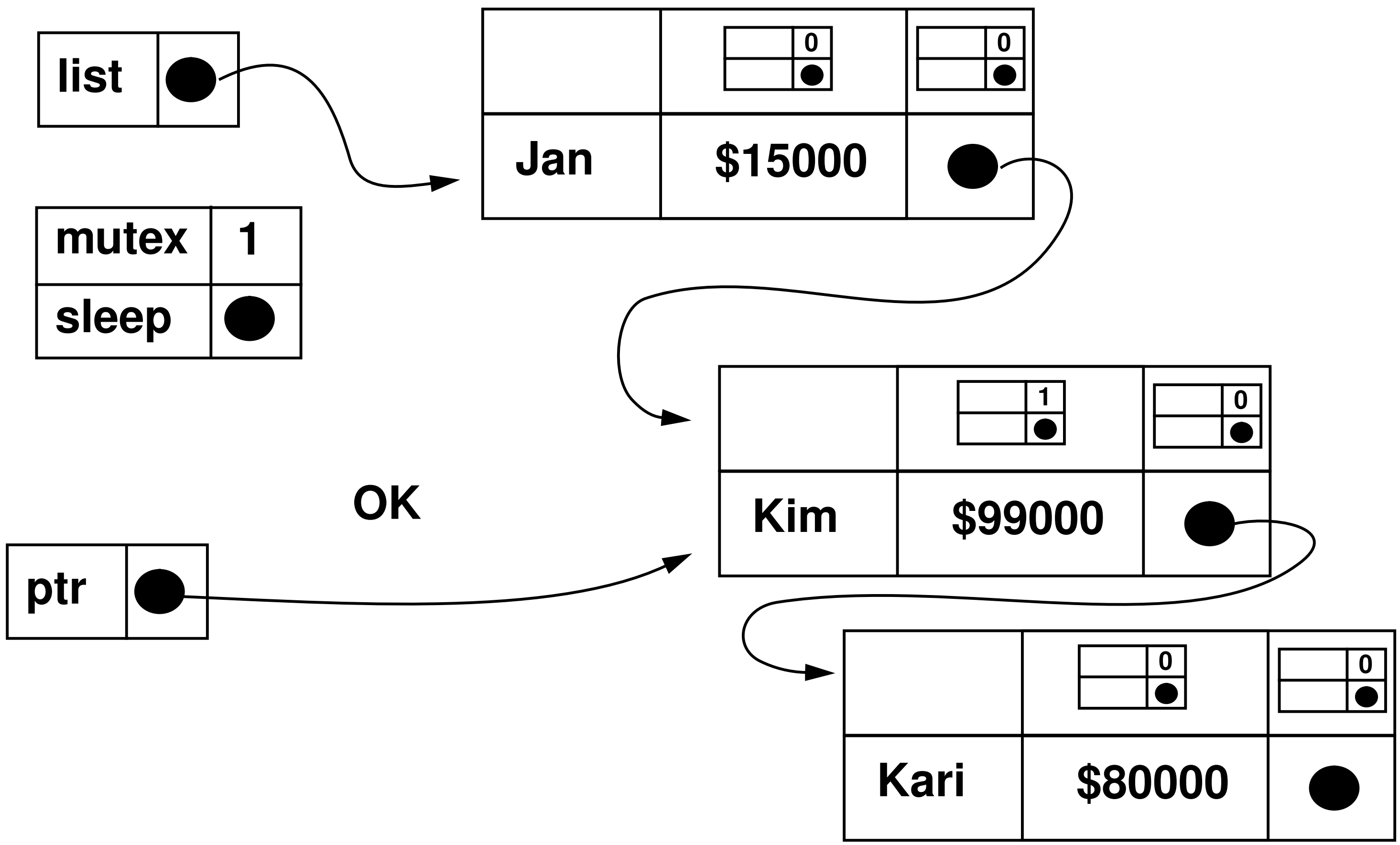
	Default Delays	Zero Delay (use 10,000 people)
Global Mutex		
Global RW + Local Mutex		
Two Local Mutexes		

Linked Lists: Single Lock



With one mutex protecting each element in the list.
Outside pointers must hold the *appropriate* mutex.
(cf: Reference counting)

Linked Lists: Two Locks



One mutex protecting the structure, and one protecting the data.
Outside pointers must hold the *appropriate* mutex.

Linked Lists, 2 Local Mutexes: list_2local_locks.c

```
 /*                list_2local_locks.c                */

/*

  Make a list of people (with names like "Jan_1234") and their salaries.
  Then spawn some threads to run down that list and give raises to all
  the people I like.  Spawn some more threads to delete from the list
  the people you don't like.  (Some people I like might also be people
  you don't like.)

  Insert delays during the raising and deleting functions to simulate disk
  I/O and make the program more interesting.

  Now consider the following parameters:
        o  Raise delay
        o  Liquidation delay
        o  Time for average search

  Which of the various locking designs works fastest for which combination
  of the above?



  This program allocates two mutexes per element, one to protect the
  next-pointer and following member, the other to protect the salary of
  the current member.  It is *vital* that you know exactly what each
  lock is going to protect.

         ___________________________________
        |              |  salary   |  next  |
        |              |  lock     |  lock  |
        |--------------|-----------|--------|
        |    Name      |  salary   |  next  |
        |  "Jan_123"   |  100000   |    o---|------>
        |______________|___________|________|

*/

/*
cc -o list_2local_locks list_2local_locks.c -L. -R. -g -lpthread -lthread -
```

```
pthread_extensions -lposix4
*/


#define _POSIX_C_SOURCE 199506L

#include <signal.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include <stdlib.h>
#include "thread_extensions.h"


typedef struct person_struct
{char *                 name;
 int                    salary;
 struct person_struct * next;
 pthread_mutex_t        next_lock;
 pthread_mutex_t        salary_lock;
} person_t;

sem_t          barrier;
person_t        people;

int             E_THREADS = 1;        /* # threads running liquidate */
int             F_THREADS = 10;       /* # threads running raise */
int             N_PEOPLE = 1000;      /* # of people in initial list */
int             N_FRIENDS = 100;      /* # of people to give raises */
int             LIQUIDATE_DELAY = 100; /* sleep time during liquidation */
int             RAISE_DELAY = 100;    /* sleep time during raise giving */


void count_conflicts(int i)             /* Note the encapsulation */
{static int count=0, old_count=0, max_count = 0;
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("Conflicts: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
}

void count_finds(int i)         /* Note the encapsulation */
{static int count=0, old_count=0, max_count = 0;
```

```
 static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 if (i == -1) printf("Total raises + liquidations: %d\n", max_count);
 if (count > max_count)
   max_count = count;
 pthread_mutex_unlock(&count_lock);
}


void print_people(person_t *p)  /* Debugging function. Not 100% MT safe */
{int i = 0;

  while(p != NULL)
    {printf("(%s  %d) ", p->name, p->salary);
     p = p->next;
     i++;
     if ((i % 5) == 0) printf("\n");
   }
}


void add_to_people(char *name)
{person_t *p;

 p = (person_t *) malloc(sizeof(person_t));
 p->name = name;
 pthread_mutex_init(&(p->next_lock), NULL);
 pthread_mutex_init(&(p->salary_lock), NULL);
 pthread_mutex_lock(&(people.next_lock));
 p->next = people.next;
 people.next = p;
 pthread_mutex_unlock(&(people.next_lock));
}

pthread_mutex_lock_conflicts(pthread_mutex_t *people_lock)
{if (pthread_mutex_trylock(people_lock) == EBUSY)
      {count_conflicts(1);
       pthread_mutex_lock(people_lock);
  }
}

person_t *find_person(char *name)
{person_t *p = &people, *p1, *p2;

 pthread_mutex_lock(&(p->next_lock));
 p1 = p->next;
 while(p1 != NULL)
```

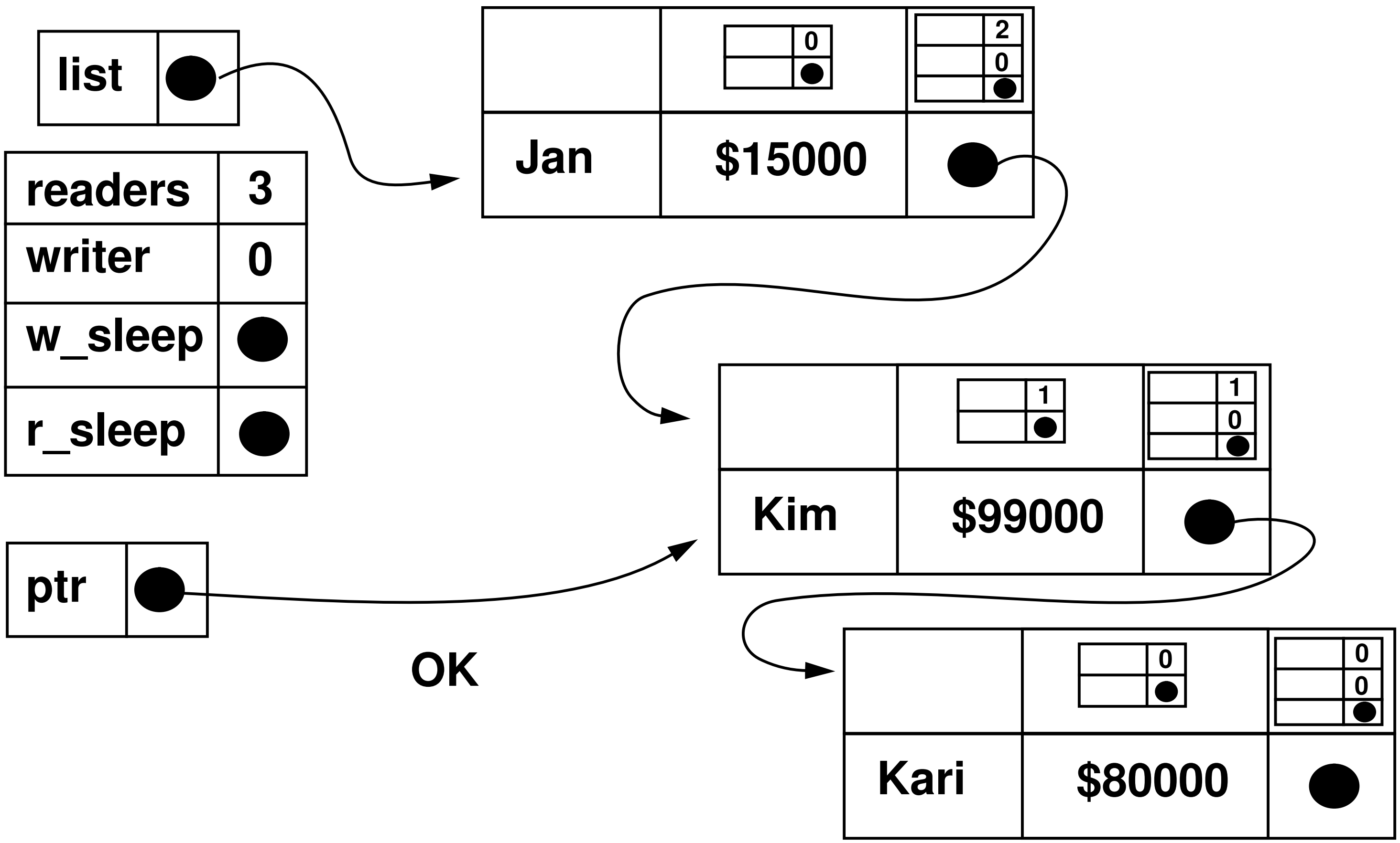
```
   {if (strcmp(p1->name, name) == 0)
      /* Found! Return *previous* person, Still holding p->lock ! */
      {count_finds(1); return(p);}
    pthread_mutex_lock_conflicts(&(p1->next_lock));
    pthread_mutex_unlock(&(p->next_lock));
    p = p1;
    p1 = p->next;
  }
 pthread_mutex_unlock(&(p->next_lock));
 return(NULL);   /* Couldn't find `em!  Too bad. No Locks held */
}


void give_raise(person_t *p)
{person_t *p1;

 if (p == NULL)  return;
 p1 = p->next;
 pthread_mutex_lock(&(p1->salary_lock));
 pthread_mutex_unlock(&(p->next_lock)); /* Unlock here!! */
 p1->salary++;
 delay(RAISE_DELAY, 0);
 pthread_mutex_unlock(&(p1->salary_lock));
}


void *give_friends_raise(void *arg)
{person_t *p, *friends = (person_t *) arg;

  while(friends != NULL)
    {p = find_person(friends->name);
     give_raise(p);
     friends = friends->next;
   }
  sem_post(&barrier);
}



void liquidate_person(person_t *p)
{person_t  *p1;

 p1 = p->next;
 pthread_mutex_lock(&(p1->next_lock));
 p->next = p1->next;
 pthread_mutex_unlock(&(p->next_lock));

 DEBUG(printf("Killing: (%s %d)\n", p1->name, p1->salary))
 pthread_mutex_lock(&(p1->salary_lock));
```

```
 pthread_mutex_unlock( &(p1->next_lock));
 pthread_mutex_destroy(&(p1->next_lock));
 pthread_mutex_unlock( &(p1->salary_lock));
 pthread_mutex_destroy(&(p1->salary_lock));
 free(p1->name);
 free(p1);
 delay(LIQUIDATE_DELAY, 0);
 return;
}


void *liquidate_enemies(void *arg)
{person_t *p, *enemies = (person_t *) arg;

  while(enemies != NULL)
    {p = find_person(enemies->name);
     if (p)
       liquidate_person(p);
     enemies = enemies->next;
   }
 DEBUG({printf("\nAfter the purge...\n"); print_people(people.next);})
 sem_post(&barrier);
}


person_t *make_list_of_friends(int seed1)
{int i, r;
 unsigned int seed = (int) seed1;
 char s[20], *s1;
 person_t *p, friends;

 friends.next = NULL;

 for (i=0; i<N_FRIENDS; i++)
   {r = rand_r(&seed) % N_PEOPLE;
    sprintf(s, "Jan_%d", r);
    s1 = (char *) malloc(strlen(s)+1);
    strcpy(s1, s);
    p = (person_t *) malloc(sizeof(person_t));
    p->name = s1;
    p->next = friends.next;
    friends.next = p;
  }
 return(friends.next);
}


person_t *make_list_of_enemies(int s)
{
```

```
  return(make_list_of_friends(s));
}



void *make_list_of_people()
{int i, n;
 char s[20], *s1;

 for (i=0; i<N_PEOPLE; i++)
   {sprintf(s, "Jan_%d", i);
    s1 = (char *) malloc(strlen(s)+1);
    strcpy(s1, s);
    add_to_people(s1);
  }
}


main(int argc, char *argv[])
{int i;
 pthread_attr_t attr;
  pthread_t tid;

 SEM_INIT(&barrier, NULL, 0);
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 people.next = NULL;
 people.name = (char *) -1;             /* SEGV if you touch this ! */
 pthread_mutex_init(&(people.next_lock), NULL);


 if (argc >= 2) LIQUIDATE_DELAY = atoi(argv[1]);
 if (argc >= 3) RAISE_DELAY     = atoi(argv[2]);
 if (argc >= 4) N_PEOPLE        = atoi(argv[3]);
 if (argc >= 5) N_FRIENDS       = atoi(argv[4]);
 if (argc >= 6) E_THREADS       = atoi(argv[5]);
 if (argc >= 7) F_THREADS       = atoi(argv[6]);

 printf(
 "list_2local_locks(LIQUIDATE: %d RAISE: %d N_PEOPLE: %d\n N_FRIENDS: %d
E_THREADS: %d F_THREADS: %d)\n",
 LIQUIDATE_DELAY, RAISE_DELAY, N_PEOPLE, N_FRIENDS, E_THREADS, F_THREADS );


 make_list_of_people();
 iobench_start();
 for (i=0; i<F_THREADS; i++)
   PTHREAD_CREATE(&tid, &attr, give_friends_raise, make_list_of_friends(i*10));
```

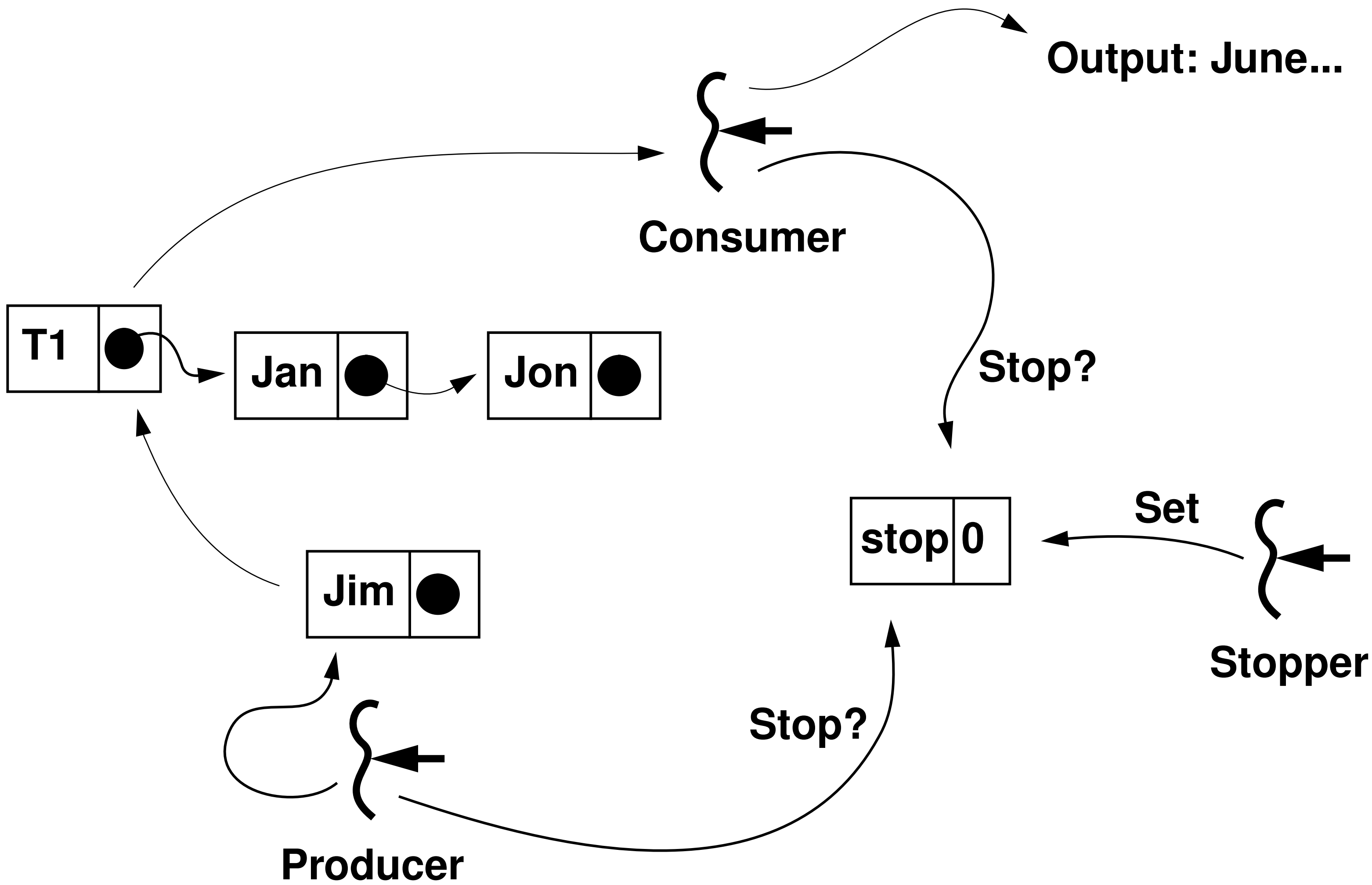
```
 for (i=0; i<E_THREADS; i++)
   PTHREAD_CREATE(&tid, &attr, liquidate_enemies, make_list_of_enemies(i));

 thread_single_barrier(&barrier, F_THREADS + E_THREADS);
 iobench_end();
 count_finds(-1);
 count_conflicts(-1);
 iobench_report();
 printf("Done\n");
 exit(0);

}
```

Linked Lists: R/W Lock & Lock



**R/W lock protecting the structure, and mutex protecting the data.
Outside pointers must hold the *appropriate* mutex/lock.**

Stoppable Producer/Consumer



Exercise Objective

Edit the code in `stop_queue_problem.c` and use the new global variable, `stop`. When `stop==FALSE`, the program runs as before. When it gets changed to `TRUE`, then the producers will stop adding new elements and exit. (They may complete whatever they're working on.) The consumers will also stop consuming and exit. (They MUST complete their current job.)

Print out the length of the queue after they have exited. Set `stop` back to `FALSE` and create some new consumers. *After* the consumers have emptied the queue, start up some new producers.

I have a thread which calls `stopper()`, which sleeps for 4 seconds, then sets `stop` to `TRUE`. It must do whatever is necessary to wake up sleeping consumers or producers, then exit itself.

Don't use UNIX signals!

My solution is in `stop_queues_solution.c`. Many minor (and major) variations are possible.

Wild Guess at Time to Completion:

30 minutes think/design time.
2 hours programming/debugging

Stoppable Queue: stop_queue_problem.c

```
/*                  stop_queue_problem.c                */

/*
  This is the prototypical producer/consumer example done by
  building a singly linked list. Every few seconds we stop the
  queue completely, letting the consumers complete their current
  request, but not taking anything new from the queue.  Upon
  restart, the consumers get to empty the list before the producers
  start.

  Delays are inserted to simulate disk I/O (and to make things
  run slower!).
*/

/*
cc -o stop_queue_problem stop_queue_problem.c -L. -R. -g -lpthread -lthread -
lthread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include "thread_extensions.h"


typedef struct request_struct
{int                     client;
 char                    *buf;
 pthread_t               tid;
 struct request_struct   *next;
} request_t;

request_t                *requests = NULL;
int                      length = 0;
pthread_cond_t           requests_producer = PTHREAD_COND_INITIALIZER;
pthread_cond_t           requests_consumer = PTHREAD_COND_INITIALIZER;
pthread_mutex_t          requests_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_attr_t           attr;
```

```
sem_t                       barrier;
int                         GET_DELAY = 20, PROCESS_DELAY = 250;
int                         stop = FALSE;
int                         N_PROD = 2, N_CONS = 2;
int                         n_produced = 0, n_processed = 0;



void print_requests()
{request_t *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->client);
    r = r->next;
  }
}


request_t *get_request()
{static pthread_mutex_t mr  = PTHREAD_MUTEX_INITIALIZER;
 int c;
 request_t *request;
 pthread_t tid = pthread_self();
 char *name = thread_name(tid);

 delay(GET_DELAY, 0);
 pthread_mutex_lock(&mr);
 c=n_produced++;
 pthread_mutex_unlock(&mr);

 request = (request_t *) malloc(sizeof(request_t));
 request->client = c;                           /* Cannot use count here! */
 request->tid = tid;

 DEBUG(printf("[%s] Creating request  %4d\n", name, c));
 return(request);
}


void process_request(request_t *request)
{static pthread_mutex_t mr  = PTHREAD_MUTEX_INITIALIZER;
 pthread_t tid = pthread_self();
 char *name = thread_name(tid);

 delay(PROCESS_DELAY, 0);
 DEBUG(printf("[%s] Processed request %4d [%s]\n",
             name, request->client, thread_name(request->tid)));
 free(request);
 pthread_mutex_lock(&mr);
```

```
 n_processed++;
 pthread_mutex_unlock(&mr);
}


request_t *remove_request()
{request_t *request;

 request = requests;
 requests = requests->next;
 length--;
 return(request);
}


void add_request(request_t *request)
{
 request->next = requests;                         /* Insert at head of list */
 requests = request;
 length++;
}


void *producer(void *arg)
{request_t *request;
 while(TRUE)
   {request = get_request();
    pthread_mutex_lock(&requests_lock);
    while (length >= 10)
      {delay(20, 0);             /* Leave in!  Makes lost wakeup more likely. */
       pthread_cond_wait(&requests_producer, &requests_lock);}
    add_request(request);
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&requests_consumer);
   }
 pthread_mutex_unlock(&requests_lock);
 sem_post(&barrier);
 pthread_exit(NULL);
}


void *consumer(void *arg)
{request_t *request;

 while(TRUE)
   {pthread_mutex_lock(&requests_lock);
    while (length == 0)
      {delay(20, 0);                     /* Leave in!  Makes lost wakeup more
likely. */
```

```
       pthread_cond_wait(&requests_consumer, &requests_lock);}
    request = remove_request();
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&requests_producer);
    process_request(request);
  }
 pthread_mutex_unlock(&requests_lock);
 sem_post(&barrier);
 pthread_exit(NULL);
}


void *stopper(void *arg)
{
  sleep(4);
  stop = TRUE;
  printf("Time to stop!\n");
  pthread_exit(NULL);
}


main(int argc, char **argv)
{int i, j;
 pthread_t tid;

 SEM_INIT(&barrier, NULL,  0);
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

 if (argc >= 2) GET_DELAY     = atoi(argv[1]);
 if (argc >= 3) PROCESS_DELAY = atoi(argv[2]);
 if (argc >= 4) N_PROD        = atoi(argv[3]);
 if (argc >= 5) N_CONS        = atoi(argv[4]);
 printf("GET_DELAY (ms) = %d PROCESS_DELAY (ms) = %d N_PROD = %d N_CONS = %d\n",
        GET_DELAY, PROCESS_DELAY, N_PROD, N_CONS);
 PTHREAD_CREATE(&tid, &attr, killer, (void *) 60);

 for (j=0; j < 3; j++)
   {printf("Starting consumers. List length: %d Produced: %d Consumed %d.\n",
           length, n_produced, n_processed);
    for (i=0; i<N_CONS; i++)
      PTHREAD_CREATE(&tid, &attr, consumer, NULL);


    printf("Starting producers. List length: %d Produced: %d Consumed %d.\n",
           length, n_produced, n_processed);
    for (i=0; i<N_PROD; i++)
      PTHREAD_CREATE(&tid, &attr, producer, NULL);
```

```
    PTHREAD_CREATE(&tid, &attr, stopper, NULL);
    thread_single_barrier(&barrier, N_PROD+N_CONS);
    stop = FALSE;
    printf("All exited. List length: %d Produced: %d Consumed %d.\n",
           length, n_produced, n_processed);
    sleep(4);
   }
 printf("Done\n");
 exit(0);
}
```

