RTOS project

Xenomai tasks

Name, index no., department	Marcin Kochalski 275514 WEFIM
Report submission date	22.05.2025

Full repo can be found at https://github.com/MarcinKochalski/RtosXeno

Task 0 - Use the snippet of code, fill the gaps and compile it.

```
/* Create task
 * Arguments: &task,
 *            name,
 *            stack size (0=default),
 *            priority,
 *            mode (FPU, start suspended, ...)
 */
rt_task_create(&hello_task, str, 0, 50, 0);

/*  Start task
 * Arguments: &task,
 *            task function,
 *            function argument
 */
rt_task_start(&hello_task, &helloWorld, 0);
```

For creation and startt of the task, we use syntax shown above, with arguments as commented.

For make, I've used the following makefile:

```
XENO_CONFIG := xeno-config

CFLAGS := $(shell $(XENO_CONFIG) --posix --alchemy --cflags)
LDFLAGS := $(shell $(XENO_CONFIG) --posix --alchemy --ldflags)

CC := gcc
EXECUTABLE := task3

all: $(EXECUTABLE)

%: %.c
	$(CC) -o $@ $< $(CFLAGS) $(LDFLAGS)

clean:
	rm -f $(EXECUTABLE)
```

The output is following:

```
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ make
gcc     -o task0 task0.c -I/usr/include/xenomai/cobalt -I/usr/include/xenomai -D_
GNU_SOURCE -D_REENTRANT -fasynchronous-unwind-tables -D__COBALT__ -D__COBALT_WRA
P__ -I/usr/include/xenomai/alchemy -Wl,--no-as-needed -Wl,@/usr/lib/cobalt.wrapp
ers -Wl,@/usr/lib/modechk.wrappers -lalchemy -lcopperplate /usr/lib/xenomai/boot
strap.o -Wl,--wrap=main -Wl,--dynamic-list=/usr/lib/dynlist.ld -L/usr/lib -lcoba
lt -lmodechk -lpthread -lrt
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task0
start task
Hello World!
Task name : hello
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$
```

Task1. Prepare simple Xenomai application using the periodic interrupt based on system timer.

```
rt_task_create(&periodic_task, "periodic", 0, 60, 0);
rt_task_start(&periodic_task, &task_function, 0);
pause();
```

We create and start the periodic task with the params as given.

```
void task_function(void *arg)
{
  rt_task_set_periodic(NULL, TM_NOW, period_ns);
  while (1) {
    printf( format: "Hello World!\n");
    rt_task_wait_period(NULL);
  }
  return;
}
```

As argument, we can choose the wait period defined earlier, corresponding to one second (value in ns)

```
#define period_ns 1000000000
```

The output is Hello World being printed every second.

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Task2. Prepare simple application for showing thread concurrency problem and propose solution for that problem

```
rt_task_create(&t1, "task1", 0, 1, 0);
rt_task_create(&t2, "task2", 0, 1, 0);
```

If we create the task and do nothing about race condition, the second task will start once the first one is finished, basing on the system scheduler, but we hope to achieve both tasks running at the same time with our own implementation of a scheduler. To achieve that, we create mutexes as semaphores:

```
static RT_SEM mutex1;
static RT_SEM mutex2;
```

Then, we implement them in the functions:

```
void taskOne(void *arg)
{
    int i;
    for (i=0; i < ITER; i++) {
    /*int rt_sem_p(RT_SEM *sem, RTIME timeout)*/
    rt_sem_p(&mutex1,TM_INFINITE);
        printf( format: "I am taskOne and global = %d.................\n", ++global);
    rt_sem_v(&mutex2);
    }
}

void taskTwo(void *arg)
{
    int i;
    for (i=0; i < ITER; i++) {
    /*int rt_sem_p(RT_SEM *sem, RTIME timeout)*/
    rt_sem_p(&mutex2,TM_INFINITE);
        printf( format: "I am taskTwo and global = %d----------------\n", --global);
    rt_sem_v(&mutex1);
    }
}
```

Without use of semaphores, the program counts to the ITER, then runs from 10 to zero. We need to obviously first initialize the semaphores with correct setup, so that the taskOne runs first and sets the semaphore up for the second function, but only for one iteration, and after that they keep waiting for one iteration after each other:

```
rt_sem_create(&mutex1, "mutex1", 1, S_FIFO);
rt_sem_create(&mutex2, "mutex2", 0, S_FIFO);
rt_task_create(&t1, "task1", 0, 1, 0);
rt_task_create(&t2, "task2", 0, 1, 0);
```

The output is – as expected from the task description – as follows:

```
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task2
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
I am taskOne and global = 1................
I am taskTwo and global = 0----------------
```

Task3. Use a Xenoimai's Message queue services for estimation of π value base on Monte Carlo Method (described in previous task list).

As in the previous list, we use the MonteCarlo function.

```
float MonteCarlo(int points) {
  int k = 0;
  float x, y;
  for (int j = 1; j <= points; j++) {
    x = (float)rand() / RAND_MAX;
    y = (float)rand() / RAND_MAX;
    if (x*x + y*y <= 1) {
      k++;
    }
  }

  return 4.0 * k / (float)points;
}
```

But this time, instead of forking, we create queue:

```
for (int i = 0; i < threads; ++i) {
  char task_name[32];
  sprintf(task_name, format: "%s-%d", QUEUE_NAME, i);
  rt_queue_create(&mq[i], task_name, sizeof(float), MAX_MSG_COUNT, Q_FIFO);
  rt_task_create(&pi_task[i], task_name, 0, 50, 0);
  rt_task_start(&pi_task[i], &task, (void *)(uintptr_t)i);
}
```

Task function (which is as follows) calculates the pi value and stores it inside the queue.

```
rt_queue_bind(&q, task_name, TM_INFINITE);

pi = MonteCarlo(pointsPerTask);

rt_queue_write(&q, &pi, sizeof(float), Q_NORMAL);
rt_queue_unbind(&q);
```

The queue is read afterwards in a FIFO manner. The sum is calculated by iterating.

```
for (int i = 0; i < threads; ++i) {
  float pi;
  rt_queue_read(&mq[i], &pi, sizeof(float), TM_INFINITE);
  sum += pi;
  rt_queue_delete(&mq[i]);
  rt_task_delete(&pi_task[i]);
}
```

And then, the sum is used to calculate the pi (which is final pi, not to be confused with the pi from MonteCarlo function)

```
double pi = sum / threads;
printf( format: "Pi: %.5f\n", pi);
```

The result is as follows:

```
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task3
Pi: 3.14800
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task3
Pi: 3.14720
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task3
Pi: 3.14360
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task3
Pi: 3.14520
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task3
Pi: 3.14720
xeno@xeno-VirtualBox:~/Desktop/Xeno/Task$ ./task3
Pi: 3.14720
```