Writing Multithreaded Code in Solaris

Steven Kleiman, Bart Smaalders, Dan Stein, Devang Shah

SunSoft Inc.
Mountain View, California

Abstract

SunOS 5.0 is the operating system component of Solaris 2.0. SunOS 5.0 contains the kernel support for multiple threads of control in a single process address space. This allows a single application to efficiently overlap I/O operations and to take advantage of more than one processor, if available. We describe some of the issues in using and converting libraries to the multithreaded environment. In addition, we give several example of different uses of threads in user applications.

1: SunOS 5.0 MT architecture

SunOS 5.0 is the operating system component of Solaris 2.0. SunOS 5.0 contains the kernel support for multiple threads of control in a single process address space. In the SunOS multithread (MT) architecture [1] threads are lightweight abstractions implemented by a thread library. The library controls how threads are scheduled onto lightweight processes (LWPs) which are the independent execution entities within the process and are supported by the kernel. This allows many hundreds of threads to exist in a process while the number of LWPs can be tailored to the actual concurrent need for system resources. The overall architecture is shown in Figure 1.

In many cases, an application need not be aware of the number of LWPs used as the library creates as many LWPs as necessary to avoid deadlock due to lack of execution resources. However, this may not be the optimal number for performance. When required, the size of the pool of LWPs used to schedule threads can be controlled by the application. Threads can also be bound to LWPs when there is some aspect of an LWP that is required by a thread, such as system-wide, real-time priority. An analogous situation is the `stdio` package whose interfaces provide an efficient, buffered interface that can be tailored by the application.

In general, the use of threads by a process is not visible from outside the process.

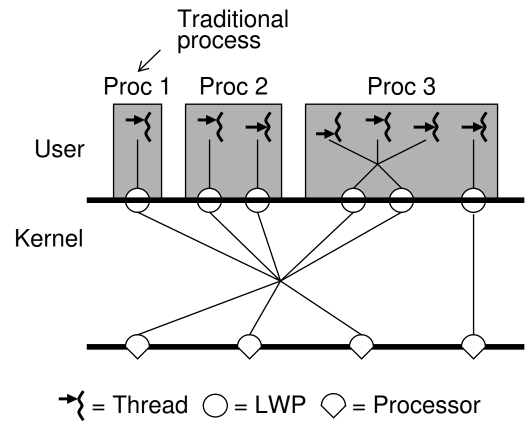

Figure 1: SunOS 5.0 MT architecture

1.1: Synchronization

Threads synchronize via a variety of synchronization primitives, such as:

- Mutual exclusion (mutex) locks
- Condition variables
- Counting semaphores
- Multiple reader, single writer (readers/writer) locks.

The synchronization primitives can be allocated statically in structures and need only be initialized to zero to achieve correct default behavior. They can also be used in a memory-mapped file that is shared between processes.

2: Multithreaded libraries

The general goal of converting existing libraries to a multithreaded environment is to provide correct operation when a library interface is entered by more than one thread simultaneously (i.e. the library is “MT-safe”). In addition, it is usually desirable that long operations such as I/O not block other threads from using the library while the operation completes. In libraries that are not computationaly intensive (most system libraries), it is much less important to allow many processors to execute library code simulta-

```
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_p(&emptybuf_sem);
        buf[i].size =
            read(0, buf[i].data, BSIZE);
        sema_v(&fullbuf_sem);
        if (buf[i].size <= 0)
            break;
        i ^= 1;
    }
}
```

```
writer()
{
    int i = 0;

    while (1) {
        sema_p(&fullbuf_sem);
        if (buf[i].size <= 0)
            break;
        write(1, buf[i].data, buf[i].size);
        sema_v(&emptybuf_sem);
        i ^= 1;
    }
}

main()
{
    thread_id_t treader, twriter;

    treader = thread_create(NULL, NULL,
        reader, NULL, THREAD_NEW_LWP);
    twriter = thread_create(NULL, NULL,
        writer, NULL,
        THREAD_NEW_LWP | THREAD_WAIT);
    thread_wait(twriter);
}
```

Figure 2: File copy

neously. In some libraries the interfaces cannot work effectively in a multithreaded environment and they must be changed.