Stoppable Queue: stop_queue_solution.c

```
 /*                 stop_queue_solution.c                */

/*
  This is the prototypical producer/consumer example done by
  building a singly linked list. Every few seconds we stop the
  queue completely, letting the consumers complete their current
  request, but not taking anything new from the queue.  Upon
  restart, the consumers get to empty the list before the producers
  start.

  Delays are inserted to simulate disk I/O (and to make things
  run slower!).
*/

/*
cc -o stop_queue_solution stop_queue_solution.c -L. -R. -g -lpthread -lthread -
pthread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include "thread_extensions.h"


typedef struct request_struct
{int                     client;
 char                    *buf;
 pthread_t               tid;
 struct request_struct   *next;
} request_t;

request_t                *requests = NULL;
int                      length = 0;
pthread_cond_t           requests_producer = PTHREAD_COND_INITIALIZER;
pthread_cond_t           requests_consumer = PTHREAD_COND_INITIALIZER;
pthread_mutex_t          requests_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_attr_t           attr;
```

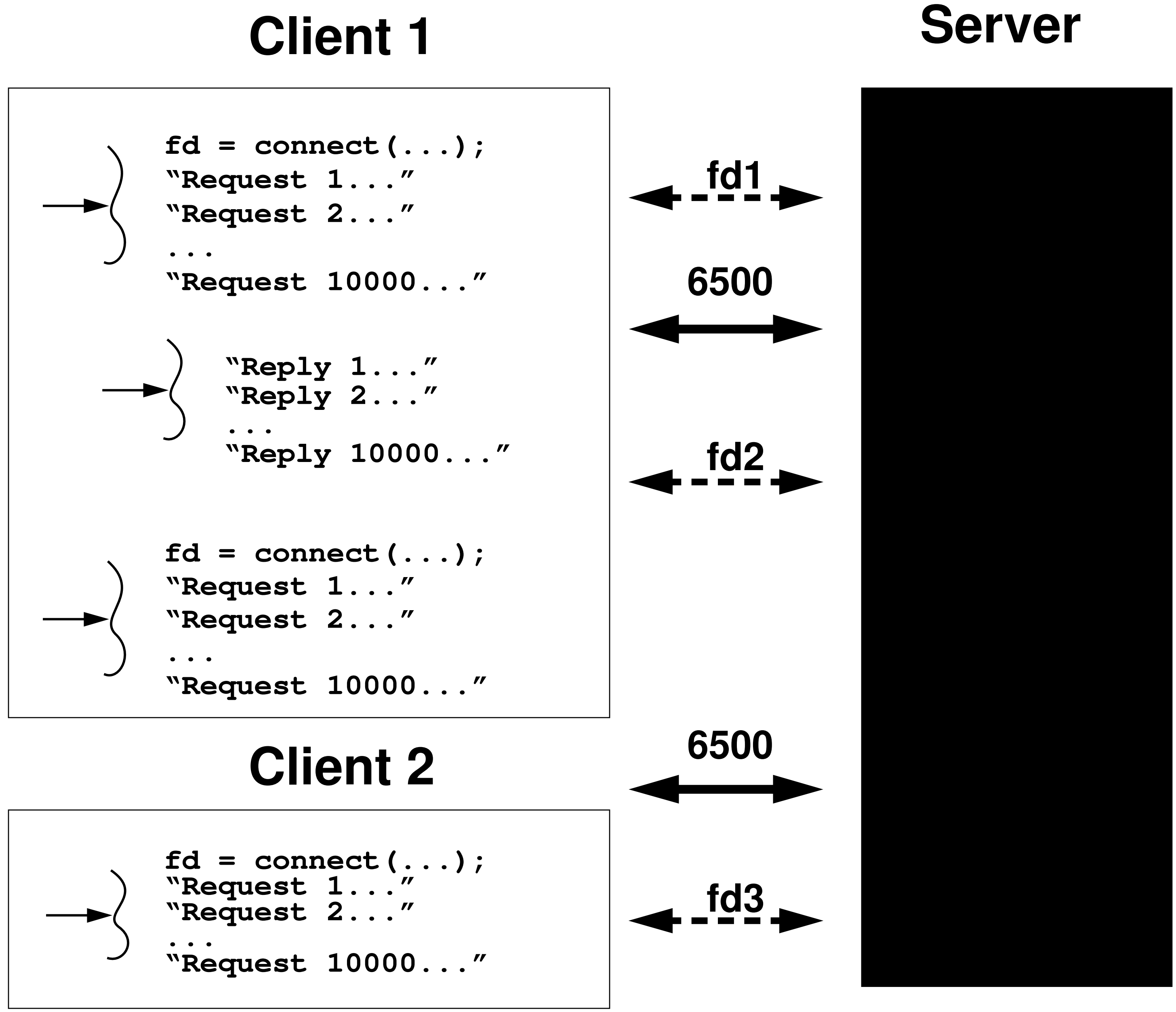
```
sem_t                    barrier;
int                      GET_DELAY = 200, PROCESS_DELAY = 250;
int                      stop = FALSE;
int                      N_PROD = 2, N_CONS = 2;
int                      n_processed = 0, n_produced = 0;



void print_requests()
{request_t *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->client);
    r = r->next;
  }
}


request_t *get_request()
{static pthread_mutex_t mr  = PTHREAD_MUTEX_INITIALIZER;
 int c;
 request_t *request;
 pthread_t tid = pthread_self();
 char *name = thread_name(tid);

 delay(GET_DELAY, 0);
 pthread_mutex_lock(&mr);
 c=n_produced++;
 pthread_mutex_unlock(&mr);

 request = (request_t *) malloc(sizeof(request_t));
 request->client = c;                      /* Cannot use count here! */
 request->tid = tid;

 DEBUG(printf("[%s] Creating request  %4d\n", name, c));
 return(request);
}


void process_request(request_t *request)
{static pthread_mutex_t mr  = PTHREAD_MUTEX_INITIALIZER;
 pthread_t tid = pthread_self();
 char *name = thread_name(tid);

 delay(PROCESS_DELAY, 0);
 DEBUG(printf("[%s] Processed request %4d [%s]\n",
              name, request->client, thread_name(request->tid)));
 free(request);
 pthread_mutex_lock(&mr);
```

```
 n_processed++;
 pthread_mutex_unlock(&mr);
}



request_t *remove_request()
{request_t *request;

 request = requests;
 requests = requests->next;
 length--;
 return(request);
}


void add_request(request_t *request)
{
 request->next = requests;                      /* Insert at head of list */
 requests = request;
 length++;
}


void *producer(void *arg)
{request_t *request;
 while(TRUE)
   {request = get_request();
    pthread_mutex_lock(&requests_lock);
    while ((length >= 10) && (!stop))
      pthread_cond_wait(&requests_producer, &requests_lock);
    add_request(request);
    if (stop) break;
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&requests_consumer);
  }
 pthread_mutex_unlock(&requests_lock);
 sem_post(&barrier);
 pthread_exit(NULL);
}


void *consumer(void *arg)
{request_t *request;

 while(TRUE)
   {pthread_mutex_lock(&requests_lock);
    while ((length == 0) && (!stop))
      pthread_cond_wait(&requests_consumer, &requests_lock);
```

```
    if (stop) break;
    request = remove_request();
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&requests_producer);
    process_request(request);
   }
 pthread_mutex_unlock(&requests_lock);
 sem_post(&barrier);
 pthread_exit(NULL);
}


void *stopper(void *arg)
{
  sleep(4);
  pthread_mutex_lock(&requests_lock);
  stop = TRUE;                          /* No mutex? Lost wakeup!! */
  pthread_mutex_unlock(&requests_lock);
  printf("Time to stop!\n");
  pthread_cond_broadcast(&requests_producer);
  pthread_cond_broadcast(&requests_consumer);
  pthread_exit(NULL);
}



main(int argc, char **argv)
{int i, j;
 pthread_t tid;

 SEM_INIT(&barrier, NULL,  0);
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

 if (argc >= 2) GET_DELAY     = atoi(argv[1]);
 if (argc >= 3) PROCESS_DELAY = atoi(argv[2]);
 if (argc >= 4) N_PROD        = atoi(argv[3]);
 if (argc >= 5) N_CONS        = atoi(argv[4]);
 printf("GET_DELAY (ms) = %d PROCESS_DELAY (ms) = %d N_PROD = %d N_CONS = %d\n",
        GET_DELAY, PROCESS_DELAY, N_PROD, N_CONS);
 PTHREAD_CREATE(&tid, &attr, killer, (void *) 60);

 for (j=0; j < 3; j++)
   {printf("Starting consumers. List length: %d Produced: %d Consumed %d.\n",
           length, n_produced, n_processed);
    for (i=0; i<N_CONS; i++)
      PTHREAD_CREATE(&tid, &attr, consumer, NULL);
    pthread_mutex_lock(&requests_lock);
```

```
    while (length != 0)
      pthread_cond_wait(&requests_producer, &requests_lock);

    printf("Starting producers. List length: %d Produced: %d Consumed %d.\n",
           length, n_produced, n_processed);
    pthread_mutex_unlock(&requests_lock);
    for (i=0; i<N_PROD; i++)
      PTHREAD_CREATE(&tid, &attr, producer, NULL);

    PTHREAD_CREATE(&tid, &attr, stopper, NULL);
    thread_single_barrier(&barrier, N_PROD+N_CONS);
    stop = FALSE;
    printf("All exited. List length: %d Produced: %d Consumed %d.\n",
           length, n_produced, n_processed);
    sleep(4);
   }
 printf("Done\n");
 exit(0);
}
```

Sockets: Client Side



Threads and Sockets: client.c

```
 /*                client.c                */

/*
  The client side simply sends a hugh number of requests to the
  chosen server as fast as requested, spawing one thread to do the
  sending and another to receive the replies. You may request multiple
  pairs of senders & receivers each with their own socket. The
  senders & receivers run asynchronously intentionally, thus replies
  may be received out-of-order.

  The client will try the requested port first, then the next higher
  port and keep going (in case the server is actually on another port).

  Nothing interesting here.


cc -o client client.c -L. -R. -g -lpthread -pthread_extensions -lposix4 -lnsl -
lsocket
*/


#define _POSIX_C_SOURCE 199506L
#define __EXTENSIONS__

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/resource.h>
#include "thread_extensions.h"


/* Constants etc. rlimit set to MAX_FDS in main() */
#define                 MESSAGE_SIZE 70
#define                 MAX_FDS  1000
```

```
/* 127.0.0.1 is the loopback address.  You may change it. */
const char              *SERV_HOST_ADDR =   "127.0.0.1";
int                     REQUESTED_TCP_PORT = 6500, ACTUAL_TCP_PORT;
int                                MAX_PORT_RANGE = 100;
int                     N_REQUESTS = 10000;
int                     SLEEP = 0, SPIN= 0, N_CONSUMERS=10;
int                     PID;
thread_single_barrier_t client_barrier;
const char              *END_CLIENT_MSG =
  "End\0                                                                            ";


/* A bunch of simple helper functions */

void set_up_signal_mask() {
 sigset_t               s, s1, s2;
 static struct          sigaction action;

/* Block everything first (in particular SIGPIPE) */
 sigfillset(&s);
 pthread_sigmask(SIG_BLOCK, &s, &s2);
 sigemptyset(&s);

 sigaddset(&s, SIGINT);                 /* For ^C */
 pthread_sigmask(SIG_UNBLOCK, &s, &s1);
}



/* Send out N_REQUESTS requests to server */
void *send_requests(void *arg) {
  int            i, j, sockfd = (int) arg, err;
  char           obuf[MESSAGE_SIZE], ibuf[MESSAGE_SIZE];
  char           *name = thread_name(pthread_self());

  set_up_signal_mask();
  printf("Client_%d[%s] \tSending %d requests on fd#%d...\n",
      PID, name, N_REQUESTS, sockfd);
  for (j=0; j<N_REQUESTS; j++) {
    sprintf(obuf, "Client_%d[%s] Request: %d", PID, name, j);
    err=write(sockfd, obuf, MESSAGE_SIZE);
    if (err == -1) {
      printf("Write failed from Client_%d[%s]\n", PID, name);
      perror("");
      pthread_exit(NULL);
    }
    DEBUG(printf("Client_%d[%s] sent: '%s'\n", PID, name, obuf));
    delay(SLEEP, SPIN);
```

```
    if ((j%1000 == 0) && ! (j==0))
      printf("Client_%d[%s] \tSent %d segments on fd#%d.\n",
          PID, name, j, sockfd);
  }
  err=write(sockfd, END_CLIENT_MSG, MESSAGE_SIZE);
  if (err == -1) {
    printf("Write END failed from Client_%d[%s]\n", PID, name);
    perror("");
    thread_single_barrier_post(&client_barrier);
    pthread_exit(NULL);
  }
  DEBUG(printf("Client_%d[%s] sent: '%s'\n", PID, name, END_CLIENT_MSG));
  printf("Client_%d[%s] \tDone sending on fd#%d.\n", PID, name, sockfd);
}


/* Listen to segments until "End" */
void *receive_replies(void *arg) {
  int                    i, j, sockfd = (int) arg;
  struct sockaddr_in     serv_addr;
  char                   obuf[MESSAGE_SIZE], ibuf[MESSAGE_SIZE];
  char                   *name = thread_name(pthread_self());

  set_up_signal_mask();
  printf("Client_%d[%s] \tReceiving segments on fd#%d...\n",
      PID, name, sockfd);

  for (j=1; TRUE; j++)
    {int         nbytes;

    nbytes = read(sockfd, ibuf, MESSAGE_SIZE);
    if (nbytes != MESSAGE_SIZE) {
      printf("Read failed from Client_%d[%s]\n", PID, name);
      if (nbytes < 0) perror("");
      thread_single_barrier_post(&client_barrier);
      pthread_exit(NULL);
    }
    DEBUG(printf("Client_%d[%s] received: '%s'\n", PID, name, ibuf));

    if (strncmp(ibuf, END_CLIENT_MSG, 3) == 0) break;
    if ((j%1000 == 0) && ! (j==0))
      printf("Client_%d[%s] \tReceived %d segments on fd#%d.\n",
             PID, name, j, sockfd);
    }

  close(sockfd);
  if (j < N_REQUESTS)
    printf("Client_%d[%s] \tDone receiving on fd#%d.\
Server dropped %d requests!\n",
```

```
      PID, name, sockfd, N_REQUESTS-j);
  else
    printf("Client_%d[%s] \tDone receiving on fd#%d.\n", PID, name, sockfd);
  thread_single_barrier_post(&client_barrier);
}


int open_a_socket() {
  struct sockaddr_in     serv_addr;
  int sockfd, j;

  for (j=REQUESTED_TCP_PORT; j<REQUESTED_TCP_PORT+MAX_PORT_RANGE;j++) {
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
      {perror("clientsoc: can't open stream socket "); abort();}
    memset((char *) &serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(j);
    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
      {
     DEBUG(printf("Can't connect to server on port %d\n", j));
     close(sockfd);
      }
    else
      break;
    if (j==REQUESTED_TCP_PORT+MAX_PORT_RANGE-1) { /* Maybe server was down? */
      printf("Maybe server was down? Starting again...\n");
      j=REQUESTED_TCP_PORT-1;
      sleep(5);/* Don't rush. Let the server restart */
    }
  }
  ACTUAL_TCP_PORT=j;
  return(sockfd);
}


main(int argc, char **argv) {
  int                    i, j, sockfd, N_SOCKETS = 1;
  pthread_attr_t         attr;
  pthread_t              tid;
  struct rlimit          r;


  PID = getpid();
  if (argc > 1) REQUESTED_TCP_PORT = atoi(argv[1]);
  if (argc > 2) SLEEP = atoi(argv[2]);
  if (argc > 3) SPIN = atoi(argv[3]);
  if (argc > 4) N_SOCKETS = atoi(argv[4]);
  if (argc > 5) N_REQUESTS = atoi(argv[5]);
```

```
  if (N_SOCKETS > MAX_FDS)
      printf("No more than %d connections allowed!\n", MAX_FDS), abort();
  printf("Client_%d(PORT %d SLEEP %dms SPIN %dus N_SOCKETS %d \
N_REQUESTS %d)\n",
      PID, REQUESTED_TCP_PORT, SLEEP, SPIN, N_SOCKETS, N_REQUESTS);



  getrlimit(RLIMIT_NOFILE, &r);
  r.rlim_cur=MAX_FDS+10;
  setrlimit(RLIMIT_NOFILE, &r);
  /*  printf("Current limit: %d Max limit: %d\n", r.rlim_cur, r.rlim_max); */
  set_up_signal_mask();


  PTHREAD_ATTR_INIT(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  thread_single_barrier_init(&client_barrier, N_SOCKETS);

  iobench_start();

  for (i=0; i < N_SOCKETS; i++) {
    sockfd=open_a_socket();
    printf("Connected to server on port %d\n", ACTUAL_TCP_PORT);
    PTHREAD_CREATE(&tid, &attr, send_requests, (void *) sockfd);
    PTHREAD_CREATE(&tid, &attr, receive_replies, (void *) sockfd);
  }

  thread_single_barrier_wait(&client_barrier);
  iobench_end();
  iobench_report();
  exit(NULL);
}
```

T&S: No Threads: server.c

```
/*                server_simple.c                */

/*

  A simple server program.  It sets up a TCP port for the client
  program to connect to.

  This version is unthreaded and very slow if there is any I/O delay
  simulated.

  On most errors this programs just aborts.

*/



/*
cc -o serve_simpler server_simple.c -L. -R. -g -lpthread  -lthread_extensions -
lposix4 -lnsl -lsocket
*/

#define _POSIX_C_SOURCE 199506L
/* Note that you need to define  __EXTENSIONS__ when using sockets. */
#define __EXTENSIONS__



#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>
#include <limits.h>
#include <errno.h>
#include "thread_extensions.h"

/* Constants etc. */
```

```
#define                 MESSAGE_SIZE 70


const int               MAX_FDS = OPEN_MAX-10;          /* Solaris constant */
 /* String sent by & to client when done */
const char              *END_CLIENT_MSG =
  "End\0                                                         ";




/* The structures for clients, and requests */

typedef struct client_struct {

  int                   socket;                /* socket fd */
  char                  name[30];              /* "Client_123[T@23]" */
  int                   outstanding_requests;  /* No. unanswered */

  pthread_mutex_t       lock;
  pthread_cond_t        cv;
} client_t;

typedef struct request_struct {
  client_t              *client;
  char                  data[MESSAGE_SIZE]; /* "Client_7[T@4] Request: 4" */
  struct request_struct *next;
} request_t;




/* Optional command line arguments (+ defaults) */
int                     SLEEP = 0, SPIN= 0, N_CONSUMERS=10;
int                        REQUESTED_TCP_PORT = 6500, ACTUAL_TCP_PORT,
                           MAX_PORT_RANGE=100;
int                     STOPPER_SLEEP=-1, KILLER_SLEEP=-1;




/* Other useful bits */
int                     PID;
pthread_attr_t          attr;
int                     port_socket;
int                     n_replies=0;
```

```
/* A List of REQUESTS with protection and related varaibles. */
request_t               *requests =     NULL;
int                     length =        0;
pthread_mutex_t         requests_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t          producer_cv =   PTHREAD_COND_INITIALIZER;
pthread_cond_t          consumer_cv =   PTHREAD_COND_INITIALIZER;



/* Some function declarations */

client_t *new_client(int);
void *producer(void *);
request_t *get_request(client_t *);
void *process_request(request_t *);


/* A bunch of simple helper functions */

void set_up_signal_mask() {
 sigset_t               s, s1, s2;
 static struct          sigaction action;

/* Block everything first (in particular SIGPIPE) */
 sigfillset(&s);
 pthread_sigmask(SIG_BLOCK, &s, &s2);
 sigemptyset(&s);
 sigaddset(&s, SIGUSR1);                /* For stopper() */
 sigaddset(&s, SIGINT);                 /* For ^C */
 pthread_sigmask(SIG_UNBLOCK, &s, &s1);
}


/* Keep track of the current number of clients for stats & fd limit */
int client_max_count = 0;

int count_clients(int i) {
  static pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;
  static int             client_count = 0;
  int j;
```

```
  pthread_mutex_lock(&counter_lock);
  client_count += i;
  if (client_count > client_max_count)
    client_max_count = client_count;
  j = client_count;
  pthread_mutex_unlock(&counter_lock);
  return(j);
}
```

```
int count_requests(int i) {
  static int count=0;
  static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 i = count;
 pthread_mutex_unlock(&count_lock);
 return(i);                           /* Can't return count */
}

void print_requests()
{request_t              *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->client->socket);
    r = r->next;
  }
}


/* Synchronize on the number of OUTSTANDING REQUESTS per client. */

void wait_for_requests_to_zero(client_t *client) {
```

```
}

int decrement_requests_for(client_t *client) {
  int i;








  return(i);
}

void increment_requests_for(client_t *client) {




}


/* Build a new REQUEST */

request_t *new_request(client_t *client, char *ibuf) {
  request_t             *request;

  pthread_t tid = pthread_self();
  DEBUG(printf("Server_%d[%s] get_request() received: \"%s\"\n",
               PID, thread_name(tid), ibuf));
  request = (request_t *) malloc(sizeof(request_t));
  request->client = client;
  strcpy(request->data, ibuf);
  count_requests(1);









  return(request);
}


int delete_request(request_t *request) {
  int i;
```

```
  free(request);
  return(i);
}



/*
  Send "End" and close the socket.


*/

void close_client(client_t *client) {
  int j;

  DEBUG(printf("Closing socket %s...\n", client->name));

  pthread_mutex_lock(&client->lock);
  if (client->socket != -1) { /* Ensure only close() once. */
    write(client->socket, END_CLIENT_MSG, MESSAGE_SIZE);   /* Ignore errors */
    close(client->socket);
    client->socket = -1;


    j = count_clients(-1);


    DEBUG(printf("Closed %s. (%d active clients)\n",
                 client->name, count_clients(0)));
  }
  pthread_mutex_unlock(&client->lock);
}




void write_msg(client_t *client, char *obuf) {
  int                    nbytes, socket, active;
  char                   *cname;
  char                   *t_name = thread_name(pthread_self());

  DEBUG(printf("Server_%d[%s] write_msg() sending: \"%s\"\n",
               PID, t_name, obuf));
  pthread_mutex_lock(&client->lock);
  socket = client->socket;
  cname  = client->name;

  pthread_mutex_unlock(&client->lock);
```

```
  retry: nbytes = write(socket, obuf, MESSAGE_SIZE);

  if (nbytes == MESSAGE_SIZE) {
    return;
  }

  if (nbytes > -1) {
    printf("Only wrote %d bytes to %s. (Impossible?)", nbytes, cname);
    abort();
  }

  if (errno == EINTR) {
    printf("EINTR while trying to write to %s on fd#%d. retrying...\n",
           cname, socket);
    goto retry;
  }

  if ((errno == EPIPE) || (errno == EBADF)) {
    printf("Failed trying to write to %s. \n", cname);
    perror("Closing socket");



    close_client(client);
    return;
  }
  else
    perror("Unknown bug (Impossible?)"), abort();
}



/* Manipulate the requests list. Caller must own requests_lock. */
request_t *remove_request() {
  request_t *request;

  request = requests;
  requests = requests->next;
  length--;
  return(request);
}

void add_request(request_t *request) {
  request->next = requests;              /* Insert at head of list */
  requests = request;
  length++;
}
```

