Sample Application Code

A

The following sample programs give you an idea of how to use multithreading in a variety of ways.

File Copy

Generating several I/O requests at once so that the I/O access time can be overlapped is often advantageous. A simple example of this is file copying. If the input and output files are on different devices, the read access for the next block can be overlapped with the write access for the last block.
Code Example A-1 shows some of the code.

The main routine creates two threads: one to read the input, and one to write the output.

The reader thread reads from the input and places the data in a double buffer. The writer thread gets the data from the buffer and continuously writes it out. The threads synchronize using two counting semaphores; one that counts the number of buffers emptied by the writer and one that counts the number of buffers filled by the reader.

Note that the reader thread initializes semaphore emptybuf_sem because it needs a nonzero initial value. The writer thread need not explicitly initialize semaphore fullbuf_sem because it is allocated in zeroed memory.

Code Example A-1 File Copy Example With a Semaphore

```
sema_t emptybuf_sem, fullbuf_sem;

/* double buffer */
struct {
    char data[BSIZE];
    int size;
} buf[2];

reader()
{
    int i = 0;

    sema_init(&emptybuf_sem, 2, 0, NULL);
    while (1) {
    sema_wait(&emptybuf_sem);
    buf[i].size = read(0, buf[i].data, BSIZE);
    sema_post(&fullbuf_sem);
    if (buf[i].size <= 0)
        break;
    i ^= 1;
    }
}

writer()
{
    int i = 0;

    while (1) {
    sema_wait(&fullbuf_sem);
    if (buf[i].size <= 0)
        break;

    write(1, buf[i].data, buf[i].size);
    sema_post(&emptybuf_sem);
    i ^= 1;
    }
}
```

```
main()
{
    thread_t twriter;

    (void)thr_create( NULL, NULL, reader, NULL, THR_DETACHED,
NULL)
    (void)thr_create( NULL, NULL, writer, NULL, , &twriter, NULL);
    thr_join(twriter, NULL, NULL);
}
```

The example is a bit contrived because the system already generates asynchronous read-ahead and write-behind requests when accessing regular files. The example is still useful when the files to be copied are raw devices, since raw-device access is synchronous.

Matrix Multiplication

Computationally intensive applications benefit from the use of all available processors. Matrix multiplication is a good example of this.

When the matrix multiplication function is called, it acquires a mutex lock to ensure that only one matrix multiplication is in progress. This relies on mutex locks that are statically initialized to zero. The requesting thread checks whether its worker threads have been created. If not, it creates one for each CPU.

Once the worker threads are created, the requesting thread sets up a counter of work to do and signals the workers with a condition variable. Each worker selects a row and column from the input matrices, then updates the row and column variables so that the next worker will get the next row or column or both.

It then releases the mutex lock so that computing the vector product can proceed in parallel. When the results are ready, the worker reacquires the mutex lock and updates the counter of work completed. The worker that completes the last bit of work signals the requesting thread.

Code Example A-2 Matrix Multiplication

```
struct {
    mutex_t lock;
    cond_t start_cond, done_cond;
    int (*m1)[SZ][SZ], (*m2)[SZ][SZ], (*m3)[SZ][SZ];
    int row, col;
    int todo, notdone, workers;
} work;
mutex_t mul_lock;

void *
matmul(int (*m1)[SZ][SZ], int (*m2)[SZ][SZ], int (*m3)[SZ][SZ]);
{
    int i;

    mutex_lock(&mul_lock);
    mutex_lock(&work.lock);
    if (work.workers == 0) {
    work.workers = sysconf (_SC_NPROCESSORS_ONLN);
    for (i = 0; i < work.workers; i++) {
        (void)thr_create (NULL, NULL, worker, (void *)NULL,
                      THR_NEW_LWP|THR_DETACHED, NULL);
    }
    }

    work.m1=m1; work.m2=m2; work.m3=m3;
    work.row = work.col = 0;
    work.todo = work.notdone = SZ*SZ;
    cond_broadcast(&work.start_cond);
    while (work.notdone)
    cond_wait(&work.done_cond, &work.lock);
    mutex_unlock(&work.lock);
    mutex_unlock(&mul_lock);
}
void *
worker()
{
    int (*m1)[SZ][SZ], (*m2)[SZ][SZ], (*m3)[SZ][SZ];
    int row, col, i, result;
```

```
    while (1) {
    mutex_lock(&work.lock);
    while (work.todo == 0)
        cond_wait(&work.start_cond, &work.lock);
    work.todo--;
    m1=work.m1; m2=work.m2; m3=work.m3;
    row = work.row; col = work.col;
    work.col++;
    if (work.col == SZ) {
        work.col = 0;
        work.row++;
        if (work.row == SZ)
            work.row = 0;
    }
    mutex_unlock(&work.lock);
    result = 0;
    for (i = 0; i < SZ; i++)
        result += (*m1)[row][i] * (*m2)[i][col];
    (*m3)[row][col] = result;
    mutex_lock(&work.lock);
    work.notdone--;
    if (work.notdone == 0)
        cond_signal(&work.done_cond);
    mutex_unlock(&work.lock);
    }
}
```