2.1: System interfaces

POSIX P1003.4a [2] has defined reentrant versions of the POSIX P1003.1 system interfaces. In most cases the interfaces are either completely reentrant or any locking for shared data can be hidden in the routine implementation. A good example of the latter is `malloc()`. Different threads can simultaneously enter `malloc()` and the implementation provides enough synchronization so that the threads don't interfere with each other and each thread returns with an independent allocation of memory.

In some cases the interface is inherently non-reentrant. A good example of this is `errno`. If one thread makes a system call which sets `errno`, then the value in `errno` can be changed by another thread making another system call. POSIX.4a defines `errno` to be uniquely allocated to each thread, so that threads making simultaneous system calls don't interfere with each other.

Another example of this is `getpwnam()`. This interface returns a pointer to a static data area. If a second thread enters `getpwnam()` before the thread that called `getpwnam()` first has completely consumed the entry in the static buffer, the entry could be overwritten. One solution is to put the buffer in a thread specific data area. This allows threads to call `getpwnam()` independently. This approach has the disadvantage that a data area must be allocated for each thread that uses the interface.

An alternative approach is to define new, reentrant interfaces to these functions. For functions that return pointers to static data areas, the interface can be changed to have the caller pass in pointer(s) to the memory in which the results can be stored. This is the approach taken by POSIX.

For example, POSIX defines a new interface, `getpwnam_r()`, which takes three additional arguments; a pointer to a `struct passwd` for the result, a buffer in which strings pointed to by the returned `struct passwd` are placed, and the size of the supplied buffer. This approach keeps the per-thread storage to a minimum and it allows the calling function to manage the required memory as appropriate.

In the cases where a new interface has been defined, the old interfaces still remain. They are still usable provided they are either called from a single thread or the application provides the appropriate locking before calling any of these routines.

Some interfaces can be made reentrant, but the overhead involved in hiding the required locking beneath the interface is too great. An example is the `stdio` library function `putc()`. By default, `putc()` is implemented with the required locking of the I/O buffers. However the overhead of locking and unlocking the I/O buffers for each character can be too great in some situations. POSIX defines three new interfaces to help in these situations; `flockfile()`, `funlockfile()`, and `putc_unlocked()`. The first two interfaces serialize multiple access to a `FILE`. Once the `FILE` is locked, `putc_unlocked()` outputs characters un-

```
struct {
    mutex_t lock;
    condvar_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;

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

    mutex_enter(&mul_lock);
    mutex_enter(&work.lock);
    if (work.workers == 0) {
        for (i = 0; i < NCPU; i++) {
            thread_create(NULL, NULL,
              worker, (void *)NULL,
              THREAD_NEW_LWP);
        }
        work.workers = NCPU;
    }
    work.m1=m1; work.m2=m2; work.m3=m3;
    work.row = work.col = 0;
    work.todo = work.notdone = SZ*SZ;
    cv_broadcast(&work.start_cond);
    while (work.notdone)
        cv_wait(&work.done_cond, &work.lock);
    mutex_exit(&work.lock);
    mutex_exit(&mul_lock);
}
```

```
worker()
{
    int (*m1)[SZ][SZ], (*m2)[SZ][SZ],
        (*m3)[SZ][SZ];
    int row, col, i, result;

    while (1) {
        mutex_enter(&work.lock);
        while (work.todo == 0)
            cv_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_exit(&work.lock);
        result = 0;
        for (i = 0; i < SZ; i++)
            result +=
              (*m1)[row][i] * (*m2)[i][col];
        (*m3)[row][col] = result;
        mutex_enter(&work.lock);
        work.notdone--;
        if (work.notdone == 0)
            cv_signal(&work.done_cond);
        mutex_exit(&work.lock);
    }
}
```

Figure 3: Matrix multiply

til the `FILE` is unlocked. This allows the application to control the locking granularity to suit its needs.

A good discussion of the trade-offs in making libraries MT-safe can be found in [3].

3: Multithreading examples

The remainder of this paper is several examples of situations in which threads can be used effectively. The code shown in the figures is somewhat sketchy due to space limitations and should be taken as an outline. The thread interfaces used are described in [1].