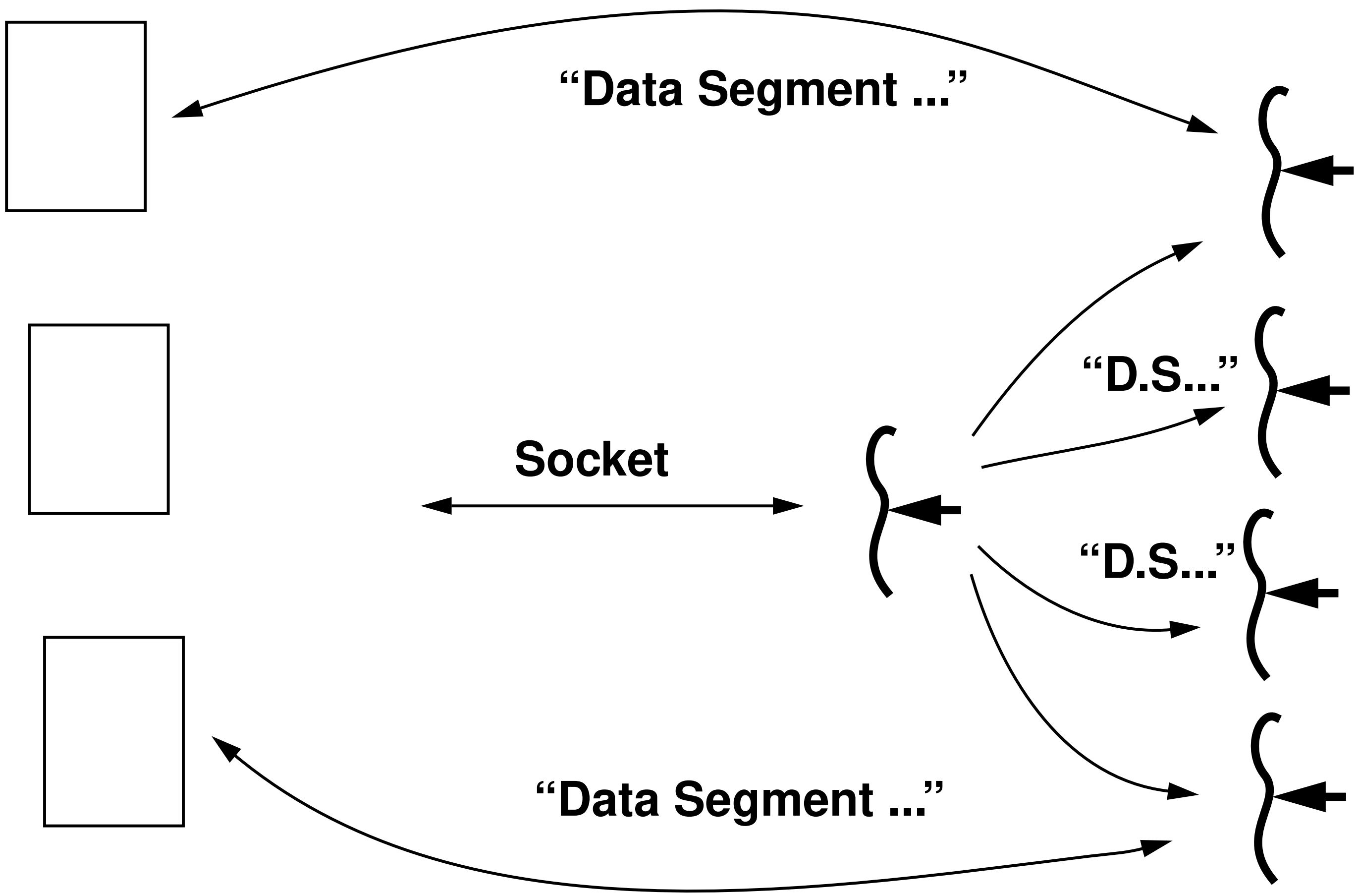
```
int open_port() {
  struct sockaddr_in    cli_addr, serv_addr;
  int                   port_socket, clilen, i;

  /* Open a new port. Try to get requested port. */
  if((port_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    perror("Can't create new socket"), abort();

  for (i=REQUESTED_TCP_PORT; i< REQUESTED_TCP_PORT+MAX_PORT_RANGE; i++) {
    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(i);
    if (bind(port_socket, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
      {DEBUG(printf("Can't bind to port %d\n", i));}
    else {
      printf("Server bound to port: %d\n", i);
      ACTUAL_TCP_PORT=i;
      listen(port_socket, 5);
      return(port_socket);
    }
  }
  printf("Could not find an available port from %d to %d",
         REQUESTED_TCP_PORT, REQUESTED_TCP_PORT+MAX_PORT_RANGE);
  abort();
}



client_t *accept_a_connection(int port_socket) {
  int                   new_socket, clilen;
  struct sockaddr_in    cli_addr;

  clilen = sizeof(cli_addr);
  new_socket = accept(port_socket, (struct sockaddr *) &cli_addr, &clilen);
  if (new_socket < 0)
    perror("Accept failed. (Impossible?)"), abort();

  return(new_client(new_socket));
}


void *reporter(void *arg) {

  while (TRUE) {
    sleep(30);
    printf("Server up on port %d. Processed %d requests.\n",
```

```
          ACTUAL_TCP_PORT, count_requests(0));
  }
}


int startup_initialize() {
  int              i;

  set_up_signal_mask();

  PTHREAD_ATTR_INIT(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

}


void *acceptor(void *arg) {
  client_t *client;
  pthread_t tid;
  int port_socket = (int) arg;

  while (TRUE) {
    client = accept_a_connection(port_socket);
    PTHREAD_CREATE(&tid, &attr, producer, (void *) client);
  }
}



client_t *new_client(int socket) {
  client_t *client = (client_t *) malloc(sizeof(client_t));

  client->socket = socket;

  client->outstanding_requests=0;
  count_clients(1);
  DEBUG(printf("New Client fd#%d. (%d active clients)\n",
               socket, count_clients(0)));
  return(client);
}

void *producer(void *arg) {
  request_t *request;
  client_t *client = (client_t *) arg;

  DEBUG(printf("Producer starting.\n"));

  while (TRUE) {
```

```
    request = get_request(client);
    if (request == NULL)  {
      close_client(client);
      return;
    }
    process_request(request);
  }
}

request_t *get_request(client_t *client) {
  request_t       *request;
  char            ibuf[MESSAGE_SIZE];
  int             nbytes;

  nbytes = read(client->socket, ibuf, MESSAGE_SIZE);
  if (nbytes != MESSAGE_SIZE) {printf("Read failed!\n"); abort();}
  if (strncmp(ibuf, END_CLIENT_MSG, 3) == 0) return(NULL);

  request = new_request(client, ibuf);
  return(request);
}



/* Send a reply back to client */
void *process_request(request_t *request) {
  client_t              *client = request->client;
  char                  *cname;
  char                  obuf[MESSAGE_SIZE];
  char                  *t_name = thread_name(pthread_self());
  int                   i, nbytes, finishing, socket;

  DEBUG(printf("Server_%d[%s] processing: \"%s\"\n",
               PID, t_name, request->data));

  delay(SLEEP, SPIN);
  sprintf(obuf, "Server_%d[%s] reply %d to %s",
          PID, t_name, n_replies++, request->data);
  write_msg(client, obuf);
  delete_request(request);
}

main(int argc, char **argv) {
  int                   i;
  pthread_t tid;

  PID = getpid();

  if (argc > 1) REQUESTED_TCP_PORT = atoi(argv[1]);
```

```
  if (argc > 2) SLEEP = atoi(argv[2]);
  if (argc > 3) SPIN = atoi(argv[3]);
  if (argc > 4) N_CONSUMERS = atoi(argv[4]);
  if (argc > 5) STOPPER_SLEEP = atoi(argv[5]);
  if (argc > 6) KILLER_SLEEP = atoi(argv[6]);
  printf("Server_%d(TCP_PORT %d SLEEP %dms SPIN %dus N_CONSUMERS \n\
%d STOPPER %ds KILLER %ds)\n",
         PID, REQUESTED_TCP_PORT, SLEEP, SPIN, N_CONSUMERS,
         STOPPER_SLEEP, KILLER_SLEEP);

  startup_initialize();

  PTHREAD_CREATE(&tid, &attr, reporter, NULL);

  port_socket = open_port();
  if (KILLER_SLEEP > 0)
    PTHREAD_CREATE(&tid, &attr, killer, (void *) KILLER_SLEEP);

  acceptor((void *)port_socket);
}
```

Master/Slave (2) “Thread per Request”



Worker thread exits after serving the one request.

Exercise Objective

Edit the code in `server.c` and change it so that instead of processing every request as it comes in, the incoming requests are put onto a queue. Then create 10 consumer threads which execute the classic Producer/Consumer code to remove those requests and send replies back (as in the existing code).

Extra: Add a method of stopping the server, such as in `stop_queue_solution.c`.

Hint: Socket connections take ~5 minutes to time-out if your program crashes (e.g., you ^C it). You can run the code with different sockets:

```
% server 6501&
% client 6501
```

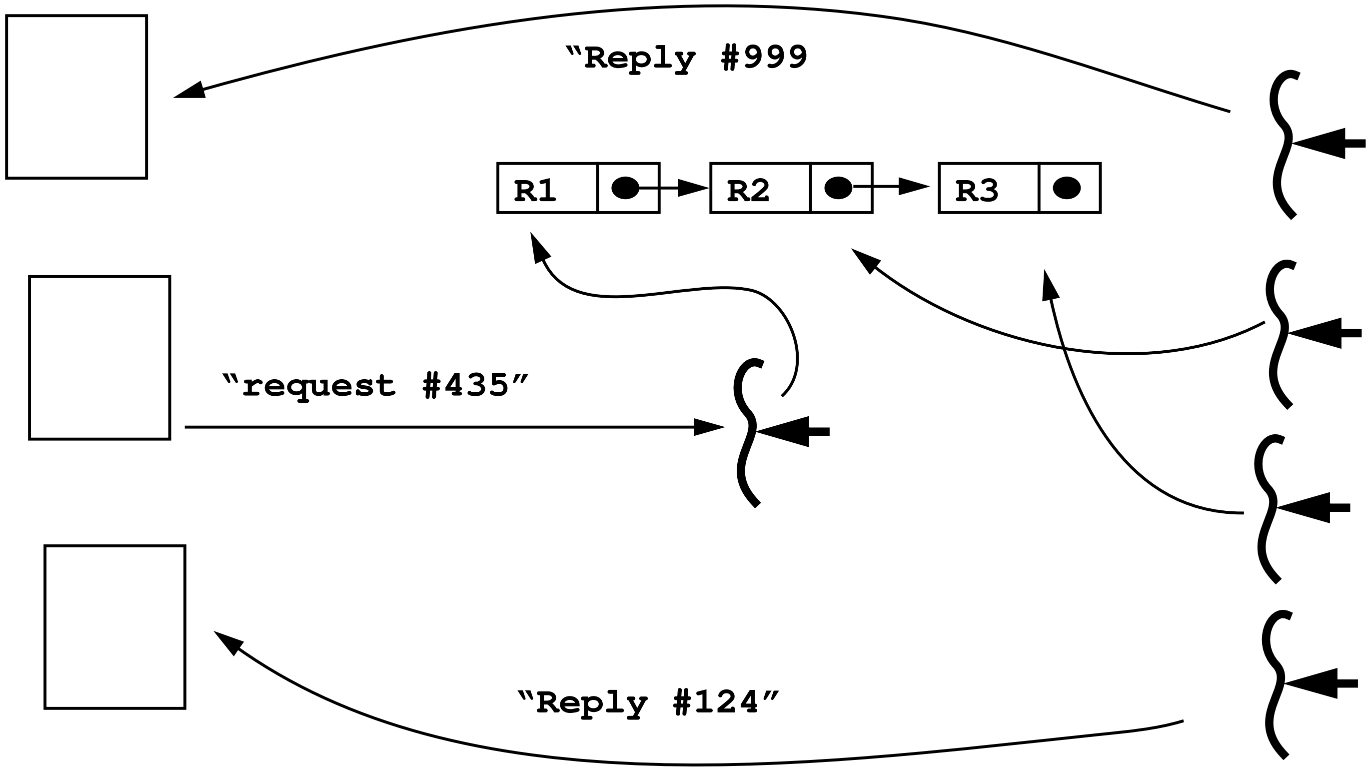
Make sure that you send replies to *all* the requests. The client will tell you if it didn't get all the replies:

```
Client[T@5] Done recving. Server dropped 85 requests!
```

My solution is in `server_pc.c`. Many minor variations of the same basic code are possible.

Wild Guess at Time to Completion:
30 minutes think/design time.
2 hours programming/debugging

Workpile (Producer/Consumer)



Worker threads take requests from queue, execute them, then go back for more requests.

T&S: Producer/Consumer: server_pc.c

```
/*                 server_pc.c                */
/* 4567890 234567890 234567890 234567890 234567890 234567890 234567890 2345678*/
/*

  A simple server program.  It sets up a TCP port for the client
  program to connect to.

  This version is really just a producer/consumer program that happens
  to run across a socket.

  This version checks to ensure all replies have been sent out before
  closing the socket.

  On most errors this programs just aborts.


*/




#define _POSIX_C_SOURCE 199506L
/* Note that you need to define  __EXTENSIONS__ when using sockets. */
#define __EXTENSIONS__




#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>
#include <limits.h>
#include <errno.h>
#include "thread_extensions.h"

/* Constants etc. */
#define                 MESSAGE_SIZE 70
```

```
const int               MAX_FDS = OPEN_MAX-10;          /* Solaris constant */
 /* String sent by & to client when done */
const char              *END_CLIENT_MSG =
  "End\0                                                                      ";



/* The structures for clients, and requests */

typedef struct client_struct {

  int                   socket;                 /* socket fd */
  char                  name[30];               /* "Client_123[T@23]" */
  int                   outstanding_requests;   /* No. unanswered */

  pthread_mutex_t       lock;
  pthread_cond_t        cv;
} client_t;

typedef struct request_struct {
  client_t              *client;
  char                  data[MESSAGE_SIZE]; /* "Client_7[T@4] Request: 4" */
  struct request_struct *next;
} request_t;



/* Optional command line arguments (+ defaults) */
int                     SLEEP = 0, SPIN= 0, N_CONSUMERS=10;
int                        REQUESTED_TCP_PORT = 6500, ACTUAL_TCP_PORT,
                           MAX_PORT_RANGE=100;
int                     STOPPER_SLEEP=-1, KILLER_SLEEP=-1;



/* Other useful bits */
int                     PID;
pthread_attr_t          attr;
int                     port_socket;
int                     n_replies=0;
```

```
/* A List of REQUESTS with protection and related varaibles. */
request_t               *requests =     NULL;
int                     length =        0;
pthread_mutex_t         requests_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t          producer_cv =   PTHREAD_COND_INITIALIZER;
pthread_cond_t          consumer_cv =   PTHREAD_COND_INITIALIZER;



/* Some function declarations */

client_t *new_client(int);
void *producer(void *);
request_t *get_request(client_t *);
void *process_request(request_t *);


/* A bunch of simple helper functions */

void set_up_signal_mask() {
 sigset_t               s, s1, s2;
 static struct          sigaction action;

/* Block everything first (in particular SIGPIPE) */
 sigfillset(&s);
 pthread_sigmask(SIG_BLOCK, &s, &s2);
 sigemptyset(&s);

 sigaddset(&s, SIGINT);                 /* For ^C */
 pthread_sigmask(SIG_UNBLOCK, &s, &s1);
}


/* Keep track of the current number of clients for stats & fd limit */
int client_max_count = 0;

int count_clients(int i) {
  static pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;
  static int             client_count = 0;
  int j;

  pthread_mutex_lock(&counter_lock);
```

```
  client_count += i;
  if (client_count > client_max_count)
    client_max_count = client_count;
  j = client_count;
  pthread_mutex_unlock(&counter_lock);
  return(j);
}
```

```
int count_requests(int i) {
  static int count=0;
  static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 i = count;
 pthread_mutex_unlock(&count_lock);
 return(i);                           /* Can't return count */
}

void print_requests()
{request_t              *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->client->socket);
    r = r->next;
  }
}


/* Synchronize on the number of OUTSTANDING REQUESTS per client. */

void wait_for_requests_to_zero(client_t *client) {
  pthread_mutex_lock(&client->lock);
  while (client->outstanding_requests > 0)
    pthread_cond_wait(&client->cv, &client->lock);
  pthread_mutex_unlock(&client->lock);
}
```

```
int decrement_requests_for(client_t *client) {
  int i;

  pthread_mutex_lock(&client->lock);
  client->outstanding_requests--;
  i = client->outstanding_requests;
  if (client->outstanding_requests == 0)
    pthread_cond_signal(&client->cv);
  pthread_mutex_unlock(&client->lock);
  return(i);
}

void increment_requests_for(client_t *client) {
  pthread_mutex_lock(&client->lock);
  client->outstanding_requests++;
  pthread_mutex_unlock(&client->lock);
}


/* Build a new REQUEST */

request_t *new_request(client_t *client, char *ibuf) {
  request_t             *request;

  pthread_t tid = pthread_self();
  DEBUG(printf("Server_%d[%s] get_request() received: \"%s\"\n",
               PID, thread_name(tid), ibuf));
  request = (request_t *) malloc(sizeof(request_t));
  request->client = client;
  strcpy(request->data, ibuf);
  count_requests(1);
  increment_requests_for(client);








  return(request);
}


int delete_request(request_t *request) {
  int i;

  i=decrement_requests_for(request->client);
  free(request);
```

```
  return(i);
}



/*
  Send "End" and close the socket.


*/

void close_client(client_t *client) {
  int j;

  DEBUG(printf("Closing socket %s...\n", client->name));

  pthread_mutex_lock(&client->lock);
  if (client->socket != -1) { /* Ensure only close() once. */
    write(client->socket, END_CLIENT_MSG, MESSAGE_SIZE);   /* Ignore errors */
    close(client->socket);
    client->socket = -1;


    j = count_clients(-1);
    if (j == 0)
      DEBUG(printf("Max open sockets: %d\n", client_max_count));
    DEBUG(printf("Closed %s. (%d active clients)\n",
                 client->name, count_clients(0)));
  }
  pthread_mutex_unlock(&client->lock);
}




void write_msg(client_t *client, char *obuf) {
  int                    nbytes, socket, active;
  char                   *cname;
  char                   *t_name = thread_name(pthread_self());

  DEBUG(printf("Server_%d[%s] write_msg() sending: \"%s\"\n",
               PID, t_name, obuf));
  pthread_mutex_lock(&client->lock);
  socket = client->socket;
  cname  = client->name;

  pthread_mutex_unlock(&client->lock);
```

```
  retry: nbytes = write(socket, obuf, MESSAGE_SIZE);

  if (nbytes == MESSAGE_SIZE) {
    return;
  }

  if (nbytes > -1) {
    printf("Only wrote %d bytes to %s. (Impossible?)", nbytes, cname);
    abort();
  }

  if (errno == EINTR) {
    printf("EINTR while trying to write to %s on fd#%d. retrying...\n",
           cname, socket);
    goto retry;
  }

  if ((errno == EPIPE) || (errno == EBADF)) {
    printf("Failed trying to write to %s. \n", cname);
    perror("Closing socket");



    close_client(client);
    return;
  }
  else
    perror("Unknown bug (Impossible?)"), abort();
}



/* Manipulate the requests list. Caller must own requests_lock. */
request_t *remove_request() {
  request_t *request;

  request = requests;
  requests = requests->next;
  length--;
  return(request);
}

void add_request(request_t *request) {
  request->next = requests;              /* Insert at head of list */
  requests = request;
  length++;
}
```

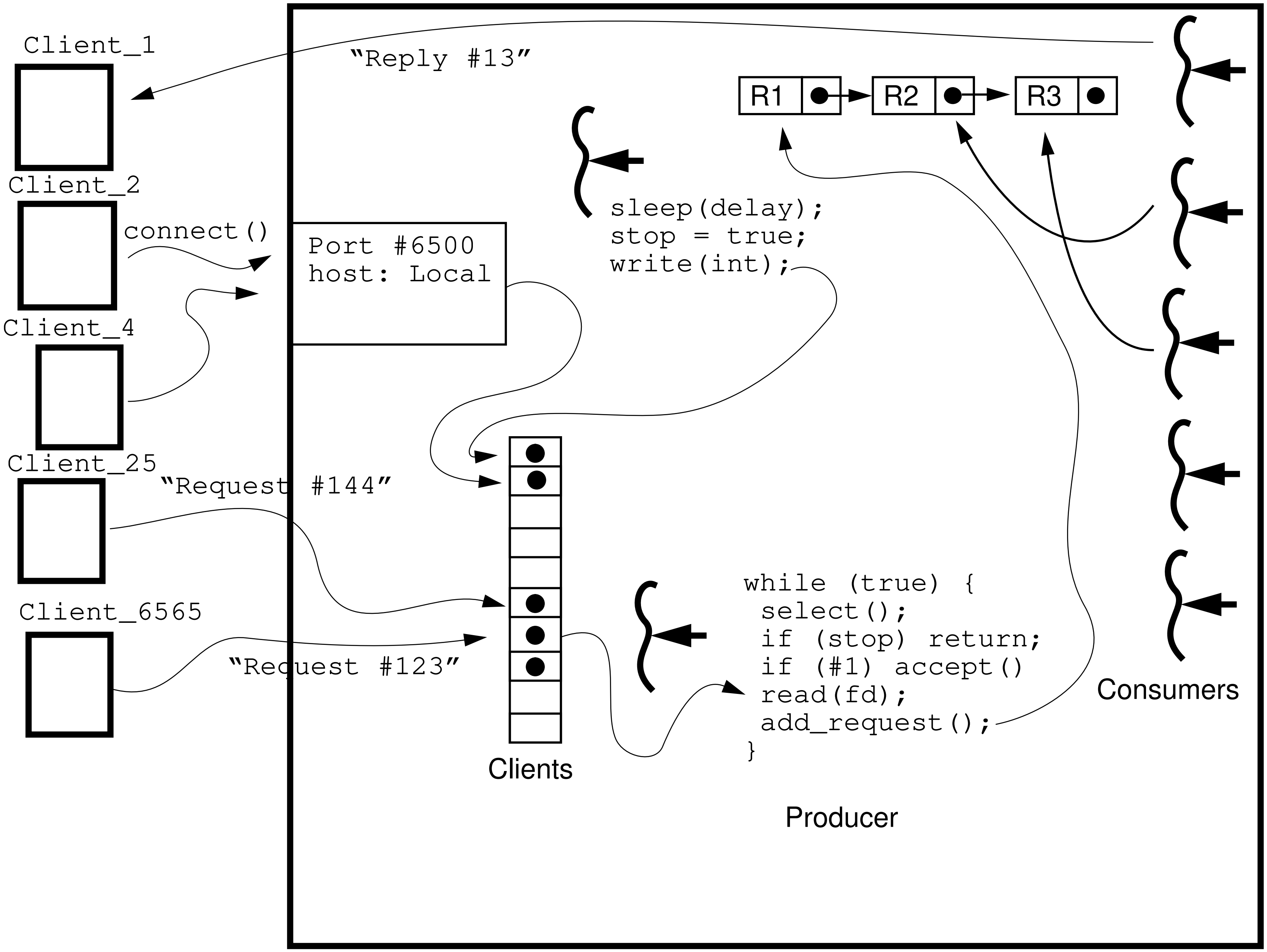
```
int open_port() {
  struct sockaddr_in    cli_addr, serv_addr;
  int                   port_socket, clilen, i;

  /* Open a new port. Try to get requested port. */
  if((port_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    perror("Can't create new socket"), abort();

  for (i=REQUESTED_TCP_PORT; i< REQUESTED_TCP_PORT+MAX_PORT_RANGE; i++) {
    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(i);
    if (bind(port_socket, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
      {DEBUG(printf("Can't bind to port %d\n", i));}
    else {
      printf("Server bound to port: %d\n", i);
      ACTUAL_TCP_PORT=i;
      listen(port_socket, 5);
      return(port_socket);
    }
  }
  printf("Could not find an available port from %d to %d",
         REQUESTED_TCP_PORT, REQUESTED_TCP_PORT+MAX_PORT_RANGE);
  abort();
}



client_t *accept_a_connection(int port_socket) {
  int                   new_socket, clilen;
  struct sockaddr_in    cli_addr;

  clilen = sizeof(cli_addr);
  new_socket = accept(port_socket, (struct sockaddr *) &cli_addr, &clilen);
  if (new_socket < 0)
    perror("Accept failed. (Impossible?)"), abort();

  return(new_client(new_socket));
}


void *reporter(void *arg) {

  while (TRUE) {
    sleep(30);
    printf("Server up on port %d. Processed %d requests.\n",
           ACTUAL_TCP_PORT, count_requests(0));
```

```
  }
}


int startup_initialize() {
  int            i;

  set_up_signal_mask();

  PTHREAD_ATTR_INIT(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

}


void *acceptor(void *arg) {
  client_t *client;
  pthread_t tid;
  int port_socket = (int) arg;

  while (TRUE) {
    client = accept_a_connection(port_socket);
    PTHREAD_CREATE(&tid, &attr, producer, (void *) client);
  }
}



request_t *get_request(client_t *client) {
  request_t       *request;
  char            ibuf[MESSAGE_SIZE];
  int             nbytes;

  nbytes = read(client->socket, ibuf, MESSAGE_SIZE);
  if (nbytes != MESSAGE_SIZE) {printf("Read failed!\n"); abort();}
  if (strncmp(ibuf, END_CLIENT_MSG, 3) == 0) return(NULL);

  request = new_request(client, ibuf);
  return(request);
}

client_t *new_client(int socket) {
  client_t *client = (client_t *) malloc(sizeof(client_t));

  client->socket = socket;

  client->outstanding_requests=0;
  count_clients(1);
```

```
  DEBUG(printf("New Client fd#%d. (%d active clients)\n",
             socket, count_clients(0)));
  return(client);
}

/* Send a reply back to client */
void *process_request(request_t *request) {
  client_t              *client = request->client;
  char                  *cname;
  char                  obuf[MESSAGE_SIZE];
  char                  *t_name = thread_name(pthread_self());
  int                   i, nbytes, finishing, socket;

  DEBUG(printf("Server_%d[%s] processing: \"%s\"\n",
             PID, t_name, request->data));

  delay(SLEEP, SPIN);
  sprintf(obuf, "Server_%d[%s] reply %d to %s",
         PID, t_name, n_replies++, request->data);
  write_msg(client, obuf);
  delete_request(request);
}


void *producer(void *arg) {
  request_t *request;
  client_t *client = (client_t *) arg;

  DEBUG(printf("Producer starting.\n"));

  while (TRUE) {
    request = get_request(client);
    if (request == NULL) {
      wait_for_requests_to_zero(client);
      close_client(client);
      return;
    }
    pthread_mutex_lock(&requests_lock);
    while ((length >= 10))
      pthread_cond_wait(&producer_cv, &requests_lock);
    add_request(request);
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&consumer_cv);
  }
}


void *consumer(void *arg) {
  request_t *request;
```

```
  DEBUG(printf("Consumer starting.\n"));
  set_up_signal_mask();

  while(TRUE) {
    pthread_mutex_lock(&requests_lock);
    while ((length == 0))
      pthread_cond_wait(&consumer_cv, &requests_lock);
    request = remove_request();
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&producer_cv);
    process_request(request);
  }
}



main(int argc, char **argv) {
  int                   i;
  pthread_t tid;

  PID = getpid();

  if (argc > 1) REQUESTED_TCP_PORT = atoi(argv[1]);
  if (argc > 2) SLEEP = atoi(argv[2]);
  if (argc > 3) SPIN = atoi(argv[3]);
  if (argc > 4) N_CONSUMERS = atoi(argv[4]);
  if (argc > 5) STOPPER_SLEEP = atoi(argv[5]);
  if (argc > 6) KILLER_SLEEP = atoi(argv[6]);
  printf("Server_%d(TCP_PORT %d SLEEP %dms SPIN %dus N_CONSUMERS \n\
%d STOPPER %ds KILLER %ds)\n",
         PID, REQUESTED_TCP_PORT, SLEEP, SPIN, N_CONSUMERS,
         STOPPER_SLEEP, KILLER_SLEEP);

  startup_initialize();

  PTHREAD_CREATE(&tid, &attr, reporter, NULL);

  for (i=0; i<N_CONSUMERS; i++) {
    PTHREAD_CREATE(&tid, &attr, consumer, (void *) NULL);
  }

  port_socket = open_port();
  if (KILLER_SLEEP > 0)
    PTHREAD_CREATE(&tid, &attr, killer, (void *) KILLER_SLEEP);

  acceptor((void *)port_socket);
}
```

