CS415 Module 3 Part C - Thread Safety and Thread Level Storage

Athens State University

Contents

1 Thread Safety

Thread Safety

- **Thread-safe**: a function is *thread-safe* if it can be safely invoked by multiple threads at the same time
- This means that a function that is not thread-safe cannot be called from one thread if it is being executed in another thread

Thread Safety: What Things Make This Function Non-thread-safe?

```c
static int glob = 0;

static void incr(int loops) {
   int loc, j;
   for (j = 0; j < loops; j++) {
     loc = glob;
     loc++;
     glob = loc;
   }
}
```

If multiple threads invoke this function, then the final value of `glob` cannot be predicted. This is one of the primary reasons that a function will not be thread-safe: the use of global or static variables that are shared by all threads. It's a special case of resource sharing.

Fixing the problem requires that we have finer-grained synchronization mechanisms that what we get out of system calls such as `waitpid()` or `pthread_join()`. We'll look at those in the next module.

Thread-safeness and System Libraries

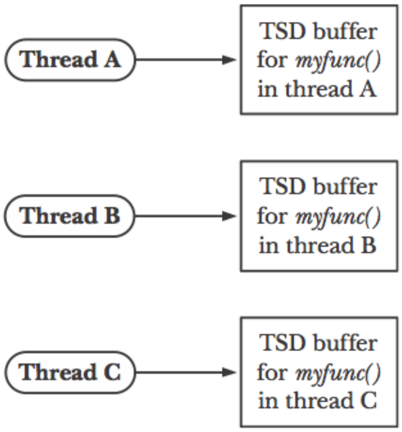
- Much as we saw with async-safe functions, there are a set of functions in the C/C++ libraries that are not thread-safe (see Table 31-1 in the Kerrisk book)
- One must be particularly careful with input and output functions
-

Re-entrant Functions

- Recall that a *reentrant* function is a thread-safe function that avoids the use of global and static variables
- Not all functions can be made reentrant:
  - Functions such as `new` and `malloc` as they must maintain a global list of free blocks on the heap
  - Some functions have an interface that is, by definition, nonreentrant as they return pointers to storage statically allocated by the function or statically maintain state between calls

2 Thread-Specific Data

Thread-Specific Storage


Thread A
TSD buffer for *myfunc()* in thread A
Thread B
TSD buffer for *myfunc()* in thread B
Thread C
TSD buffer for *myfunc()* in thread C

- Thread-specific storage (TLS) allows a function to maintain a separate copy of a variable for each thread that calls a function
- TLS is persistent... for the thread, it behaves as if global
- But it has a performance impact

Some design considerations

- A function using TLS must allocation a separate block of storage for each thread that calls the function
- Functions need to be able to access this storage
- Different function may each need data from TLS. Functions must be able to identify their data in TLS
- Function has no direct control over what happens when the thread terminates, so special code is required to clean up TLS

2.1 The Pthreads Thread-Specific Data API

The Steps to Follow to use TLS with Pthreads

- Create a key using `pthread_key_create()` passing a pointer to the function that serves as the destructor of the TLS data
- Use the C `malloc()` function to allocate space for each thread that is using TLS
- Use the `pthreads_setspecific()` and `pthreads_getspecific()` functions to set and get the pointer used to access TLS data

Example: Using Pthreads TLS

```c
include <stdio.h>
#include <string.h>
#include <pthread.h>
static void *threadFunc(void *arg) {
   char *str;
   str = strerror(EPERM)
   return NULL
}
int main(int argc, char *argv[]) {
   pthread_t t;
   int s;
   char *str;
   str = strerror(EINVAL)
   s = pthread_create(&t, NULL, threadFunc, NULL);
   if (s != 0)  exit(s);
   s = pthread_join(t, NULL);
   if (s != 0) exit(s);
   exit(EXIT_SUCCESS);
}
```

Not good. Rewrite strerror() to be thread safe

```c
char *strerror(int err) {
   int s;
   char *buff;
   s = pthread_once(&once, createKey);
   if (s != 0) exit(s);
   buf = pthread_getspecific(strerrorkey);
   if (buf == NULL) // first call, malloc buffer
   {
     buf = malloc(MAX_ERROR_LEN);
     if (buf == NULL) exit(-1);
     s = pthread_setspecific(strerrorkey, bufe);
     if (s != 0) exit(s);
   }
   if (eff < 0 || err >= _sys_nerr || _sys_errlist[err] == NULL) {
      sprintf(buf, MAX_ERROR_LEN, "Unknown error %d"), err);
   } else {
      strncopy(buf, _sys_errlist[err], MAX_ERROR_LEN - 1);
      buf[MAX_ERROR_LEN - 1] = '\0'
   }
}
```

```c
#include <stdio.h>
#include <string.h>
#include <pthread.h>

static pthread_once_t once = PTHREAD_ONCE_INIT;
static pthread_key_t strerrorkey;
#define MAX_ERROR_LEN 256
static void destructor(void *buf) { free(buf); }

static void createKey(void) {
  int s;
  s = pthread_key_create(&strerrorkey, destructor);
  is (s != 0) exit(s);

}
```