Note that each iteration computed the results of one entry in the result matrix.

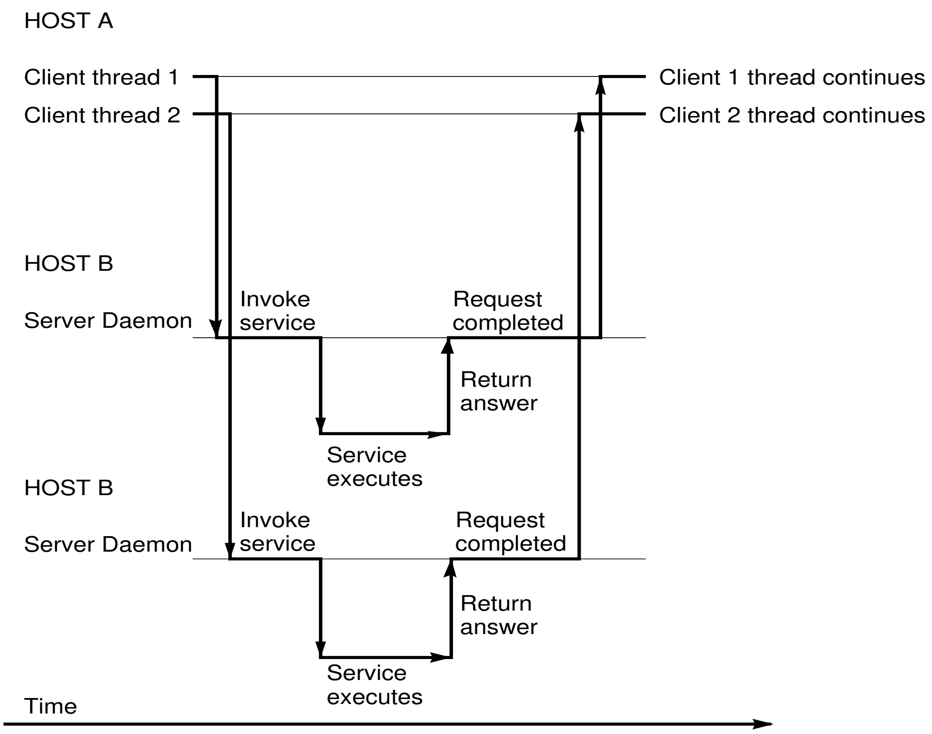
In some cases the amount of work is not sufficient to justify the overhead of synchronizing. In these cases it is better to give each worker more work per synchronization. For example, each worker could compute an entire row of the output matrix.

RPC Program

In a multithreaded client program, a thread can be created to issue each RPC request. When multiple threads share the same client handle, only one thread at a time can make a RPC request. The other threads must wait until the outstanding request is complete.

However, when multiple threads make RPC requests using unique client handles, the requests are carried out concurrently. Figure A-1 illustrates the possible timing of a multithreaded client implementation consisting of two client threads using different client handles.

Figure A-1 Two Client Threads Using Different Client Handles (Realtime)



Code Example A-3 shows the implementation of an `rstat` program with a multithreaded client and single-threaded servers. The client program creates a thread for each host. Each thread creates its own client handle and makes various RPC calls to a specified host. Because each client thread uses its own handle to make the RPC calls, the threads can carry out the RPC calls concurrently.

