REAL-TIME TASKS MODELS IN LINUX

The assignment involves developing a program using POSIX threads to implement periodic and a periodic tasks which can be expressed in real-time systems as endless loops with time and event based triggers. The task body is defined in the BNF form and the input to the program is a specification of a task set which is assumed to be free of errors. The program reads in a task set specification and creates a thread for each periodic and aperiodic task given in the input file. The scheduling events of the tasks are verified using “trace-cmd” command by collecting “sched_switch” events from the Linux internal tracer ftrace. The traced records are then viewed using kernelshark.

The input file is read line by line. The first line of the input files decides the number of tasks and the total execution time of the program. The main thread reads the next line and parses the line to extract the following information from the input:

1. Periodic, period, priority, task body
2. Aperiodic, event, priority, task body

The task body is then parsed and saved in a linked list consisting of type (L-lock, U-unlock, and I-iteration) and value indicating the iterations. All these details are saved in a ‘struct param’ array instance with the index representing the task. Once the input is parsed the main thread initialises all the semaphores and mutexes used in the program.

```
int sem_init(sem_t *, int, unsigned int);
int pthread_mutexattr_init(pthread_mutexattr_t *);
int pthread_mutexattr_destroy(pthread_mutexattr_t *);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *, int ); // protocol: PTHREAD_PRIO_INHERIT
int pthread_mutex_init(pthread_mutex_t *, const pthread_mutexattr_t *);
```

Next it creates a thread for each task using the below API’s:

```
int pthread_attr_init(pthread_attr_t *); // Initialises the thread attribute object
Int pthread_attr_setschedpolicy(pthread_attr_t *, int); // Set to FIFO.
int pthread_create(pthread_t *, const pthread_attr_t *, void *(*) (void *), void *);
```

The thread start routine is a standard function which registers a thread exit handler and calls the specific task function. The threads are initialized and wait for activation (wait on semaphore). All the threads are activated by the main thread but start the execution at the same time because of the barrier (pthread_barrier_t). The next activation time for the period task is calculated and saved in the ‘struct param’ instance for that thread. The aperiodic task waits for the event to occur (semaphore wait).

The main thread creates another thread that listens to key-press events and sends a signal to the destination thread when the event occurs. This thread opens the device node of the keyboard and gets blocked on the read system call. When the key-press event occurs it then signals the destination threads by doing a semaphore post.

```
int open(const char *, int);
size_t read(int, void *, size_t);
```

The main thread then activates the all the task threads that were waiting on a semaphore and sleeps till the total execution time is completed. The periodic thread gets its work details from the linked list and performs its work.

The below API are used to lock and unlock mutex in the task body.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

If the periodic task finishes the work within its period then the threads sleeps till the current period ends before moving on to next iteration.

```
int clock_nanosleep(clockid_t, int, const struct timespec *, struct timespec *);
```

The aperiodic task waits for the event to occur until it receives key-press event. Once it receives the event it executes its task and again waits for the event to occur (wait on a semaphore).

The main thread when wakes up after the sleep till the total execution time is complete and sends a signal to the all the threads to exit by sending a signal. The critical section inside the task body is protected by disabling the cancel state before and enabling it after the critical section. Below are the API's,

```
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
```

Once the cancel signal is received by the threads the exit handler is called where all the resources such the linked list nodes and the mutexes allocated are released. The API used for registering the exit handler are below:

```
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

The main thread then waits for waits for all the threads to join and then finally the program terminates.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

trace-cmd COMMAND [OPTIONS]

The trace-cmd command interacts with the ftrace tracer (Linux kernel internal tracer). It interfaces with the ftrace specific files found in the debugfs file system under the tracing directory. A COMMAND must be specified to tell trace-cmd what to do.
For eg:
record - record a live trace and write a trace.dat file to the local disk.

Trace-cmd provides a tiny GUI call kernelshark which can be used to analyse the trace files.