One to one user threading model

Thread create

- In one to one model each **user thread** will **map** to one **kernel thread**. Hence, **no user level scheduling** is required. Scheduling is managed by the kernel scheduler itself.
- A single process may have many threads. All the threads in the same process will have the **same parent process id, process id**. The only **different** identification will be the **thread id**.
- The creation of thread in this case will involve creating a schedulable kernel entity. This is done using the **clone(2)** system call. The clone(2) call takes as argument the starting routine of the thread and its argument. Various flags are passed to clone(2) for enabling resource sharing between the caller thread and the cloned thread. Few flags are required to initialize the thread id of the new thread. Few more flags are used to set hardware registers with some values.
- Each thread will share many resources with other threads in the same process, but will have seperate stack and register file. For this seperate stack is to be allocated for each thread. The stack is allocated using the **mmap(2)** system call. Each thread is allocated a stack of maximum size. A stack guard is provided below each stack to prevent the stack from increasing beyond the maximum size.
- For user level thread management the state of each thread is maintained in a structure called **Thread Control Block (TCB)**. This structure stores all the information required for implementing all the features provided by this library. For ex. it contains thread id, address of the start function, address of the arguments, base address of the thread stack, size of the stack, etc. On creation of the thread, it's TCB is allocated and initialized accordingly.
- The user may perform operations of the thread or using the thread. In order to identify the threads in the application program using the thread library, a **thread handle** is provided, which is nothing but the address of its TCB.
- For generalizing the start routine signature and providing generalized arguments (i.e. changing the **start function signature** as compared to what is supported by clone), the start function provided by the user is **wrapped** around an **actual** thread **start function**. This function launches the start function specified by the user with the specified arguments.

Thread join

- A thread may **wait** for the **completion** of another thread. This is termed as waiting for a thread to join to the caller. It provides the return value of thread either provided by **exit** function or **return** statement.
- If the caller thread waits for an incomplete thread to join, then it will halt till the thread exits. This halt is implemented using the **futex(2)** system call.
- When the thread was created, a member variable of the TCB was set to its thread id. By passing a special flag to the clone(2) system call, it is made sure that this **member variable bears the thread id** as long as the thread is alive. However, by passing another special flag to clone, it is also made sure that this **member variable is set to zero** when the thread exits.
- Hence using the futex(2) call the caller thread waits for the target thread's TCB member variable to be zero. When the target thread exits, the member variable is set to zero and only one of the **waiting thread** is **woke up**.
- Even if a thread has been executed completely, none of its allocated resources data structures are freed. All the maintained **data structures** and **resources** of the thread are released, once a join is performed on it.
- A thread must be **joined only once**. Any further joins on already joined thread will produce unexpected results.

Thread exit

- A thread can **stop** its **execution** using the exit function. The thread can **return some value** while it is exiting.
- When the thread was created, the thread wrapper which is the actual start function of the thread, stores the **address of the code** where the **exit function must return** in is TCB. This is done using **setjmp(3)** function.
- When the thread exits, the **return value** is **stored** in its TCB. Then the exit function **jumps** to the address which was stored for exiting in the TCB. This is done using the **longjmp(3)** function.

Thread self

- The library maintains **no globals** for any kind of book-keeping of the threads. However it may happen that the currently running thread requires the **address of its own TCB**. This is provided by the thread self function.

- While creating the thread using the clone(2), **special flags** were provided to set the address of it's TCB in a special register called the **FS** register. This is a special **segment** register (or **selector**).
- Hence whenever the thread wants it's TCB address, then it can just read the **FS** register. This is done using the **arch_prctl(2)** system call

Thread spinlock

- The library provides **spinlock** as a user level **synchronization primitive**.
- A spinlock structure is defined which consists of an **integer** which is updated whenever the lock is acquired or released. The structure also carries the TCB address to identify the **owner** of the thread.
- As there is a **race** for **updating** the **integer** part of the lock, an atomic instruction is needed to atomically compare and exchange the integer part. For this **atomic_compare_exchange_strong(3)** function is used.
- The two **operations** that can be performed on the spinlock are **locking** and **unlocking**.
- While locking if a thread finds that the **lock is free**, then it acquires the lock and sets the **owner** of the lock as **itself**. However if the thread finds that the **lock** is already **acquired** then the thread **waits** for the lock to be released by the current owner. This is done using the futex(2) system call. If the thread finds that the lock is already acquired by itself, then it returns immediately.
- While unlocking the caller thread **releases** the lock and **wakes** up only one thread that is waiting for the same lock. This is also done using the futex(2) system call.
- futex(2) system call is used to **prevent processor consumption** while the thread is **busy waiting**. This call halts the waiting process until the one who holds the lock releases it.

Thread kill

- To **deliver signals** to the threads or to the thread group, the thread kill functionality is used.
- For this **tgkill(2)** system call is used to direct the signals to the target thread. Depending on the nature of the signal, it is directed to the entire thread group or only the thread.

Miscellaneous

Stack Smashing

gcc by default uses some protective measures to check if the program, has not accessed elements more than it can in the stack frame. This is done by placing **stack canary** in the stack. The stack canary is referred with respect to the **FS** register. **gcc** in the assembly prologue stores this value in the stack and then in the assembly epilogue confirms that this value was not updated if yes then **stack smashing** error is raised. However we are using that register for pointing to the TCB. As the TCB **members values** are **dynamic** i.e. they change during the course of execution, our program may result in stack smashing even if we are not violating any bounds. This is handled by padding some bytes in the TCB and not changing their value so that when **gcc** uses this value in prologue and epilogue it is found to be equal.

Structure padding

While handling the stack smashing problem some unexpected results were observed, as **gcc** was padding and aligning the member variables of the thread control block for efficiency. Hence the structure variables are placed at respective offsets currently so as to prevent any kind of padding.

Bugs

- Using **printf(3)** after creating threads with **clone(2)** system call, causes the segmentation fault.
- The spinlock initialization function does not prevent multiple initialization, which may cause multiple threads to enter critical section.