Linux Kernel Training

Concurrency and Synchronization

General Concepts

Agenda

1. Processes & Threads
    - Manual creation and management
2. Mutexes
3. Conditional Variables
4. Semaphores
5. Pros & Cons

Process

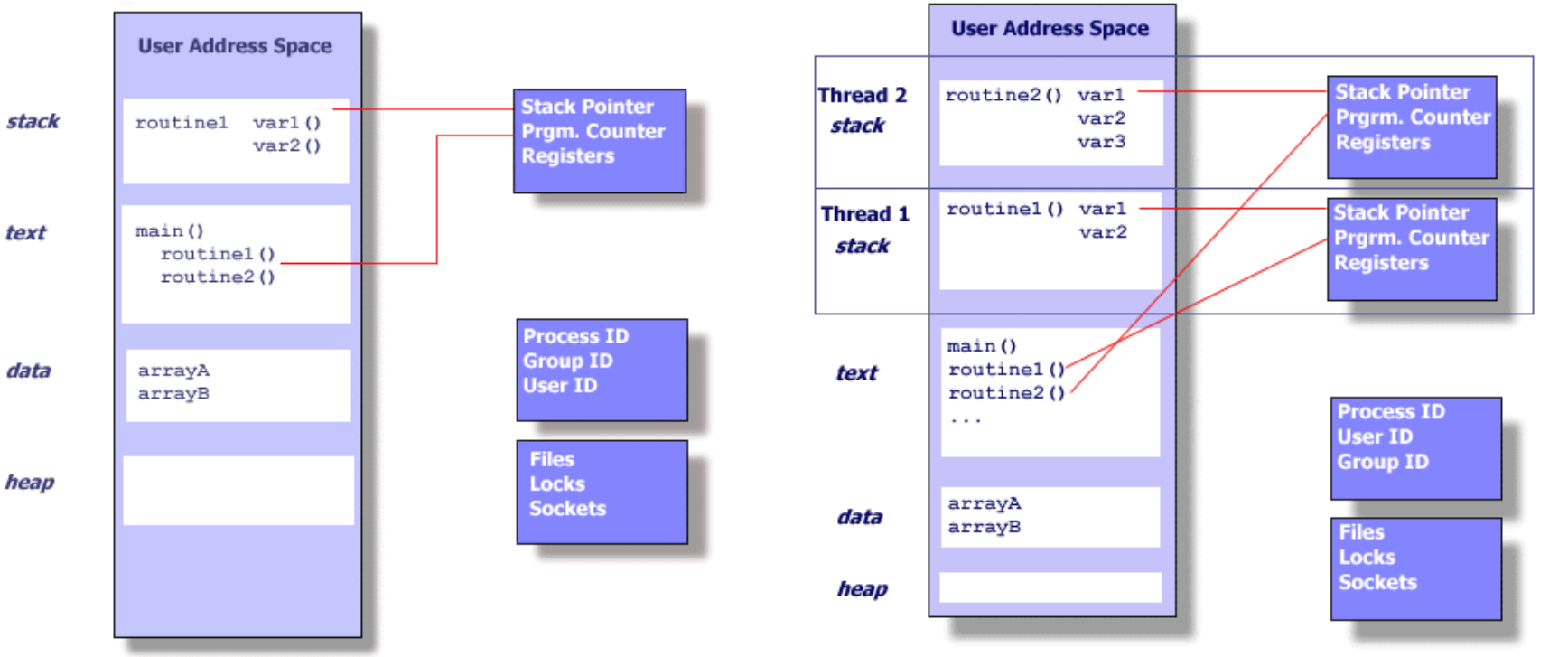
Processes contain information about program resources and program state:

- PID,PGID, UID, GID, EUID, EGID, etc.
- Environment
- Working directory
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (message queues, pipes, semaphores, or shared memory).

A Thread?

