CS415 Module 3 Part B - Introduction to Pthreads

Athens State University

Contents

1 What is a thread?

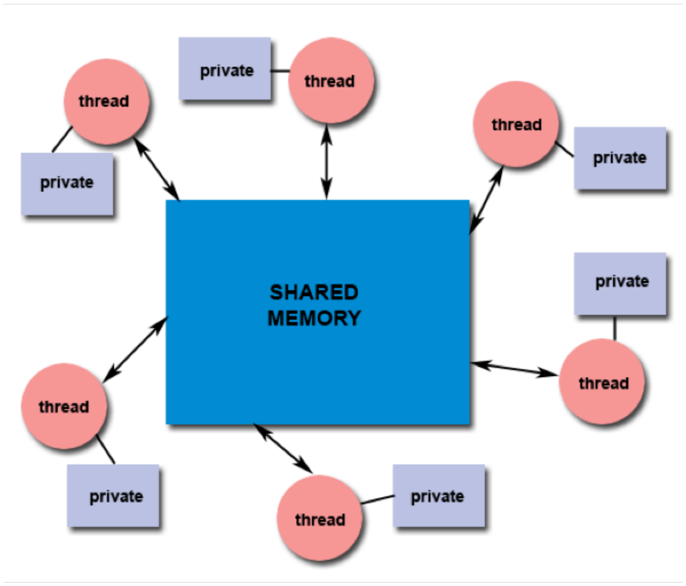
A thread:

- Exists within a process and uses the process's resources
- Has its own independent flow of control so long as the parent process exists
- Duplicates only the essential resources it needs to be independently scheduled
- May share the process resources with other threads
- Dies if the parent process dies - or something similar
- Is "lightweight" because most of the overhead was dealt with by creation of the process

Threads share resources

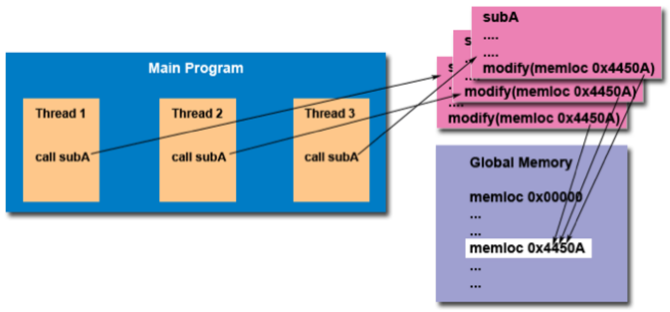
- Changes made by one thread to shared system resources will be seen by all threads
  - Think about what happens when you close a file
- Two pointers having the same value point to the same data
- Reading and writing to the same memory location is possible
  - Requires explicit synchronization by the developer

Shared Memory Model



- All threads have access to the same global memory
- Threads also have their own private data
- Programmers have to make certain threads don't stomp on each other

Thread-safeness



- **Thread-safe**: the ability of an application to execute multiple threads without negatively impacting other threads
- Must be very careful about the libraries you use with multi-threaded programs

2 The Pthreads Library

Pthreads

- A POSIX standard API for thread creation and synchonization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, most implementation details left up to the developer
- Common in UNIX operating systems (Linux, Solaris, macOS)

2.1 Cross-library concepts

Cross-library Concepts: Pthreads data types

Data type	Description
pthread_t	Thread identifier
pthread_mute_t	Mutex
pthread_mutexattr_t	Mutex attributes object
pthread_cond_t	Condition variable
pthread_condattr_t	Condition variable attributes object
pthread_key_t	Key for thread-specific data
pthread_once_t	One-time initialization control context
pthread_attr_t	Thread attributes object

The POSIX `pthreads` standard does not specify how these data types should be represented. Result is that you have to treat them as opaque data: your program must avoid any reliance on knowledge of the internal structure of data of these types. One particular annoyance is the fact you cannot do comparisons of variables of these types using the C/C++ `==` operator.