3.1: File copy

On either a uniprocessor or a multiprocessor it can be advantageous to generate several I/O requests at once so that the I/O access time can be overlapped. A simple example of this is file copying. If the input file and the output file are on different devices the read access for the next block can be overlapped with the write access for the last block. Figure 2 shows some of the code.

The main routine creates two threads; one to read the input, one to write the output. Each `thread_create()` also adds an LWP to the pool of LWPs upon which threads can be scheduled (`THREAD_NEW_LWP`), since the application will require full system resources for each thread. This is an optimization since the library ensures that the threads will make progress. Note that the LWPs are not permanently bound to the thread so the threads package can destroy any that are not utilized.

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.

The example is somewhat contrived in that normally the system already asynchronously generates read-ahead requests and write blocks behind when accessing regular files. The example is still useful if the files to be copied are raw devices, since raw device access is synchronous.

```
main(int argc, char *argv[])
{
    /* initialize gui and # of hosts */
    for (i = 0; i < hosts; i++) {
        thread_create(NULL, NULL,
          do_host, argv[i+1], 0);
    }
    run_gui(); /* only returns when done */
    exit(0);
}
```

```
do_host(char *host)
{
    meter_t meter = init_meter(host);

    while (1) {
        client =
          get_rstat_clnt(metername(meter));
        if (client == NULL) {
            meter_down(meter);
            /* don't thrash */
            sleep(sleeptime);
            continue;
        }
        while (
          clnt_rstat_call(client,&stat)) {
            update_meter(meter, &stat);
            sleep(sleeptime);
        }
        clnt_destroy(client);
        meter_down(meter);
    }
}
```

Figure 4: RPC client

3.2: Matrix multiply

Computationally intensive applications benefit from the use of all available processors. Matrix multiplication is a good example of this; see Figure 3.

When the matrix multiply is called, it acquires a mutex to ensure that only one matrix multiply is in progress. This relies on mutexes that are statically initialized to zero. The requesting thread then checks whether its worker threads have been created. If not, it creates one for each CPU. Once the worker threads have been created, it sets up a counter of work to do and then signals the workers via a condition variable. Each worker picks off a row and column from the input matrices then updates the counter of work so that the next worker will get the next item. It then releases the mutex so that computing the vector product can proceed in parallel. When the results are ready, the worker reacquires the mutex and updates the counter of work completed. The worker that completes the last bit of work signals the requesting thread.

Note that each iteration computed the results of one entry in the result matrix. In some cases this 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.

3.3: RPC client

Windowing applications that are also RPC clients have traditionally had trouble maintaining acceptable levels of interactivity if communication with the RPC server is slow or intermittent. By using threads the application can ensure that the windowing code continues to process user events and repaint the screen during long duration RPC calls.

A simple example of this application is the multi-host graphical CPU monitor shown in Figure 4. Here, the main thread creates as many threads as there are hosts to be monitored. The main routine then runs the window system event loop until the application terminates. Each host thread attempts to build a RPC client handle to its host, flagging the host as down on the display if this fails. Once a client handle has been successfully created, the thread performs the RPC call, updates its meter on the screen, and sleeps until the update period has elapsed. If the RPC call fails, the host is marked as down, the client handle destroyed, and the thread starts trying to build a new client handle.

This example relies on MT-safe RPC client and window system toolkit libraries. A simple MT-safe RPC library allows multiple requests on different client handles, but only allow a single request at a time on the same client handle. One could also construct an RPC library which used helper threads that actually wrote the requests and waited for replies via poll(). This could substantially reduce the amount of system resource required since only one or two LWPs would be required for the I/O threads.

