1994

Ariadne User Manual Version 2.0

Edward Mascarenhas

Vernon J. Rego
_Purdue University_, rego@cs.purdue.edu

Vaidy Sunderam

Report Number:
94-081


This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
1 Introduction

Ariadne or Aria-Threads is a user level-threads package that provides light-weight processes or threads as a basic unit of computation. Its design was motivated by the need for portability and thread mobility on multiprocessor systems. Its design is conceptually simple, partly motivated by systems such as Rex and SunLWP, utilizing standard Unix primitives to enable context-switching and synchronization, and partly based on the need for thread portability and mobility in the ACES system.

Each Aria thread runs on an independent stack and can receive up to sixteen arguments (integers, floats or doubles\(^1\)) as parameters from its creator. CPU control is switched from one executing thread to another using the rule that, at any given time, the highest-priority runnable thread must execute. Synchronization may be done using either thread-priority or counting semaphores, and time-slicing may be done between runnable threads of the same priority. Ariadne provides a novel thread-migration feature, supporting transparent or user-level thread movement across machines in a homogeneous environment.

Ariadne is small and very portable. Almost all code is written in the C programming language, with thread initialization code in assembler. Context-switching is accomplished using setjmp() and longjmp() primitives. Ariadne is currently supported on Sun4, Intel iPSC, Sequent Symmetry, SGI, and IBM RS/6000 machines. Ports to other architectures are possible with little effort. Thread migration has currently been implemented on the Sun4, Sequent Symmetry and Intel iPSC environments.

2 Ariadne Initialization and Termination

Aria threads are dynamic structures based on minimal context areas and run-time stacks. Each thread created within Ariadne is associated with the two-component structure

\[
\text{struct thread_t} \{ \\
\text{int} \quad \text{thread_id;} \\
\text{char} \quad *\text{thread_key}; \\
\};
\]

where id is a unique integer identifying a thread, and key is a pointer to its context. The Ariadne library is initialized using the call:

\[
\text{ariadne(struct thread_t *} \text{t_ptr, int t_priority, int t_unit_stacksize);}\]

\(^1\) Currently, only integer and double arguments are supported on the Intel i860 and IBM RS/6000 processors.
or the call:

```c
ariadne(struct thread_t *t_ptr, int t_priority, DEFAULTSIZE);
```

where

- `t_ptr` is a pointer to a thread struct which identifies the thread,
- `t_priority` is a non-zero integer defining the priority of the invoking routine (i.e., `main()`), and
- `t_unit_stacksize` is an integer defining the stack-size of the smallest basic stack on which a thread can run.

Note that Ariadne primitives will not function correctly until the thread system is initialized through the `ariadneO` call. (`ariaO` may also be used instead of `ariadneO`). Thread priorities are integers in the range 1 through 8. Larger integers correspond to higher priorities.

The Ariadne thread-system is terminated through the call

```c
void a_exit(void)
```

which must be placed in `main()`. When control returns to the initiating program through a return from `a_exit()`, it can be assumed that subsequent execution of the program is thread-free.

### 2.1 Stack sizes

Ariadne provides the user with five basic stack types. These are identified by the keywords `TINY`, `SMALL`, `MEDIUM`, `LARGE` and `HUGE`. During initialization, the user-defined `t_unit_stacksize` defines the size `TINY`. Thereafter, the size of each type is twice the size of the previous type, so that `SMALL` is twice the size of `TINY`, and `MEDIUM` is twice the size of `SMALL`, etc. The keyword `DEFAULTSIZE` in the initialization call defines `TINY` to be 8192 bytes.

A stack size other than one of the five basic types described above can be obtained by specifying the required integer size as the stack-size argument in the `a_create()` call. These are called user-defined stacks.

A thread that is bound to a stack of a certain type is said to be a thread of that stack type. Thus, a thread bound to a `TINY` stack is a thread of stack type `TINY`. 
Common Stacks - stack type COMMON

Ariadne also supports several threads that can use a common stack. An internal stack defined for this purpose is shared among threads that are created by using the keyword COMMON. This stack size is presently set to 64K. When such a thread runs its stack (saved in a private area) is copied to the common stack area, and the thread can then continue execution. Note that there is an overhead involved with the use of the common stack due to the need to copy stacks to and from the common area and a thread private area. For most applications errors due to stack overflow can be avoided by using the common stack. Also the run-time requirements of memory is reduced with the use of the common stack.

Stack free-lists

Ariadne maintains its own free lists, with a user-defined limit on how many threads of a given stack type are to be maintained in each list. Dead threads are not returned to the system but instead are placed on these free lists for possible reuse. The call

\[
\text{a_buffer(basic\_size, length);}\
\]

defines a free list with a maximum of length threads, each with a stack of size basic\_size.

Thus, for example,

\[
\text{a\_buffer(TINY, 1000)};\
\]
\[
\text{a\_buffer(HUGE, 10)};\
\]

requests a free list containing a maximum of 1000 threads with TINY stacks, and a free list containing a maximum of 10 threads with HUGE stacks. During creation, threads are obtained from free lists whenever possible, thereby reducing the number of calls to `malloc()` (and consequently to `free()`) routines. Free lists are not maintained for user-defined stacks.