Producer/Consumer w/`select()`



Robust Server Using `select()`: server_select.c

```
/*                 server_select.c                  */
/* 4567890 234567890 234567890 234567890 234567890 234567890 234567890 2345678*/
/*

  This version of the server is designed to be robust under client
  failure and implements a complete shutdown, first closing the port,
  then shutting down the producer thread, finally closing the Interruptor
  pipe. The consumers do not exit, but they continue and empty the queue,
  then they sit around waiting for the server to restart. (This is
  trickier than it may seem!)

  This uses a single producer thread which calls select() on all clients
  and on the port socket. When a new connect comes it, it gets accepted
  and the new socket added to the select() list. When a new request
  comes in, it is read and returned as a request_t.


  Any IO failures are handled by printing out an error message, closing the
  socket in question, then ignoring it. The Client program is sent an “End”
  message then, and left to its own devices to deal with the fact that the
  socket has been closed early.

  Shutdown is accomplished by setting “stop” variables and then ensuring
  that the producer, which may be blocked on select() is woken up by doing a
  sending data on a special “Interruptor” pipe. Both read() and accept() are
  assumed never to block because they are only called after select()
  returns. Calls to write() are also assumed never to block (although an
  uncooperative client could cause this by never reading replies and letting
  the socket fill up (~64k!)).

*/

/*                                              replies
                                     <--------------------------
                                                              |
                   _______     _______     _______            |
   Requests       |_______|-->|_______|-->|_______|           |
                                  ^            |              |
                                  |            |              |
          requests __________     |            |              |
Client1 -------->| PRODUCER |     |            ----------> | CONSUMER |
Client2 -------->| select() |     |                        |          |
```

```
Client3 ------->|  read()  |---                       |          |
Client4 =======>|  accept()|<-------------------       |__________|
       connect  |__________|                   |
                            Interruptor        |
                            Pipe               |
                                               |        __________
                 __________                    |       | CONSUMER |
                | STOPPER  |                   |       |          |
                |          |                   |       |          |
                |stop vars |--------------------       |__________|
                |__________|           "STOP"
                                                             ...

*/



#define _POSIX_C_SOURCE 199506L
/* Note that you need to define  __EXTENSIONS__ when using sockets. */
#define __EXTENSIONS__



#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>
#include <limits.h>
#include <errno.h>
#include <sys/resource.h>
#include "thread_extensions.h"

/* Constants etc. rlimit set to MAX_FDS in main() */
#define                MESSAGE_SIZE 70
#define                READ 0
#define                WRITE 1
#define                MAX_FDS  1024
int                    ACTUAL_MAX_FDS=MAX_FDS;

 /* String sent by & to client when done */
const char             *END_CLIENT_MSG =
  "End\0                                                                    ";
/* String used by Stopper thread to wake up Producer */
const char             *STOP_MSG =
  "STOP\0                                                                   ";
```

```
/* The structures for clients, and requests */

typedef struct client_struct {
  int                    active;                /* Is the socket valid? */
  int                    socket;                /* socket fd */
  char                   name[30];              /* "Client_123[T@23]" */
  int                    outstanding_requests;  /* No. unanswered */
  int                    finishing;             /* "End" received */
  pthread_mutex_t        lock;
  pthread_cond_t         cv;
} client_t;

typedef struct request_struct {
  client_t               *client;
  char                   data[MESSAGE_SIZE]; /* "Client_7[T@4] Request: 4" */
  struct request_struct *next;
} request_t;




/* Optional command line arguments (+ defaults) */
int                      SLEEP = 0, SPIN= 0, N_CONSUMERS=10;
int                      REQUESTED_TCP_PORT = 6500, ACTUAL_TCP_PORT,
                         MAX_PORT_RANGE=100;
int                      STOPPER_SLEEP=-1, KILLER_SLEEP=-1;

/* Array of clients */
client_t                 clients[MAX_FDS];

/* Other useful bits */
int                      PID;
pthread_attr_t           attr;
int                      port_socket;
int                      interrupter_pipe[2];


/* Barriers and variables used for shutting down the server */
thread_single_barrier_t producer_barrier;
thread_single_barrier_t stopper_barrier;
thread_barrier_t         consumer_barrier1, consumer_barrier2;
pthread_t                producer_thread;
int                      stop_producer = FALSE;
int                      stop_consumers= FALSE;


/* A List of REQUESTS with protection and related varaibles. */
request_t                *requests =     NULL;
int                      length =        0;
```

```
pthread_mutex_t          requests_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t           producer_cv =   PTHREAD_COND_INITIALIZER;
pthread_cond_t           consumer_cv =   PTHREAD_COND_INITIALIZER;



/* Some function declarations */

client_t *new_client(int);
void *producer(void *);
request_t *get_request();
void *process_request(request_t *);


/* A bunch of simple helper functions */

void set_up_signal_mask() {
 sigset_t               s, s1, s2;
 static struct          sigaction action;

/* Block everything first (in particular SIGPIPE) */
 sigfillset(&s);
 pthread_sigmask(SIG_BLOCK, &s, &s2);
 sigemptyset(&s);

 sigaddset(&s, SIGINT);                 /* For ^C */
 pthread_sigmask(SIG_UNBLOCK, &s, &s1);
}


/* Keep track of the current number of clients for stats & fd limit */
int client_max_count = 0;

int count_clients(int i) {
  static pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;
  static int             client_count = 0;
  int j;

  pthread_mutex_lock(&counter_lock);
  client_count += i;
  if (client_count > client_max_count)
    client_max_count = client_count;
  j = client_count;
  pthread_mutex_unlock(&counter_lock);
  return(j);
}


/* Used for debugging */
```

```
void dump_client_data() {
  int i;

  for (i=0; i<ACTUAL_MAX_FDS; i++) {
    if (clients[i].active)
      printf("%s on fd#%d\n", clients[i].name, clients[i].socket);
  }
}


int count_requests(int i) {
  static int count=0;
  static pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;

 pthread_mutex_lock(&count_lock);
 count += i;
 i = count;
 pthread_mutex_unlock(&count_lock);
 return(i);                              /* Can't return count */
}

void print_requests()
{request_t              *r = requests;

 while (r != NULL)
   {printf("<Request: %d>\n", r->client->socket);
    r = r->next;
  }
}


/* Synchronize on the number of OUTSTANDING REQUESTS per client. */

void wait_for_requests_to_zero(client_t *client) {
  pthread_mutex_lock(&client->lock);
  while (client->outstanding_requests > 0)
    pthread_cond_wait(&client->cv, &client->lock);
  pthread_mutex_unlock(&client->lock);
}

int decrement_requests_for(client_t *client) {
  int i;

  pthread_mutex_lock(&client->lock);
  client->outstanding_requests--;
  i = client->outstanding_requests;
  if (client->outstanding_requests == 0)
    pthread_cond_signal(&client->cv);
  pthread_mutex_unlock(&client->lock);
```

```
  return(i);
}

void increment_requests_for(client_t *client) {
  pthread_mutex_lock(&client->lock);
  client->outstanding_requests++;
  pthread_mutex_unlock(&client->lock);
}


/* Build a new REQUEST */

request_t *new_request(client_t *client, char *ibuf) {
  request_t             *request;

  pthread_t tid = pthread_self();
  DEBUG(printf("Server_%d[%s] get_request() received: \"%s\"\n",
               PID, thread_name(tid), ibuf));
  request = (request_t *) malloc(sizeof(request_t));
  request->client = client;
  strcpy(request->data, ibuf);
  count_requests(1);
  increment_requests_for(client);

  pthread_mutex_lock(&client->lock);
  if (client->name[0] == ' ') {         /* Hack to get a client name! */
    strncpy(client->name, ibuf, 18);    /* Runs only on first request */
    client->name[18] = 0;
  }
  pthread_mutex_unlock(&client->lock);
  return(request);
}


int delete_request(request_t *request) {
  int i;

  i=decrement_requests_for(request->client);
  free(request);
  return(i);
}



/*
  Send "End" and close the socket.
  It's OK if the socket has already been closed.
  This is only called when outstanding_requests == 0.
*/
```

```
void close_client(client_t *client) {
  int j;

  DEBUG(printf("Closing socket %s...\n", client->name));

  pthread_mutex_lock(&client->lock);
  if (client->socket != -1) {                   /* Ensure only close() once. */
    write(client->socket, END_CLIENT_MSG, MESSAGE_SIZE);   /* Ignore errors */
    close(client->socket);
    client->socket = -1;
    client->active = FALSE;
    client->finishing = FALSE;
    j = count_clients(-1);
    if (j == 0)
      DEBUG(printf("Max open sockets: %d\n", client_max_count));
    DEBUG(printf("Closed %s. (%d active clients)\n",
                 client->name, count_clients(0)));
  }
  pthread_mutex_unlock(&client->lock);
}
```

```
void write_msg(client_t *client, char *obuf) {
  int                    nbytes, socket, active;
  char                   *cname;
  char                   *t_name = thread_name(pthread_self());

  DEBUG(printf("Server_%d[%s] write_msg() sending: \"%s\"\n",
               PID, t_name, obuf));
  pthread_mutex_lock(&client->lock);
  socket = client->socket;
  cname  = client->name;
  active = client-> active;
  pthread_mutex_unlock(&client->lock);

  if (active)  { /* Another consumer got an error & closed socket */
  retry: nbytes = write(socket, obuf, MESSAGE_SIZE);

  if (nbytes == MESSAGE_SIZE) {
    return;
  }

  if (nbytes > -1) {
    printf("Only wrote %d bytes to %s. (Impossible?)", nbytes, cname);
    abort();
```

```
  }

  if (errno == EINTR) {
    printf("EINTR while trying to write to %s on fd#%d. retrying...\n",
           cname, socket);
    goto retry;
  }

  if ((errno == EPIPE) || (errno == EBADF)) {
    printf("Failed trying to write to %s. \n", cname);
    perror("Closing socket");
    pthread_mutex_lock(&client->lock);
    client->active = FALSE;
    pthread_mutex_unlock(&client->lock);
    close_client(client);
    return;
  }
  else
    perror("Unknown bug (Impossible?)"), abort();
  }
}


/* Manipulate the requests list. Caller must own requests_lock. */
request_t *remove_request() {
  request_t *request;

  request = requests;
  requests = requests->next;
  length--;
  return(request);
}

void add_request(request_t *request) {
  request->next = requests;                     /* Insert at head of list */
  requests = request;
  length++;
}


int open_port() {
  struct sockaddr_in    cli_addr, serv_addr;
  int                   port_socket, clilen, i;

  /* Open a new port. Try to get requested port. */
  if((port_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    perror("Can't create new socket"), abort();

  for (i=REQUESTED_TCP_PORT; i< REQUESTED_TCP_PORT+MAX_PORT_RANGE; i++) {
```

```
    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(i);
    if (bind(port_socket, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
      {DEBUG(printf("Can't bind to port %d\n", i));}
    else {
      printf("Server bound to port: %d\n", i);
      ACTUAL_TCP_PORT=i;
      listen(port_socket, 5);
      return(port_socket);
    }
  }
  printf("Could not find an available port from %d to %d",
         REQUESTED_TCP_PORT, REQUESTED_TCP_PORT+MAX_PORT_RANGE);
  abort();
}


/* This will never block. Only called when select() indicates a new client. */
client_t *accept_a_connection(int port_socket) {
  int                   new_socket, clilen;
  struct sockaddr_in    cli_addr;

  clilen = sizeof(cli_addr);
  new_socket = accept(port_socket, (struct sockaddr *) &cli_addr, &clilen);
  if (new_socket < 0)
    perror("Accept failed. (Impossible?)"), abort();

  return(new_client(new_socket));
}


void *reporter(void *arg) {

  while (TRUE) {
    sleep(30);
    printf("Server up on port %d. Processed %d requests. %d currently connected
clients.\n",
           ACTUAL_TCP_PORT, count_requests(0), count_clients(0));
  }
}


int startup_initialize() {
  int           i;
  struct rlimit         r;

  set_up_signal_mask();
```

```
  thread_barrier_init(&consumer_barrier1, N_CONSUMERS+1);
  thread_barrier_init(&consumer_barrier2, N_CONSUMERS+1);
  thread_single_barrier_init(&producer_barrier, 1);
  thread_single_barrier_init(&stopper_barrier, 1);
  PTHREAD_ATTR_INIT(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  getrlimit(RLIMIT_NOFILE, &r);
  r.rlim_cur=r.rlim_max;
  ACTUAL_MAX_FDS = r.rlim_max-10;
  setrlimit(RLIMIT_NOFILE, &r);
  /*  getrlimit(RLIMIT_NOFILE, &r);
      printf("Current limit: %d Max limit: %d\n", r.rlim_cur, r.rlim_max); */


  for (i=0; i<ACTUAL_MAX_FDS; i++) {
    pthread_mutex_init(&clients[i].lock, NULL);
  }
}


int initialize_clients() {
  int i;
  client_t *client;

  count_clients(-count_clients(0)); /* Zero it out */
  for (i=0; i<ACTUAL_MAX_FDS; i++) {
    client = &clients[i];
    pthread_mutex_lock(&client->lock);
    close(client->socket);      /* Could be left open by dirty exit */
    client->socket = -1;
    client->name[0] = ' ';
    client->active = FALSE;
    client->finishing = FALSE;
    client->outstanding_requests = 0;
    pthread_mutex_unlock(&client->lock);
  }

  if (pipe(interrupter_pipe) < 0)
    printf("Can't open Interruptor pipe\n"), abort();

  /* clients[0] is the local socket for announcing new clients. */
  clients[0].socket = interrupter_pipe[READ];
  strcpy(clients[0].name, "Interruptor");
  clients[0].active = TRUE;

  port_socket = open_port();
```

```
  /* clients[1] is the local socket for announcing new clients. */
  clients[1].socket = port_socket;
  strcpy(clients[1].name, "Acceptor");
  clients[1].active = TRUE;


}






client_t *new_client(int socket) {
  int i, c;
  client_t *client;

  for (i=0; i<ACTUAL_MAX_FDS; i++) { /* Find an open slot in array */
    client=&clients[i];
    pthread_mutex_lock(&client->lock);

    if (!client->active & (client->outstanding_requests == 0)) {
      /* If client died dirty, there could still be outstanding requests. */
      client = &clients[i];
      client->socket = socket;
      client->active = TRUE;
      client->outstanding_requests=0;
      client->finishing = FALSE;
      client->name[0] = ` `;         /* Hack! (see new_request()) */
      c =count_clients(1);
      DEBUG(printf("New Client fd#%d. (%d active clients)\n",
                   socket, c));
      pthread_mutex_unlock(&client->lock);
      return(client);
    }

    pthread_mutex_unlock(&client->lock);
  }

  pthread_mutex_unlock(&client->lock);
  write(socket, END_CLIENT_MSG, MESSAGE_SIZE);            /* Ignore errors */
  close(socket);
  printf("Too many clients (max = %d). Ignoring new one.\n", ACTUAL_MAX_FDS-2);
}


/* Wait for any client to send a request & return that request. */
/* Only one thread may call get_request(). */
request_t *get_request() {
  fd_set                readset, errorset;
```

```
int                     i, num_ready, num_now, nbytes, socket;
char                    buf[MESSAGE_SIZE];
request_t               *request;
client_t                *client;
pthread_t               tid = pthread_self();


while (TRUE) {

  /* Reset the bit arrays for select() */
  FD_ZERO(&readset);
  FD_ZERO(&errorset);
  for (i = 0; i < ACTUAL_MAX_FDS; i++) {
    client = &clients[i];
    pthread_mutex_lock(&client->lock);
    if (client->active) {
      FD_SET(client->socket, &readset);
      FD_SET(client->socket, &errorset);
    }
    pthread_mutex_unlock(&client->lock);
  }

  num_ready = select(OPEN_MAX, &readset, 0, &errorset, 0);

  /* Time to exit? */
  pthread_mutex_lock(&requests_lock);
  if (stop_producer) {
    pthread_mutex_unlock(&requests_lock);
    return(NULL);           /* Clean exit */
  }
  pthread_mutex_unlock(&requests_lock);


  if (num_ready < 0) {
    if (errno == EBADF) goto bottom;  /* An fd was closed, retry */
    perror("Error in select(). Impossible"); abort();
  }


  /* Find the socket with data ready */
  for (i = 0; i < ACTUAL_MAX_FDS;  i++) {
    client = &clients[i];
    pthread_mutex_lock(&client->lock);

    if (client->active && FD_ISSET(client->socket, &errorset)) {
      /* Error condition pending on this fd. (What does that mean??) */
      client->active = FALSE;
      pthread_mutex_unlock(&client->lock);
      close_client(client);
```

```
      printf("Error condition on %s, fd#%d from select(). Impossible?\n",
             client->name, client->socket);
      goto bottom;
    }


    if (client->active && FD_ISSET(client->socket, &readset)) {
      /* This is the socket with data ready! */
      socket = client->socket;
      pthread_mutex_unlock(&client->lock);

      if (i == 1) {
        DEBUG(printf("Accepting...\n"));
        accept_a_connection(socket);
        goto bottom;
      }
      nbytes = read(socket, buf, MESSAGE_SIZE);

      if ((nbytes == -1) || (nbytes != MESSAGE_SIZE)) {
        printf("Read only %d bytes on %s!\n", nbytes, client->name);
        if (i == 0) {
          printf("Interrupt pipe failed. Impossible!\n");
          abort();
        }
        pthread_mutex_lock(&client->lock);
        client->active = FALSE;
        pthread_mutex_unlock(&client->lock);
        close_client(client);
        goto bottom;
      }

      if (i == 0) {    /* Should have exited on stop_producer */
        printf("Bogus message on Interruptor. Impossible!\n");
        abort();
      }

      if (nbytes == MESSAGE_SIZE) {/* All is well, return new request_t. */
        DEBUG(printf("Server_%d[%s] get_request() received: \"%s\"\n",
                     PID, thread_name(tid), buf));
        request = new_request(client, buf);
        return(request);
      }
    }
    pthread_mutex_unlock(&client->lock);
  }

  /* select() returned 0 ready, or failed to set readset */
  if (num_ready == 0)
    printf("select() returned 0 ready. Impossible?\n");
```

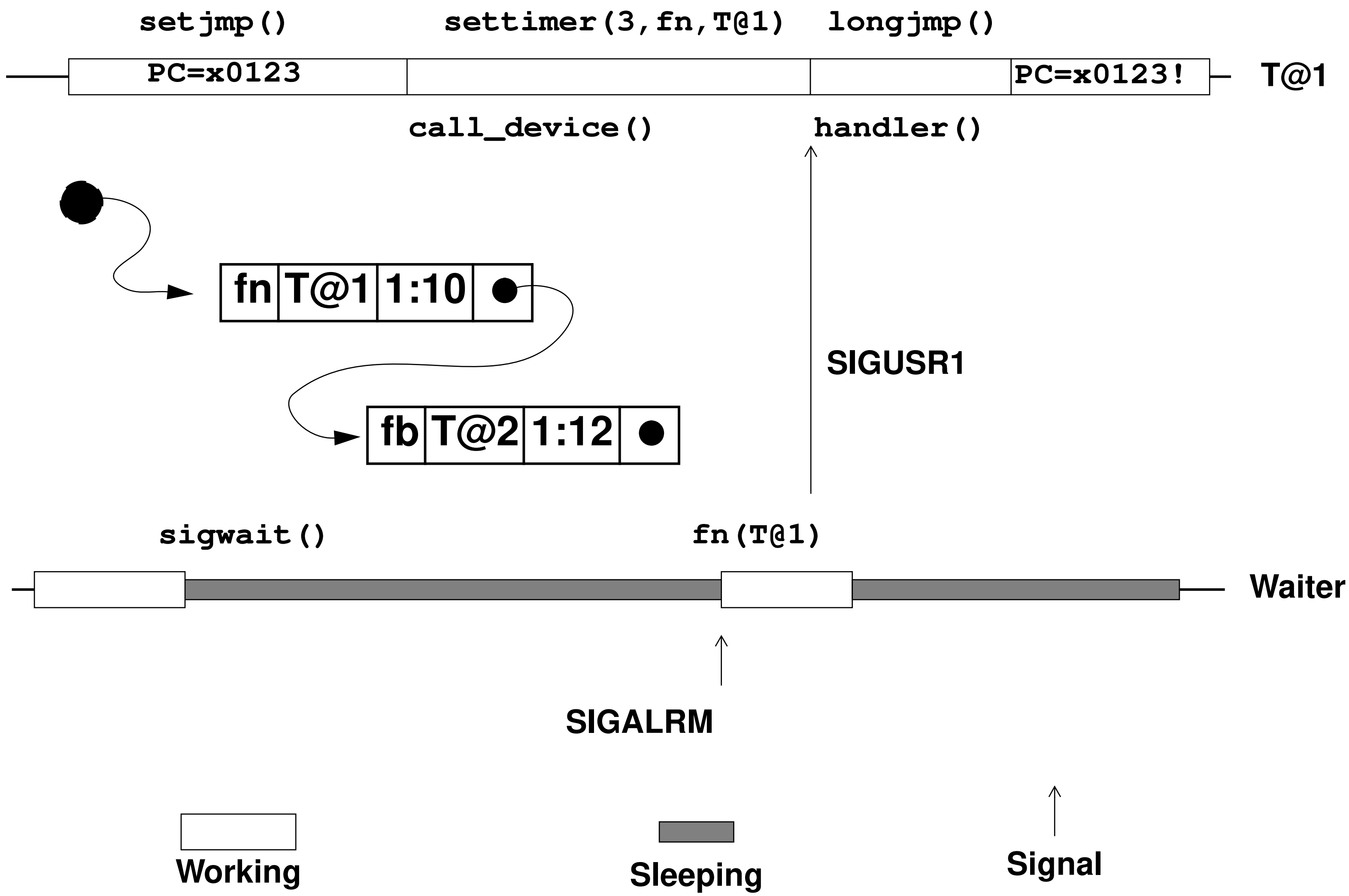
```
    else
      printf("select() returned %d ready (readset not set) Impossible?\n",
             num_ready);
    abort();

  bottom: {}
  }     /* End while(TRUE) */
}

/* Send a reply back to client */
void *process_request(request_t *request) {
  client_t              *client = request->client;
  char                  *cname;
  char                  obuf[MESSAGE_SIZE];
  char                  *t_name = thread_name(pthread_self());
  int                   i, nbytes, finishing, socket;

  DEBUG(printf("Server_%d[%s] processing: \"%s\"\n",
               PID, t_name, request->data));

    pthread_mutex_lock(&client->lock);
    socket = client->socket;
    cname = client->name;
    pthread_mutex_unlock(&client->lock);

    if (strncmp(request->data, END_CLIENT_MSG, 3) == 0) {
      pthread_mutex_lock(&client->lock);
      client->finishing = TRUE;
      pthread_mutex_unlock(&client->lock);
    }
    else {
      delay(SLEEP, SPIN);
      sprintf(obuf, "Server_%d[%s] reply %d to %s",
              PID, t_name, count_requests(1), request->data);
      write_msg(client, obuf);
    }
  i = delete_request(request);
  if (i == 0) {
    pthread_mutex_lock(&client->lock);
    finishing = client->finishing;
    pthread_mutex_unlock(&client->lock);
    if (finishing) close_client(client);
  }
}
void *producer(void *arg) {
  request_t *request;
  int j;

  DEBUG(printf("Producer starting.\n"));
```

```
  set_up_signal_mask();

  while (TRUE) {
    request = get_request();

    if (request == NULL) {
      pthread_mutex_lock(&requests_lock);
      if (!stop_producer) {
        printf("get_request() returned NULL w/o stop. Impossible.");
        abort();
      }
      pthread_mutex_unlock(&requests_lock);
      DEBUG(printf("Producer exiting via get_request().\n"));
      thread_single_barrier_post(&producer_barrier);
      pthread_exit(NULL);
    }


    pthread_mutex_lock(&requests_lock);
    while ((length >= 10) && (!stop_producer))
      pthread_cond_wait(&producer_cv, &requests_lock);
    add_request(request);
    if (stop_producer) break;
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&consumer_cv);
  }


  pthread_mutex_unlock(&requests_lock);
  DEBUG(printf("Producer exiting via wait.\n"));
  thread_single_barrier_post(&producer_barrier);
  pthread_exit(NULL);
}


void *consumer(void *arg) {
  request_t *request;

  DEBUG(printf("Consumer starting.\n"));
  set_up_signal_mask();

  while(TRUE) {
    pthread_mutex_lock(&requests_lock);
    while ((length == 0) && (!stop_consumers))
      pthread_cond_wait(&consumer_cv, &requests_lock);
    if ((length == 0) && (stop_consumers)) {
      /* Empty the queue first, then indicate done. */
      pthread_mutex_unlock(&requests_lock);
      thread_barrier_wait(&consumer_barrier1); /* consumers are stopped */
```

```
      thread_barrier_wait(&consumer_barrier2); /* consumers may continue */
      continue;         /* Continue when everyone is done. */
    }
    request = remove_request();
    pthread_mutex_unlock(&requests_lock);
    pthread_cond_signal(&producer_cv);
    process_request(request);
  }
}



void *stopper(void *arg) {
  pthread_t waker_thread;

  sleep(STOPPER_SLEEP);
  printf("Time to stop!\n");

  /* Wake up producer if blocked in select() wait() */
  pthread_mutex_lock(&requests_lock);
  stop_producer = TRUE;
  pthread_mutex_unlock(&requests_lock);
  pthread_cond_broadcast(&producer_cv);
  /* Wake producer */
  write(interrupter_pipe[WRITE], STOP_MSG, MESSAGE_SIZE);
  thread_single_barrier_wait(&producer_barrier);
  close(interrupter_pipe[WRITE]);


  /* Wake up all CV sleepers and stop them last. Otherwise the producer
   could still add a request after they all stopped. */
  pthread_mutex_lock(&requests_lock);
  stop_consumers=TRUE;
  pthread_mutex_unlock(&requests_lock);
  pthread_cond_broadcast(&consumer_cv);
  thread_barrier_wait(&consumer_barrier1);

  DEBUG(printf("Stopper finished.\n"));
  thread_single_barrier_post(&stopper_barrier);
  pthread_exit(NULL);
}

main(int argc, char **argv) {
  int                  i, j;
  pthread_t            tid;


  PID = getpid();
```

```
  if (argc > 1) REQUESTED_TCP_PORT = atoi(argv[1]);
  if (argc > 2) SLEEP = atoi(argv[2]);
  if (argc > 3) SPIN = atoi(argv[3]);
  if (argc > 4) N_CONSUMERS = atoi(argv[4]);
  if (argc > 5) STOPPER_SLEEP = atoi(argv[5]);
  if (argc > 6) KILLER_SLEEP = atoi(argv[6]);
  printf("Server_%d(TCP_PORT %d SLEEP %dms SPIN %dus N_CONSUMERS \
%d STOPPER %ds KILLER %ds)\n",
         PID, REQUESTED_TCP_PORT, SLEEP, SPIN, N_CONSUMERS,
         STOPPER_SLEEP, KILLER_SLEEP);

  startup_initialize();

  for (i=0; i<N_CONSUMERS; i++) {
    PTHREAD_CREATE(&tid, &attr, consumer, (void *) NULL);
  }

  PTHREAD_CREATE(&tid, &attr, reporter, NULL);

  for (j=0; j<1000; j++) {         /* Start/Stop 1000 times */
    printf("Starting up interation %d.\n", j);
    initialize_clients();

    PTHREAD_CREATE(&producer_thread, &attr, producer, (void *) NULL);
    if (STOPPER_SLEEP > 0)
      PTHREAD_CREATE(&tid, &attr, stopper, (void *) NULL);
    if (KILLER_SLEEP > 0)
      PTHREAD_CREATE(&tid, &attr, killer, (void *) KILLER_SLEEP);

    thread_single_barrier_wait(&stopper_barrier);
    printf("Shutdown successful. %d replies sent.\n\n\n", count_requests(0));

    pthread_mutex_lock(&requests_lock);
    stop_producer=FALSE;
    stop_consumers=FALSE;
    pthread_mutex_unlock(&requests_lock);
    thread_barrier_wait(&consumer_barrier2); /* NOW restart consumers */
  }
  printf("Exiting Server Normally after 1000 iterations.\n");
}
```