Cross-library Concepts: Threads and `errno`

- The `errno` system variable is a global variable defined in the system header files that contains the error number of the most recently processed error
- Threads invalidate this assumption, the compiler does some voodoo with compiler macros to create a thread-specific `errno` in thread-safe storage
- The macros allow you to continue to use `errno` in the common manner

Cross-library Concepts: Return Value from `Pthreads` functions

- The coding pattern for returning status from system calls is to return a 0 on success and -1 on error
- The Pthreads API uses a different convention: return 0 on success or a positive value containing the system error number for the error
- So, don't directly assign the return value of a call to a `Pthreads` function to `errno`, capture it into a local variable and process accordingly

Cross-library Concepts: Compiling and Linking

- Use the `-pthread` option on the compiler and linker command lines
- This option forces the inclusion of thread management code in your object files
- Links your program with the GNU `libpthread` library

2.2 Working with `Pthreads`

Thread Creation: `pthread_create()`

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start)(void *),
                   void *arg);
```

A call to this function starts a new thread running within your process. The new thread commences execution by calling the function identified by the function pointer `start` using the value of `arg` as an argument.

The calling function continues execution at the next statement following the call to `pthread_create()`.

The `thread` parameter points to a buffer into which the thread id is copied before `pthread_create()` returns. One needs to careful here as there are timing issues about when this happens.

The `attr` parameter is a pointer to an `pthread_attr_t` object whose values are used by `pthread_create` to configure settings for the new thread.

Note the extensive use of the `void *` construct. This means we can pass a pointer to `start()` that points to any type of object. If you have nothing to pass to the `start()` function, then set the value of `arg` to be `NULL`.

If you need to pass multiple values to the `start()` function, then have `arg` point to a structure that contains the arguments as separate fields.

Note that the return type of `start()` is `void *`. Do avoid using a casted integer as a return value from `start()` as this will conflict with system values within the `Pthreads` library.

Thread Termination

Execution of thread terminates in one of the following ways:

- The thread's `start()` function performs a `return` specifying a return value for the thread
- The thread explicitly exits by calling `pthread_exit()`
- The thread is canceled via `pthread_cancel()`
- Any of the threads calls `exit()` or the main thread performs a `return` in the `main()` function, which causes all threads to terminate immediately

Thread Termination: `pthread_exit()`

```
#include <pthread.h>

void pthread_exit(void *retval);
```

- Calling this function is equivalent to calling `return` in a thread's `start()` function.
- Difference is that this function can be called from any function called from the thread's `start()` function

The `retval` argument specifies the return value for the thread. The value pointed to by `retval` *`MUST NOT`* be located on the thread's stack as a thread's stack is deleted when a thread terminates.

If the main thread calls `pthread_exit()` instead of calling `exit()` or performing a `return`, then the other threads in the program will continue to execute.

Joining Theads: `pthread_join()`

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

- This function waits for the identified thread to terminate
- If `retval` is a non-`NULL` pointer, then it receives a copy of the terminated thread's return value

Never call `pthread_join()` for a thread ID that has been previously joined as that leads to unpredictable behavior.

2.3 A Few Examples

Hello World, Pthreads style

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   int tid;
   tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Simple create and join

```c
#include <pthread.h>
static void *threadFunction(void *arg) {
  char *s = (char *)arg;
  printf("%s",s);
  return (void *) strlen(s);
}

int main(int argc, char *argv) {
  pthread_t t1;
  void *res;
  int s;
  s = pthread_create(&t1, NULL, threadFunc, "Hello World\n");
  if (s != 0) {
     exit(s)
  }
  printf("Message from main()\n");
  s = pthread_join(t1, &res);
  if (s != 0) {
     exit(s);
  }
  printf("Thread returned %ld\n", (long) res);
  exit(EXIT_SUCCESS);
}
```

2.4 Working with Thread IDs

Self and equal

```c
#include <pthread.h>

pthread_t pthread_self(void);

int pthread_equal(pthread_t t1, pthread_t t2);
```

- `pthread_equal()` returns a nonzero value if the two threads are equal, otherwise return 0

Detaching a Thread

```c
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

- This marks a thread as *detached*, when means we don't care the thread's return status, just clean up and remove the thread when it terminates.