The default maximum buffering is DEFAULTBUFFER (currently 1000) threads for each basic stack. The call

\[
\text{a\_buffer(ALL, DEFAULTBUFFER);}\
\]

ensures that free lists are set up for each of the basic types, with a DEFAULTBUFFER maximum size.
Warning
Currently, there is no provision for recovery from stack errors which arise from choice of a smaller than required stack size. Should such stack errors occur, the stack size for each thread, in particular the one causing the error, should be increased. Ariadne provides some minimal protection against such segmentation errors in the form of warnings (i.e., [STACK WARNING] id) giving the id of the offending thread. This warning is given whenever an executing thread takes control with remaining stack-space less than TOLERANCE bytes. TOLERANCE is currently set at 800 bytes. If stack overflow occurs during a thread’s execution, one of two things happens. The thread may execute with a memory access that is probably incorrect, with a possible warning message given on the next context-switch to the thread. Alternately, a segmentation error may occur. In either case, this is an indication that stack sizes should be increased.

2.2 Thread limits
For each basic-stack or user-defined stack type, Ariadne allows the user to set a limit on the maximum number of threads that can co-exist in the system. If such a user-defined limit is reached, Ariadne terminates with the message "[THREAD LIMIT: type] id", where "type" indicates which one of the basic stack-types or user-defined stacks (USER_STK) has exceeded the predefined limit, and "id" is the identity of the offending (creating) thread. For example, the calls

```
a_limit(TINY, 1000);
a_limit(USER_STK, 200);
```

set the maximum number of TINY threads and the maximum number of threads with user-defined stack sizes that can co-exist in the system to be 1000 and 200, respectively.

The default limit is DEFAULTLIMIT (currently 1000) threads for each of the basic stack types or for user-defined stacks. The call

```
a_limit(ALL, DEFAULTLIMIT);
```

sets the limit to DEFAULTLIMIT for threads of all stack types.

3 Thread creation and destruction
Ariadne threads are created using the call
a_create(t_ptr,t-name,t_priority,t_size,j,k,m,i_1,...,i_j,f_1,...,f_k,d_1,...,d_m);

where

- **struct thread_t *t_ptr** is a pointer to a two-component structure identifying the thread,
- **(void) (*t_name)()** is the name of a function representing the thread,
- **int t_priority** is a non-zero integer (between 1 and 8) defining the priority of the thread,
- **int t_size** is a size (as explained in Section 2.1),
- **int j,k,m** are the number of integer, float and double arguments, respectively, and
- **i_1,...,i_j,f_1,...,f_k,d_1,...,d_m** represent the integer, float and double arguments. The total number of arguments is currently limited to 16, i.e., \( i + j + k \leq 16 \).

When a thread is created with a priority that is higher than the priority of its creator, the creator is placed on the ready queue, and the newly created thread is given control. Otherwise, the newly created thread is placed on the ready queue, and the creator continues to run. It is always the case that the thread in control is the highest priority thread in the system, or belongs to the highest priority class in the system.

A thread is destroyed by invoking the function

\[ \text{a_destroy(struct thread_t thr)} \]

which returns the dead thread to the free list. The invoker may destroy any thread, including itself. The target thread is removed from the execution pool and placed on a free list. If the invoker destroys itself, control transfers to the next thread in line to execute. If a thread attempts to destroy a nonexistent thread, a warning (such as [DESTROY WARNING] id) is printed, giving the id of the offending thread (i.e., one attempting the destroy operation), and control returns to the invoker.

### 4 Control switching

Ariadne allows an executing thread to relinquish control to another specific thread through use of the primitive
a_yield (struct thread_t thr);

where thr identifies the target thread. If the priority of the target thread is not less than the priority of the invoking thread, control transfers to the target thread. Otherwise, the yield fails, and control simply returns to the invoker. If the target thread does not exist, the yield fails, a warning message (such as [YIELD WARNING] id) is printed, giving the id of the offending (invoking) thread, and control returns to the invoker.

If a thread simply wants to yield to the “next” thread in its own priority class, without having to specify an id for the target thread, the function

a_yieldc();

forces the required reschedule. If all other existing threads are of lower priority, the invoker regains control from the scheduler.

4.1 Semaphores

Semaphores are used to provide synchronization or mutual exclusion for the execution of critical sections, and may cause transfer of control between threads. Calls are available to create a counting semaphore which can be initialized to any value, to wait on a semaphore and to signal a semaphore either once or several times.

struct sem *a_creates(int count);

Allocates memory for the semaphore structure and creates a semaphore initialized to the value count.

void a_waits(struct sem *s);

The executing thread waits on the semaphore s if the call determines the semaphore count to be zero or less.

void a_signals(struct sem *s);

Signals the semaphore once. If a thread is currently blocked on this semaphore, the thread becomes ready for execution. It executes immediately if its priority is higher than the priority of the signaling thread.
void a_signalns(struct sem *s, int k);

Signals the semaphore k times. The highest priority runnable thread will execute.

int a_counts(struct sem *s);

Returns the count of the semaphore.

4.2 Processor Sharing

Ariadne allows threads to share a processor through time-slicing. That is, control passes from one thread to another, within the same (highest) priority class after each thread has received a service quantum from the CPU. Allocation is done in round-robin fashion. Two types are provided for processor sharing. The function

a_defslice (int type, int pri, int sec, int usec):

defines the length of a time slice (in units of seconds and microseconds) for priority class pri. Time-slicing can be turned off for a given priority class by setting sec = usec = 0 for this class. The first parameter type determines whether processor sharing is based on actual time or on execution time. The type parameter is either

REAL: time-slice length corresponds to real-time, or

EXEC: time-slice length corresponds to execution time.