Signals and Timers




```
% callout_tests
```

Creating Thread-Local Signals:callout_tests.c

```
   /*                callout_tests.c               */

/*
================================================================
  I want to read from an unreliable device and interrupt it if
it should fail to return from the read after some amount of time.
So I set a per-thread timer and longjmp() out of the device
driver should it expire.
================================================================


cc -o callout_tests callout_tests.c -L. -R. -lcallout -g  -lpthread -lthread -
pthread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L

#include <signal.h>
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <setjmp.h>
#include "callout.h"
#include "thread_extensions.h"





int                     TIMEOUT = 3, DEVICE_SPEED = 3;
pthread_key_t           JMPBUF_KEY;


void my_SigHandler(int signo)
{void                   *envp;
 pthread_t              tid = pthread_self();
 char                   *name = thread_name(tid);

 /*   Note that printf() and thread_name() are not signal safe! */
 printf("Got signal in thread: %s. Signal %d\n", name, signo);
 envp = pthread_getspecific(JMPBUF_KEY);
```

```
 siglongjmp((int *) envp, 1);
}

void slow_device_interrupter(void *arg)
{pthread_t *ptid;

 ptid = (pthread_t *) arg;
 printf("Interrupting the device on thread %s...\n", thread_name (*ptid));
 pthread_kill(*ptid, SIGUSR1);   /* Interrupt the device. */
}


device_read(int speed)  /* Fake a read from the actual device */
{
  delay(speed*1000, 0);
}


void call_device(int speed, int timeout)
{int                    timer_number;
 pthread_t              tid = pthread_self();

 timer_number = thread_settimer(timeout, slow_device_interrupter,
                                  (void *) &tid); /* Should not pass tid! */
 device_read(speed);
 thread_deletetimer(timer_number);
 printf("Device responded in time!\n");
}


/* This thread doesn't do anything useful for this program.  It does
   prove that you can have different threads calling sigwait() on
   different signal sets.
*/
void *my_sigwaiter(void *arg)
{int            sig, err;
 sigset_t       s;

 sigfillset(&s);
 pthread_sigmask(SIG_BLOCK, &s, NULL);  /* Take no interrupts */

 sigemptyset (&s);
 sigaddset (&s, SIGINT);
 sigaddset (&s, SIGTSTP);
 sigaddset (&s, SIGQUIT);
 sigaddset (&s, SIGHUP);

 /* User program waits for everything EXCEPT USR1 and ALRM */
 while(1)
```

```
    {err = sigwait(&s, &sig);
    if (err != 0) {
        printf("%s\n", strerror(err));
        abort();
     }
     switch (sig)
       {case SIGINT  :  printf("Exit on SIGINT\n"), exit(1);
        case SIGTSTP :  printf("Exit on SIGSTP\n"), exit(1);
        case SIGQUIT :  printf("Exit on SIGQUIT\n"), exit(1);

        default:        printf("Default handler for signal: %d\n", sig);
      }
   }
}


main(int argc, char *argv[])
{jmp_buf                        env;
 sigset_t                       s, s1, s2;
 static struct sigaction        action;
 int                            i;
 pthread_t                      tid;



 /* Initialize things */
 action.sa_handler = (void (*)(void)) my_SigHandler;
 sigaction(SIGUSR1, &action, NULL);

 sigfillset(&s);
 pthread_sigmask(SIG_BLOCK, &s, &s2); /* Block everything, then... */
 sigemptyset(&s);
 sigaddset(&s, SIGUSR1);
 pthread_sigmask(SIG_UNBLOCK, &s, &s1); /* Unblock USR1 */

 PTHREAD_CREATE(&tid, NULL, my_sigwaiter, NULL);
 pthread_key_create(&JMPBUF_KEY, NULL);

#if !defined(__sun) && !defined(__alpha)
 initialize_thread_timer();
#endif

 /* The real program begins here. */

 for (i=0; i < 4; i++)

   {DEVICE_SPEED = i; TIMEOUT = 4 - i;
    printf("DEVICE_SPEED=%d TIMEOUT=%d\n", DEVICE_SPEED, TIMEOUT);
    if (sigsetjmp(env, 1) != 0)
```

```
      {printf("Device timed out!\n");
       continue;
      }
    pthread_setspecific(JMPBUF_KEY, (void *)env);
    call_device(DEVICE_SPEED, TIMEOUT);
  }

 printf("Done\n");
 exit(0);
}
```

Creating Thread-Local Signals: callout.c (library)

```
/*                  callout.c                 */



/*

 The interface to the per-thread alarm package.  This is fully MT-safe
 and will handle any number of alarms / sleeps.

   int thread_settimer(int delay, void fn(void *), void *arg)
   thread_deletetimer(int timer_number)
   void thread_sleep(int t)



 typically:

   i = thread_settimer(20, foo, TID);
   ...
   thread_deletetimer(i);

 "Run function foo(TID) in 20 seconds."

 If the thread gets to the deletetimer() line before the alarm expires,
 only that one alarm will be affected.

 TID can be any value, and foo() is allowed to send a signal to specific
 threads if desired (eg, TID).  That thread would have to install a signal
 handler in that case (presumably SIGUSR1).


 thread_sleep(3);

 "Put the current thread to sleep for 3 seconds."

 sleep() uses SIGALRM, so it will not work with other uses of alarm().
 thread_sleep() is based upon thread_settimer() and works reliably with any
 number of threads.  Note that the sleeping thread will become RUNNABLE
 at wake-up time.  When it actually gets a CPU is a scheduling question.
 (Yes, nanosleep() makes more sense, but this is a nice proof-of-concept.)
```

```
 The user code must not handle SIGALRM, but that is the only limitation.
 (Mask it out & don't sigwait() on it.)  The user code may install handlers
 or sigwait() for any other signal.

 This library uses alarm(), hence its finest granularity is 1 second.  Higher
 resolution timers can easily be built on top of this, and are left to the
 reader as an exercise.


  Build a dynamic library:

cc  -G -o libcallout.so callout.c -g -lpthread -lthread -lposix4
*/



#define _POSIX_C_SOURCE 199506L

#include <signal.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include "thread_extensions.h"


 /*  This runs before main() and starts up the sigwait() loop for SIGALRM. */

void initialize_thread_timer();

#ifdef __sun
#pragma init (initialize_thread_timer)
#endif

#if defined (__alpha) && defined (__osf__)
__init_thread_timer()
{initialize_thread_timer();}
#endif


typedef struct timer_struct {
  int                   expire_time;   /* UTC time */
  int                   number;        /* Unique identifier */
  void                  (*fn)(void *); /* Function to call */
  void                  *arg;          /* Argument to fn */
  struct timer_struct   *next;         /* Pointer to next */
} timer_queue_t;
```

```
timer_queue_t           *timer_queue;
pthread_t               _SIGWAIT_TID;
pthread_mutex_t         timer_queue_mutex;
sem_t                   timer_sema;
int                     timer_number;

void UnusedSigHandler(int signo)
{
  /* This should never happen.
     Note that printf() and thread_name() are not signal safe!
     */
  printf("Got SIGALRM (%d) in handler for thread %d?!\n",
         signo, thread_name(pthread_self()));
  exit(1);
}

 /* To wake up a sleeping thread, just release its semaphore.  The
   thread will run as soon as the scheduler decides to schedule it.
 */
void thread_timer_wakeup(void *s)
{
  sem_post((sem_t *) s);
}

 /* Set a sleep time.  This function will set up the ALRM, then sleep
   on a semaphore it creates for the purpose. */
void thread_sleep(int t)
{sem_t s;

 if (t < 1) return;
 SEM_INIT(&s, 0, NULL);
 thread_settimer(t, thread_timer_wakeup, &s);
 while(sem_wait(&s) != 0) DEBUG(printf("Sem Interrupted\n");)
 sem_destroy(&s);
}


initialize_timer()
{
  _SIGWAIT_TID = pthread_self();
  timer_queue = (timer_queue_t *) malloc(sizeof(timer_queue_t));
  timer_queue->next = NULL;
  timer_queue->expire_time = -1;
  timer_number = 1;
  sem_post(&timer_sema);               /* Used only for initialization */
}

thread_timer_setalarm()
```

```
{int now = time(NULL);
 int delay;

 pthread_mutex_lock(&timer_queue_mutex);
 if (timer_queue->next == NULL)
   {pthread_mutex_unlock(&timer_queue_mutex);
    return;}
 delay = timer_queue->next->expire_time - now;
 if (delay < 1) delay = 1;
  alarm(delay);
 pthread_mutex_unlock(&timer_queue_mutex);
}

 /* Delete the node for the 1st matching entry.  Leave the ALRM. */
void thread_deletetimer(int timer_number)
{
  timer_queue_t         *tq, *tq1, *tq2, *ts;

  pthread_mutex_lock(&timer_queue_mutex);
  tq2 = timer_queue;
  for (tq = timer_queue, tq1 = timer_queue->next;
       tq1 != NULL;
       tq1 = tq1->next, tq = tq->next)
    {if (tq1->number == timer_number)
     {tq->next = tq1->next;
      free(tq1);}}
 pthread_mutex_unlock(&timer_queue_mutex);
}


int thread_settimer(int delay, void fn(void *), void *arg)
{
  timer_queue_t         *tq, *tq1, *tq2, *ts;
  int                   expire_time;
  int                   killp = 0;

  ts  = (timer_queue_t *) malloc(sizeof(timer_queue_t));
  expire_time = (int) time(NULL)+delay;
  ts->expire_time=expire_time;
  ts->fn = fn;
  ts->arg = arg;

  pthread_mutex_lock(&timer_queue_mutex);
  timer_number++;
  ts->number = timer_number;
  tq2 = timer_queue;
  DEBUG(if (tq2->next == NULL) printf("New Timer\n"););
  for (tq = timer_queue, tq1 = timer_queue->next;
       1;
```

```
      tq1 = tq1->next, tq = tq->next)
    {if ((tq1 == NULL) || (tq1->expire_time > expire_time))
     {tq->next = ts;
      ts->next = tq1;
      if (tq2->next == ts) killp = 1;
      pthread_mutex_unlock(&timer_queue_mutex);

      thread_timer_setalarm();
      return(timer_number);}
   }
}

thread_expired_timer()
{
  timer_queue_t          *tq, *tq1;
  int                    now = time(NULL);
  void                   (*fn) (void *);
  void                   *arg;


  pthread_mutex_lock(&timer_queue_mutex);
  tq = timer_queue->next;
  tq1 = tq->next;

  while(tq->expire_time <= now)
    {fn = tq->fn;
     arg = tq->arg;
     free(tq);
     tq = tq1;
     timer_queue->next = tq;

     pthread_mutex_unlock(&timer_queue_mutex);
     fn(arg);

     if (tq1 == NULL) {DEBUG(printf("No timer.\n");) return;}
     pthread_mutex_lock(&timer_queue_mutex);
     tq1 = tq1->next;
   }
  alarm(tq->expire_time - now);
  pthread_mutex_unlock(&timer_queue_mutex);
}

void *sigwaiter(void *arg)
{int                             sig, err;
 sigset_t                        s;
 static struct sigaction         action;

 action.sa_handler = (void (*)(void)) UnusedSigHandler;
 sigaction(SIGALRM, &action, NULL);
```

```
 sigfillset(&s);
 pthread_sigmask(SIG_BLOCK, &s, NULL);  /* Take no interrupts */
 initialize_timer();

 sigemptyset(&s);
 sigaddset(&s, SIGALRM);

 while(1)
    {err = sigwait(&s, &sig);
     if (err) {errno = err; perror("Sigwait error!");}
     switch (sig)
       {case SIGALRM :           thread_expired_timer();
                                 break;
       default:                  printf("IMPOSSIBLE! signal: %d\n", sig);
      }
   }
}


void initialize_thread_timer()
{ pthread_attr_t        attr;
  int                   err;
  pthread_t             tid;

  SEM_INIT(&timer_sema, 0, 0);
  PTHREAD_ATTR_INIT(&attr);
  err = pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  pthread_mutex_lock(&timer_queue_mutex);
  PTHREAD_CREATE(&tid, &attr, sigwaiter, NULL);
  while(sem_wait(&timer_sema) != 0) ;
  pthread_mutex_unlock(&timer_queue_mutex);
}
```

Passing Shared Data: matrix_problem.c

```
/*                  matrix_problem.c                */

/*

This program illustrates the dangers of inadvertantly using shared
data, and the problems which result.

What is particularly interesting about this program, is that the
thread start times must be considered as random, making this into
a big race condition.  It is possible that you will get the correct
answer.  It is possible that all the threads will get the same values
for {i,j}, it is possible for any combination of values! A very nice
program indeed.

*/


/*
cc -o matrix_problem matrix_problem.c -L. -R. -g -lpthread -lthread -pthread -
lposix4
*/


#define _POSIX_C_SOURCE 199506L

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include "thread_extensions.h"



#define SIZE            4

pthread_mutex_t         unused_lock = PTHREAD_MUTEX_INITIALIZER;
sem_t                   barrier;
int                     a[SIZE] [SIZE], b[SIZE] [SIZE], c[SIZE] [SIZE];

typedef struct _v
```

```
{int    i;
 int    j;
} element_t;



void *add_element(void *arg)
{element_t *e = (element_t *) arg;

 delay(100, 100);
 c[e->i] [e->j] =  a[e->i] [e->j] +  b[e->i] [e->j];
 sem_post(&barrier);
 pthread_exit(NULL);
}


void fill_matrix(int m[SIZE][SIZE])
{int i, j, n = 0;

 for (i=0; i<SIZE; i++)
   for (j=0; j<SIZE; j++)
     m[i][j] = n++;
}

void print_matrix(int m[SIZE][SIZE])
{int i, j, n = 0;

 for (i=0; i<SIZE; i++)
   {printf("\n\t| ");
    for (j=0; j<SIZE; j++)
      printf("%2d ", m[i][j]);
    printf("|");
  }
}


main(int argc, char *argv[])
{int            i, j, n = 0;
 element_t      element;
 pthread_attr_t attr;
 pthread_t      tid;

 SEM_INIT(&barrier, NULL, 0);
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

 fill_matrix(a);
 fill_matrix(b);
```

```
 iobench_start();
 for (i=0; i<SIZE; i++)
   for (j=0; j<SIZE; j++)
     {element.i = i; element.j = j;
      PTHREAD_CREATE(&tid, &attr, add_element, &element);
      delay(0, 40000);
    }
 thread_single_barrier(&barrier, SIZE*SIZE);
 iobench_end();

 printf("\n\n\n");
 print_matrix(a);
 printf("\n\n\n\t       + \n");
 print_matrix(b);
 printf("\n\n\n\t       = \n");
 print_matrix(c);
 printf("\n\n\n");
 iobench_report();
 printf("Done\n");
 exit(0);
}
```

Exercise Objective

Edit the code in `matrix_problem.c` and fix it so that the answer is correct.

Don't peek at `matrix_solution.c` or `matrix_solution2.c` until you have figured it out on your own! (The objective is not to get the answer, the objective is for you to think about the issue, so that when you encounter something similar in real code, you'll be able to deal with it!)

There are numerous simple solutions to the problem.

Wild Guess at Time to Completion:

30 minutes think/design time.
30 minutes programming/debugging
My solution changed 4 lines of code.

Passing Stack Data: matrix_solution.c

```
/*                  matrix_solution.c                */

/*

This program illustrates the dangers of inadvertantly using shared
data, and the problems which result.

What is particularly interesting about this program, is that the
thread start times must be considered as random, making this into
a big race condition.  It is possible that you will get the correct
answer.  It is possible that all the threads will get the same values
for {i,j}, it is possible for any combination of values! A very nice
program indeed.

This shows just one of many possible solutions.  You could pass a
structure, you could pass a single integer which encoded i and j,
you could use a global array, you could make it a producer/consumer
program, etc.

*/


/*
cc -o matrix_solution matrix_solution.c -L. -R. -g -lpthread -lthread -pthread -
lposix4
*/


#define _POSIX_C_SOURCE 199506L

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include "thread_extensions.h"



#define SIZE            4
```

```
pthread_mutex_t         unused_lock = PTHREAD_MUTEX_INITIALIZER;
sem_t                   barrier, one_at_a_time;
int                     a[SIZE] [SIZE], b[SIZE] [SIZE], c[SIZE] [SIZE];

typedef struct _v
{int    i;
 int    j;
} element_t;



void *add_element(void *arg)
{element_t *e = (element_t *) arg;
 int i = e->i, j = e->j;

 delay(100, 100);
 sem_post(&one_at_a_time);
 c[i] [j] =  a[i] [j] +  b[i] [j];
 sem_post(&barrier);
 pthread_exit(NULL);
}


void fill_matrix(int m[SIZE][SIZE])
{int i, j, n = 0;

 for (i=0; i<SIZE; i++)
   for (j=0; j<SIZE; j++)
     m[i][j] = n++;
}

void print_matrix(int m[SIZE][SIZE])
{int i, j, n = 0;

 for (i=0; i<SIZE; i++)
   {printf("\n\t| ");
    for (j=0; j<SIZE; j++)
      printf("%2d ", m[i][j]);
    printf("|");
  }
}


main(int argc, char *argv[])
{int i, j, n = 0;
 element_t element;
 pthread_attr_t attr;
 pthread_t tid;
```

```
 SEM_INIT(&barrier, NULL, 0);
 SEM_INIT(&one_at_a_time, NULL, 0);
 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

 fill_matrix(a);
 fill_matrix(b);

 iobench_start();
 for (i=0; i<SIZE; i++)
   for (j=0; j<SIZE; j++)
     {element.i = i; element.j = j;
      PTHREAD_CREATE(&tid, &attr, add_element, &element);
      delay(0, 40000);
      SEM_WAIT(&one_at_a_time);
    }
 thread_single_barrier(&barrier, SIZE*SIZE);
 iobench_end();

 printf("\n\n\n");
 print_matrix(a);
 printf("\n\n\n\t       + \n");
 print_matrix(b);
 printf("\n\n\n\t       = \n");
 print_matrix(c);
 printf("\n\n\n");
 iobench_report();
 printf("Done\n");
 exit(0);
}
```

Exercise Objective

Look at the code in `test_recursive_mutex.c.` It needs a recursive mutex to function. Unfortunately, POSIX mutexes are not recursive.

Right now, it runs using the recursive mutex I wrote (named `thread_recursive_mutex`), but I'm sure you could build a better one. So do a global replacement of "`thread_recursive_mutex`" with "`my_thread_recursive_mutex`" and then implement it.

You will need to write three functions and a datatype:

```
typedef struct _rm {...} my_thread_recursive_mutex_t;

int thread_recursive_mutex_lock (my_thread_recursive_mutex_t *)

int thread_recursive_mutex_unlock (my_thread_recursive_mutex_t *)
```

Extra Credit: Write the other two functions:

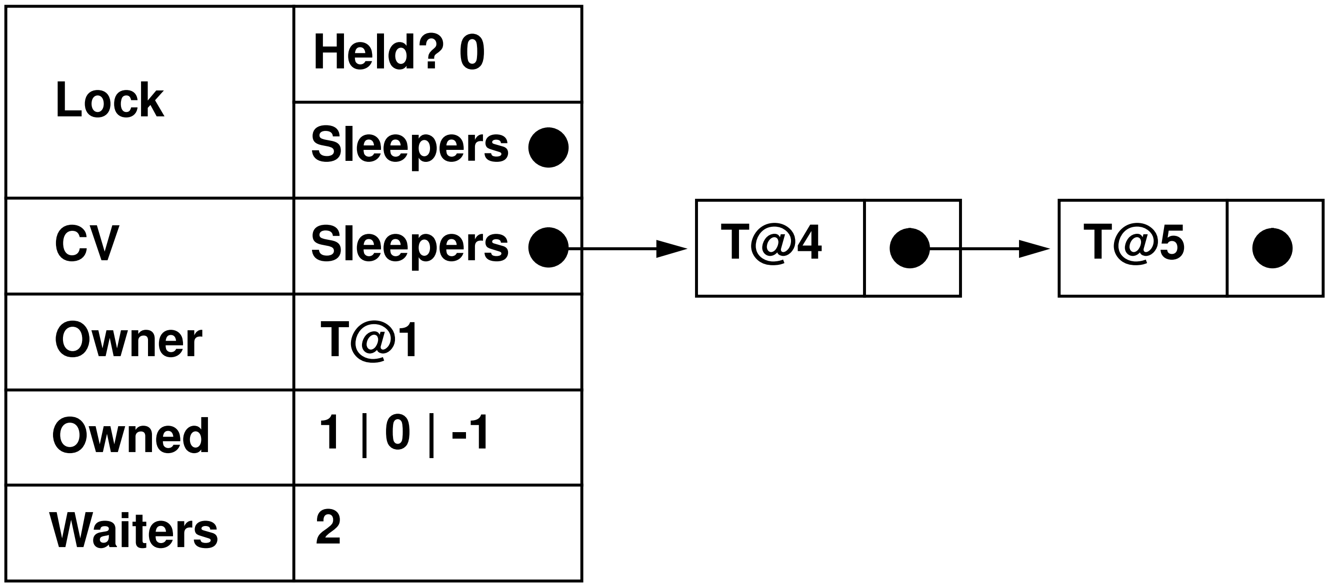
```
int my_thread_recursive_mutex_trylock (thread_recursive_mutex_t *)

int my_thread_recursive_mutex_destroy (thread_recursive_mutex_t *)
```

There is really only one simple solution to the problem (with lots of little variations). You may look at the definition of FIFO mutexes for guidance.