- Definition: sequence of related instructions executed independently of other instruction sequences
- A thread can create another thread
- Each thread maintains its current machine state
- Relationship between user-level and kernel-level threads – 1:1 (user-level thread maps to kernel-level thread)
- Threads share same address space but have their own private stacks
- Thread states: ready, running, waiting (blocked), or terminated

Process and Threads within a

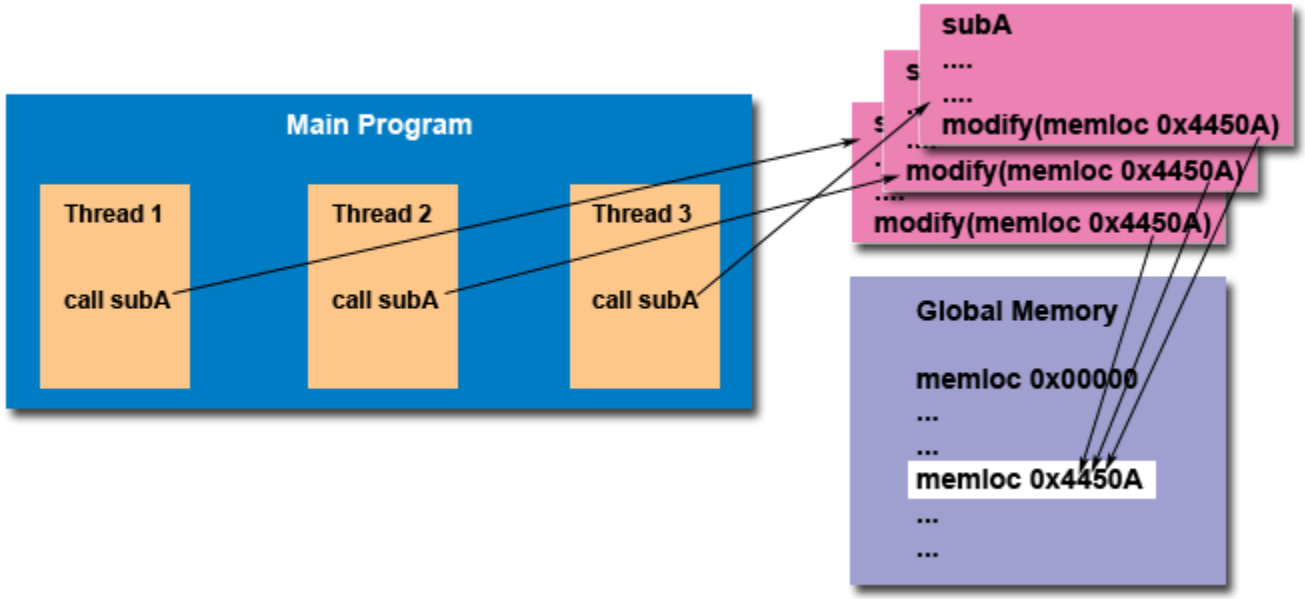


Thread-safeness

Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions. For example, suppose that your application creates several threads, each of which makes a call to the same library routine:

- This library routine accesses/modifies a global structure or location in memory.
- As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
- If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.

Thread-safeness



The Pthreads API

The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

Thread management: Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes: Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

The Pthreads API

Condition variables: Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
Synchronization: Routines that manage read/write locks and barriers.

Naming conventions: All identifiers in the threads library begin with **pthread_**.

The Pthreads API

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Thread Creation

Initially, main() comprises a default thread.

pthread_create(thread, attr, start_routine, arg) creates a new thread and makes it executable.

- 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, 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.

