Parlib API Reference

Release 1.0

Kevin Klues, Barret Rhoden

February 07, 2013

CONTENTS

CHAPTER

ONE

WELCOME TO PARLIB'S API REFERENCE!

Parlib is a library meant to ease the development of software written for highly parallel systems. It was originally written for the Akaros operating system, and has since been ported to Linux. It was originally designed as an emulation layer on top of Linux that would allow developers to write applications and test them on a linux system before deplying them on Akaros. Since then, however, Parlib has proved itself useful in its own right as a standalone libary for Linux. Most notably, as the backend for the Lithe implementation.

The official Home Page for Parlib is: http://akaros.cs.berkeley.edu/parlib.

At present, Parlib provides 6 primary services to developers:

1. A user-level abstraction for managing physical cores in a virtualized-namespace (vcores)
2. A user-level thread abstraction (uthreads)
3. User-space MCS locks, MCS barriers, spinlocks and spinbarriers
4. Static and dynamic user-level thread local storage
5. Pool and slab allocator dynamic memory management abstractions
6. A standardized API for common atomic memory operations

CHAPTER

TWO

API REFERENCE

2.1 Vcores

The vcore abstraction gives user-space an abstraction of a physical core (within a virtual namespace) on top of which it can schedule its user-level threads. When running code associated with a vcore, we speak of being in **vcore context**.

Vcore context is analogous to the interrupt context in a traditional OS. Vcore context consists of a stack, a TLS, and a set of registers, much like a basic thread. It is the context in which a user-level scheduler makes its decisions and handles signals. Once a process leaves vcore context, usually by starting a user-level thread, any subsequent entrances to vcore context will be at the top of the stack at vcore_entry ().

The vcore abstraction allows an application to manage its cores, schedule its threads, and get the best performance it possibly can from the hardware.

To access the vcore API, include the following header file:

```
#include <parlib/vcore.h>
```

2.1.1 Constants

MAX_VCORES
The maximum number of vcores supported on this platform

LOG2_MAX_VCORES
The log-base-2 of the maximum number of vcores supported on this platform

2.1.2 Types

struct **vcore**
vcore_t
An opaque type used to maintain state associated with a vcore

2.1.3 External Symbols

extern void **vcore_entry** ()
User defined entry point for each vcore. If vcore_saved_ucontext is set, this function should just restore it, otherwise, it is user defined.

2.1.4 Global Variables

vcore_t ***__vcores**
An array of all of the vcores available to this application

void ****__vcore_tls_descs**
An array of pointers to the TLS descriptor for each vcore.

__thread ucontext_t **vcore_context**
Context associated with each vcore. Serves as the entry point to this vcore whenever the vcore is first brough up, a usercontext yields on it, or a signal / async I/O notification is to be handled.

__thread ucontext_t ***vcore_saved_ucontext**
Current user context running on each vcore, used when interrupting a user context because of async I/O or signal handling. Vcore 0's vcore_saved_ucontext is initialized to the continuation of the main thread's context the first time it's vcore_entry () function is invoked.

__thread void ***vcore_saved_tls_desc**
Current tls_desc of the user context running on each vcore, used when interrupting a user context because of async I/O or signal handling. Hard Thread 0's vcore_saved_tls_desc is initialized to the **tls_desc_** of the main thread's context the first time it's vcore_entry () function is invoked.

2.1.5 API calls

int vcore_lib_init();
Initialization routine for the vcore subsystem.

void **vcore_reenter** (void *(*entry_func)(void)*)
Function to reenter a vcore at the top of its stack.

int **vcore_request** (int *k*)
Requests k additional vcores. Returns -1 if the request is impossible. Otherwise, blocks the calling vcore until the request is granted and returns 0.

void vcore_yield();
Relinquishes the calling vcore.

int **vcore_id** (void)
Returns the id of the calling vcore.

size_t **num_vcores** (void)
Returns the current number of vcores allocated.

size_t **max_vcores** (void)
Returns the maximum number of allocatable vcores.

bool in_vcore_context() {
Returns whether you are currently running in vcore context or not.

void clear_notif_pending(uint32_t vcoreid);
Clears the flag for pending notifications

void enable_notifs(uint32_t vcoreid);
Enable Notifications

void disable_notifs(uint32_t vcoreid);
Disable Notifications

#define **vcore_begin_access_tls_vars** (vcoreid)
Begin accessing TLS variables associated with a specific vcore (possibly a different from the current one). Matched one-to-one with a following call to vcore_end_access_tls_vars () within the same function.