Wild Guess at Time to Completion:

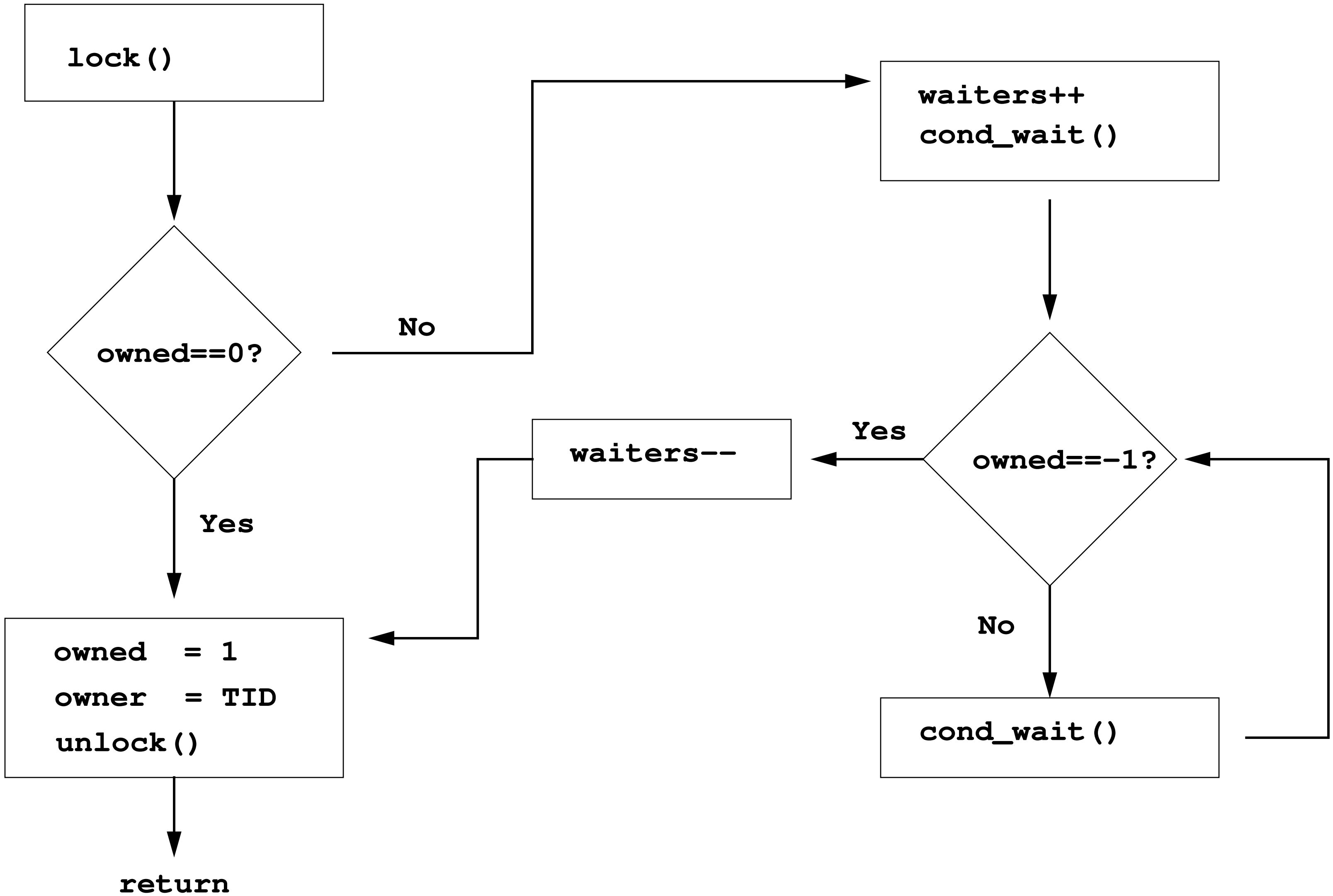
30 minutes think/design time.
30 minutes programming/debugging

FIFO Mutex

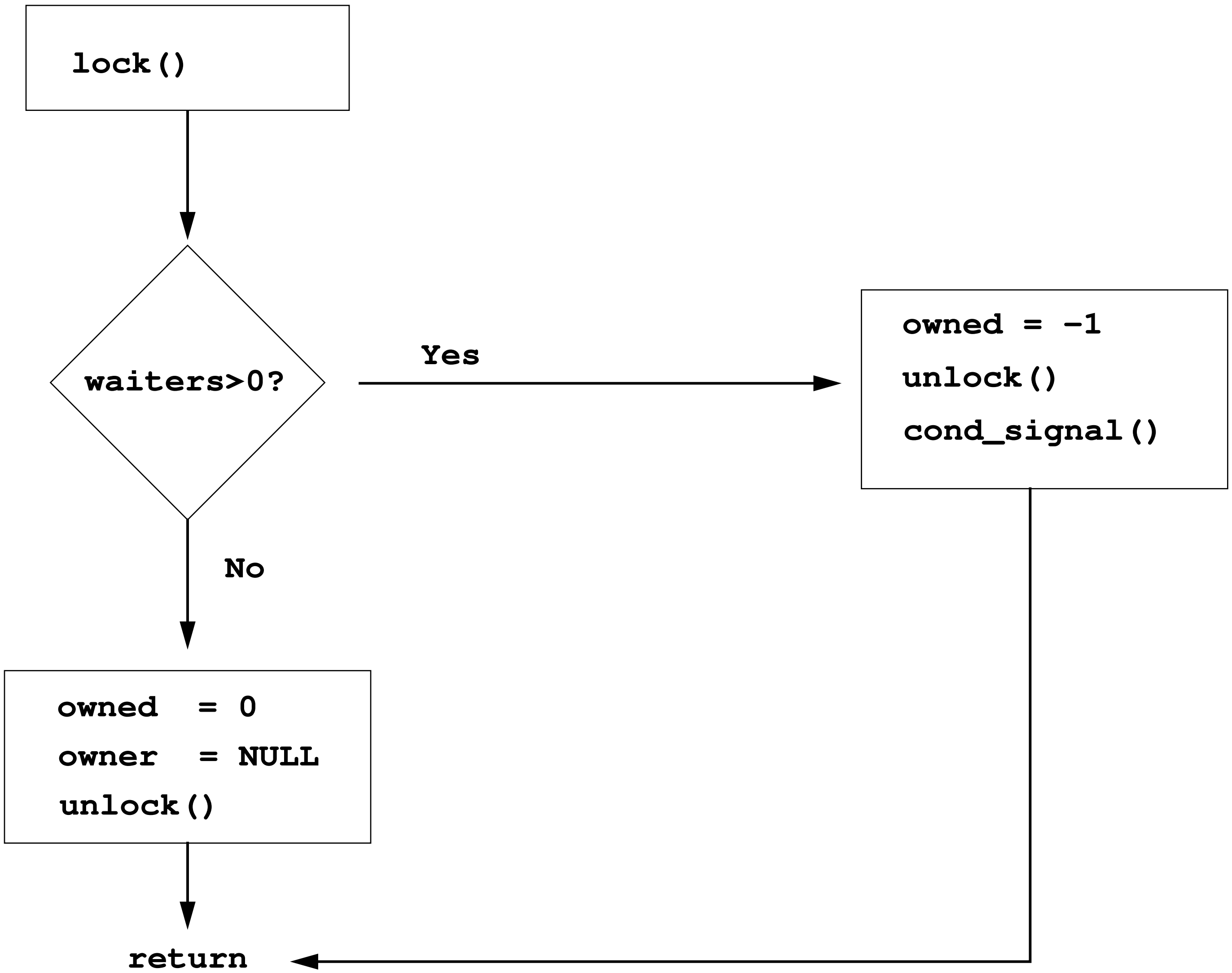


The lock protects the owner word and waiters count. If owned, then Owner contains the TID. If Owned == 0, it's free. If Owned == -1, then it's being passed to a thread which is presently waking up. Waiters is the number of threads on the CV's sleep queue.
Note that Owner is defined as pthread_t, hence it cannot contain anything else (e.g., 0 or -1). Therefore we need the Owner field.

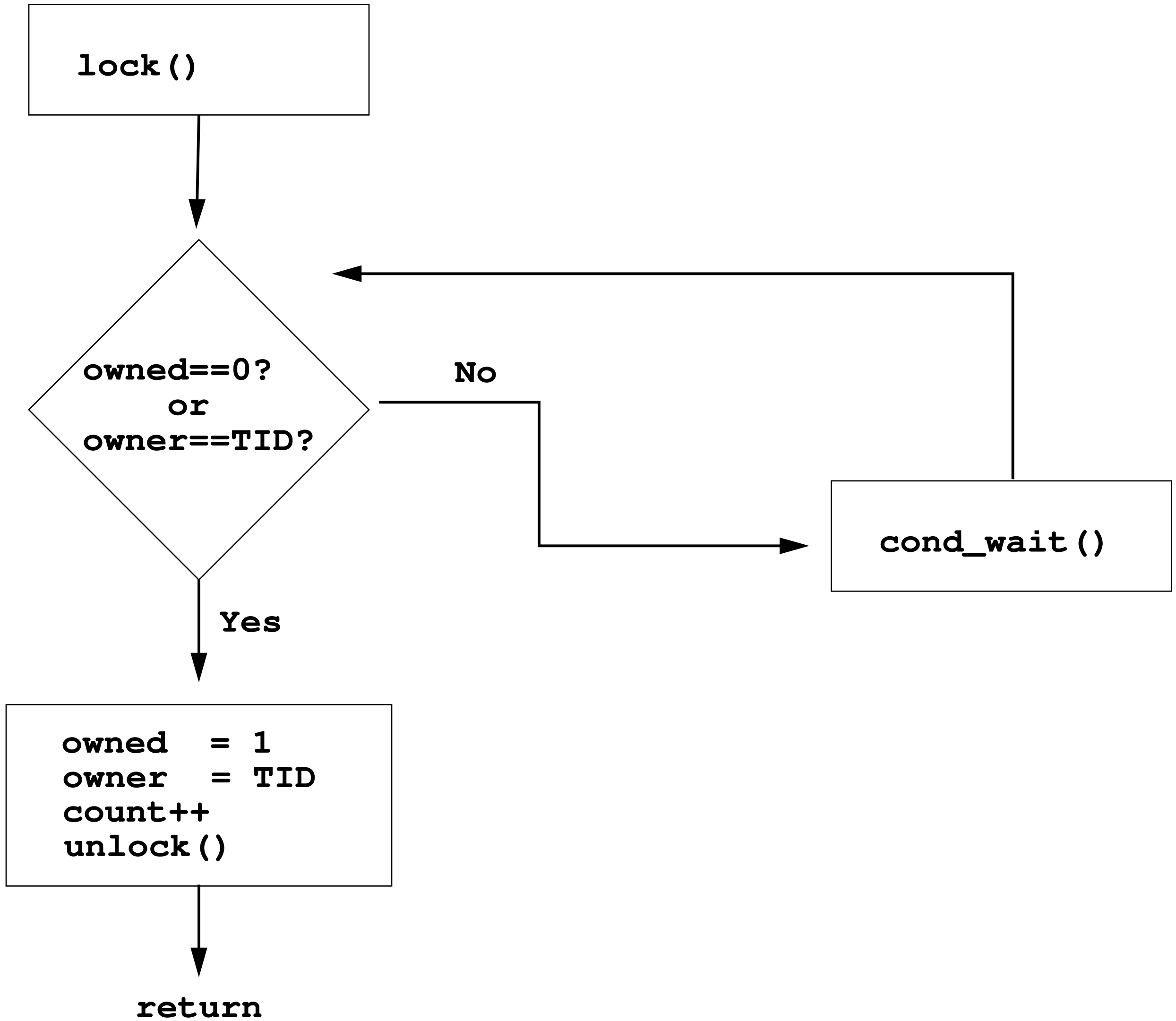
FIFO Mutex (lock)



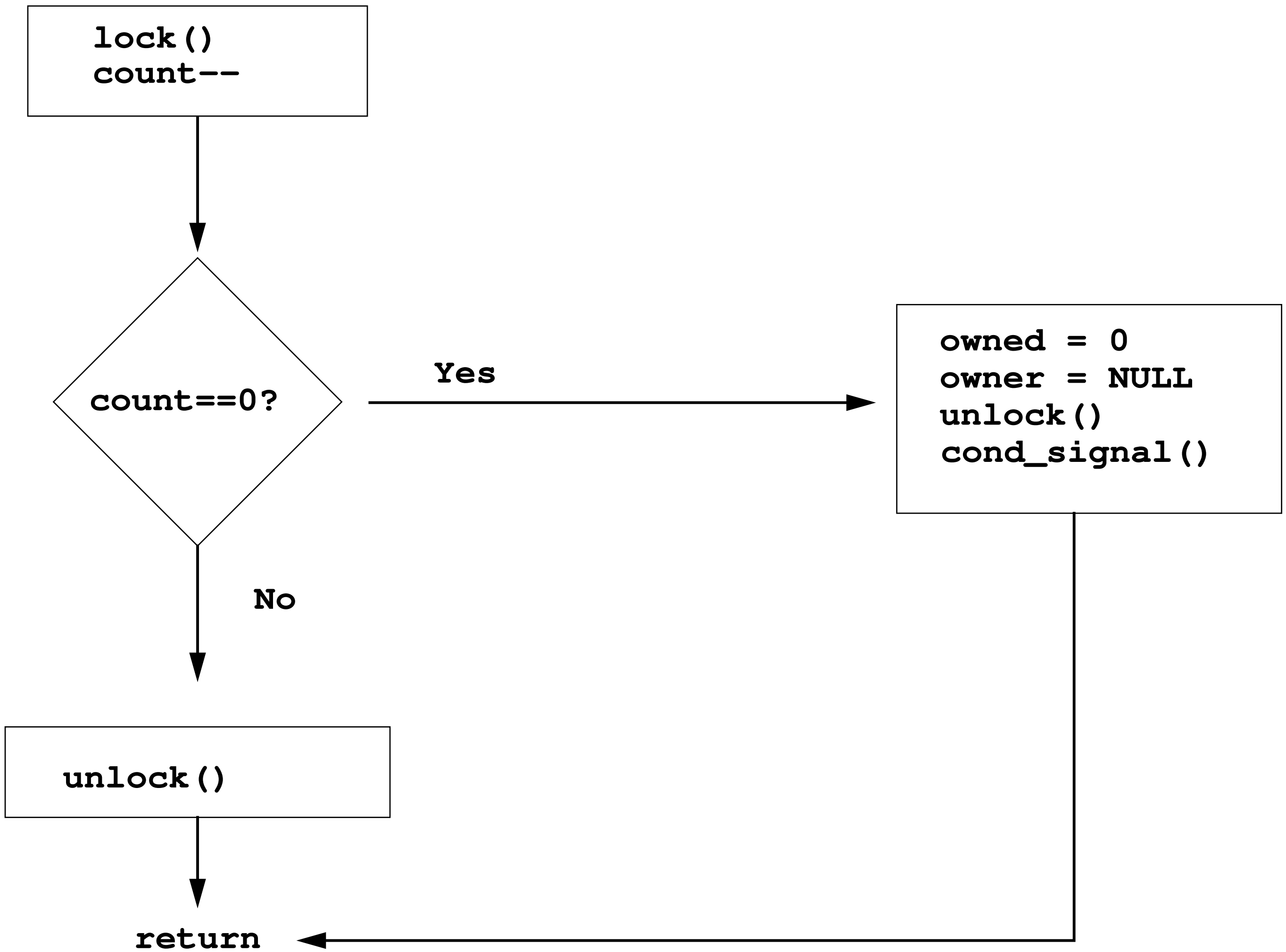
FIFO Mutex (unlock)



Recursive Mutex (lock)



Recursive Mutex (unlock)



Testing Recursive Mutexes: test_recursive_mutex.c

```
/*                 test_recursive_mutex.c                */

/*

  Test your recursive mutex.  Replace "thread_recursive_mutex" with
  "my_thread_recursive_mutex", then write it!

*/


/*
cc -o test_recursive_mutex test_recursive_mutex.c -L. -R. -g -lpthread -lthread
-thread_extensions -lposix4
*/

#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include "thread_extensions.h"

typedef struct {
  pthread_mutex_t       lock;           /* lock for structure */
  pthread_cond_t        cv;             /* waiting list control */
  pthread_t             owner;          /* NULL_TID or TID */
} my_thread_recursive_mutex_t;


char *s[6] = {"", "*", "**", "***", "****", "*****"};


thread_recursive_mutex_t m;


void *recursive(void *arg)
{int            i = (int) arg;
 pthread_t      tid = pthread_self();
 char           *name = thread_name(tid);
```

```
 if (i > 0)
   {printf("%s Trying to enter level %d \t%s\n", name, 6-i, s[6-i]);
    thread_recursive_mutex_lock(&m);
    printf("%s Entered          level %d \t%s\n", name, 6-i, s[6-i]);
    delay(200, 0);
    recursive((void *)(i-1));
    thread_recursive_mutex_unlock(&m);
    printf("%s Leaving          level %d \t%s\n", name, 6-i, s[6-i]);
  }
 return;
}


main(int argc, char **argv)
{int                    i;
 pthread_attr_t         attr;
 pthread_t              tid;

 thread_recursive_mutex_init(&m, NULL);

 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

 PTHREAD_CREATE(&tid, &attr, recursive, (void *) 5);
 PTHREAD_CREATE(&tid, &attr, recursive, (void *) 5);
 PTHREAD_CREATE(&tid, &attr, recursive, (void *) 5);

 pthread_exit(0);
}
```

Bil's Extensions (Library): thread_extensions.h

```
/*
        A set of useful functions and macros for pthreads programs.
*/

#include <pthread.h>
#include <semaphore.h>


#define _POSIX_C_SOURCE 199506L

/* Some useful constants */
#ifndef NULL
# define NULL           ((void*)0)
#endif
#ifndef TRUE
# define TRUE           1
#endif
#ifndef FALSE
# define FALSE          0
#endif
#define DEBUG(arg)      {if (getenv("DEBUG")) arg;}
/* This NULL_TID may not work on all implementations! */
#define NULL_TID        (pthread_t) 0L


/*
   ================================================================
                        General List
   ================================================================
*/

typedef struct _list_t {
        void            *frst;
        struct _list_t  *next;
} list_t;

extern          list_t  *cons(void *, void *);
extern          int     empty(list_t *);
extern          void    *first(list_t *);
extern          list_t  *rest(list_t *);
```

```
/*
   ================================================================
                        Readers / Writer Lock
   ================================================================
*/

/*
   Definitions for rw locks adapted from Sun's SPILT package.
   RWLocks are  not part of the POSIX threads standard, but
   they are part of UNIX98.
*/

#define THREAD_RWLOCK_INITIALIZER \
 {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, \
    PTHREAD_COND_INITIALIZER, 0, 0}

typedef struct {
  pthread_mutex_t       lock;           /* lock for structure           */
  pthread_cond_t        readers;        /* waiting readers              */
  pthread_cond_t        writers;        /* waiting writers              */
  int                   state;          /* -1:writer,0:free,>0:readers  */
  int                   waiters;        /* number of waiting writers    */
} thread_rwlock_t;

typedef int thread_rwlock_attr_t;       /* Not implemented */

extern  int     thread_rwlock_init(thread_rwlock_t *,
                                        thread_rwlock_attr_t *);
extern  int     thread_rwlock_destroy(thread_rwlock_t *);
extern  int     thread_rw_rdlock(thread_rwlock_t *);
extern  int     thread_rw_tryrdlock(thread_rwlock_t *);
extern  int     thread_rw_wrlock(thread_rwlock_t *);
extern  int     thread_rw_trywrlock(thread_rwlock_t *);
extern  int     thread_rw_unlock(thread_rwlock_t *);



/*
   ================================================================
                        Fifo Mutex Lock
   ================================================================
*/

typedef struct _fifo_mutex_queue_t {
  pthread_t                     tid;
  sem_t                         semaphore;
  struct _fifo_mutex_queue_t    *next;
} fifo_mutex_queue_t;
```

```
#define THREAD_FIFO_MUTEX_INITIALIZER \
 {PTHREAD_MUTEX_INITIALIZER, NULL, FALSE, NULL_TID}


typedef struct  {
  pthread_mutex_t       lock;            /* lock for structure           */
  fifo_mutex_queue_t    *waitersQueue;   /* queue of waiting writers     */
  fifo_mutex_queue_t    *waitersQueueLast; /* last of waiting writers   */
  int                   owned;           /* TRUE | FALSE | -1:awaiting   */
  pthread_t             owner;           /* tid of owner                 */
} thread_fifo_mutex_t;

extern  int     thread_fifo_mutex_init(thread_fifo_mutex_t *,
                                          pthread_mutexattr_t *);
extern  int     thread_fifo_mutex_destroy(thread_fifo_mutex_t *);
extern  int     thread_fifo_mutex_lock(thread_fifo_mutex_t *);
extern  int     thread_fifo_mutex_trylock(thread_fifo_mutex_t *);
extern  int     thread_fifo_mutex_unlock(thread_fifo_mutex_t *);


/*
   ================================================================
                        Debug Mutex Lock
   ================================================================
*/


#define PTHREAD_DMUTEX_INITIALIZER \
 {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, \
    NULL, NULL, 0, 0, FALSE, NULL_TID}

typedef struct _sleep_queue_t {
  pthread_t                    tid;
  struct _sleep_queue_t        *next;
} sleep_queue_t;


typedef struct _dmutex {
  pthread_mutex_t       lock;            /* lock for structure   */
  pthread_cond_t        cv;              /* waiting list control */
  char                  *name;           /* The declared name    */
  sleep_queue_t         *sleepers;       /* List of sleepers     */
  int                   n_locked;        /* Number of times locked */
  int                   n_failed;        /* Times woken w/o locking */
  int                   owned;           /* TRUE | FALSE         */
  pthread_t             owner;           /* NULL_TID or TID      */
} pthread_dmutex_t;
```

```
typedef struct _dmutex_queue_t {
  struct _dmutex                *mutex;
  struct _dmutex_queue_t        *next;
} dmutex_queue_t;

extern  int     pthread_dmutex_init(pthread_dmutex_t *, pthread_mutexattr_t *);
extern  int     pthread_dmutex_destroy(pthread_dmutex_t *);
extern  int     pthread_dmutex_lock(pthread_dmutex_t *);
extern  int     pthread_dmutex_trylock(pthread_dmutex_t *);
extern  int     pthread_dmutex_unlock(pthread_dmutex_t *);

#define pthread_dmutex_init(m, a) pthread_dmutex_init1(m, a, #m)

/*
   ================================================================
                     Recursive Mutex Lock
   ================================================================
*/

#define THREAD_RECURSIVE_MUTEX_INITIALIZER \
 {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0, FALSE, NULL_TID}


typedef struct  {
  pthread_mutex_t       lock;           /* lock for structure           */
  pthread_cond_t        cv;             /* waiting list control         */
  int                   count;          /* times locked recursively     */
  int                   owned;          /* TRUE | FALSE         */
  pthread_t             owner;          /* NULL_TID or TID      */
} thread_recursive_mutex_t;

extern  int     thread_recursive_mutex_init(thread_recursive_mutex_t *,
                                         pthread_mutexattr_t *);
extern  int     thread_recursive_mutex_destroy(thread_recursive_mutex_t *);
extern  int     thread_recursive_mutex_lock(thread_recursive_mutex_t *);
extern  int     thread_recursive_mutex_trylock(thread_recursive_mutex_t *);
extern  int     thread_recursive_mutex_unlock(thread_recursive_mutex_t *);

/*
   ================================================================
                             Barrier
   ================================================================
*/



typedef struct {
  pthread_mutex_t       lock;           /* lock for structure           */
```

```
  pthread_cond_t        cv;               /* waiting list control         */
  int                   count;            /* Number of threads to wait for*/
  int                   n_sleepers;       /* Number of threads to waiting */
  int                   releasing;        /* Still waking up sleepers     */
} thread_barrier_t;

extern  int     thread_barrier_init(thread_barrier_t *, int);
extern  int     thread_barrier_destroy(thread_barrier_t *);
extern  int     thread_barrier_wait(thread_barrier_t *);
extern  int     thread_barrier_trywait(thread_barrier_t *);


/*
   ================================================================
                          Single Barrier
   ================================================================
*/



typedef struct  {
  pthread_mutex_t       lock;             /* lock for structure           */
  pthread_cond_t        cv;               /* waiting list control         */
  int                   count;            /* Number of threads to post    */
  int                   n_posters;        /* Number of threads who posted */
  int                   n_waiters;        /* Number of threads waiting    */
  int                   releasing;        /* Still waking up sleepers     */
} thread_single_barrier_t;

extern  int     thread_single_barrier_init(thread_single_barrier_t *, int);
extern  int     thread_single_barrier_destroy(thread_single_barrier_t *);
extern  int     thread_single_barrier_wait(thread_single_barrier_t *);
extern  int     thread_single_barrier_post(thread_single_barrier_t *);

/*
   ================================================================
                      Timed Mutex Lock
   ================================================================
*/

#define THREAD_TIMED_MUTEX_INITIALIZER  \
 {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, FALSE, NULL_TID, 0}


typedef struct _timed_mutex {
  pthread_mutex_t       lock;             /* lock for structure           */
  pthread_cond_t        cv;               /* waiting list control         */
  int                   owned;            /* TRUE | FALSE                 */
  pthread_t             owner;            /* NULL_TID or TID              */
```

```
  const struct timespec *abstime;       /* absolute time before timeout */
} thread_timed_mutex_t;

extern  int     thread_timed_mutex_init(thread_timed_mutex_t *,
                                                 pthread_mutexattr_t *);
extern  int     thread_timed_mutex_destroy(thread_timed_mutex_t *);
extern  int     thread_timed_mutex_lock(thread_timed_mutex_t *,
                                            const struct timespec *);
extern  int     thread_timed_mutex_trylock(thread_timed_mutex_t *);
extern  int     thread_timed_mutex_unlock(thread_timed_mutex_t *);

/*
   ================================================================
                     Assorted Useful Functions
   ================================================================
*/

extern  char    *thread_name(pthread_t);


int PTHREAD_CREATE(pthread_t *new_thread_ID,
                   const pthread_attr_t *attr,
                   void * (*start_func)(void *), void *arg);

int PTHREAD_ATTR_INIT(pthread_attr_t *a);

int PTHREAD_CONDATTR_INIT(pthread_condattr_t *a);

int PTHREAD_MUTEXATTR_INIT(pthread_mutexattr_t *a);

int SEM_INIT(sem_t *sem, int pshared, unsigned int value);

int SEM_WAIT(sem_t *arg);

void delay(int sleep_ms, int cpu_us);

void *killer(void *arg);

/*----------------------------------------------------------------------------
 *
 * Library Name: UTIL
 * Module Name:  iobench
 *
 * Designer:    R. C. Sullivan
 * Programmer:  R. C. Sullivan
 * Date:        Sep 22, 1995
 *
 * History Of Changes:
 *      Name         Date            Description
```

```
 *      ----         ----            -----------
 *      RCS     Jan 17, 1996     Inital release
 *
 * Purpose:
 *   To report resource usage statistics that will be correct for
 * programs using threads on a Solaris system.
 *
 * Notes:
 *
 *-----------------------------------------------------------------------------
 */

extern struct prusage prusagebuf_start, prusagebuf_end;
extern int procfd;
extern double real_time, user_time, system_time, trap_time, wait_time;
extern unsigned long minor_pfs, major_pfs, input_blocks, output_blocks, iochars;

void iobench_start();
void iobench_end();
void iobench_report();
```