You can compile and run this program with:

```
% cc -D_REENTRANT -o example example.c -lnsl -lrpcsvc -lthread
% example host1 host2 host3...
```

Code Example A-3 RPC `rstat` Program With Multithreaded Client

```
/* @(#)rstat.c2.3 88/11/30 4.0 RPCSRC */
/*
 * Simple program that prints the status of a remote host, in a
 * format similar to that used by the 'w' command.
 */

#include <thread.h>
#include <synch.h>
#include <stdio.h>
#include <sys/param.h>
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>
#include <errno.h>

mutex_t tty;/* control of tty for printf's */
cond_t cv_finish;
int count = 0;
int nthreads = 0;

main(argc, argv)
    int argc;
    char **argv;
{
    int i;
    thread_t tid;
    void *do_rstat();

    if (argc < 2) {
    fprintf(stderr, "usage: %s \"host\" [...]\n", argv[0]);
    exit(1);
    }
```

```
    mutex_lock(&tty);

    for (i = 1; i < argc; i++) {
    if (thr_create(NULL, 0, do_rstat, argv[i], 0, &tid) != 0) {
        fprintf(stderr, "thr_create failed: %d\n", i);
        exit(1);
    } else
        fprintf(stderr, "tid: %d\n", tid);
    }
nthreads = argc - 1;
    while (count < nthreads) {
    printf("argc = %d, count = %d\n", nthreads, count);
    cond_wait(&cv_finish, &tty);

    }

    exit(0);
}

bool_t rstatproc_stats();

void *
do_rstat(host)
    char *host;
{
    CLIENT *rstat_clnt;
    statstime host_stat;
    bool_t rval;
    struct tm *tmp_time;
    struct tm host_time;
    struct tm host_uptime;
    char days_buf[16];
    char hours_buf[16];

    mutex_lock(&tty);
    printf("%s: starting\n", host);
    mutex_unlock(&tty);
```

```
/* client handle to rstat */
rstat_clnt = clnt_create(host, RSTATPROG, RSTATVERS_TIME,
            "udp");
if (rstat_clnt == NULL) {
mutex_lock(&tty);/* get control of tty */
clnt_pcreateerror(host);
count++;
cond_signal(&cv_finish);
mutex_unlock(&tty);/* release control of tty */

thr_exit(0);

}

rval = rstatproc_stats(NULL, &host_stat, rstat_clnt);
if (!rval) {
mutex_lock(&tty);/* get control of tty */
clnt_perror(rstat_clnt, host);
count++;
cond_signal(&cv_finish);
mutex_unlock(&tty);/* release control of tty */

thr_exit(0);

}

tmp_time = localtime_r(&host_stat.curtime.tv_sec,
            &host_time);

host_stat.curtime.tv_sec -= host_stat.boottime.tv_sec;

tmp_time = gmtime_r(&host_stat.curtime.tv_sec,
            &host_uptime);

if (host_uptime.tm_yday != 0)
sprintf(days_buf, "%d day%s, ", host_uptime.tm_yday,
(host_uptime.tm_yday > 1) ? "s" : "");
else
days_buf[0] = '\0';

if (host_uptime.tm_hour != 0)
sprintf(hours_buf, "%2d:%02d,",
    host_uptime.tm_hour, host_uptime.tm_min);
```

```
    else if (host_uptime.tm_min != 0)
    sprintf(hours_buf, "%2d mins,", host_uptime.tm_min);
    else

    hours_buf[0] = '\0';

    mutex_lock(&tty);/* get control of tty */
    printf("%s: ", host);
    printf(" %2d:%02d%cm up %s%s load average: %.2f %.2f %.2f\n",
    (host_time.tm_hour > 12)  ? host_time.tm_hour - 12

    : host_time.tm_hour,
    host_time.tm_min,
    (host_time.tm_hour >= 12) ? 'p'
    : 'a',
    days_buf,
    hours_buf,
    (double)host_stat.avenrun[0]/FSCALE,
    (double)host_stat.avenrun[1]/FSCALE,
    (double)host_stat.avenrun[2]/FSCALE);
    count++;
    cond_signal(&cv_finish);
    mutex_unlock(&tty);/* release control of tty */
    clnt_destroy(rstat_clnt);

    sleep(10);
    thr_exit(0);
}

/*
Client side implementation of MT rstat program
*/

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

bool_t
rstatproc_stats(argp, clnt_resp, clnt)
    void *argp;
    statstime *clnt_resp;
    CLIENT *clnt;
{
```

```
    memset((char *)clnt_resp, 0, sizeof (statstime));
    if (clnt_call(clnt, RSTATPROC_STATS,
    (xdrproc_t) xdr_void, (caddr_t) argp,
    (xdrproc_t) xdr_statstime, (caddr_t) clnt_resp,
    TIMEOUT) != RPC_SUCCESS) {
    return (FALSE);

    }
    return (TRUE);
}
```

Window System Server

A networked window system server tries to handle each client application as independently as possible. Each application should get a fair share of the machine resources, and any blocking on I/O should affect only the connection that caused it.