#define **vcore_end_access_tls_vars** ()
End access to a vcore's TLS variables. Matched one-to-one with a previous call to vcore_begin_access_tls_vars () within the same function.

#define **vcore_set_tls_var** (name, val)
Set a single variable in the TLS of the current vcore. Mostly useful when running in uthread context and want to set something vcore specific.

#define **vcore_get_tls_var** (name)
Get a single variable from the TLS of the current vcore. Mostly useful when running in uthread context and want to get something vcore specific.

2.2 Uthreads

To access the uthread API, include the following header file:

```
#include <parlib/uthread.h>
```

2.2.1 Constants

UTH_EXT_BLK_MUTEX
One, of possibly many in the future, reasons that a uthread has blocked externally. This is required for proper implementation of the uthread_has_blocked() API call.

2.2.2 Types

struct **uthread**
uthread_t
An opaque type used to reference and manage a user-level thread

struct **schedule_ops**
schedule_ops_t
A struct containing the list of callbacks a user-level scheduler built on top of this uthread library needs to implement.

```
struct schedule_ops {
    void (*sched_entry)(void);
    void (*thread_runnable)(struct uthread *);
    void (*thread_paused)(struct uthread *);
    void (*thread_blockon_sysc)(struct uthread *, void *);
    void (*thread_has_blocked)(struct uthread *, int);
    void (*preempt_pending)(void);
    void (*spawn_thread)(uintptr_t pc_start, void *data);
};
```

void **schedule_ops_t.sched_entry** ()

void **schedule_ops_t.thread_runnable** (struct uthread *)

void **schedule_ops_t.thread_paused** (struct uthread *)

void **schedule_ops_t.thread_blockon_sysc** (struct uthread *, void *)

void **schedule_ops_t.thread_has_blocked** (struct uthread *, int)

void **schedule_ops_t.preempt_pending** ()

void **schedule_ops_t.spawn_thread** (uintptr_t *pc_start*, void **data*)

2.2.3 External Symbols

extern struct schedule_ops ***sched_ops**

A reference to an externally defined variable which contains pointers to implementations of all the schedule_ops callbacks.

2.2.4 Global Variables

__thread uthread_t ***current_uthread**

2.2.5 API calls

int **uthread_lib_init** ()

void **uthread_cleanup** (struct uthread **uthread*)

void **uthread_runnable** (struct uthread **uthread*)

void **uthread_yield** (bool *save_state*, void (*yield_func)(struct *uthread**, void*), void **yield_arg*)

void **save_current_uthread** (struct uthread **uthread*)

void **highjack_current_uthread** (struct uthread **uthread*)

void **run_current_uthread** (void)

This function does not return.

void **run_uthread** (struct uthread **uthread*)

This function does not return.

void **swap_uthreads** (struct uthread **__old*, struct uthread **__new*)

init_uthread_tf (uthread_t **uth*, void *(*entry)(void)*, void **stack_bottom*, uint32_t *size*)

#define **uthread_begin_access_tls_vars** (uthread)

#define **uthread_end_access_tls_vars** ()

#define **uthread_set_tls_var** (uthread, name, val)

#define **uthread_get_tls_var** (uthread, name)

2.3 MCS Locks

MCS locks are a spinlock style lock designed for more efficient execution on multicore processors. They are designed to mitigate cache clobbering and TLB shootdowns by having each call site spin on a core-local lock variable, rather than the single lock variable used by traditional spinlocks. These locks should really only be used while running in vcore context.

