Pthreads Overview

What is a Thread?

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
- To go one step further, imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.

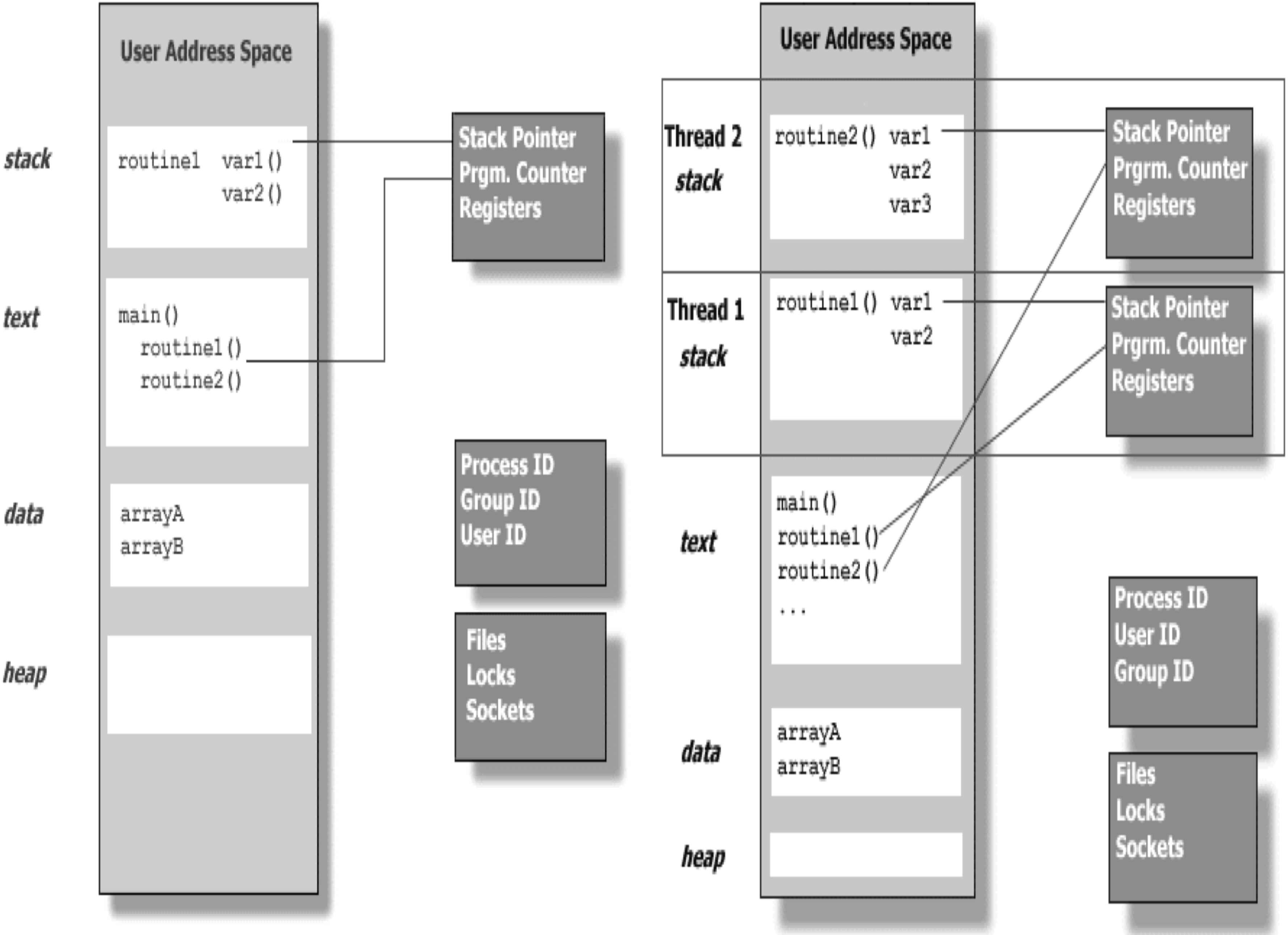
Pthreads Overview

Why threads?