For example, the call

a_defslice(REAL, 6, 0, 100);

sets the real-time time-slice length for priority class 6 to be 100 usecs, and the call

a_defslice(EXEC, 5, 0, 0);

sets time-slicing off for threads in priority class 5. In this case, threads are assigned to the CPU in FIFO order.
4.3 Thread Suspension for interval

Ariadne threads can be suspended from execution for a specified interval in seconds and microseconds. However the actual suspension interval may be an arbitrary amount longer than that specified due to other processes in the system. Time-slicing based on the REAL (wall-clock time) criterion may not be on while threads are being suspended using the primitive defined here. The call

\[
\text{void a_sleep(long seconds, long microseconds);}\]

suspends the current thread from execution for the specified interval. Some other ready thread that has the highest priority is given control.

5 Utilities

Ariadne currently provides a small set of generic calls for thread manipulation.

\[
\text{int a_setpri(struct thread_t thr, int newpri);}\]

allows a thread to reset its own priority or the priority of another thread. It assigns a priority of \text{newpri} to the specified thread \text{thr} and returns the old priority of this thread to the invoker. If the specified thread does not exist, a warning message (such as [SETPRI WARNING] id) is printed, giving the id of the offending thread (i.e., one attempting the priority reset), and control returns to the invoker. If \text{newpri} exceeds the maximum allowable priority (currently 8), it is replaced by the maximum priority. If \text{newpri} is greater than the priority of the invoking thread, the invoking thread is retired to the ready queue, and control transfers to thread \text{thr}.

\[
\text{int a_mypri();}\]

allows a thread to determine its own priority.

\[
\text{int a_ping (struct thread_t thr);}\]

allows a thread to determine if another thread is still alive. It returns a 0 if the specified thread is alive, and 1 otherwise.
int a_suicide();

allows a thread to remove itself from the execution pool. It is placed on a free list.

void a_self(struct thread_t thr);

allows a thread to determine its own id through a subsequent access to thr.thread_id.

void a_previous(struct thread_t thr);

allows a thread to determine the id of its predecessor at the CPU, through a subsequent access to thr.thread_id. Note that the id returned may be the id of a system thread (internal to Ariadne) and not a user-thread, if the last thread to use the CPU was a system thread.

int a_myid();

returns the integer id of the invoking thread.

int a_interruptoff();

Returns the interrupt mask at the time of the call and disables time slicing if it was set on. This call is necessary and should be made whenever non-entrant routines are called, including calls from the standard i/o library.

void a_interrupton(int mask);

This sets the signal mask to the value of the parameter. In general it is used to re-enable time slicing if it was disabled earlier.
6 Program execution (Single Process)

A minimal Ariadne program contains the following code.

```c
#include "aria.h"
main()
{
    struct thread_t main_t;
    int main_pri;

    /* program initialization */
    ariadne(&main_t, main_pri, DEFAULTSIZE);

    /* Body of the program may create and manage threads */
    a_exit();
    /* print results if required */
}
```

Programs using Aria-Threads are written in standard C or C++, with a main program `main(int argc, char **argv)`. Ariadne does its own global initialization before calling the user's `main()` function. `Main()` is turned into a thread upon return from the `ariadne()` function call. Therefore `ariadne()` should be the first Ariadne function called, to initialize the thread system. The last `ariadne` call should be `a_exit()`. Any subsequent processing after this call would take place in a thread-free process.

The thread system terminates when `a_exit()` is invoked, or when all user-threads have completed execution. `Main()` may continue to execute after a return from `a_exit()`, if necessary. Upon completion, Ariadne performs some cleanup, and prints runtime statistics in a file called "aria.log".

7 Ariadne in a Distributed Memory Environment

Ariadne can be customized by the user to run in a distributed environment selected by the user. PVM is an example of such an environment that provides a message passing mechanism over a distributed virtual machine. Such systems allow the user to define a machine configuration and provide for the automatic creation and destruction of remote processes. In distributed environments access of remote data is one of the important issues. The use of the communication primitives, for example send and receive, allow data to be
shipped from one process to another. Ariadne provides another method for data access - by migrating the thread to the location of the data.

7.1 Thread migration

With the help of communication primitives, Ariadne allows users to migrate executing threads across different machines. ² Ariadne currently exists in the form of a library. The file libthread.a is a stand-alone library with no support for thread migration. The file libthreadd.a is a library that can be combined with the communications library for low-level thread-migration support. A thread may migrate itself from a host process to another process through the call

```c
void a_migrate(int process_id)
```

where `process_id` is an integer representing the id of the target process, possibly on a different machine³. Execution of the migrant thread resumes on the target process at the point of return from `a_migrate()`. The state of the thread is preserved during migration (i.e., all local variables are left intact and available when the thread resumes on the destination machine). All references to global variables after the migration will involve instances of the global variables on the destination machine. To update pointer variables to local data on the destination machine, however, it is necessary to invoke the function

```c
void a_updatep(int *pvar);
```

where `pvar` is the pointer address of the object of interest. This pointer is valid on the destination machine upon a return from this call.

For example:

```c
foo()
{
    int i;
    int *ip = &i;

    /* processing, before migration */
```

² Currently, thread migration is possible with the Conch and PVM message passing libraries.
³ For example, Conch processes are numbered 0,1,2,... etc.
The Ariadne system maintains unique thread identifiers for all threads in the distributed system. Thread ids for distributed threads are distinct from local thread ids, and can be determined through the call

```c
int a_mydid(void)
```

which returns a unique integer (over the distributed thread-system) identifying the invoking thread.

