

Advanced Embedded Software - Homework 3

Que 2:

The APIs used in pthread are explored and an example code is created and submitted "1pthreads.c". The following is the explanation of some of the pthread APIs,

pthread_create():

This function is used to create the child threads and take off the load from the parent. The child will share the same address space with the parent thread. The function will give the PID of the new thread, and executes the function in which the thread should start.

pthread_join():

This function helps in synchronous threading by enabling the parent thread to wait for the child to finish its work before the parent exits. This allows to have return value from the child to parent thread as well.

pthread_exit():

This functions exits the calling thread with a return value. This is similar to calling a return statement.

pthread_self():

This function returns the ID of the calling thread.

pthread_getattr_np():

This function gets the attributes of a thread and stores it in the pthread_attr_t variable type. The thread can be selected using the first parameter.

pthread_getattr_default_np():

This function gets the default parameters of the thread creation and stores it in the pthread_attr_t variable type. Pthread_create uses these parameters to create a thread when the attribute argument is set to NULL.

pthread_set*:

There are many functions available to set a particular attribute in a pthread_attr_t variable and also a function which can set the default attributes of the system which can take effect when the pthread_create is passed with NULL attributes.

pthread_mutex_init():

This function is used to initialise the mutex variable used in the process. When the mutex is initialized it will be in unlocked state.

pthread_mutex_destroy():

This function is used to destroy the mutex allocated with pthread_init() function. The function sets the mutex to some invalid value. Destroying a locked mutex will result in undefined behavior.

pthread_mutex_lock():

The function locks the mutex object called from the thread. Mutex locks blocks access to other threads, so that synchronization is achieved. When a locked mutex is requested, the thread is moved into waiting state. Based on the mutex type, the mutex can take different functionalities like recursive and error checking.

pthread_mutex_trylock():

This function behaves same as mutex lock but does not wait on an unlocked mutex. It returns zero if a lock is acquired and an error number on other cases

pthread_mutex_unlock():

The function unlocks the acquired mutex and performs the opposite of the mutex lock. The scheduling policy is used to determine whether any other thread is waiting on the mutex.

pthread_cond_init():

The condition variables are used to control the order in which the mutexes are locked. For example, in a producer-consumer environment the consumer cannot read anything until the producer writes into it. The function initializes the condition variable with default attributes when a NULL attribute is passed. Upon success the condition variable is initialised.

pthread_cond_wait():

This function is a cancellation point. This function should be called after locking a mutex. This function forms a dynamic binding between the cond and the mutex object. The call releases the acquired mutex and waits on the condition variable. The condition wait is released when it is signaled by the other threads. Then the mutex will be acquired and the critical section will be performed.

pthread_cond_signal():

This function unblocks one of the threads which are blocked on a condition variable. The function may also be called from threads which doesn't own the dynamically linked mutex. This will have no effect when nothing is blocked on the condition. This can be used for producer-consumer problems.

pthread_cond_destroy():

This function destroys the cond object that was created. This should be done when no threads are using the condition variable. Referring to destroyed objects will show an undefined behavior.

In my example code two threads, are created from the parent. The second thread waits on the condition variable after acquiring the mutex and the first thread signals the condition variable. This helps in executing the thread 2 after thread 1.

The mutex lock, try lock and unlock is also used to synchronize the threads to read and write from a memory location.

pthread_getattr_default_np(), pthread_attr_setdetachstate(), pthread_attr_getschedpolicy() are some of the functions explored in this example.

Que 3: Profiling

In this question, we profiled the time taken to execute the thread creation APIs using the time library. The time library used for profiling in user space is clock_gettime() from time.h. The fork and exec and pthread_create are implemented in same function and are shown below,

Pthread_create and fork/exec:

