Operating Systems 2017/2018 | API Threads

API Threads

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

For more information please consult

'Advanced Programming in the UNIX® Environment, 3rd Edition,

W. Richard Stevens and Stephen A. Rago, Addison Wesley'

Sections 11.1–11.6

Thread Identification

```
#include <pthread.h>

pthread_t pthread_self(void);
// * returns the thread ID of the calling thread
// * a thread ID is represented by the pthread_t data structure

int pthread_equal(pthread_t tid1, pthread_t tid2);
// * most implementations represent the thread IDs as integers
//   but a portable implementations can't assume that
// * portable implementations must use the pthread_equal()
//   function to compare thread IDs
// * returns nonzero if the thread IDs are equal, 0 otherwise
```

Thread Creation

```
#include <pthread.h>

int pthread_create(pthread_t *tidp, pthread_attr_t *attr,
                   void *(*start_rtn)(void *), void *arg);
// * creates a new thread of execution to start running the
//   start_rtn() function and sets tidp with the thread ID of
//   the newly created thread
// * the attr argument can be used to customize various thread
//   attributes, setting it to NULL creates a thread with the
//   default attributes
// * the arg argument can be used to pass information to the
//   start_rtn() function which takes arg as its single argument
// * returns 0 if successful, error number on failure
```

Thread Termination

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
// * terminates thread execution with return code rval_ptr,
//   without terminating the entire process (the same happens
//   when the thread simply returns from the start routine)
// * note that, if any thread within a process calls exit(),
//   then the entire process terminates
// * returns 0 if successful, error number on failure
```

Thread Termination

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);
// * waits for a specific thread to terminate and blocks until
//   the specified thread calls pthread_exit() or returns from
//   its start routine
// * the thread return code is then made available in the
//   rval_ptr argument
// * returns 0 if successful, error number on failure
```

Thread Creation/Termination: Example

```
void run_thread(long arg, long ret) {
  pthread_t tid;
  void *rval;
  if (pthread_create(&tid, NULL, thr_fun, (void *) arg) != 0)
    { /* pthread_create error */ }
  ...
  if (pthread_join(tid, &rval) != 0) { /* pthread_join error */ }
  ret = (long) rval;
}
void *thr_fun(void *thr_arg) {
  long ret, arg = (long) thr_arg;
  ... // do something
  return ((void *) ret);
}
```

Thread Synchronization

- When multiple threads of control share the same memory, we need to make sure that **all threads see a consistent view of the data**
  - If each thread uses variables that other threads don't read or modify, no consistency problems will exist
  - If a variable is read-only, there is no consistency problem with more than one thread reading its value at the same time
  - If one thread can modify a variable that other threads can also read or modify, we need to **synchronize the threads** to ensure that they don't read or write an invalid value when accessing the variable's memory contents

Thread Synchronization: Problem Example

Thread A	Thread B	Contents of i
fetch i into register (register = 5)		5
increment the contents of the register (register = 6)	fetch i into register (register = 5)	5
store the contents of the register into i (register = 6)	increment the contents of the register (register = 6)	6
	store the contents of the register into i (register = 6)	6

Mutexes

- To protect our data and **ensure access by only one thread at a time**, we can use the pthreads mutual-exclusion (mutex) interface

- A **mutex is basically a lock** that we set (lock) before accessing a shared resource and release (unlock) when we're done
  - If a mutex is locked, any thread that tries to set it will block until it is released
  - If more than one thread is blocked when a mutex is unlock, then the first thread to run will be able to set the lock
  - The others will see that the mutex is still locked and go back to waiting for it to become available again
  - In this way, **only one thread will proceed at a time**

Mutexes

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
// * a mutex can be initialized by either setting it to
//   PTHREAD_MUTEX_INITIALIZER (if statically allocated) or
//   by calling pthread_mutex_init() (if dynamically allocated)
// * for a dynamically allocated mutex, we need to call
//   pthread_mutex_destroy() before freeing its memory
// * the attr argument customizes various mutex attributes,
//   a NULL value initializes it with the default attributes
// * both functions return 0 if OK, error number on failure
```

Mutexes

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// * pthread_mutex_lock() locks a mutex and, if the mutex is
//   already locked, it blocks the calling thread until the
//   mutex is unlocked
// * pthread_mutex_trylock() locks a mutex but, if the mutex is
//   already locked, it fails without blocking
// * pthread_mutex_unlock() unlocks a mutex
// * all functions return 0 if OK, error number on failure
```

Static Mutex: Example

```
void mutexes_fun() {
  pthread_mutex_t mutex_static = PTHREAD_MUTEX_INITIALIZER;
  ...
  execute_critical_region(&mutex_static);
  ...
}

void execute_critical_region(pthread_mutex_t *mutex) {
  pthread_mutex_lock(mutex);
  ... // critical region
  pthread_mutex_unlock(mutex);
}
```

Dynamic Mutex: Example

```
void mutexes_fun() {
  struct xpto *str_xpto;
  if ((str_xpto = malloc(sizeof(struct xpto))) != NULL) {
    // struct initialization
    pthread_mutex_init(&str_xpto->mutex_dynamic, NULL);
    ...
  }
  ...
  execute_critical_region(&str_xpto->mutex_dynamic);
  ...
  pthread_mutex_destroy(&str_xpto->mutex_dynamic);
  free(str_xpto);
}
```