### 7.2 Building a Distributed Threads Environment

Support is available to the user to implement distributed functions like thread migration, and distributed termination. The user must provide functions based on the communications system for remote creation of processes, distributed shutdown, reading messages from other processes in the system, etc. These functions may be added for other communication systems at low programming cost, using existing implementations as examples. The minimal number of functions that may be required to be developed for the use of Ariadne with a new communications system are given in Table 1. For the Conch library these functions are in the file `aria_conch.c` and for PVM in `aria_pvm.c`. The corresponding object files for the particular environment must be linked with the application object files when using these environments.

The support functions are given in the Appendix.
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_dinit()</code></td>
<td>Initializes the distributed environment</td>
</tr>
<tr>
<td><code>a_dexit()</code></td>
<td>Shutdown of system/processes</td>
</tr>
<tr>
<td><code>a_getprocid()</code></td>
<td>Returns the process identifier of the process</td>
</tr>
<tr>
<td><code>a_getnumprocs()</code></td>
<td>Returns the number of processes in the system</td>
</tr>
<tr>
<td><code>a_dtermiate_check()</code></td>
<td>Check for distributed termination</td>
</tr>
<tr>
<td><code>(*a_usr.cswitch_process)()</code></td>
<td>A pointer to a user written function that is called to perform special actions after a context switch</td>
</tr>
<tr>
<td><code>a_migrate()</code></td>
<td>Migrate the calling thread</td>
</tr>
<tr>
<td><code>recv_any_messages()</code></td>
<td>Receive and handle a thread migration, destruction, or process termination message</td>
</tr>
<tr>
<td><code>a_printf()</code></td>
<td>Remote message printing</td>
</tr>
</tbody>
</table>

Table 1: Distributed Thread Functions

### 7.3 Distributed Program Execution

A simple skeleton of a distributed threads program is as follows:

```c
#include "aria.h"
main()
{
    struct thread_t main_t;
    int main_pri;

    /* program initialization */
    a_dinit(...); /* distributed system initialization */
    ariadne(&main_t, main_pri, DEFAULTSIZE); /* ariadne initialization */

    /* processing with one or more threads */
    a_exit();
    a_dexit(); /* distributed system exit */

    /* print results */
}
```
Distributed program execution is accomplished with support from a communications system. Each process in the distributed system begins execution in the main() function. The distributed system is initialized with a call to a_dinit(). This call also sets up the required mechanism to determine when the distributed program has terminated. For example, in existing implementation termination is assumed when there are no more live threads in the distributed system. As in the uniprocessor situation, the thread system is initialized through a call to the ariadne() function after the return from a_dinit().

Following a call to ariadne(), execution proceeds as in the sequential case. On each process, the Ariadne system frequently examines input buffers to process migrating threads and incorporate them in the computation. Because blocking calls prevent the system from attending to incoming threads, it is recommended that the use of blocking calls be avoided whenever possible.

Ariadne effects distributed termination on a given process when this process has no more live threads. A special (internal) bridge thread begins to run at this time. This thread either runs until the system terminates, or suspends itself if new incoming threads are received at this process. The system terminates when no live threads exist on any process. A call to a_dexit() at this point will result in the process leaving the distributed environment.

Ariadne creates execution log files for each process in the system. Process id k writes onto log file “aria.logk”. These files give information on performance and statistics on migrations.

8 Thread Scheduler Customization

The ariadne kernel provides facilities for context-switching, thread initialization and destruction. A built-in priority queue based scheduler maintains the runnable threads. This scheduler has been provided, keeping in mind the needs of some applications that may be developed using a threads package. There may be many other applications that do not need a sophisticated scheduler (a simple LIFO or FIFO queueing discipline may suffice), and there may be other applications that require a different kind of scheduler. For example, in a simulation, the next thread to run is based on which thread hosts the event with the minimum time-stamp. For such needs, Ariadne provides the means to replace the built-in scheduler, with a user-defined scheduler, and also provides support to users that need to access Ariadne variables while developing such a scheduler. The following paragraphs describe how a custom scheduler may be installed, and what support is available. Ariadne threads must have a priority assigned to them at creation time. If a user does not need to use this value, any one of the allowable values (1 to 8) may be specified to all the threads.
8.1 Scheduler Customization Functions

int a_getpri(struct thread_t thr)

This function gets the priority of the specified thread. A thread may obtain its own priority using a_mypri(), or pass the Ariadne constant ASELF to this function.

int a_getmaxpri(void)

This function returns the maximum priority of any runnable thread in the system. Even though the user may not assign any meaning to the priority field, Ariadne must maintain a notion of the maximum priority of all the runnable threads in the system.

void a_sched_register(void (*insert_ready)(void *),
                      void (*insert_top)(void *),
                      void * (*delete_ready)(void *),
                      void * (*delete_named)(void *))

This function installs the custom scheduler routines. When installed, the Ariadne kernel calls these routines instead of invoking its built-in scheduler. The kernel requests for threads to run using delete_ready and delete_named, and returns threads back to user space through insert_ready and insert_top. The user may use a specialized data structure to store and retrieve threads. The actual implementation of these scheduler routines is application dependent. The specific situations in which the Ariadne kernel makes calls to these user defined scheduler routines is mentioned below. If these Aria functions/features are not used by the user in their application programs, then the corresponding scheduler functions can be passed as NULL (left undefined). It is an error to define a function as NULL, and make use of the features which call the scheduler function. In such cases the program will fail. It must be noted that only user created threads are returned to the user. All system threads are saved on internal queues, and are never handed out of the Aria kernel.

