Linux System Programming Part 9 - Threads (synchronization)

IBA Bulgaria
2018

Exiting the Threads

A thread can arrange for functions to be called when it exits. These functions are known as **thread cleanup handlers**. More than one cleanup handler can be established for a thread. The handlers are recorded in a **stack**, which means that they are executed in the reverse order from that with which they were registered.

When a thread is canceled or terminates by calling **pthread_exit**(), all of the stacked clean-up handlers are popped and executed in the reverse of the order in which they were pushed onto the stack.

Managing thread exit with C

```c
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

- The **pthread_cleanup_push**() function pushes routine onto the top of the stack of clean-up handlers. When routine is later invoked, it will be given arg as its argument.
- The **pthread_cleanup_pop**() function removes the routine at the top of the stack of clean-up handlers, and optionally executes it if execute is nonzero.

Clean thread exit

- Define exit function, which frees up the allocated memory
- Define thread function:
  Lock the thread cancellation
- Allocate 1K memory and print notification
- Add exit function passing the new memory as parameter
- Unlock the thread cancellation
- Print 4 messages in 4 seconds
- Pop exit function, executing it

exittest.c

```c
void exit_func(void * arg)
{
  free(arg);
  printf("Freed the allocated memory.\n");
}
void * thread_func(void * arg)
{
  int i;
  void * mem;
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  mem = malloc(1024);
  printf("Allocated some memory.\n");
  pthread_cleanup_push(exit_func, mem);
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  for (i = 0; i < 4; i++) {
    sleep(1);
    printf("I'm still running!!!\n");
  }
  pthread_cleanup_pop(1);
}
```

Why is synchronization needed?

Thread synchronization is the concurrent execution of two or more threads that share critical resources.

Threads should be synchronized to avoid critical resource use conflicts.

Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time.

Synchronization mechanisms

The threads library provides three synchronization mechanisms:

- **mutexes** - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- **joins** - Make a thread wait till others are complete (terminated).
- **condition variables** - While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

We already know about joins, so we'll focus on mutexes now...

Mutexes

Mutexes are used to prevent data inconsistencies due to operations by multiple threads upon the same memory area performed at the same time or to prevent **race conditions** where an order of operation upon the memory is expected.

A contention or **race condition** often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

Mutexes are used for serializing shared resources such as memory. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.

Mutex example

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

In the above example, a mutex should be used to lock the "Balance" while a thread is using this shared data resource.

Mutex example

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Read balance: $1200	$1200
	Deposit $200	$1200
	Update balance $1200+$200	$1400

In the above example, thread 1 locks the balance resource and thread 2 is blocked until the mutex is released.

Typical use of mutexes

A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

When several threads compete for a mutex, the losers block at that call - an unblocking call is available with **trylock**() instead of the **lock** call.

When protecting shared data every thread that needs to use a mutex should do so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

Problems with threads - race conditions

Race conditions: While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they are created.

They may also execute at different speeds. When threads are executing (racing to complete) they may give unexpected results (race condition).

Mutexes and **joins** must be utilized to achieve a predictable execution order and outcome.

Problems with threads - thread safe code

Thread safe code: The threaded routines must call functions which are "thread safe".

This means that there are no static or global variables which other threads may clobber or read assuming single threaded operation. If static or global variables are used then **mutexes** must be applied or the functions must be re-written to avoid the use of these variables.

In C, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is thread-safe.

Thread-unsafe functions may be used by only one thread at a time in a program and the uniqueness of the thread must be ensured.

Problems with threads - mutex deadlock

Mutex Deadlock: This condition occurs when a mutex is applied but then not "unlocked".

This causes program execution to halt indefinitely. It can also be caused by poor application of **mutexes** or **joins**.

Be careful when applying two or more mutexes to a section of code. If the first **`pthread_mutex_lock`** is applied and the second **`pthread_mutex_lock`** fails due to another thread applying a mutex, the first mutex may eventually lock all other threads from accessing data including the thread which holds the second mutex.

The threads may wait indefinitely for the resource to become free causing a deadlock. It is best to test and if failure occurs, free the resources and stall before retrying.

Creating and destroying mutexes

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Mutex variables must be declared with type **`pthread_mutex_t`**, and must be initialized before they can be used. There are two ways to initialize a mutex variable:

- Statically, when it is declared.
- Dynamically, with the **`pthread_mutex_init`**() routine. This method permits setting mutex object attributes, attr. **The mutex is initially unlocked.**

Locking and unlocking mutexes

```c
#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()`** – used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- **`pthread_mutex_trylock()`** – will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
- **`pthread_mutex_unlock()`** – will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data.

Mutex example

Define mutex and **balance** variables

Define thread function:
Lock the mutex

Add the **deposit** amount to the **balance**

Print the **balance** and unlock the mutex

Create thread one with **deposit** = 200, notify if failed

Create thread two with **deposit** = 200, notify if failed

Wait for both threads and exit with success

mutex.c

```c
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  balance = 1000;
void *deposit(void *arg) {
   pthread_mutex_lock( &mutex1 );
   int dep_amount = * (int *) arg;
   balance += dep_amount;
   printf("balance value: %d\n",balance);
   pthread_mutex_unlock( &mutex1 );
}
void main() {
   int rc1, rc2;
   pthread_t thread1, thread2;
   int deposit_amount = 200;
   if( (rc1=pthread_create( &thread1, NULL, &deposit,
                            &deposit_amount)) ) {
      printf("Thread creation failed: %d\n", rc1);
   }
   if( (rc2=pthread_create( &thread2, NULL, &deposit,
                            &deposit_amount)) ) {
      printf("Thread creation failed: %d\n", rc2);
   }
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);
   exit(EXIT_SUCCESS);
}
```

Synchronizing with semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

`sem_init()` – initializes the unnamed semaphore at the address pointed to by **`sem`**. The **`value`** argument specifies the initial value for the semaphore.

`sem_wait()` – decrements (locks) the semaphore pointed to by **`sem`**. If the semaphore currently has the value zero, then the call blocks.

`sem_post()` – increments (unlocks) the semaphore pointed to by **`sem`**.

`sem_destroy()` - destroys the unnamed semaphore at the address pointed to by **`sem`**.

Semaphore Threads

Initialize a **semaphore**

Define thread function - Process the argument and release **semaphore**.

In main function -
Set the common **id** variable to 1

Initialize **semaphore** with 0 (busy)

Start the first thread and pass **id**

Wait the **semaphore**, set **id** to 2 and start the second thread with the new **id**

Join both threads and destroy **semaphore**

```c
sem_t sem;
void * thread_func(void * arg)
{
    int i;
    int loc_id = * (int *) arg;
    sem_post(&sem);
    ...
}
int main(int argc, char * argv[]) {
    int id, result;
    pthread_t thread1, thread2;
    id = 1;
    sem_init(&sem, 0, 0);
    result = pthread_create(&thread1, ...
    ...
    sem_wait(&sem);
    id = 2;
    result = pthread_create(&thread2, ...
    ...
    result = pthread_join(thread1, NULL);
    ...
    result = pthread_join(thread2, NULL);
    ...
    sem_destroy(&sem);
```

Exercise

Project *AdvancedChat*:

Modify the network server/client pair of programs from the previous lecture with the following changes:

- The server should support additional **optional** argument '**-p**', which if present, identifies that the communication is in '**polite mode**'. In this mode, the server should accept the clients' messages in rounds - when a client sends a message the next message from the same client will not be processed until **all** other connected clients send their messages. You may change the communication protocol between clients and server if you need to.
- The server should store the chat history into a file '**chat.log**'.