You could assure that each application gets a fair share of machine resources by allocating a bound thread for each client application. While this would work, it is wasteful since more than a small subset of the clients are rarely active at any one time.

Allocating an LWP for each connection ties up large amounts of kernel resources basically for waiting. On a busy desktop, this can be several dozen LWPs. (A window system server designed to run with a single-level threads model would have different considerations about kernel resources and could be designed quite differently.)

The code shown in Code Example A-4 takes a different approach. It allocates two unbound threads for each client connection, one to process display requests and one to write out results.

This approach allows further input to be processed while the results are being sent, yet it maintains strict serialization within the connection. A single control thread looks for requests on the network. The relationship between threads is shown in Figure A-2.

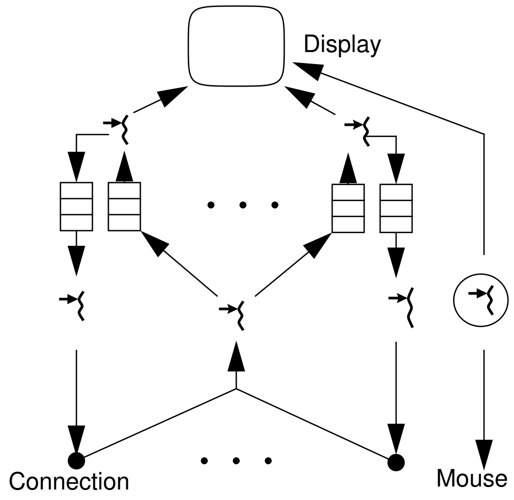


Figure A-2 Window Server Threads

With this arrangement, an LWP is used for the control thread and whatever threads happen to be active concurrently. The threads synchronize with queues. Each queue has its own mutex lock to maintain serialization, and a condition variable to inform waiting threads when something is placed on the queue. A bound thread processes mouse events to provide a quick response to inputs.

Code Example A-4 Window Server

```
main ()
{
    /* set up server and listen port */
    for(;;) {
    poll(&fds, nfds, 0);
    for (i = 0; i < nfds; i++) {
        if (fds[i].revents & POLLIN)
            checkfd(fds[i].fd)
    }
    }
}

checkfd (int fd)
{
    struct connection *connp;

    if (fd == listenfd) {
    /* new connection request */
    connp = create_new_connection();
    (void)thread_create (NULL, NULL, svc_requests, connp,
                THR_DETACHED, NULL);
    (void) thread_create (NULL, NULL, send_replies, connp,
                 THR_DETACHED, NULL);
    } else {
    requestp = new_msg();
    requestp->len =
        t_rcv (fd, requestp->data, BUFSZ, &flags);
    connp = find_connection (fd);
    put_q (connp->input_q, requestp);
    }
}

send_replies (struct connection *connp)
{
    struct msg *replyp;

    while (1) {
    replyp = get_q (connp->output_q);
    t_snd (connp->fd, replyp->data, replyp->len, &flags);
    }
}
```

```
svc_requests (struct connection *connp)
{
    struct msg *requestp, *replyp;

    while (1) {
    requestp = get_q (connp->input_q);
    replyp = do_request (requestp);
    if (replyp)
        put_q (connp->output_q, replyp);
    }
}

put_q (struct queue *qp, struct msg *msgp)
{
    mutex_lock (&qp->lock);
    if (list_empty (qp->list))
    cond_signal (&qp->notempty_cond);
    add_to_tail (msgp, &qp->list);
    mutex_unlock (&qp->lock);
}


struct msg *
get_q struct queue *qp)
{
    struct msg *msgp;

    mutex_lock (&qp->lock);
    while (list_empty (qp->list))
    cond_wait (&qp->notempty_cond, &qp->lock);
    msgp = get_from_head (&qp->list);
    mutex_unlock (&qp->lock);
    return (msgp);
}
```