- delete_ready() The scheduler function delete_ready is called when the kernel requires a thread to run. This function must be defined. It is called when Aria switches context from an old thread to a new thread. The function must return the tca (thread context area) pointer to a thread that should be run next by the kernel. The tca of a thread is the key component of the thread_t structure described earlier.
**insert_ready(char *thr)** The scheduler function `insert_ready` is called when the old thread is returned back to user space. This function must be defined. The user must store the thread that is passed as a parameter by Aria, as a runnable thread and return it back to Aria only when it is to be run next (through a return from a call of the `delete_ready()` function).

**delete_named(char *thr)** The function `delete_named` is called when a specific thread is required. This is required to be defined only if the user needs to use the `a_yield` call, the `a_setpri`, or the `a_destroy` call. This function must return the `tca` pointer of the thread that is passed as a parameter.

**insert_top(char *thr)** The function `insert_top` is required in cases when a thread must be given special consideration. Aria calls this function when a thread must be temporarily returned to user space. The understanding is that an immediately following call made to `delete_ready` will return the thread that was passed in the `insert_top` call. This routine need not be defined if the `a_setpri` function and the thread migration/common stack features of Aria are not used.

**8.2 Example - priority based scheduler**

This is an example of Ariadne's built-in scheduler implemented as a user scheduler. The application program changes minimally. After the following scheduler code is written in a file `sched.c` that is separately compiled and linked with the other modules, it only requires to make the call to `create_scheduler` (defined below), in the application to replace the built-in scheduler. The function `create_scheduler` must be invoked in the `main()` function before the call to `ariadne()`.

```c
/* sched.c - file implementing the user scheduler - replace this file
 * with another to define different schedulers
 * This code makes use of the following queue manipulation primitives
 * (tca = the thread context area, schedq = queue data structure)
 * tca* hq(schedq*) - deletes and returns the head of the queue
 * tca* dq(schedq*, tca*) - deletes and returns requested thread from queue
 * void ioq(schedq*, tca*) - inserts thread at the head of the queue
 */
#include <stdio.h>
#include "sched.h"

static struct lpq schedq[MAXPRI]; /* defines an array of queues */

/*
 * delete_ready - this function is used to return the next thread that is
```
struct tca*
delete_ready(void)
{
    struct tca* nextp;
    int i;

    for (i=a_getmaxpri()+1; i--; ) {
        nextp = hoq (&schedq[i]);
        if (nextp && (!a_ping(nextp) /* thread must be alive */
                      break;
    }
    return nextp;
}

/*
 * delete_named - this function deletes a particular thread from the
 * scheduler data structures and returns it to
 * ariadne.
 */
struct tca*
delete_named(struct tca* thptr )
{
    return doq(&schedq[a_getpri(thptr)], (struct tca*)thptr);
}

/*
 * insert_ready - insert a thread into the scheduling queues. This thread
 * is inserted in a state ready to run. It may have been
 * returned because it was preempted by some higher priority
 * thread or because its time-slice was completed.
 */
void
insert_ready(struct tca* thptr)
{
    toq (&schedq[a_getpri(thptr)], (struct tca*)thptr);
}

/*
 * insert_top - insert with priority. this is an indication that this
 * thread must be inserted with highest priority. It may be placed
 * as the next thread that will run in its priority class.
 */
void
insert_top(struct tca* thptr)
{
ioq(&schedq[a_getpri(thptr)], (struct tca* )thptr); /* add at head of queue */
}
/*
* create.scheduler - creates and initializes the data structures that are
* required. User must call this function before ariadne().
* It also registers the scheduling functions.
*/
void
create_scheduler(void)
{
    int i;
    struct lpq *q;

    for (i=MAXPRI, q=schedq; i--; q++) {
        q->head = q->tail = 0;
    }
    a_sched_register(insert_ready, insert_top, delete_ready, delete_named);
}

9 Ariadne in a Shared Memory Environment

Ariadne provides shared memory support so that multi-processors can be exploited and multiple threads can run concurrently. Programming with shared memory is similar to the stand-alone mode except that some of the objects and threads are now placed in shared memory and several processes are created. Each process can pick up a ready thread and schedule it for execution. The creation and execution of multiple processes is transparent to the user. To exploit shared memory multi-processors a scheduler implemented in shared memory is usually created using the functions described in the previous section.

The following user callable functions are available for shared-memory programming.

void a_set_shared(int sz)

Sets shared memory usage on, and the size of the Ariadne internal shared memory segment is set to sz. Passing 0 as a parameter to this function will get the maximum possible internal shared area. In addition, users can create their own external shared memory segments using the primitives described below.

void a_shared_begin(int nprocs)
This function creates \texttt{nprocs} processes and sets them up for shared access. The routine
determines the number of processors available on the system, and uses that value if the
parameter \texttt{nprocs} is larger than the maximum number of processors that are available.
It sets up the internal shared memory segment and some semaphores required for running
Ariadne in a shared memory environment.

\begin{verbatim}
void a_shared_exit()
\end{verbatim}

This function cleans up internal Ariadne shared segments and semaphores that were cre­
ated by the \texttt{a_shared_begin()} function. The parent process also sends a signal to all
children, which makes the children to wake up, if they were blocked. A user defined routine
\texttt{a_shared_finish()} is then called automatically. The programmer may write code in this
function to unblock the process, or perform any other action that will allow the processes
to exit gracefully. If clean up is not required then this function may be empty.