Bil's Extensions (Library): thread_extensions.c

```
 /*                thread_extensions.c                */

/*
  Lots of useful functions for all sorts of neat things.

   I changed the names to avoid all conflicts / confusion
   with Solaris threads.
*/

/*
cc  -G -o libthread_extensions.so thread_extensions.c -g -lpthread
        -lthread -lposix4

*/




#include <stdio.h>
#include <errno.h>
#include "thread_extensions.h"


#define _POSIX_C_SOURCE 199506L


pthread_key_t   sleep_queue_key, FIFO_MUTEX_QUEUE_KEY;

void dmutex_key_initializer();

#ifdef __sun
 /*  This runs before main() and declares the TSD key. */
#pragma init(dmutex_key_initializer)
#pragma init(fifo_key_initializer)
#endif

#if defined(__alpha) && (__osf__)
void __init_dmutex_key()
{
  dmutex_key_initializer();
```

```
  fifo_key_initializer();
}
#endif




/*
   ====================================================================
                          General List
   ====================================================================
*/

list_t *cons(void *obj, void *lst) {
  list_t                *l = (list_t *) malloc(sizeof(list_t));

  l->frst = obj;
  l->next = lst;
  return(l);
}

int empty(list_t *l) {return(l == NULL);}
void *first(list_t *l) {return(l->frst);}
list_t *rest(list_t *l) {return(l->next);}


/*
   ====================================================================
                          Stringify (for printing) TIDs
   ====================================================================
*/


typedef struct _pt
{pthread_t      tid;
 char           *name;
 struct _pt     *next;
} thread_name_t;

#if (!defined(__sun) && !defined(__sgi))

char *thread_name(pthread_t tid)
{char s[100];
 thread_name_t *n;
 static int tid_count = 1000;
 static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
 static thread_name_t *thread_names = NULL;

 if (pthread_equal(NULL_TID, tid)) tid = pthread_self();
```

```
 pthread_mutex_lock(&lock);

   for (n = thread_names; n != NULL; n=n->next)
     {if (pthread_equal(n->tid, tid))
        goto L;
    }
 n = (thread_name_t *) malloc(sizeof(thread_name_t));
 n->tid = tid;
#if defined(__alpha) && defined(__osf__)
 sprintf(s, "T@%d", pthread_getsequence_np(tid));
#else
 sprintf(s, "T@%d", tid_count);
 tid_count++;
#endif
 n->name = (char *) malloc(strlen(s)+1);
 strcpy(n->name, s);
 n->next = thread_names;
 thread_names = n;

L:pthread_mutex_unlock(&lock);
 return(n->name);
}

#else
/* For known implementations, use the underlying TID numbering */

char *thread_name(pthread_t tid)
{char s[100];
 thread_name_t *n;

 static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
 static thread_name_t *thread_names = NULL;

 if (pthread_equal(NULL_TID, tid)) tid = pthread_self();
 pthread_mutex_lock(&lock);

   for (n = thread_names; n != NULL; n=n->next)
     {if (pthread_equal(n->tid, tid))
        goto L;
    }
 n = (thread_name_t *) malloc(sizeof(thread_name_t));
 n->tid = tid;
 sprintf(s, "T@%d", (int) tid);
 n->name = (char *) malloc(strlen(s)+1);
 strcpy(n->name, s);
 n->next = thread_names;
 thread_names = n;

L:pthread_mutex_unlock(&lock);
```

```
 return(n->name);
}
#endif


/*
   ================================================================
                        Readers / Writer Lock
   ================================================================
*/

int thread_rwlock_init(thread_rwlock_t *rwp,
                           thread_rwlock_attr_t *attr)
{int                     ptype;
 pthread_mutexattr_t    mutex_attr;
 pthread_condattr_t     cond_attr;

 pthread_mutexattr_init(&mutex_attr);
 pthread_condattr_init(&cond_attr);

    /* Not Implemented ...
    ptype = thread_rwlock_attr_getshared(&attr);
    pthread_mutexattr_setpshared(&mutex_attr, ptype);
    pthread_condattr_setpshared(&cond_attr, ptype);
    */

 pthread_mutex_init(&rwp->lock, &mutex_attr);
 pthread_cond_init(&rwp->readers, &cond_attr);
 pthread_cond_init(&rwp->writers, &cond_attr);

 rwp->state     = 0;
 rwp->waiters   = 0;

 return(0);
}


int thread_rwlock_destroy(thread_rwlock_t *rwp)
{
  pthread_mutex_destroy(&rwp->lock);
  pthread_cond_destroy(&rwp->readers);
  pthread_cond_destroy(&rwp->writers);

  return(0);
}


int thread_rw_rdlock(thread_rwlock_t *rwp)
{pthread_mutex_t        *lkp    = &rwp->lock;
```

```
 pthread_mutex_lock(lkp);
 pthread_cleanup_push(pthread_mutex_unlock, lkp);
 /* active or queued writers            */
 while ((rwp->state < 0) || rwp->waiters)
   pthread_cond_wait(&rwp->readers, lkp);

 rwp->state++;
 pthread_cleanup_pop(1);

 return(0);
}


int thread_rw_tryrdlock(thread_rwlock_t *rwp)
{int    status  = EBUSY;

 pthread_mutex_lock(&rwp->lock);
 /* available and no writers queued     */
 if ((rwp->state >= 0) && !rwp->waiters) {
   rwp->state++;
   status       = 0;
 }

 pthread_mutex_unlock(&rwp->lock);

 return(status);
}


void thread_rw_wrlock_cleanup(void *arg)
{thread_rwlock_t        *rwp    = (thread_rwlock_t *) arg;

 /*
  * Was the only queued writer and lock is available for readers.
  * Called through cancellation clean-up so lock is held at entry.
  */
 if ((--rwp->waiters == 0) && (rwp->state >= 0))
   pthread_cond_broadcast(&rwp->readers);

 pthread_mutex_unlock(&rwp->lock);

 return;
}


int thread_rw_wrlock(thread_rwlock_t *rwp)
{pthread_mutex_t        *lkp    = &rwp->lock;
```

```
 pthread_mutex_lock(lkp);
 rwp->waiters++;                /* another writer queued                */
 pthread_cleanup_push(thread_rw_wrlock_cleanup, rwp);

 while (rwp->state)
   pthread_cond_wait(&rwp->writers, lkp);

 rwp->state     = -1;
 pthread_cleanup_pop(1);

 return(0);
}


int thread_rw_trywrlock(thread_rwlock_t *rwp)
{int    status  = EBUSY;

 pthread_mutex_lock(&rwp->lock);
 /* no readers, no writers, no writers queued   */
 if (!rwp->state && !rwp->waiters) {
   rwp->state   = -1;
   status               = 0;
 }

 pthread_mutex_unlock(&rwp->lock);

 return(status);
}


int thread_rw_unlock(thread_rwlock_t *rwp)
{pthread_mutex_lock(&rwp->lock);

 if (rwp->state == -1) {        /* writer releasing     */
   rwp->state   = 0;            /* mark as available    */

   if (rwp->waiters)            /* writers queued       */
     pthread_cond_signal(&rwp->writers);
   else
     pthread_cond_broadcast(&rwp->readers);
 } else {
   if (--rwp->state == 0)       /* no more readers      */
     pthread_cond_signal(&rwp->writers);
 }

 pthread_mutex_unlock(&rwp->lock);

 return(0);
}
```

```
/*
   =================================================================
                        Fifo Mutex Lock
   =================================================================

        Guarantees that all sleepers will get the mutex in correct
        order.
*/


int thread_fifo_mutex_init(thread_fifo_mutex_t *m,
                               pthread_mutexattr_t *attr)
{
  pthread_mutex_init(&(m->lock), NULL);
  m->waitersQueue = NULL;
  m->owned = FALSE;
  m->owner = NULL_TID;
}


int thread_fifo_mutex_destroy(thread_fifo_mutex_t *m)
{
  pthread_mutex_destroy(&(m->lock));
}


void fifo_queue_destroyer(void *arg)
{
  fifo_mutex_queue_t *fifo_mutex_queue = (fifo_mutex_queue_t *) arg;

  free(fifo_mutex_queue);
}


void fifo_key_initializer()
{
  pthread_key_create(&FIFO_MUTEX_QUEUE_KEY, fifo_queue_destroyer);
}



fifo_mutex_queue_t *get_fifo_mutex_queue()
{fifo_mutex_queue_t *fifo_mutex_queue =
               (fifo_mutex_queue_t *) pthread_getspecific(FIFO_MUTEX_QUEUE_KEY);
int i;
 if (fifo_mutex_queue == NULL)
        {fifo_mutex_queue = (fifo_mutex_queue_t *) mal-
```

```
loc(sizeof(fifo_mutex_queue_t));
        i++;
         fifo_mutex_queue->tid = pthread_self();
         sem_init(&fifo_mutex_queue->semaphore, FALSE, 0);
         pthread_setspecific(FIFO_MUTEX_QUEUE_KEY, fifo_mutex_queue);
        }
 return(fifo_mutex_queue);
}


int thread_fifo_mutex_lock(thread_fifo_mutex_t *m)
{pthread_t self = pthread_self();
 fifo_mutex_queue_t *fifo_mutex_queue = get_fifo_mutex_queue();

  pthread_mutex_lock(&(m->lock));
  if (!(m->owned))
    {m->owner = self;
     m->owned = TRUE;
     pthread_mutex_unlock(&(m->lock));
     return(0);}

  fifo_mutex_queue->next = NULL;
  if (m->waitersQueue == NULL)
    m->waitersQueue = fifo_mutex_queue;
  else
    m->waitersQueueLast->next = fifo_mutex_queue;
  m->waitersQueueLast = fifo_mutex_queue;
  pthread_mutex_unlock(&(m->lock));
  SEM_WAIT(&(fifo_mutex_queue->semaphore));

  pthread_mutex_lock(&(m->lock));
  if (m->owned != -1) {abort(1);}  /* Cannot happen */
  m->owner = self;
  m->owned = TRUE;
  m->waitersQueue = m->waitersQueue->next;
  pthread_mutex_unlock(&(m->lock));
  return(0);
}


int thread_fifo_mutex_trylock(thread_fifo_mutex_t *m)
{

  pthread_mutex_lock(&(m->lock));
  if (!(m->owned))
    {m->owner = pthread_self();
     m->owned = TRUE;
     pthread_mutex_unlock(&(m->lock));
     return(0);}
```

```
  pthread_mutex_unlock(&(m->lock));
  return(EBUSY);
}


int thread_fifo_mutex_unlock(thread_fifo_mutex_t *m)
{
  pthread_mutex_lock(&(m->lock));
  m->owner = NULL_TID;

  if (m->waitersQueue == NULL)
    {m->owned = FALSE;
     pthread_mutex_unlock(&(m->lock));
     return(0);}

  m->owned = -1;
  sem_post(&(m->waitersQueue->semaphore));
  pthread_mutex_unlock(&(m->lock));
  return(0);
}



/*
   ==================================================================
                     Simple (wrong) Fifo Mutex Lock
   ==================================================================

        Guarantees that a sleeper will get the mutex, even if
        some other thread trys first.  But several sleeping threads
        will NOT be FIFO.  But the implementation is simple.



int thread_fifo_mutex_init(thread_fifo_mutex_t *m,
                               pthread_mutexattr_t *attr)
{
  pthread_mutex_init(&(m->lock), NULL);
  pthread_cond_init(&(m->cv), NULL);
  m->waiters = 0;
  m->owned = FALSE;
  m->owner = NULL_TID;
}


int thread_fifo_mutex_destroy(thread_fifo_mutex_t *m)
{
  pthread_mutex_destroy(&(m->lock));
  pthread_cond_destroy(&(m->cv));
```

```
}


int thread_fifo_mutex_lock(thread_fifo_mutex_t *m)
{

  pthread_mutex_lock(&(m->lock));
  if (!(m->owned))
    {m->owner = pthread_self();
     m->owned = TRUE;
     pthread_mutex_unlock(&(m->lock));
     return(0);}

  m->waiters++;
  /* Don't check for -1 the first time because
     that -1 is intended for a sleeper.
  pthread_cond_wait(&(m->cv), &(m->lock));

  while (m->owned != -1)
    pthread_cond_wait(&(m->cv), &(m->lock));
  /* A spurious wakeup at just the wrong moment will give
     ownership to the wrong thread.  So we're only 99.999% FIFO.
     But NEVER incorrect.
  m->owner = pthread_self();
  m->owned = TRUE;
  m->waiters--;
  pthread_mutex_unlock(&(m->lock));
  return(0);
}


int thread_fifo_mutex_trylock(thread_fifo_mutex_t *m)
{

  pthread_mutex_lock(&(m->lock));
  if (!(m->owned))
    {m->owner = pthread_self();
     m->owned = TRUE;
     pthread_mutex_unlock(&(m->lock));
     return(0);}
  pthread_mutex_unlock(&(m->lock));
  return(EBUSY);
}


int thread_fifo_mutex_unlock(thread_fifo_mutex_t *m)
{
  pthread_mutex_lock(&(m->lock));
  m->owner = NULL_TID;
```

```
  if (m->waiters == 0)
    {m->owned = FALSE;
     pthread_mutex_unlock(&(m->lock));
     return(0);}

  m->owned = -1;
  pthread_mutex_unlock(&(m->lock));
  pthread_cond_signal(&(m->cv));
  return(0);
}

*/



/*
   ================================================================
                          Debug Mutex Lock
   ================================================================

        For debugging programs.  Do a global replace of _mutex_ with
        _dmutex_ and then the dmutex will detect self deadlocks,
        non-owner unlocks, and record the owner's TID for use from
        the debugger.

        If you call pthread_dmutex_init(), then a macro will place
        the dmutex on a global list and record its name.  If you only
        use PTHREAD_DMUTEX_INITIALIZER, then no recording.  You may
        choose specific names via pthread_dmutex_init1().

        print_dmutexes() will print out the state (and sleep queue)
        for all mutexes on the list.  From the debugger:
        (debugger) call print_dmutexes()

*/

pthread_mutex_t dmutex_queue_lock = PTHREAD_MUTEX_INITIALIZER;
dmutex_queue_t  *dmutex_queue;


void dmutex_key_initializer()
{
  pthread_key_create(&sleep_queue_key, NULL);
 }

void insert_dmutex(pthread_dmutex_t *m)
{dmutex_queue_t *dq;
```

```
 dq = (dmutex_queue_t *) malloc(sizeof(dmutex_queue_t));
 dq->mutex = m;
 pthread_mutex_lock(&dmutex_queue_lock);
 dq->next = dmutex_queue;
 dmutex_queue = dq;
 pthread_mutex_unlock(&dmutex_queue_lock);
}

void remove_dmutex(dmutex_queue_t **q, pthread_dmutex_t *m)
{dmutex_queue_t *dq, *dq1;

 pthread_mutex_lock(&dmutex_queue_lock);

 if ((*q)->mutex == m)
   {dq1 = dmutex_queue->next;
    *q = dq1->next;
    free(dq1);
  }
 else
   for (dq = *q; dq->next != NULL; dq = dq->next)
     if (dq->next->mutex == m)
       {dq1 = dq->next;
        dq->next = dq1->next;
        free(dq1);
        break;
      }

 pthread_mutex_unlock(&dmutex_queue_lock);
}

void remove_sleeper(sleep_queue_t **q, pthread_t tid)
{sleep_queue_t *dq, *dq1;

 if (pthread_equal((*q)->tid, tid))
   {
     *q = (*q)->next;
   }
 else
   for (dq = *q; dq->next != NULL; dq = dq->next)
     if (pthread_equal(dq->next->tid, tid))
       {dq1 = dq->next;
        dq->next = dq1->next;
        break;
      }

}


void print_dmutex(pthread_dmutex_t *m)
```

```
{sleep_queue_t  *sq;

 pthread_mutex_lock(&(m->lock));
 flockfile (stdout);
   if (!m->owned)
   printf("%s ---- Times locked: %3d,  failed: %3d.  Sleepers: ( ",
          m->name, m->n_locked, m->n_failed);
 else
   printf("%s %s  Times locked: %3d,  failed: %3d.  Sleepers: ( ",
          m->name, thread_name(m->owner), m->n_locked, m->n_failed);

 for (sq = m->sleepers; sq != NULL; sq = sq->next)
   printf("%s ", thread_name(sq->tid));
 printf(")\n");
 funlockfile (stdout);
 pthread_mutex_unlock(&(m->lock));
}



void print_dmutexes()
{dmutex_queue_t *dq;
 pthread_dmutex_t *m;

 pthread_mutex_lock(&dmutex_queue_lock);
 for (dq = dmutex_queue; dq != NULL; dq = dq->next)
   {m = dq->mutex;
    print_dmutex(m);
  }
 pthread_mutex_unlock(&dmutex_queue_lock);
}



int pthread_dmutex_init1(pthread_dmutex_t *m, pthread_mutexattr_t *attr,
                         char *name)
{
  pthread_mutex_init(&(m->lock), NULL);
  pthread_cond_init(&(m->cv), NULL);
  m->owned = FALSE;
  m->owner = NULL_TID;
  m->name = name;
  m->sleepers = NULL;
  m->n_locked = 0;
  m->n_failed = 0;
  insert_dmutex(m);
}

int pthread_dmutex_destroy(pthread_dmutex_t *m)
```

```
{
  pthread_mutex_destroy(&(m->lock));
  pthread_cond_destroy(&(m->cv));
  remove_dmutex(&dmutex_queue, m);
}



int pthread_dmutex_lock(pthread_dmutex_t *m)
{pthread_t tid = pthread_self();
 sleep_queue_t *dq = (sleep_queue_t *) pthread_getspecific(sleep_queue_key);

 if (dq == NULL)
   {dq = (sleep_queue_t *) malloc(sizeof(sleep_queue_t));
    pthread_setspecific(sleep_queue_key, dq);
    dq->tid = tid;
  }

 pthread_mutex_lock(&(m->lock));
 if (pthread_equal(m->owner, tid))
   {if (m->name)
      printf("Error! %s deadlocking dmutex %s.\n", thread_name(tid), m->name);
   else
     printf("Error! %s deadlocking dmutex x%x.\n", thread_name(tid), m);
    abort();    /* I don't want it to exit.  I want it to crash. */
  }
 while (m->owned)
   {dq->next = m->sleepers;
    m->sleepers = dq;
    m->n_failed++;
    pthread_cond_wait(&(m->cv), &(m->lock));
    remove_sleeper(&(m->sleepers), tid);
   }
 m->owner = tid;
 m->owned = TRUE;
 m->n_locked++;
 pthread_mutex_unlock(&(m->lock));
 return(0);
}


int pthread_dmutex_trylock(pthread_dmutex_t *m)
{pthread_t tid = pthread_self();

 pthread_mutex_lock(&(m->lock));
 if (!(m->owned))
    {m->owner = tid;
     m->owned = TRUE;
     m->n_locked++;
```

```
    pthread_mutex_unlock(&(m->lock));
    return(0);
  }
 m->n_failed++;
 pthread_mutex_unlock(&(m->lock));
 return(EBUSY);
}


int pthread_dmutex_unlock(pthread_dmutex_t *m)
{pthread_t tid = pthread_self();

 pthread_mutex_lock(&(m->lock));
 if (!pthread_equal(m->owner, tid))
   {if (m->name)
      printf("Error! %s unlocking dmutex %s owned by %s\n",
             thread_name(tid), m->name, thread_name(m->owner));
   else
     printf("Error! %s unlocking dmutex x%x owned by %s\n",
            thread_name(tid), m, thread_name(m->owner));
    abort();    /* I don't want it to exit.  I want it to crash. */
    }

 m->owned = FALSE;
 m->owner = NULL_TID;
 pthread_mutex_unlock(&(m->lock));
 pthread_cond_signal(&(m->cv));
 return(0);
}




/*
   ================================================================
                     Recursive Mutex Lock
   ================================================================

       Allow the same thread to lock the mutex recursively
       (requiring the same number of unlocks).

       Use of these is NOT recommended for any program!  Write
       your program better so that you don't WANT them.
*/


int thread_recursive_mutex_init(thread_recursive_mutex_t *m,
                                    pthread_mutexattr_t *attr)
{
```

```
  pthread_mutex_init(&(m->lock), NULL);
  pthread_cond_init(&(m->cv), NULL);
  m->owned = FALSE;
  m->owner = NULL_TID;
  m->count = 0;
}


int thread_recursive_mutex_destroy(thread_recursive_mutex_t *m)
{
  pthread_mutex_destroy(&(m->lock));
  pthread_cond_destroy(&(m->cv));
}



int thread_recursive_mutex_lock(thread_recursive_mutex_t *m)
{pthread_t tid = pthread_self();

  pthread_mutex_lock(&(m->lock));
  while (m->owned && !pthread_equal(m->owner, tid))
    pthread_cond_wait(&(m->cv), &(m->lock));
 m->owner = tid;
 m->owned = TRUE;
 m->count++;
 pthread_mutex_unlock(&(m->lock));
 return(0);
}


int thread_recursive_mutex_trylock(thread_recursive_mutex_t *m)
{pthread_t tid = pthread_self();

 pthread_mutex_lock(&(m->lock));
 if (!m->owned || pthread_equal(m->owner, tid))
    {m->owner = tid;
     m->owned = TRUE;
     m->count++;
     pthread_mutex_unlock(&(m->lock));
     return(0);
   }
  pthread_mutex_unlock(&(m->lock));
  return(EBUSY);
}


int thread_recursive_mutex_unlock(thread_recursive_mutex_t *m)
{
  pthread_mutex_lock(&(m->lock));
```

```
  m->count--;
  if (m->count == 0)
    {m->owned = FALSE;
     m->owner = NULL_TID;
     pthread_mutex_unlock(&(m->lock));
     pthread_cond_signal(&(m->cv));
     return(0);
   }
  pthread_mutex_unlock(&(m->lock));
  return(0);
}


/*
   ================================================================
                         Barrier
   ================================================================

        Full barrier: all threads wait until the last thread arrives
        before waking up.
*/


int thread_barrier_init(thread_barrier_t *m, int i)
{
  pthread_mutex_init(&(m->lock), NULL);
  pthread_cond_init(&(m->cv), NULL);
  m->n_sleepers = 0;
  m->releasing = FALSE;
  m->count = i;
}


int thread_barrier_destroy(thread_barrier_t *m)
{
  pthread_mutex_destroy(&(m->lock));
  pthread_cond_destroy(&(m->cv));
}



int thread_barrier_wait(thread_barrier_t *m)
{int release = FALSE;

 pthread_mutex_lock(&(m->lock));
 while (m->releasing)          /* If previous cycle still releasing, wait */
   pthread_cond_wait(&(m->cv), &(m->lock));

 m->n_sleepers++;
```

```
 if (m->n_sleepers == m->count)
   release = m->releasing = TRUE;
 else
   while (!m->releasing)
     pthread_cond_wait(&(m->cv), &(m->lock));

 m->n_sleepers--;
 if (m->n_sleepers == 0)
   {m->releasing = FALSE;
    release = TRUE;             /* Wake up waiters (if any) for next cycle */
  }
 pthread_mutex_unlock(&(m->lock));

 if (release)
   pthread_cond_broadcast(&m->cv);
 return(0);
}


/*
   ================================================================
                          Barrier
   ================================================================

        Full barrier: all threads wait until the last thread arrives
        before waking up.
*/


int thread_single_barrier_init(thread_single_barrier_t *b, int i)
{
  pthread_mutex_init(&(b->lock), NULL);
  pthread_cond_init(&(b->cv), NULL);
  b->n_posters = 0;
  b->n_waiters = 0;
  b->releasing = FALSE;
  b->count = i;
}


int thread_single_barrier_destroy(thread_single_barrier_t *b)
{
  pthread_mutex_destroy(&(b->lock));
  pthread_cond_destroy(&(b->cv));
}



int thread_single_barrier_wait(thread_single_barrier_t *b) {
```

```
  int release = FALSE;

  b->n_waiters++;
  while (!b->releasing)
    pthread_cond_wait(&(b->cv), &(b->lock));

  b->n_waiters--;
  if (b->n_waiters == 0) {
    b->releasing = FALSE;
    release = TRUE;              /* Wake up waiters (if any) for next cycle */
    b->n_posters=0;
    }
  pthread_mutex_unlock(&(b->lock));

  if (release)
    pthread_cond_broadcast(&b->cv);
  return(0);
}




int thread_single_barrier_post(thread_single_barrier_t *b)
{int release = FALSE;

 pthread_mutex_lock(&(b->lock));
 while (b->releasing)            /* If previous cycle still releasing, wait */
   pthread_cond_wait(&(b->cv), &(b->lock));

 b->n_posters++;
 if (b->n_posters == b->count)
   {b->releasing = TRUE;
    release = TRUE;              /* Wake up waiters (if any) for next cycle */
  }
 pthread_mutex_unlock(&(b->lock));

 if (release)
   pthread_cond_broadcast(&b->cv);
 return(0);
}




/*
   ================================================================
                     Error Detection
   ================================================================
```

```
        For those functions which can return a "reasonable" runtime
        error, go ahead and detect them.  And then abort.
*/


int PTHREAD_CREATE(pthread_t *new_thread_ID,
                   const pthread_attr_t *attr,
                   void * (*start_func)(void *), void *arg)
{int err;
 pthread_t tid;

 if (!new_thread_ID)
   {printf("TID cannot be NULL.\n");
    abort();
  }
 if (err = pthread_create(new_thread_ID, attr, start_func, arg))
   {printf("%s\n", strerror(err));
    abort();
  }}

int PTHREAD_ATTR_INIT(pthread_attr_t *a)
{int err;
 if (err = pthread_attr_init(a))
   {printf("%s\n", strerror(err));
    abort();
  }}

int PTHREAD_CONDATTR_INIT(pthread_condattr_t *a)
{int err;
 if (err = pthread_condattr_init(a))
   {printf("%s\n", strerror(err));
    abort();
  }}

int PTHREAD_MUTEXATTR_INIT(pthread_mutexattr_t *a)
{int err;
 if (err = pthread_mutexattr_init(a))
   {printf("%s\n", strerror(err));
    abort();
  }}

int SEM_INIT(sem_t *sem, int pshared, unsigned int value)
{int err;
 if (err = sem_init(sem, pshared, value))
   {perror("");
    abort();
  }}
```

```
int SEM_WAIT(sem_t *arg)        /* Ignore signal interruptions */
{while (sem_wait(arg) != 0) {}}


/*
   ================================================================
                          Random useful functions
   ================================================================

*/

int delay_cycles_per_us = 8;            /* App. number for 110MHz SS4

/* sleep time will be quantized at 10ms, CPU time is VERY approximate. */

void delay(int sleep_ms, int cpu_us)
{int i, cycles = cpu_us*delay_cycles_per_us;
 volatile j=0;
 struct timespec tv;

 if (sleep_ms != 0)
   {tv.tv_sec = sleep_ms/1000;
    tv.tv_nsec = (sleep_ms%1000) * 1000000;
    nanosleep(&tv, NULL);
  }

 if (cpu_us != 0)
   for (i=0; i<cycles; i++) j++;
}


int calibrate_delay()
{
  delay_cycles_per_us = 8;
  iobench_start();
  delay(0, 1000000);
  iobench_end();
  DEBUG(iobench_report());
  DEBUG(printf("Calibration pass: %d delay cycles -> %f seconds\n",
               delay_cycles_per_us, user_time));
  delay_cycles_per_us = (delay_cycles_per_us * 1.0) / user_time;

  iobench_start();
  delay(0, 1000000);
  iobench_end();
  DEBUG(iobench_report());
  DEBUG(printf("Verification pass: %d delay cycles -> %f seconds\n",
               delay_cycles_per_us, user_time));
```

```
}

/* A barrier for one thread to wait for lots of others. */
void thread_single_barrier(sem_t *barrier, int count)
{
  while (count > 0)
    {SEM_WAIT(barrier);
     count--;
    }
}


/*
   ================================================================
                        Timed Mutex Lock
   ================================================================

        Just like a timed CV, only for a mutex!
*/


int thread_timed_mutex_init(thread_timed_mutex_t *m,
                                    pthread_mutexattr_t *attr)
{
  pthread_mutex_init(&(m->lock), NULL);
  pthread_cond_init(&(m->cv), NULL);
  m->owned = FALSE;
  m->owner = NULL_TID;
  m->abstime = 0;
}


int thread_timed_mutex_destroy(thread_timed_mutex_t *m)
{
  pthread_mutex_destroy(&(m->lock));
  pthread_cond_destroy(&(m->cv));
}



int thread_timed_mutex_lock(thread_timed_mutex_t *m,
                                const struct timespec *abstime)
{int err;
 pthread_t tid = pthread_self();

  pthread_mutex_lock(&(m->lock));
  while (m->owned)
    {err=pthread_cond_timedwait(&(m->cv), &(m->lock), abstime);
     if (err)
```

```
       {pthread_mutex_unlock(&(m->lock));
        return(err);
      }}
 m->owner = tid;
 m->owned = TRUE;
 pthread_mutex_unlock(&(m->lock));
 return(0);
}


int thread_timed_mutex_trylock(thread_timed_mutex_t *m)
{pthread_t tid = pthread_self();

 pthread_mutex_lock(&(m->lock));
 if (!m->owned)
    {m->owner = tid;
     m->owned = TRUE;
     pthread_mutex_unlock(&(m->lock));
     return(0);
   }
  pthread_mutex_unlock(&(m->lock));
  return(EBUSY);
}


int thread_timed_mutex_unlock(thread_timed_mutex_t *m)
{
  pthread_mutex_lock(&(m->lock));
  m->owned = FALSE;
  m->owner = NULL_TID;
  pthread_mutex_unlock(&(m->lock));
  pthread_cond_signal(&(m->cv));
}




/*
   ================================================================
                         Wait before fork()
   ================================================================

  The idea is that you count the number of threads you have, then have
the forker thread wait for them all to reach a fork-safe location (where
they will pause).  The forker then forks and wakes up all of the other
threads.

  The other threads will call thread_wait_for_forker() whenever they
are at a fork-safe location.  If no thread is waiting to fork, they
```

```
simply continue.  If there is one, they will then sleep until awaken.

  The forker thread (there can only be one!) calls thread_wait_for_others()
before forking, and thread_awake_others() afterwards.  The child does
not need to call thread_awake_others() unless it plans to call fork()
also.  (A bug w/spurious wakeups if child also forks?)

*/

struct _fork_waiter
{
  pthread_mutex_t       lock;
  pthread_cond_t        forker_cv;
  pthread_cond_t        others_cv;
  int                   runners;
  int                   owned;
  pthread_t             owner;
};

struct _fork_waiter fork_waiter =
{PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, PTHREAD_COND_INITIALIZER,
   0, FALSE, NULL_TID};


void thread_wait_for_others(int i)
{
  pthread_mutex_lock(&fork_waiter.lock);
  if (fork_waiter.owned)
    {printf("Fatal Error: Multiple calls to thread_wait_for_others!");
     abort();
   }
  fork_waiter.owned = TRUE;
  fork_waiter.owner = pthread_self();
  fork_waiter.runners=i;
  while (fork_waiter.runners>0)
    pthread_cond_wait(&fork_waiter.forker_cv, &fork_waiter.lock);
  pthread_mutex_unlock(&fork_waiter.lock);
}


void thread_awake_others()
{
  pthread_mutex_lock(&fork_waiter.lock);
  fork_waiter.owned = FALSE;
  fork_waiter.owner = NULL_TID;
  pthread_mutex_unlock(&fork_waiter.lock);
  pthread_cond_broadcast(&fork_waiter.others_cv);
}
```

```
void thread_wait_for_forker()
{
  pthread_mutex_lock(&fork_waiter.lock);
  if (fork_waiter.owned)
     {fork_waiter.runners--;
      while (fork_waiter.owned)
        {if (fork_waiter.runners == 0)
pthread_cond_signal(&fork_waiter.forker_cv);
         pthread_cond_wait(&fork_waiter.others_cv, &fork_waiter.lock);
    }}
  pthread_mutex_unlock(&fork_waiter.lock);
}



/*
   ==================================================================
                                Killer
   ==================================================================
   */

void *killer(void *arg) {
  int time = (int) arg;
  sleep(time);
  printf("Killer thread timing out. (Program hung?)  Die!  Die!  Die!\n");
  exit(-1);
}



/*
   ==================================================================
                                Spin locks
   ==================================================================

  Two versions of spin locks.  The good versions (only defined for
  Solaris right now) will not flood the memory bus.

  The bad version of spin lock will flood the memory bus with ldstub
  ops.  Try setting BAD_SPIN_COUNT = 10^9 and then running this on
  a couple of CPUs, while trying to get work done on the other CPUs.

*/


#ifdef __sun2_5
const int SPIN_COUNT = 1000;           /* About .1us per loop, SS10 */

void spin_lock(pthread_mutex_t *m)
```

```
{int i;
 for (i = 0; i < SPIN_COUNT; i++)
   {if (m->pthread_mutex_lock.pthread_mutex_owner64 == 0) /* Check w/o ldstub */
      if (pthread_mutex_trylock(m) != EBUSY)
        return;                         /* Got it! */
    /* Didn't get it, continue the loop */
    }
 pthread_mutex_lock(m);                 /* Give up and block */
}

#else
const int BAD_SPIN_COUNT = 100;         /* About 1us  per loop, SS10 */

void spin_lock(pthread_mutex_t *m)
{int i;
 for (i=0; i < BAD_SPIN_COUNT; i++)
   {if (pthread_mutex_trylock(m) != EBUSY)      /* Trash the memory bus! */
      return;    }                              /* got the lock! */
 pthread_mutex_lock(m);                         /* give up and block. */
 return;                                       /* got the lock after blocking! */
}



#endif




/*----------------------------------------------------------------------------
 *
 * Library Name: UTIL
 * Module Name:  iobench
 *
 * Designer:    R. C. Sullivan
 * Programmer:  R. C. Sullivan
 * Date:        Sep 22, 1995
 *
 * History Of Changes:
 *      Name         Date            Description
 *      ----         ----            -----------
 *      RCS     Jan 17, 1996     Inital release
 *
 * Purpose:
 *   To report resource usage statistics that will be correct for
 * programs using threads on a Solaris system.
 *
 * Notes:
 *
```

```
 *----------------------------------------------------------------------------
 */

#ifdef __sun
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/procfs.h>

struct stat statbuf;
struct prusage prusagebuf_start, prusagebuf_end;
int procfd;
double real_time, total_real_time, user_time, system_time, trap_time, wait_time;
unsigned long minor_pfs, major_pfs, input_blocks, output_blocks, iochars;

void iobench_start() {
  char pfile[80];

  sprintf(pfile, "/proc/%ld", getpid());
  procfd = open(pfile, O_RDONLY);

  ioctl(procfd, PIOCUSAGE, &prusagebuf_start);
}

void iobench_end() {
  ioctl(procfd, PIOCUSAGE, &prusagebuf_end);
  close(procfd);

  real_time = (double) prusagebuf_start.pr_tstamp.tv_sec +
    (double) prusagebuf_start.pr_tstamp.tv_nsec / NANOSEC;
  real_time = (double) prusagebuf_end.pr_tstamp.tv_sec +
    (double) prusagebuf_end.pr_tstamp.tv_nsec / NANOSEC - real_time;

  total_real_time = (double) prusagebuf_start.pr_rtime.tv_sec +
    (double) prusagebuf_start.pr_rtime.tv_nsec / NANOSEC;
  total_real_time = (double) prusagebuf_end.pr_rtime.tv_sec +
    (double) prusagebuf_end.pr_rtime.tv_nsec / NANOSEC - real_time;

  user_time = (double) prusagebuf_start.pr_utime.tv_sec +
    (double) prusagebuf_start.pr_utime.tv_nsec / NANOSEC;
  user_time = (double) prusagebuf_end.pr_utime.tv_sec +
    (double) prusagebuf_end.pr_utime.tv_nsec / NANOSEC - user_time;

  system_time = (double) prusagebuf_start.pr_stime.tv_sec +
    (double) prusagebuf_start.pr_stime.tv_nsec / NANOSEC;
  system_time = (double) prusagebuf_end.pr_stime.tv_sec +
    (double) prusagebuf_end.pr_stime.tv_nsec / NANOSEC - system_time;
```

```
  trap_time = (double) prusagebuf_start.pr_ttime.tv_sec +
    (double) prusagebuf_start.pr_ttime.tv_nsec / NANOSEC;
  trap_time = (double) prusagebuf_end.pr_ttime.tv_sec +
    (double) prusagebuf_end.pr_ttime.tv_nsec / NANOSEC - trap_time;

  wait_time = (double) prusagebuf_start.pr_wtime.tv_sec +
    (double) prusagebuf_start.pr_wtime.tv_nsec / NANOSEC;
  wait_time = (double) prusagebuf_end.pr_wtime.tv_sec +
    (double) prusagebuf_end.pr_wtime.tv_nsec / NANOSEC - wait_time;

  minor_pfs = prusagebuf_end.pr_minf - prusagebuf_start.pr_minf;
  major_pfs = prusagebuf_end.pr_majf - prusagebuf_start.pr_majf;
  input_blocks = prusagebuf_end.pr_inblk - prusagebuf_start.pr_inblk;
  output_blocks = prusagebuf_end.pr_oublk - prusagebuf_start.pr_oublk;
  /*  iochars = prusagebuf_end.pr_ioch - prusagebuf_start.pr_ioch;*/
}

void iobench_report() {
  fprintf(stderr, "\nProcess info:\n");
  fprintf(stderr, "  elapsed time  %g\n", real_time);
  /*  fprintf(stderr, "  total time    %g\n", total_real_time);*/
  fprintf(stderr, "  CPU time      %g\n", user_time + system_time);
  /*  fprintf(stderr, "  user time     %g\n", user_time);
  fprintf(stderr, "  system time   %g\n", system_time);
  fprintf(stderr, "  trap time     %g\n", trap_time);
  fprintf(stderr, "  wait time     %g\n",  wait_time);
  fprintf(stderr, "  pfs    major/minor  %6lu/%6lu\n", major_pfs, minor_pfs);
  fprintf(stderr, "  blocks input/output %6lu/%6lu\n",
          input_blocks, output_blocks); */
  /*  fprintf(stderr, "  char inp/out  %lu\n", iochars);*/
  fprintf(stderr, "\n");

  /*  fprintf(stderr, "%2.5g Mbytes/sec (real time)\n", iochars /
      real_time / 1e6);
      fprintf(stderr, "%2.5g Mbytes/sec (CPU time) \n", iochars /
      (user_time + system_time) / 1e6);*/

  fprintf(stderr, "%2.1f%% CPU usage\n\n", 100 * (user_time + system_time) /
          real_time + .05);
}

#else
void iobench_start() {}
void iobench_end() {}
void iobench_report() {}
#endif
```

