Conceptual Questions

1. Explain pthread_create, pthread_join and pthread_exit.

pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)

This function is used to create a new thread within a process. Upon successful creation, the ID of the thread is stored in the location *thread*. When the thread is created, it will execute the *start_routine* function with argument *arg*. If the start routine returns, the thread behaves as if *pthread_exit()* was called.

pthread_join(pthread_t thread, void **value_ptr)

This function will suspend execution of the calling thread until the target thread terminates. When the target thread terminates after this function is called, the returning value from the target thread will be made available in *value_ptr*. If a *pthread_join()* returns successfully, it means the target thread has been terminated.

pthread_exit(void *value_ptr)

This function will terminate the calling thread and make the *value_ptr* available to any successful join with the terminating thread.

2. Explain thread memory vs. process memory. Do threads share memory? Can threads access each others memory?

Threads are assigned a stack pointer when they are created so they use a stack data structure for their memory. Processes put their variables on the heap at startup and use stacks for function calls. Both threads and processes can use dynamic memory when using functions like *malloc* and *calloc*. This memory will be stored on the heap. All global variables can be shared by threads. Threads also share a common heap and because of this, allocation and deallocation must be synchronized. However, each thread is assigned it's own call stack which is not normally shared unless one thread contains a pointer to another thread's stack.

3. Difference between multi-threading/processing? Pros and cons of each?

Multithreading

- Execution is split among threads.
- On Linux, *pthreads* prove multithreading support at the user level
- Threads share memory so global variables must be protected through mutual exclusion mechanisms.

- Threads are generally lighter.
- Most OS implement threads and processes differently, however linux uses the same data structures.
- Shared memory is generally faster and more flexible than message passing mechanisms

Multiprocessing

- Execution is split among processes.
- On Linux, we use system calls like *fork()* to create and *exec()* to modify processes.
- Processes normally do not share memory, on linux they can through system calls.
- Data is usually exchanged through message passing mechanisms (e.g. pipes, sockets).
- Multiprocess applications are usually more reliable since one process crashing does not affect other processes.

4. Provide an explanation of mutual exclusion, what is a critical section?

Mutual exclusion or *mutex* is a mechanism that prevents simultaneous access to a shared resource. This concept is used when dealing with *critical sections* in which multiple processes or threads attempt to access a shared resource. The critical section will contain code which sets or changes the value of a shared resource. If 2 threads attempt to change the value of the critical section at the same time, we do not know which thread will reach the critical section first and therefore we cannot predict the outcome of the program. Instead, we give the thread which is currently executing the critical section a mutex which that thread must *lock*. Once that thread is done executing, it *unlocks* the mutex and the next thread in line can lock the mutex and begin it's execution. This ensures that only one thread or process can reach the critical section at time.

```
void *class_total(void *args) {
    pthread_mutex_lock(&mutex);
    int *grade = args;
    total_grade += *grade;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

This example of mutual exclusion only allows a single thread to increment the `total_grade`.

5. Explain the functions used to perform mutual exlcusion with pthreads.

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

This function initializes the the mutex refereces by *mutex* with attributes specified by *attr*. If *attr* is null, the default attributes are used. Upon success, the state of the mutex is unlocked.

int pthread_mutex_destroy(pthread_mutex_t *mutex);

This function destroys the mutex object references by *mutex*. The mutex becomes unitialized and can be reinitialized with pthread_mutex_init.

int pthread_mutex_lock(pthread_mutex_t *mutex);

This function locks the mutex referenced by *mutex*. If the mutex is already locked, the calling thread will block until the mutex becomes available. The function returns with the mtuex object referenced by *mutex* in the locked state with the calling thread as it's owner.

int pthread_mutex_trylock(pthread_mutex_t *mutex);

This function locks the mutex referenced by *mutex*. If the mutex is already locked by any thread, including the current one, the call will immediately return.

int pthread_mutex_unlock(pthread_mutex_t *mutex);

This function will release the mutex object referenced by *mutex*. The manner in which the mutex is released is dependent upon the mutex's type attribute. If there are other threads blocked on the referenced mutex, the scheduling policy is used to determine which thread aquired the mutex next.

Application Questions

All code and output can be found here.

Link to clone repository: git@github.com:AshwinK97/Operating-Systems.git