The time taken for pthread create() and fork() and exec() are shown in the figure below. The time varied when executed many times but all the execution had pthread_create using less time

compared to fork/exec. pthread_create also does a clone/exec internally but the address space is shared among the parent and child. While the fork duplicates the entire address space.

```
raj@raj-Q504UA:~/ECEN_5013_AES/HW3/Que3$ sudo ./profiler.out
Time for pthread_create is 62515 ns
The pthread_create function
Time for fork and exec is 141796 ns
This is the parent
This is the end Fork Returned:27747
raj@raj-Q504UA:~/ECEN_5013_AES/HW3/Que3$ sudo ./profiler.out
Time for pthread_create is 82403 ns
The pthread_create function
Time for fork and exec is 83312 ns
This is the parent
This is the end Fork Returned:27751
raj@raj-Q504UA:~/ECEN_5013_AES/HW3/Que3$ sudo ./profiler.out
Time for pthread_create is 68734 ns
The pthread_create function
Time for fork and exec is 159146 ns
This is the parent
This is the end Fork Returned:27755
```

The clock_gettime() profiled using the CLOCK_THREAD_CPUTIME_ID clock Id, which gives the execution time of the current thread and the time difference of start and end time are taken.

Profiling kthread_create():

The kthread_create was profiled by creating the kernel module. The Time Stamp counter was used to profile the function in kernel space get_cycles() function gives the number of CPU cycles to execute kthread_create. This time was found to be 369844 CPU cycles. In architectures where a TSC register is not supported this will return 0.

```
[ 41419.273340] kthreadProfiler::Init
[ 41419.273500] CPU Cycles before kthread_create:40508056177152
[ 41419.273503] CPU Cycles after kthread_create:40508056547036
[ 41419.273506] CPU Cycles used for kthread_create:369884
[ 41421.385576] kthreadProfiler::Exit
```

Que 4: Kernel Linked List for Kernel Threads

This exercise helped us to explore linked list in the kernel. The kernel module was created and the program traversed from the current process towards all the parent and printed some data. This was achieved using the parent member in task_struct structure. The number of children for each parent is found by traversing the child list using list_for_each() macro.

The kernel module was built and installed in the kernel. The output of the kernel module is shown below,

```
[41421.385576] kthreadProfiler::Exit
[48805.162260] ParentFinder::Exit
[48827.660503] ParentFinder::Init
[48827.660509] Thread Name:insmod
[48827.660513] Thread Id:28945
[48827.660517] Thread State:0
[48827.660520] Number of Children:0
[48827.660524] Thread Name:sudo
[48827.660527] Thread Id:28944
[48827.660530] Thread State:1
[48827.660534] Number of Children:1
[48827.660537] Thread Name:bash
[48827.660540] Thread Id:25575
[48827.660543] Thread State:1
[48827.660546] Number of Children:1
[48827.660549] Thread Name:gnome-terminal-
[48827.660552] Thread Id:2286
[48827.660555] Thread State:1
[48827.660559] Number of Children:3
[48827.660562] Thread Name:upstart
[48827.660566] Thread Id:1424
[48827.660569] Thread State:1
[48827.660598] Number of Children:61
[48827.660601] Thread Name:lightdm
[48827.660604] Thread Id:1126
[48827.660607] Thread State:1
[48827.660611] Number of Children:1
[48827.660614] Thread Name:lightdm
[48827.660617] Thread Id:888
[48827.660620] Thread State:1
[48827.660624] Number of Children:2
[48827.660627] Thread Name:systemd
[48827.660631] Thread Id:1
[48827.660634] Thread State:1
[48827.660648] Number of Children:31
[48827.660652] Thread Name:swapper/0
[48827.660655] Thread Id:0
[48827.660658] Thread State:0
[48827.660661] Number of Children:2
[48831.514172] ParentFinder::Exit
raj@raj-Q504UA:~/ECEN_5013_AES/HW3/Que4$
```

It is seen that the Thread Id 1 is systemd which is the init thread, the thread Id 0 was also found which we have to discuss in class to find the details of it.