Misusing the Cache: cache_thrash.c

```
/*                  cache_thrash.c                  */

/*
cc -o cache_thrash cache_thrash.c -L. -R. -D__sun -D__sun2_6 -fast -lpthread -
lthread -lposix4 -lthread_extensions

  Standard deviation < 1% of mean.

*/


#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include "thread_extensions.h"


#define SIZE            1000000


int                     INTERATIONS = 30;
int                     BLOCK_SIZE = 1;
int                     touches = 0;
int                     CACHE_RESIDENT = 0;
int                     a[SIZE+100];


int add_rows()
{int            sum = 0, i, j;

 for (i = 0; i <  BLOCK_SIZE; i++)
   {
     DEBUG(printf("%d %d\n", j, i));
     for (j = i; j < SIZE; j+= BLOCK_SIZE*2)
       {if (CACHE_RESIDENT)
         {sum += a[i];
         sum += a[i+BLOCK_SIZE];
         }
```

```
        else
          {sum += a[j];
          sum += a[j+BLOCK_SIZE];
          }
        }
    }
 touches += SIZE;
 return(sum);
}


main(int argc, char **argv)
{int i, j, k;

 if (argc >= 2) INTERATIONS    = atoi(argv[1]);
 if (argc >= 3) CACHE_RESIDENT = atoi(argv[2]);
 printf("INTERATIONS: %d CACHE_RESIDENT: %d\n",  INTERATIONS, CACHE_RESIDENT);

 j = add_rows();

 for (k = 1; k <= 8; k = k*2)
   {iobench_start();
   touches = 0;
   BLOCK_SIZE = k;
   for (i = 0; i <  INTERATIONS; i++)
     j = add_rows();
   iobench_end();
   printf("BLOCK_SIZE: %d.  Total of %dM touches. Sum: %d\n", BLOCK_SIZE,
touches/1000000, j);
   iobench_report();
   }

 printf("Done\n");
 exit(0);
}
```

Misusing the Cache: false_sharing.c

```
/*                 false_sharing.c                */

/*
  Demonstrates what NOT to do on an MP.  If SEPARATION = 1, then there will be
  horrible false sharing and the machine will crawl.  If SEPARATION = 16
  then there will be no false sharing and the program will run about 10x
  faster.
*/


#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include “thread_extensions.h”


int               AVOCADOS_NUMBER = 6023500;
int               N_CPUS = 4;
int              SEPARATION = 16; /* 8 64-bit words per cache-line on Ultra SPARC
*/
sem_t             exit_barrier;
int               a[128];
pthread_attr_t  attr;


void *worker(void *arg)
{int              index = (int) arg;
 int              i;

 for (i=0; i<AVOCADOS_NUMBER; i++)
   {a[index]++;}

 sem_post(&exit_barrier);
}

main(int argc, char **argv)
{int              i, j;
 pthread_t        tid;
```

```
 if (argc >= 2) N_CPUS = atoi(argv[1]);

 printf("N_CPUS = %d \n \n", N_CPUS);

 PTHREAD_ATTR_INIT(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
 SEM_INIT(&exit_barrier, 0, 0);

 for (j=1;j<17;j+=j)
   {
     SEPARATION=j;
     printf("SEPARATION = %d\n", SEPARATION);
     iobench_start();
     for (i=0;i<N_CPUS;i++)
       PTHREAD_CREATE(&tid, &attr, worker, (void *) (i * SEPARATION));

     thread_single_barrier(&exit_barrier, N_CPUS);
     iobench_end();
     iobench_report();
   }
     printf("Done\n");
     exit(0);
}
```

Timing Disk Access: time_disk.c

```
/*                  time_disk.c                */

/*
  This program runs a set of read() calls against one large file.
  It runs with one or more threads so you can see the performance
  effect of MT.  Each read() gets one byte from a random location.
  You can run it for a number of iterations & get mean & SD.

  Make sure that there are links in /tmp pointing to wherever you
  can find room.

  lrwxrwxrwx   1 bil other ... time_disk0.tmp -> /disk2/6/temp_disk_test
  ln -s /disk2/6/temp_disk_test /tmp/time_disk0.tmp

  The file must be much larger than physical memory.  10x would be great,
  but 2x will do.  Expect "performance" to improve as the mbuffers get
  loaded.  For a file 2x Physical, initial 100/s will improve to 200/s
  (as 50% of the file will become cached).  To populate the cache, you
  can run this program for a while.

  runtime = (Physical Memory / PAGE_SIZE)   / READS_PER_SECOND  eg:
  160seconds = (128MB        / (8KB/PAGE))  / 100

  CF: Same program in Java: TimeDisk/Test.java

*/


#define _POSIX_C_SOURCE 199506L

#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <time.h>
#include <tnf/probe.h>
#include <math.h>
#include <unistd.h>
#include "thread_extensions.h"
```

```
#define MAX_FILE_SIZE   1024*1024*1024
#define PAGE_SIZE       8192
#define MAX_PAGES       (MAX_FILE_SIZE/PAGE_SIZE)
#define MAX_THREADS     512
#define MAX_DENSITY     100


/*  Hack for early TNF bug */
int PTHREAD_CREATE(pthread_t *new_thread_ID,
                   const pthread_attr_t *attr,
                   void * (*start_func)(void *), void *arg)
{
     thr_create(0, NULL, start_func, arg, 0x00000001, new_thread_ID);
}



pthread_attr_t            attr;
int                       nProcessed[MAX_THREADS];
pthread_t                 threads[MAX_THREADS];
int                       MAX_READS;
int                       density[MAX_DENSITY];
int                       hits[MAX_PAGES];
int                       fd;
char                      *path = "/tmp/time_disk0.tmp";
off_t                     length;
int                       PID;
int                       stop=FALSE;
int                       N_THREADS=1, SPIN_TIME=0, RUNTIME=1, ITERATIONS=1;


extern double real_time, total_real_time, user_time,
  system_time, trap_time, wait_time;



void *test(void *arg) {
  int    me = ((int) arg) % 1000; /* Hack to get two values in one! */

  /* Don't reuse seed, even across processes */
  long  seed = ((int) arg) + PID*100000;

  int   i, fileOffset, s;
  char  *thread = thread_name(pthread_self());
  char  buf[10];

  for (i=0; i< MAX_READS; i++) {
    if (stop) break;
    fileOffset = abs((rand_r(&seed)*PAGE_SIZE) % length);
```

```
    hits[(int) fileOffset/PAGE_SIZE]++;
    TNF_PROBE_1(process_request_start,  "start", "",
                tnf_int, location, fileOffset);
    /* DEBUG(printf("%s reading at %d\n", thread, fileOffset)); */
    s = pread(fd, &buf, 1, fileOffset);
    TNF_PROBE_0(process_request_end, "end", "");
    if (s != 1) printf("Error! at %d\n", fileOffset);
    delay(0, SPIN_TIME*1000);
    nProcessed[me]++;
  }
  stop=TRUE;    /* If one thread completes MAX_READS, all quit. */
                /* Presumably never get here. */
}




main(int argc, char **argv) {
  int           i, j;
  pthread_t     tid;
  double        rates[MAX_THREADS], S = 0.0, mean, rate_sum = 0.0;
  int           totalProcessed;


  PID = getpid();
  PTHREAD_ATTR_INIT(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  if (argc > 1) N_THREADS       = atoi(argv[1]);
  if (argc > 2) SPIN_TIME       = atoi(argv[2]);
  if (argc > 3) RUNTIME         = atoi(argv[3]);
  if (argc > 4) ITERATIONS      = atoi(argv[4]);
  printf("time_disk(N_THREADS %d SPIN_TIME %dms RUNTIME %ds ITERATIONS %d)\n",
         N_THREADS, SPIN_TIME, RUNTIME, ITERATIONS);
  if (N_THREADS>MAX_THREADS) {
    printf("Too many threads! Max is %d\n", MAX_THREADS);
    abort();
  }

  fd = open(path, O_RDONLY);
  if (fd == -1)   {printf("Check %s (see comments)!", path); exit(1);}
  length = lseek(fd, 0, SEEK_END);

  /* Should never be used, except when seed==1 (to read from mbufs)*/
  MAX_READS=RUNTIME*10000;

  if (SPIN_TIME>0)  calibrate_delay();
```

```
for (j=0; j<ITERATIONS; j++) {

  for (i=0; i<MAX_PAGES; i++) hits[i]=0;
  for (i=0; i<MAX_DENSITY; i++) density[i]=0;
  for (i=0; i<N_THREADS; i++) nProcessed[i]=0;
  totalProcessed=0;

  for (i=0; i<N_THREADS; i++)
    PTHREAD_CREATE(&threads[i], &attr, test, (void *) (j*1000 + i));

  iobench_start();
  sleep(RUNTIME);
  stop = TRUE;
  for (i=0; i<N_THREADS; i++) {pthread_join(threads[i], NULL);}
  iobench_end();

  for (i=0; i<N_THREADS; i++) {
    DEBUG(printf("%s processed \t%d\n",
                 thread_name(threads[i]), nProcessed[i]));
    totalProcessed += nProcessed[i];
  }

  for (i=0; i<MAX_PAGES; i++) {
    if (hits[i] < MAX_DENSITY)
      density[hits[i]]++;
    else
      DEBUG(printf("Page %d got %d hits\n", i, hits[i]));
  }

  DEBUG(printf("nHits \t nPages \n"));
  for (i=1; i<MAX_DENSITY; i++) {
    if (density[i]>0)
      DEBUG(printf("%d \t %d\n", i, density[i]));
  }

  if (ITERATIONS==1) iobench_report();
  if (ITERATIONS==1) printf("totalProcessed: %d, rate: %f/sec\n",
                            totalProcessed, totalProcessed / real_time);
  rates[j] =  totalProcessed / real_time;
  rate_sum += totalProcessed / real_time;
  stop=FALSE;
}

if (ITERATIONS>1) {
  mean = rate_sum/ITERATIONS;

  for (i=0; i<ITERATIONS; i++) S+=(mean-rates[i])*(mean-rates[i]);
  S = sqrt(S/(ITERATIONS-1));
  printf("Mean rate: %3.2f/sec, Standard Deviation: %3.2f\n", mean, S);
```

```
    }
}
```