\section{Sharing Resources}

\subsection{Shared Semaphores}

For synchronization between threads in multiple processes in the system, shared memory
counting semaphores can be created. The interface for these is provided in table 2 and is
similar to that described in Section 4.1. In this case, however, the semaphores are identified
by a integer identifier which is returned by the creation call, and it must be used in all
other calls related to semaphores. All routines return a value of -1 if an invalid semaphore
identifier is used.

\subsection{Shared Memory}

Ariadne provides two types of shared memory areas. Users can allocate and free space from
Ariadne's internal shared memory segment and also create and destroy additional segments.
The following interface is based on Unix inter process communication calls and is simplified
for use in the Ariadne system.

\subsection{Internal Shared memory}

\begin{verbatim}
void *shmemalloc(int size)
\end{verbatim}
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sem_id = a_createsh(int sem_key, int value);</td>
<td>Create a semaphore</td>
</tr>
<tr>
<td>int a_destsh(int sem_id);</td>
<td>Destroy the semaphore with id sem_id</td>
</tr>
<tr>
<td>int a_signalsh(int sem_id);</td>
<td>Signal the semaphore once</td>
</tr>
<tr>
<td>int a_signalnsh(int sem_id, int count);</td>
<td>Signal the semaphore count times</td>
</tr>
<tr>
<td>int a_waitallsh(int sem_id);</td>
<td>Signal all the threads waiting</td>
</tr>
<tr>
<td>int a_countsh(int sem_id);</td>
<td>Wait to obtain the semaphore</td>
</tr>
<tr>
<td>int a_countsh(int sem_id);</td>
<td>Returns a count of the number of</td>
</tr>
<tr>
<td></td>
<td>threads blocked at this semaphore</td>
</tr>
</tbody>
</table>

Table 2: Shared Memory Synchronization Primitives

Gets a block of memory of size `size` from the internal segment. It returns the address of the area allocated if successful otherwise it returns -1.

```c
void *shmemfree(void *ptr, int size)
```

Unallocates the memory allocated by `shmemalloc`. The same size as used with the allocation call and the pointer returned by it must be passed to this function for it to work correctly.

External Shared Memory

```c
void *shmcreate(int key, int size, int perms, int *id)
```

Creates a shared memory segment of size `size` bytes and returns the address where the segment is located. It returns the identifier `id` that identifies the shared memory segment. The value of `perms` is the same as that used in the `shmget(2)` call in the Unix system.

```c
int shmfree(void *addr, int id)
```

This routine frees the shared memory segment with identifier `id` at address `addr` that was allocated with `shmcreate()`.
9.2 Customizing the scheduler

It is possible to customize the scheduler in a shared-memory environment. Ariadne provides two custom schedulers for use with shared memory. The first (implemented in a file named schedshm.c) schedules all available threads on available processes in order of priority. The second one (in file schedshml.c) schedules threads strictly on priority basis. All threads at a higher priority are finished first before continuing with threads at a lower priority. Support functions for creating such a scheduler is given in the appendix.

9.3 Shared Memory Programs

An example of using multiprocessors for a matrix multiplication example is provided later in the examples section of this document. A skeletal shared memory program showing correct usage is as follows:

```c
#include "aria.h"
#include "shm.h"
#include "schedshm.h"
main()
{
  struct thread_t main_t;
  int main_pri = 8;
  int nprocs;

  /* initialize program, command line processing etc */
  a_set_shared(0);
  a_shared_begin(nprocs);
  create_scheduler(); /* if custom scheduler is installed */
  ariadne(&main_t, main_pri, DEFAULTSIZE);

  /* create shared memory objects etc. */
  if (sh_process_id == 0) {
    /* create one or more threads */
  }
  a_exit(); /* all processes wait for this barrier */

  /* print results */
```

The programmer must keep the following in mind in order to use Ariadne with shared memory multi-processors. The main program is not schedulable and cannot do any useful work other than initialization and cleanup. Thus the main program must only create threads and then wait for the system to terminate. This can be accomplished by setting a high priority for the ariadne() call and lower priority for all other threads created thereafter. The call to a_exit() then ensures that the main thread remains blocked until all the other threads which have a smaller priority are finished. In actuality, the a_exit() call implements a barrier allowing all threads to complete before signaling termination.

The size of the internal shared memory segment in Ariadne dictates the number of threads that can be live at any instant after which the Ariadne system runs out of memory. Currently the size of internal shared memory is set to approximately 1 MB but this value may be increased by increasing the value of SHARED_MEM_SIZE and recompiling the Ariadne system. The current value is set based on the needs of simple applications and also the fact that most machines are not configured to allow large shared memory segments.

The Ariadne system uses shared memory segments and semaphores with key values less than 10. (These values are required by the IPC in Unix systems.) If the user decides to create new shared memory segments or semaphores, they must use key values larger than 20. Values from 11 to 20 are reserved for use by the custom shared memory schedulers.