To access the mcs lock API, include the following header file:

```
#include <parlib/mcs.h>
```

2.3.1 Constants

MCS_LOCK_INIT
Static initializer for an mcs_lock_t

MCS_QNODE_INIT
Static initializer for an mcs_lock_qnode_t

2.3.2 Types

struct **mcs_lock_qnode**
mcs_lock_qnode_t
An MCS lock qnode. MCS locks are maintained as a queue of MCS qnode pointers, and locks are granted in order of request, using the qnode pointer passed at each mcs_lock_lock () call.

struct **mcs_lock**
mcs_lock_t
An MCS lock itself. This data type keeps track of whether the lock is currently held or not, as well as the list of qnode pointers described above.

struct **mcs_dissem_flags**
mcs_dissem_flags_t
Dissemmenation flags used by the MCS barriers described below. This data type should never normally be accessed by an external library. They are used internally by the MCS barrier implementation, but exist as part of the API because the mcs_barrier struct contains them.

struct **mcs_barrier**
mcs_barrier_t
An MCS barrier. This data type is used to reference an MCS barrier across the various MCS barrier API calls.

2.3.3 API calls

void **mcs_lock_init** (struct mcs_lock **lock*)
Initializes an MCS lock.

void **mcs_lock_lock** (struct mcs_lock **lock*, struct mcs_lock_qnode **qnode*)
Locks an MCS lock, associating a call-site specific qnode with the lock in the process.

void **mcs_lock_unlock** (struct mcs_lock **lock*, struct mcs_lock_qnode **qnode*)
Unlocks an MCS lock, releasing the call-site specific qnode associated with the current lock holder.

void **mcs_lock_notifsafe** (struct mcs_lock **lock*, struct mcs_lock_qnode **qnode*)
A signal-safe implementation of mcs_lock_lock (). While the lock is held, the lockholder will not be interrrupted to run any signal handlers.

void **mcs_unlock_notifsafe** (struct mcs_lock **lock*, struct mcs_lock_qnode **qnode*)
The mcs_lock_unlock () counterpart to mcs_lock_notifsafe (). After releasing the lock, signals may be processed again.

void **mcs_barrier_init** (mcs_barrier_t* *b*, size_t *num_vcores*)
Initializes an MCS barrier with the number of vcores associated with the barrier.

void **mcs_barrier_wait** (mcs_barrier_t* *b*, size_t *vcoreid*)
Waits on an MCS barrier for the specified vcoreid.

2.4 Spinlocks

To access the spinlock API, include the following header file:

```
#include <parlib/spinlock.h>
```

2.4.1 Constants

2.4.2 Types

struct **spinlock**
spinlock_t

2.4.3 API calls

void **spinlock_init** (spinlock_t **lock*)

int **spinlock_trylock** (spinlock_t **lock*)

void **spinlock_lock** (spinlock_t **lock*)

void **spinlock_unlock** (spinlock_t **lock*)

2.5 Dynamic Thread Local Storage

Dynamic thread local storage is parlib's way of providing pthread_key_create() and pthread_set/get_specific() style sematics for uthreads and vcores. Using dtls, different libraries can dynamically define their own set of dtls keys for thread local storage. Uthreads and vcores can then access the private regions associated with these keys through the API described below.

To access the dynamic thread local storage API, include the following header file:

```
#include <parlib/dtls.h>
```

2.5.1 Types

struct **dtls_key**
dtls_key_t

A dynamic thread local storage key. Uthreads and vcores use these keys to gain access to their own thread local storage region associated with the key. Multiple keys can be created, with each key referring to a different set of thread local storage regions.

void (*dtls_dtor_t)(void*);
dtls_dtor_t

A function pointer defining a destructor function mapped to a specific dtls_key. Whenever a uthread has accessed a dtls_key that has a dtls_dtor_t mapped to it, the destructor function will be called before the uthread exits.