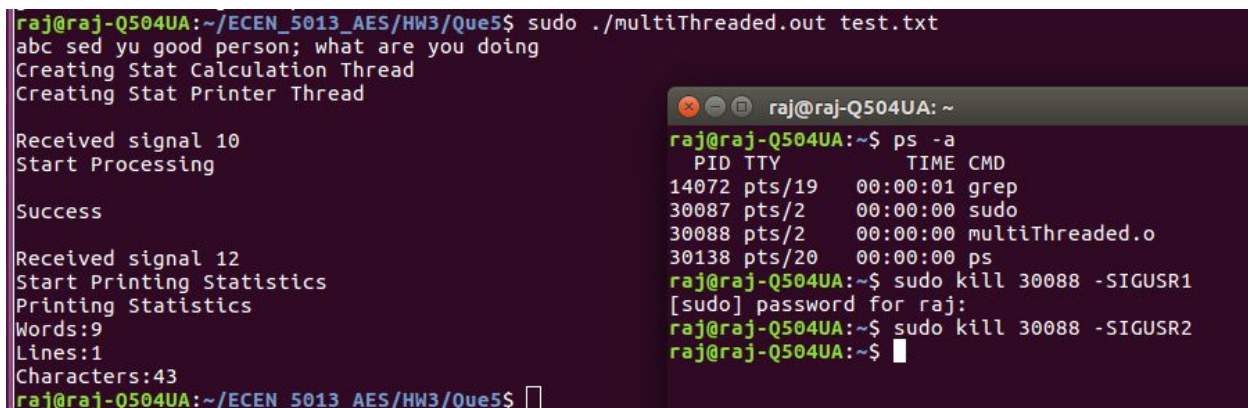
Que 5: Multithreaded I/O

The program implements creation of two threads from the parent process. The parent process(Thread 1) opens a file and gets data from the user and stores it in the file. After that two threads are created and is made to wait on a `sem_wait`.

The semaphore is signaled with the help of signal handler. A signal handler which implements for `SIGUSR1`, `SIGUSR2` and `SIGINT` was registered for the thread group with the help of `sigaction()`. When the `SIGUSR1` is sent from the command prompt, Thread 2 wakes up and process the data entered in the file by Thread 1 earlier. It then stores the values in a variable in a structure. The number of characters, words and lines are calculated.

When the `SIGUSR2` signal is sent, Thread 3 wakes up and prints the statistics on the screen. These are synchronized by the semaphore and mutexes.

The output of the program is shown below,



```
raj@raj-Q504UA:~/ECEN_5013_AES/HW3/Que5$ sudo ./multiThreaded.out test.txt
abc sed yu good person; what are you doing
Creating Stat Calculation Thread
Creating Stat Printer Thread

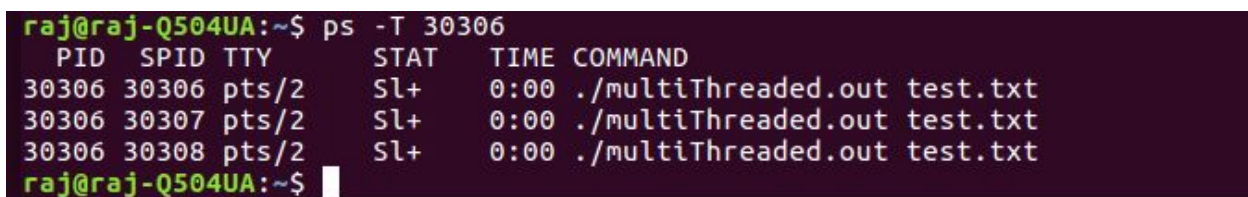
Received signal 10
Start Processing

Success

Received signal 12
Start Printing Statistics
Printing Statistics
Words:9
Lines:1
Characters:43
raj@raj-Q504UA:~/ECEN_5013_AES/HW3/Que5$
```

Here in Thread 1, I am getting the inputs from the user until ENter is pressed. To count characters I have counted spaces as well and the word count is done as usual.

The following screenshot confirms that I have created 3 threads to achieve the functionality,



```
raj@raj-Q504UA:~$ ps -T 30306
  PID  SPID  TTY      STAT  TIME  COMMAND
  30306  30306  pts/2    Sl+    0:00  ./multiThreaded.out test.txt
  30306  30307  pts/2    Sl+    0:00  ./multiThreaded.out test.txt
  30306  30308  pts/2    Sl+    0:00  ./multiThreaded.out test.txt
raj@raj-Q504UA:~$
```