The window system toolkit can use a fairly simple locking technique since the actual amount of time spent in the toolkit by any of the host threads is very small. This is discussed in detail in [5].

```
sema_t throttle;

main(int argc, char ** argv)
{
    /* set up and register server */
    sema_init(&throttle, MAX_BANDWIDTH,
      0, NULL);
    while (1) {
        poll(&fds, nfds, -1);
        for (i = 0; i < nfds; i++)
            if (fds[i].revents & POLLIN)
                checkfd(fds[i].fd);
        }
    }
}
```

```
checkfd(int fd)
{
    sema_p(&throttle);
    if (islistenfd(fd))
        thread_create(NULL, NULL,
            create_new_connection, fd, 0);
    else
        thread_create(NULL, NULL,
            service, fd, 0);
}
service(int fd)
{
    rpc_msg in, out;

    read_msg(fd, &in);
    /* handle request and format response*/
    write_msg(fd, &out);
    sema_v(&throttle);
}
```

Figure 5: RPC Server

3.4: RPC server

RPC servers have used various techniques to support long duration requests. These include forking and having the child processes handle the reply, relaying long requests to sub-processes, and RPC callbacks. Threads allow us to use a much simpler model to handle multiple pending requests. In Figure 5, we can see a simple RPC service that performs some unspecified task.

The main thread initializes the server and a counting semaphore, and sits in a poll loop. When a request comes in a new thread is created to handle it. This takes advantage of the relatively lightweight cost of thread creation in the user process. The semaphore is used to prevent a flood of service requests from creating too great a load on the system due to service processing. In the case where all the threads are already busy the main thread blocks until a service thread exits. Note that each service thread handles its own reads and writes. If a client doesn't empty the stream fast enough, the service thread will block on the write call.

3.5: 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. This can be done by allocating a bound thread for each client application. While this would work, it is wasteful in that it is rare that more than a small subset of the clients are 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.

The code shown in Figure 6 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 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 7.

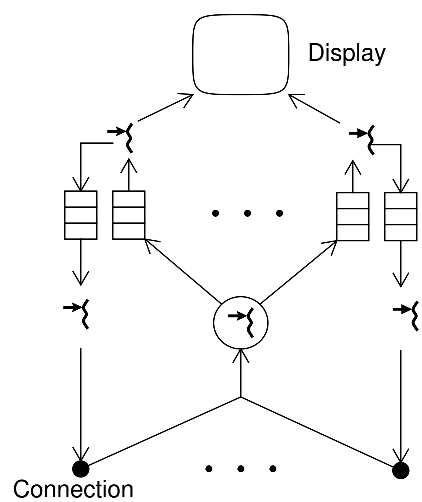

Figure 7: Window server threads

With this arrangement, an LWP is used for the control thread and for whatever number of threads happen to be active concurrently. The threads synchronize via queues. Each queue has its own mutex to maintain serialization, and a condition variable to inform waiting threads when something is placed on the queue.

4: Threads interfaces

The threads library prototype contains the threads interfaces described in [1]. These will be converted to be com-

```
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();
        thread_create(NULL, NULL,
            svc_requests, connp, 0);
        thread_create(NULL, NULL,
            send_replies, connp, 0);
    } 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_enter(&qp->lock);
    if (list_empty(qp->list))
        cv_signal(&qp->notempty_cond);
    add_to_tail(msgp, &qp->list);
    mutex_exit(&qp->lock);
}

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

    mutex_enter(&qp->lock);
    while (list_empty(qp->list))
        cv_wait(&qp->notempty_cond,
          &qp->lock);
    msgp = get_from_head(&qp->list);
    mutex_exit(&qp->lock);
    return (msgp);
}
```

Figure 6: Window server

pliant with the similar SVR4 interfaces (derived from [1]) described in [4] and will be made available along with MT-safe libraries in the future. When POSIX P1003.4a has completed the standardization process, the POSIX `pthreads` interfaces will be made available, in addition.

5: References

[1] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks. "SunOS Multi-thread Architecture", Proceedings of the Winter 1991 USENIX Conference.

[2] POSIX P1003.4a Draft 5, IEEE.

[3] M.B. Jones. "Bringing the C Libraries With Us into a Multi-Threaded Future", Proceedings of the Winter 1991 USENIX Conference.

[4] UNIX System Laboratories. "UNIX System V Release 4 ES/MP Multiprocessing Detailed Specifications".

[5] B. Smaalders, B. Warkentine, K. Clarke. "Prototyping MT-safe Xt and XView libraries", Proceedings of the 6th Annual Conference on the X Window System, 1992.