10 Examples
10.1 Single Process

This example illustrates use of simple Ariadne primitives. Main() creates thread t1. Thread t1 creates thread t2. Thread t1 attempts to yield to thread t2. The yield fails because t1 has a higher priority than t2. Thread t1 next lowers its own priority before attempting a second yield to t2. This yield is successful. The example also shows that a yield to a non-existent thread results in a run-time error.

```c
#include <stdio.h>
#include "aria.h"

thread_t Tom, Harry;
```
void t2 (int i1, float f1, double d1)
{
    struct thread_t whoami, who_last;
    a_self(&whoami);
    a_previous(&who_last);
    printf("t2: I am Thread (id=%d,key=%d) \n", whoami.thread_id, whoami.thread_key);
    printf("t2: I am Thread (id=%d,pri=%d) \n", a_myid(), a_mypri());
    printf("t2: I got control from thread %d \n", who_last.thread_id);
    printf("t2: My arguments are: %d %f %lf \n", i1, f1, d1);
    a_yield(Tom);
    printf("t2: I regain control from t1\n");
    printf("t2: I yield to t1 again \n\n");
    a_yield(Tom);
}

void t1 (int i1, float f1, double d1)
{
    struct thread_t whoami, who_last;
    a_self(&whoami);
    a_previous(&who_last);
    printf("t1: I am Thread (id=%d,key=%d) \n", whoami.thread_id, whoami.thread_key);
    printf("t1: I am Thread (id=%d,pri=%d) \n", a_myid(), a_mypri());
    printf("t1: I got control from thread %d \n", who_last.thread_id);
    printf("t1: My arguments are: %d %f %lf \n", i1, f1, d1);
    printf("t1: I create t2 \n");
    a_create(&Harry, (void (*)())t2, 2, MEDIUM, 1, 1, i1, f1, d1);
    printf("t1: I yield to t2 \n\n");
    a_yield(Harry);
    printf("t1: I was unsuccessful because my priority is higher \n");
    printf("t1: I decrease my priority and try again \n\n");
    a_setpri(whoami, 2); /* or a_setmypri(2) */
    printf("t1: I regain control from t2 \n");
    print("t1: Is t1 still alive ?\n");
    if (!a_ping(Harry)) printf("t1: YES, t2 is still alive !\n");
    printf("t1: I will yield to t2 again \n\n");
    a_yield(Harry);
    printf("t1: I regain control from t2, and now destroy t2\n");
    a_destroy(Harry);
    printf("t1: What if I yield to t2 now ?\n");
    a_yield(Harry);
}
printf ("t1: I was unsuccessful because t2 does not exist!\n");
}

main()
{
    struct thread_t whoami;
    int i1=100; float f1=200; double d1 = 300;
    aria(&whoami,5,8192);
    printf ("main: I am Thread (id=%d,key=%d) \n\n", whoami.thread_id, whoami.thread_key);
    a_create(&Tom, (void (*) () )t1, 3, MEDIUM, 1,1,1,i1,f1,d1);
    a_exit();
}

10.2 Multiple Processes - Distributed Memory

This example illustrates the use of thread migration. Given an \( N \) process Conch environment (where each process is hosted by a distinct processor), \texttt{main()} creates \( N \) threads on each process. Each thread executing on process \( k \) writes a character and then migrates to process \( (k + 1) \mod N \), for \( 0 \leq k \leq N - 1 \). A thread performs 100 migrations before it is terminated.

The following file is a sample Conch configuration file, for a 5-machine environment, that may be used with this example.

\texttt{#FILE ~edm/threads/aria/examples/ex2}
\texttt{#TREE}
\texttt{ariadne.cs}
\texttt{-bach.cs}
\texttt{-beethoven.cs}
\texttt{-carcassi.cs}
\texttt{-segovia.cs}
\texttt{#END}

\texttt{#include <conch.h>}
\texttt{#include <stdio.h>}

26
```c
#include <signal.h>
#include "aria.h"
#ifndef _sys_time_h
#include <sys/time.h>
#endif
#define OUTERLOOP 100
#define INNERLOOP 100000

struct thread_t T[100];

void t (int i)
{
    int j, k, m=0; char ch;
    struct thread_t me;
    int mask;
    int numprocs;
    numprocs = a_getnumprocs();
    ch = i + '0';
    a_self(&me);
    mask = a_interruptoff();
    printf( " I am %d \n", me.thread_id);
    a_interrupton(mask);
    for (j=1; j<OUTERLOOP; j++) {
        for(k=1;k<INNERLOOP; k++) m = m+1;
        write(1, &ch, 1);
        a_migrate((a_getprocid() + 1)%numprocs);
    }
}

main()
{
    int i;
    struct thread_t whoami;
    a_dinit(); /* initialize distributed system */
a_usr_cswitch_proc = sp_process; /* set the function called after a context switch */
    aria(&whoami,7,4096); /* initialize Ariadne system */
```
a_defslice(REAL,ALL,0,10000); /* set round robin slicing on */

for (i = 0; i < a_getnumprocs(); i++) {
    a_create(&T[i], t, 5, MEDIUM, 1,0,0,i);
}

a_exit(); /* exit from Ariadne system */
a_dexitO; /* exit Distributed System */

10.3 Multiple Processes - Shared Memory

This example shows the use of the shared memory Ariadne system. Two matrices A and B are given and the result R=AxB is required. The solution given below stores the matrices A and B in shared memory provided by the Ariadne system. New shared memory segments could have been created to store these values. The main program in the parent initializes the shared system as well as the Ariadne system and then creates a single thread creator. It then calls a_exit(). Child processes only initialize the Ariadne system and then call a_exit(). The creator thread creates as many threads as the number of elements in the result. These threads compute the dot product from the corresponding row and column in the input matrices.

