Thread Control

Thread Limitations

Name of limit	Description	Name argument (sysconf)
`PTHREAD_DESTRUCTOR_ITERATIONS`	max number of times to destroy the thread-specific data	`_SC_THREAD_DESTRUCTOR_ITERATIONS`
`PTHREAD_KEYS_MAX`	max number of keys per process	`_SC_THREAD_KEYS_MAX`
`PTHREAD_STACK_MIN`	min number of bytes per thread stack	`_SC_THREAD_STACK_MIN`
`PTHREAD_THREADS_MAX`	max number of threads per process	`_SC_THREAD_THREADS_MAX`

Name of limit	Free BSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Ubuntu 18.04
`PTHREAD_DESTRUCTOR_ITERATIONS`	4	4	4	no limit	4
`PTHREAD_KEYS_MAX`	256	1024	512	no limit	1024
`PTHREAD_STACK_MIN`	2048	16384	8192	no limit	16384
`PTHREAD_THREADS_MAX`	no limit	no limit	no limit	no limit	no limit

Thread Attributes

- `pthread_attr_init(3)`: `int pthread_attr_init(pthread_attr_t *attr);`
- `pthread_attr_destroy(3)`: `int pthread_attr_destroy(pthread_attr_t *attr);`
  - return: 0 OK, error number on failure
  - common attributes

Name	Description

Name	Description
detachstate	detached thread attribute
guardsize	guard buffer size in bytes at end of thread stack
stackaddr	lowest address of thread stack
stacksize	lowest address of thread stack

- `pthread_attr_setdetachstate(3)`: `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);`
- `pthread_attr_getdetachstate(3)`: `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);`
  - return: 0 OK, error number on failure
  - `detachstate`
    - `PTHREAD_CREATE_DETACHED`
    - `PTHREAD_CREATE_JOINABLE`
- `pthread_attr_setguardsize(3)`: `int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);`
- `pthread_attr_getguardsize(3)`: `int pthread_attr_getguardsize(const pthread_attr_t *attr, size_t *guardsize);`
  - return: 0 OK, error number on failure
  - protect stack overflow caused by a single thread
  - default is set to `PAGESIZE` bytes
- `pthread_attr_setstack(3)`: `int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize);`
- `pthread_attr_getstack(3)`: `int pthread_attr_getstack(const pthread_attr_t *attr, void **stackaddr, size_t *stacksize);`
  - return: 0 OK, error number on failure
  - `stackaddr`: lowest addressable address
  - **not recommend to use, they are considered as deprecated**
- `pthread_attr_setstacksize(3)`: `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);`
- `pthread_attr_getstacksize(3)`: `int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);`
  - return: 0 OK, error number on failure

Synchronization Attributes

> **omit `pthread_xxx(3)` and return description in following paragraph**

- `int pthread_mutexattr_init(pthread_mutexattr_t *attr);`
- `int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);`
- `int pthread_condattr_init(pthread_condattr_t *attr);`
- `int pthread_condattr_destroy(pthread_condattr_t *attr);`
- `int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);`
- `int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);`
- Mutex Attribute: Process-Shared

    - `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);`
    - `int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *pshared);`
    - `pshared`
        - `PTHREAD_PROCESS_PRIVATE`: more efficient
        - `PTHREAD_PROCESS_SHARED`: more expensive
- Mutex Attribute: Type
    - `int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);`
    - `int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr, int *restrict type);`
    - `type`

Mutex type	Description
`PTHREAD_MUTEX_NORMAL`	standard type, not do special error check or deadlock detection
`PTHREAD_MUTEX_ERRORCHECK`	provide error checking
`PTHREAD_MUTEX_RECURSIVE`	allow a thread to lock mutex multiple times (same number of unlocks to release mutex)
`PTHREAD_MUTEX_DEFAULT`	system dependent default choice of mutex type

Mutex type	Relock without unlock?	Unlock when not owned?	Unlock when unlocked?
`PTHREAD_MUTEX_NORMAL`	deadlock	undefined	undefined
`PTHREAD_MUTEX_ERRORCHECK`	return error	return error	return error
`PTHREAD_MUTEX_RECURSIVE`	allowed	return error	return error
`PTHREAD_MUTEX_DEFAULT`	system dependent	system dependent	system dependent

- Other Common Attributes
    - `int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);`
    - `int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr, int *restrict pshared);`
    - `int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);`
    - `int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr, int *restrict pshared);`
    - `int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);`
    - `int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr, int *restrict pshared);`

Thread-Specific Data

- Known as Thread-Private Data
- A Solution
  - use array based on thread id
    - thread id may be not an integer
    - need extra protections
- Steps
  - create a pthread key, done once for all threads
  - get the data associated with key for the current thread
  - if data is not available, allocate data and associate with the key
  - if data is no longer required, it can be released and de-associated
- `pthread_key_create(3p)`: `int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));`
- `pthread_key_delete(3p)`: `int pthread_key_delete(pthread_key_t key);`
  - return: 0 OK, error number on failure
  - non-NULL data address will passed to the destructor when thread exits
  - a pthread key should be created only once
  - a call to `pthread_key_delete(3p)` will not invoke the corresponding destructor
- `pthread_once(3p)`: `int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));`
  - alternative `pthread_once_t once_control = PTHREAD_ONCE_INIT;`
- Use `pthread_once_t` to Create a pthread Key

```
void destructor(void *);
pthread_key_t key;
pthread_once_t init_done = PTHREAD_ONCE_INIT;
void threaed_init(void) {
  err = pthread_key_create(&key, destructor);
}
int thread_func(void *arg) {
  pthread_once(&init_done, thread_init);
  ...
}
```

- Get, Associate, and De-associate Data
  - `pthread_getspecific(3p)`: `void *pthread_getspecific(pthread_key_t key);`
    - return: thread-specific data, `NULL` if no value associated with `key`

    - `pthread_setspecific(3p)`: `int pthread_setspecific(pthread_key_t key, const void *value);`
        - return: 0 OK, error number on failure
        - non-NULL `value` to associate the data
        - NULL `value` to de-associate the data

- Example: A Thread-Safe Implementation of `getenv(3)`

```
static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex = PTHREAD_MUTEX_INITIALIZER;
extern char **environ;
static void thread_init(void) { pthread_key_create(&key, free); }

char *getenv(const char *name) {
  int i, len;
  char *envbuf;

  pthread_once(&init_done, thread_init);
  pthread_mutex_lock(&env_mutex);
  envbuf = (char *)pthread_getspecific(key);
  if (envbuf == NULL) {
    envbuf = malloc(ARG_MAX);
    if (envbuf == NULL) {
      pthread_mutex_unlock(&env_mutex);
      return NULL;
    }
    pthread_setspecific(key, envbuf);
  }
  len = strlen(name);
  for (i = 0; environ[i] != NULL; i++) {
    if ((strncmp(name, environ[i], len) == 0) && (environ[i][len] ==
'=')) {
      strcpy(envbuf, &environ[i][len + 1]);
      pthread_mutex_unlock(&env_mutex);
      return envbuf;
    }
  }
  pthread_mutex_unlock(&env_mutex);
  return NULL;
}
```

Cancel Options

Threads and Signals

Threads and fork