2.5.2 API calls

dtls_key_t **dtls_key_create**(dtls_dtor_t *dtor*)
: Initialize a dtls key for dynamically setting/getting thread local storage on a uthread or vcore. Takes a dtls_dtor_t as a paramameter and associates it with a new dtls_key_t, which gets returned.

void **dtls_key_delete**(dtls_key_t *key*)
: Delete the provided dtls key.

void **set_dtls**(dtls_key_t *key*, void **dtls*)
: Associate a dtls storage region for the provided dtls key on the current uthread or vcore.

void ***get_dtls**(dtls_key_t *key*)
: Get the dtls storage region previously assocaited with the provided dtls key on the current uthread or vcore. If no storage region has been associated yet, return NULL.

void **destroy_dtls**()
: Destroy all dtls storage associated with all keys for the current uthread or vcore.

2.6 Thread Local Storage

To access the thread local storage API, include the following header file:

```
#include <parlib/tls.h>
```

2.6.1 Global Variables

void ***main_tls_desc**

__thread void ***current_tls_desc**

2.6.2 API calls

void ***allocate_tls**(void)

void ***reinit_tls**(void **tcb*)

void **free_tls**(void **tcb*)

void **set_tls_desc**(void **tls_desc*, uint32_t *vcoreid*)

void ***get_tls_desc**(uint32_t *vcoreid*)

2.7 Memory Pools

Memory pools are designed for efficient dynamic memory management when you have a fixed number of fixed size objects you need to manage. A large chunk of memory must first be allocated by traditional means before passing it to the pool API. Once received however, the memory is managed according to the restrictions described above, without worrying about fragmentation or other traditional dynamic memory management concerns.

To access the pool API, include the following header file:

```
#include <parlib/pool.h>
```

2.7.1 Types

struct **pool**
pool_t
Opaque type used to reference and manage a pool

2.7.2 API calls

void **pool_init** (pool_t *pool, void* buffer, void **object_queue, size_t num_objects, size_t object_size)
Initialize a pool. All memory MUST be allocated externally. The pool implementation simply manages the objects contained in the buffer passed to this function. This allows us to use the same pool implementation for both statically and dynamically allocated memory.

size_t **pool_size** (pool_t *pool)
Check how many objects the pool is able to hold

size_t **pool_available** (pool_t *pool)
See how many objects are currently available for allocation from the pool.

void* **pool_alloc** (pool_t *pool)
Get an object from the pool

int **pool_free** (pool_t* pool, void *object)
Put an object into the pool

2.8 Slab Memory Allocator

To access the slab allocator API, include the following header file:

```
#include <parlib/slab.h>
```

2.8.1 Types

struct **slab_cache**
slab_cache_t

void **(*slab_ctor_t)** (void *, size_t)
slab_ctor_t

void **(*slab_dtor_t)** (void *, size_t)
slab_dtor_t

2.8.2 API calls

struct slab_cache ***slab_cache_create** (const char *name, size_t obj_size, int align, int flags, slab_ctor_t ctor, slab_dtor_t dtor)

void **slab_cache_destroy** (struct slab_cache *cp)

void ***slab_cache_alloc** (struct slab_cache *cp, int flags)

void **slab_cache_free** (struct slab_cache *cp, void *buf)

2.9 Atomic Memory Operations

To access the atomic memory operations API, include the following header file:

```
#include <parlib/atomic.h>
```

2.9.1 Types

atomic_t

2.9.2 API calls

void **atomic_init** (atomic_t **number*, long *val*)

void ***atomic_swap_ptr** (void ***addr*, void **val*)

long **atomic_swap** (atomic_t **addr*, long *val*)

uint32_t **atomic_swap_u32** (uint32_t **addr*, uint32_t *val*)

#define **mb** ()

#define **cmb** ()

#define **rmb** ()

#define **wmb** ()

#define **wrmb** ()

#define **rwmb** ()

#define **mb_f** ()

#define **rmb_f** ()

#define **wmb_f** ()

#define **wrmb_f** ()

#define **rwmb_f** ()

CHAPTER

THREE

ABOUT

3.1 People

Current Contributors:

- Kevin Klues <klueska@cs.berkeley.edu>
- Barret Rhoden <brho@cs.berkeley.edu>

3.2 Publications

There are currently no publications on Parlib.

INDEX

S

U

V

W