/* shmatmul.c - parallel matrix multiplication program */
#include <stdio.h>
#include <math.h>
#include "aria.h" /* stand-alone ariadne include file */
#include "shm.h" /* shared memory support */
#include "schedshm.h" /* shared memory based scheduler */

void dotProduct(int, int); /* dot product calculation thread */
void creator(void); /* creator thread */
#define SHM_KEY 100 /* external shared memory segment */
struct usr_shmem {
    double *rr, *aa, *bb;
};
struct usr_shmem* usr_shmem_ptr;

#define r (usr_shmem_ptr->rr)
#define a (usr_shmem_ptr->aa)
#define b (usr_shmem_ptr->bb)
int numRowsA, numColsA, numRowsB, numColsB, nprocs;
main(int argc, char *argv[]) {
    int i, j;
    struct thread_t whoami;
    int i_id;

    /* process command line arguments */
    if (argc == 5) {
        numRowsA = atoi(argv[1]);
        numColsA = atoi(argv[2]);
        numColsB = atoi(argv[3]);
        nprocs = atoi(argv[4]);
    } else {
        printf("usage: matmult rowsA colsA colsB nprocs\n");
        exit(1);
    }

    /* read input from file or initialize internally */
    numRowsB = numColsA;

    a_set_shared(0);            /* shared memory environment */
    a_shared_begin(nprocs);     /* create multiple processes */
    create_scheduler();         /* set up custom scheduler */
    /* create external shared memory segment */
    usr_shmem_ptr = shmcreate(SHM_KEY, sizeof(struct usr_shmem), PERMS, &i_id);
    aria(&whoami, 7, 1024);
    if (sh_process_id == 0) {
        /* allocating matrices from internal shared segment */
        r = (double*)shmemalloc(sizeof(double)*numRowsA*numColsB);
        a = (double*)shmemalloc(sizeof(double)*numColsA*numRowsA);
        b = (double*)shmemalloc(sizeof(double)*numColsB*numRowsB);
        for (i=0; i< numRowsA; i++)
            for (j=0; j< numColsA; j++)
                *(a+i*numColsA + j) = i*numColsA+j;
        for (i=0; i< numRowsB; i++)
            for (j=0; j< numColsB; j++)
                *(b+i*numColsB + j) = i*numColsB + j;
        a_create(0, creator, 5, SMALL, 0, 0, 0);
    }
    a_exit();                   /* ariadne system exit */
    if (sh_process_id == 0)
        for (i=0; i< numRowsA; i++) { /* print results */
            for (j=0; j< numColsB; j++)
                printf("%10.2lf", *(r+numRowsA*i+j));
}
printf("\n");
}
destroy_scheduler(); /* destroy the scheduler */
a_shared_exit(); /* shared memory clean up */
shmfree(usr_shmem_ptr, i_id); /* release external shared memory */
}

void creator(void)
{
    int i, j;

    for (i=0; i< numRowsA; i++)
        for (j=0; j< numColsB; j++)
            a_create(0, dotProduct, 5, SMALL, 2, 0, i, j);
}

void dotProduct(int row, int col)
{
    double result;
    int i,j;

    for (j=0; j< 10000; j++) {
        result = 0.0;
        for (i=0; i < numColsA; i++)
            result += *(a+numColsA*row+i)* *(b+numColsB*i+col);
        *(r+numRowsA*row+col) = result;
    }
}
A Support for distributed threads

This section may be read by users who wish to make use of distributed threads with a message passing system other than PVM or Conch. It describes briefly the functions using which, it is possible to develop an environment that can make programming with distributed threads simpler.

int a_get_liveth(void)

Returns the number of threads (live thread count) that are live in the distributed system and which were created by the calling process.

int a_numbthreads(void)

Returns the number of live threads in the calling process, irrespective of where they were created.

int a_get_th_destr(int procid)

Return the count of the number of threads destroyed by the calling process which were created by process procid.

void a_decr_liveth(int count)

Decrement the live thread count by count.

void a_reset_th_destr(int procid)

Set the count of the number of threads destroyed for process procid to zero.

char *a_currthread(void)

Returns the tca pointer of the currently running thread.

int a_myprocid(void)

Returns the process identifier of the process that created the thread.

int a_thread_pack(char *tca, char **packed_thread, int *size)
This routine "packs" the complete context of the thread into the buffer `packed_thread` and returns the size in `size`. A return of zero from this routine indicates that the thread is being packed on the source, and a return of 1 indicates that it has been unpacked and is now running as a migrated thread on the destination process. This routine takes the help of an internal "copier" thread to do the actual packing.

```c
char *a_thread_unpack(char *packed_thread, int size)
```

Creates a new thread from the thread in the buffer `packed_thread` of size `size` and returns the pointer to the tca area of the thread.

```c
int a_mydid(void)
```

Returns the thread identifier of the thread that is unique in the distributed system.

```c
int a_sizeoftca(void)
```

Returns the size of the tca - the thread context area.

```c
int a_sizeofstack(void)
```

Returns the amount of stack space that is currently used by the thread.

## B  Creating a shared memory based scheduler

If required by their application, users write the scheduler functions using some additional support functions given below in addition to those provided in Section 8.1. The scheduler file thus created is compiled separately and linked in with the rest of the application. This will install the revised scheduler in place of the built-in scheduler.

```c
int a_shget_maxpri(void)
```

Returns the value of priority corresponding to the thread with the maximum priority in the calling process.

```c
void a_shglobal_lock(void)
```
This call can be used to prevent other processes in the shared memory system from manipulating internal shared variables. It allows the calling process to manipulate structures and at the same time ensure that shared variables in the Ariadne system will not be changed. It would be necessary to call this function (for example) when obtaining the next thread to run from a shared-memory based scheduler.

```c
void a_shglobal_lock(void)
```

Unlocks the internal shared global variables of the Ariadne system. Any function that calls `a_shglobal_lock()` must not return unless this function is also called.