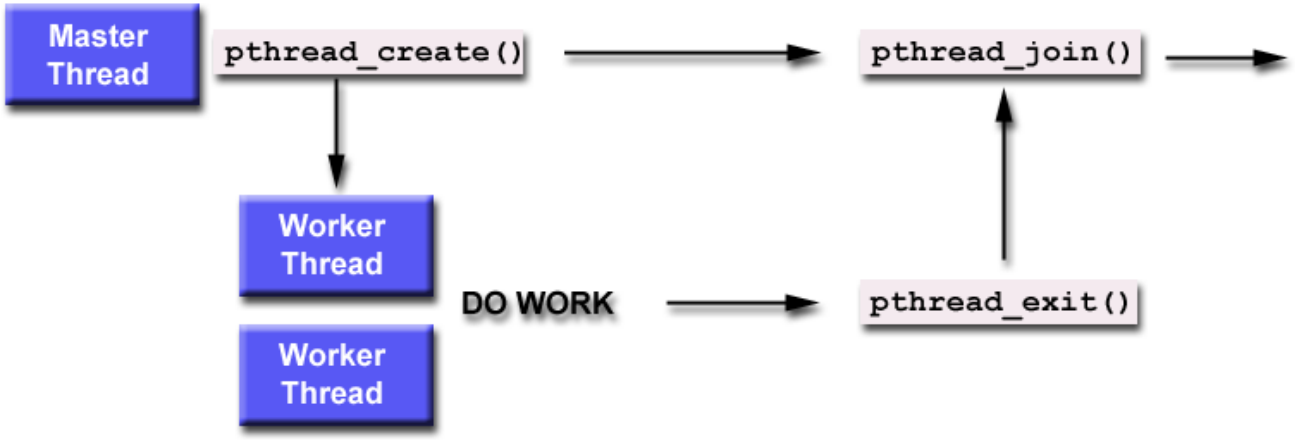
Thread Termination

pthread_exit (status)
pthread_cancel (thread)
pthread_attr_init() and pthread_attr_destroy() are used to initialize/destroy the thread attribute object.
Other routines are then used to query/set specific attributes in the thread attribute object. Attributes include:

- Detached or joinable state
- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Scheduling contention scope
- Stack size
- Stack address
- Stack guard (overflow) size

Joining and Detaching Threads

pthread_join (threadid,status)
pthread_detach (threadid)
pthread_attr_setdetachstate (attr,detachstate)
pthread_attr_getdetachstate (attr,detachstate)



Mutexes

Blocking mutual exclusion lock.
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_init (mutex,attr)
pthread_mutex_destroy (mutex)

pthread_mutexattr_init (attr)
pthread_mutexattr_destroy (attr)

pthread_mutex_lock (mutex)
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)

Conditional Variables

They allow threads to synchronize based upon the actual value of data.
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;

pthread_cond_init (condition,attr)
pthread_cond_destroy (condition)

pthread_condattr_init (attr)
pthread_condattr_destroy (attr)

pthread_cond_wait (condition,mutex)
pthread_cond_signal (condition)
pthread_cond_broadcast (condition)

Semaphore

Semaphore is a typically controls access to limited resources, i.e. permit a limited number of threads to execute a section of the code

- similar to mutexes
- should include the semaphore.h header file
- semaphore functions have sem_ prefixes

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
int sem_destroy(sem_t *sem)
int sem_post(sem_t *sem)
int sem_wait(sem_t *sem)
```

Mutex vs. Semaphore

The Toilet Example (c) Copyright 2005, Niclas Winquist ;)

Mutex:
Is a key to a toilet. One person can have the key - occupy the toilet - at the time. When finished, the person gives (frees) the key to the next person in the queue.

Semaphore:
Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.

Pros & Cons

Paper by Edward Lee, 2006

• The author argues:

– “From a fundamental perspective, threads are seriously flawed as a computation model”

– “Achieving reliability and predictability using threads is essentially impossible for many applications”

• The main points:

– Our abstraction for concurrency does not even vaguely resemble the physical world.

– Threads are dominating but not the best approach in every situation

– Yet threads are suitable for embarrassingly parallel applications

Conclusions

The logic of the paper:

– Threads are nondeterministic

– Why shall we use nondeterministic mechanisms to achieve deterministic aims??

– The job of the programmer is to prune this nondeterminism.

– This leads to poor results

Be wise