- As process is created by the operating system, it requires a fair amount of information about program resources and program execution state, including:
- Process ID, process group ID, user ID, and group ID
- Working directory.
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Pthreads Overview

- Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities.
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- This independent flow of control is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.
- Dies if the parent process dies.
- Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.



User Address Space
stack
routine1
var1()
var2()
text
main()
routine1()
routine2()
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets
User Address Space
Thread 2
stack
routine2()
var1
var2
var3
Thread 1
stack
routine1()
var1
var2
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Stack Pointer
Prgm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets

The Pthreads API

- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:
- **Thread management:** The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
- **Mutexes**: The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- Naming conventions: All identifiers in the threads library begin with **pthread_**

Creating Threads

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

```
#include <pthread.h>;
int pthread_create (pthread_t *thread, const pthread_attr_t
                    *attr, void *(*start_routine) (void *), void *arg) ;
```

Description

- pthread_create creates a new thread and makes it executable.
- Typically, threads are first created from within main() inside a single process.
- Once created, threads are peers, and may create other threads.
- The new thread inherits its creating thread's signal mask; but any pending signal of the creating thread will be cleared for the new thread.
- The maximum number of threads that may be created by a process is implementation dependent.

Creating Threads

pthread_create arguments:

- thread: An opaque, unique identifier for the new thread returned by the subroutine.
- attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- start_routine: the C routine that the thread will execute once it is created.
- arg: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Return Values

- If successful, the pthread_create function returns zero. Otherwise, an error number is returned to indicate the error.

Threads Syncronization

int **pthread_join** (pthread_t th, void **thread_return)

- pthread_join suspends the execution of the calling thread until the thread identified by th terminates, either by calling pthread_exit or by being canceled.
- If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return.
- The joined thread the must be in the joinable state: it must not have been detached using pthread_detach.
- When a joinable thread terminates, its memory resources (thread descriptor and stack) are not deallocated until another thread performs pthread_join on it. Therefore, pthread_join must be called once for each joinable thread created to avoid memory leaks.
- At most one thread can wait for the termination of a given thread. Calling pthread_join on a thread th on which another thread is already waiting for termination returns an error.

Thread Termination

void **pthread_exit** (void *retval)

- pthread_exit terminates the execution of the calling thread.
- The retval argument is the return value of the thread. It can be retrieved from another thread using pthread_join.
- The pthread_exit function never returns.

pthread_t **pthread_self** (void)

- pthread_self returns the thread **identifier** for the calling thread.

Mutexes

- A mutex is a MUTual EXclusion device is useful for protecting shared data structures from concurrent modifications.
- A mutex has two possible states:
  - unlocked (not owned by any thread), and
  - locked (owned by one thread).
- A mutex can never be owned by two different threads simultaneously.
- A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.
- None of the mutex functions is a cancellation point, not even pthread_mutex_lock, in spite of the fact that it can suspend a thread for arbitrary durations.

Mutexes

- The mutex is initially unlocked.

int **pthread_mutex_lock** (*pthread_mutex_t *mutex*))

- pthread_mutex_lock locks the given mutex.
- If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and pthread_mutex_lock returns immediately.
- If the mutex is already locked by another thread, pthread_mutex_lock suspends the calling thread until the mutex is unlocked.

int **pthread_mutex_trylock** (*pthread_mutex_t *mutex*)

- pthread_mutex_trylock behaves identically to pthread_mutex_lock, except that it does not block the calling thread if the mutex is already locked by another thread.
- Instead, pthread_mutex_trylock returns immediately with the error code EBUSY.

Mutexes

int **pthread_mutex_timedlock** (*pthread_mutex_t *mutex, const struct timespec *abstime*)

- The pthread_mutex_timedlock is similar to the pthread_mutex_lock function but instead of blocking for in indefinite time if the mutex is locked by another thread, it returns when the time specified in *abstime* is reached.

int **pthread_mutex_unlock** (*pthread_mutex_t *mutex*)

- pthread_mutex_unlock unlocks the given mutex.

int **pthread_mutex_destroy** (pthread_mutex_t *mutex)

- pthread_mutex_destroy destroys a mutex object, freeing the resources it might hold.
- If the mutex is locked by some thread, pthread_mutex_destroy returns EBUSY. Otherwise it returns 0.