List of Hashpipe Structures

Deepthi Gorthi

April 13, 2018

1 Thread Structures and Functions

1.1 hashpipe_thread_desc_t

```
struct hashpipe_thread_desc {
  const char * name;
  const char * skey;
  initfunc_t init;
  runfunc_t run;
  databuf_desc_t ibuf_desc;
  databuf_desc_t obuf_desc;
};
```

The `hashpipe_thread_desc` structure is used to store metadata describing a hashpipe thread. Typically a hashpipe plugin will define one of these hashpipe thread descriptors per hashpipe thread.

1. `const char *name`: String containing the thread name. Hashpipe threads are identified by their names which need to be registered with `register_hashpipe_thread()`. This is used to match command line thread specifiers to thread metadata so that the pipeline can be constructed as specified on the command line.

2. `const char *skey`: String containing the thread's status buffer "status" key. It is typically 8 characters or less, uppercase and ends with "STAT". If it is `non-NULL` and non-empty, `HASHPIPE` will automatically store/update this key in the status buffer with the thread's status at initialization ("init") and exit ("exit").

3. `initfunc_t init`: Pointer to thread's initialization function. The thread initialization function can be null if no special initialization is needed. If provided, it must point to a function with the following signature:

   ```
   int my_thread_init_funtion(hashpipe_thread_args_t *args)
   ```

4. `runfunc_t run`: Pointer to thread's run function. The thread run function must have the following signature:

   ```
   void my_thread_run_funtion(hashpipe_thread_args_t *args)
   ```

5. `ibuf`: Structure describing thread's input data buffer (if any)

6. `obuf`: Structure describing thread's output data buffer (if any). The data buffer description structure used for ibuf and obuf currently contains one function pointer:

   create - A pointer to a function that creates the data buffer.

   Future `HASHPIPE` versions may introduce additional data buffer fields. `ibuf.create` should be `NULL` for input-only threads and `obuf.create` should be `NULL` for output-only threads. Having both `ibuf.create` and `obuf.create` set to `NULL` is invalid and the thread will not be used. The create function must have the following signature:

   ```
   hashpipe_databuf_t *my_create_function(
        int instance_id, int databuf_id)
   ```

1.2 hashpipe_thread_args

This structure passed (via a pointer) to the application's thread initialization and run functions. The 'user_data' field can be used to pass info from the init function to the run function.

1. `hashpipe_thread_desc_t *thread_desc`
2. `int instance_id`
3. `int input_buffer`
4. `int output_buffer`

5. `unsigned int cpu_mask`: 0 means use inherited
6. `int finished`
7. `pthread_cond_t finished_c`
8. `pthread_mutex_t finished_m`
9. `hashpipe_status_t st`
10. `hashpipe_databuf_t *ibuf`
11. `hashpipe_databuf_t *obuf`
12. `void *user_data`

1.3 Useful Functions

1. `int run_threads()`: Function threads used to determine whether to keep running.
2. `register_hashpipe_thread (hashpipe_thread_desc_t *ptm)`: Function should be used by pipeline plugins to register threads with the pipeline executable.
3. `hashpipe_thread_desc_t *find_hashpipe_thread(char *name)`: This function can be used to find hashpipe threads by name. It is generally used only by the hashpipe executable. Returns a pointer to its `hashpipe_thread_desc_t` structure or `NULL` if a test with the given name is not found. Names are case sensitive.
4. `void list_hashpipe_threads(FILE *f)`: List all known hashpipe threads to file pointed to by the file pointer.
5. `unsigned int get_cpu_affinity()`: Get the CPU affinity of calling thread.

2 Data Buffer Structures and Functions

2.1 `hashpipe_databuf_t`

1. `char data_type[64]`: Type of data in the buffer
2. `size_t header_size`: Size of each block header in bytes
3. `size_t block_size`: Size of each data block in bytes.
4. `int n_block`: Number of data blocks in buffer
5. `int shmid`: ID of this shared memory segment
6. `int semid`: ID of locking semaphore set

2.2 Associated functions

1. **Key**:

   `key_t hashpipe_databuf_key()`

   Get the base key to use for **all** hashpipe databufs. The base key is obtained by calling the ftok function, using the value of `$HASHPIPE_KEYFILE`, if defined, or `$HOME` from the environment or, if `$HOME` is not defined, by using `tmp`. By default (i.e. no `HASHPIPE_KEYFILE` in the environment), this will create and connect to a user specific set of shared memory buffers (provided `$HOME` exists in the environment), but if desired users can connect to any other set of memory buffers by setting `HASHPIPE_KEYFILE` appropriately.

2. **Create Databuf**:

   ```
   hashpipe_databuf_t *hashpipe_databuf_create(int
      instance_id,int databuf_id, size_t
      header_size, size_t block_size, int n_block)
   ```

   Create a new shared mem area with given params. Returns pointer to the new area on success, or `NULL` on error. Returns error if an existing shmem area exists with the given shmid and different sizing parameters.

3. **Get Databuf**:

   ```
   hashpipe_databuf_t *hashpipe_databuf_attach(int   instance_id
   , int databuf_id)
   ```

   Return a pointer to a existing shmem segment with given id. Returns error if segment does not exist

4. **Detach Databuf**:

   ```
   int hashpipe_databuf_detach(hashpipe_databuf_t *d)
   ```

   Detach from shared mem segment

5. **Clear Databuf**:

   ```
   void hashpipe_databuf_clear(hashpipe_databuf_t *d)
   ```

6. **Reset Pointer location**:

   ```
   char *hashpipe_databuf_data(
      hashpipe_databuf_t *d, int block_id)
   ```

   Returns pointer to the beginning of the given data block.

7. **Get lock status**:

   ```
   int hashpipe_databuf_block_status(
      hashpipe_databuf_t *d, int block_id)

   int hashpipe_databuf_total_status(
      hashpipe_databuf_t *d)

   uint64_t hashpipe_databuf_total_mask(
      hashpipe_databuf_t *d)
   ```

   Returns lock status for given block_id, or total for whole array.

8. **Locking functions**:

   ```
   int hashpipe_databuf_wait_filled(
      hashpipe_databuf_t *d, int block_id)

   int hashpipe_databuf_busywait_filled(
      hashpipe_databuf_t *d, int block_id)
   ```

```
int hashpipe_databuf_set_filled(
   hashpipe_databuf_t *d, int block_id)

int hashpipe_databuf_wait_free(
   hashpipe_databuf_t *d, int block_id)

int hashpipe_databuf_busywait_free(
   hashpipe_databuf_t *d, int block_id)

int hashpipe_databuf_set_free(
   hashpipe_databuf_t *d, int block_id)
```

Databuf locking functions. Each block in the buffer can be marked as free or filled. The "wait" functions block (i.e. sleep) until the specified state happens. The "busywait" functions busy-wait (i.e. do NOT sleep) until the specified state happens. The "set" functions put the buffer in the specified state, returning error if it is already in that state.