# Stack-based Resource Access Protocol

Osman, Hamid
Li, Guanpeng
Ton, Hason
Wong, Cary

November 1, 2012

# 1 Current Scheduler

Changes in task 3 extended the scheduler to EDF scheduling with SRP. EDF scheduling provides a basis on which we will build a dynamic task server.

# 2 Modified Scheduler

Aperiodic Task: A task that does not run on a fix period

Task Server: A mechanism to schedule aperiodic tasks

Constant Bandwidth Server: A dynamic task server that supports soft-real time scheduling

Required outcome from this task: extend the scheduler to support CBS Break deadline ties using priority

Based on the requirements, we need to differentiate between a periodic task and aperiodic task. The type of task is specified by the user at task creation. It can be set to

1. BASIC_TASK
2. CBS_TASK

The definitions will help the schedule make appropriate decisions regarding deadline violations.

```
int main ()
{
    nrk_setup_ports();
    nrk_setup_uart(UART_BAUDRATE_115K2);

    nrk_init();

    nrk_time_set(0,0);

    nrk_time_get(seed);
    srand(seed->nano_secs);

    //Initialize tasks
    //Higher value higher priority
    INITIALIZE_TASK(1, BASIC_TASK);
    INITIALIZE_TASK(2, CBS_TASK);
    INITIALIZE_TASK(3, BASIC_TASK);

    nrk_start();

    return 0;
}
```

Figure 1: Task type differentiation

There are a few properties a constant bandwidth server exhibits:

All tasks referred here are assumed to be aperiodic.

1. An aperiodic task can run within a CBS; the *budget* $Q_s$ of the server decreases as the task executes
2. If $Q_s$ decreases to zero, and the task has not completed, the server is extended to continue servicing the task: $Q_s$ resets to its original value, and the deadline is increased $d_s$ by the server's period $P_s$.
3. The server is idle if there are no waiting tasks
4. Once a task requests service when the server is idle, the server recalculates its deadline, setting $d_s = \mathrm{t} + P_s$

As part of this task, we modified some of the code from *nrk_scheduler.c* to support CBS (Figure 2). The following code snippet handles property 2 of the CBS, described above. A check is performed to determine whether a task that has run out of execution time is a CBS task, and extends the timers to support the task to be run. If the task is not a task to be handled by the CBS, the scheduler will throw a *NRK_RESERVE_ERROR* exception, indicating the task has missed its deadline.

```
    // Update cpu used value for ended task
    // If the task has used its reserve, suspend task
    // Don't disable IdleTask which is 0
    // Don't decrease cpu_remaining if reserve is 0 and hence disabled
    if(nrk_cur_task_TCB->cpu_reserve!=0 && nrk_cur_task_TCB->task_ID!=NRK_IDLE_TASK_ID && nrk_cur_task_TCB->task_state!=FINISHED )
    {
        if(nrk_cur_task_TCB->cpu_remaining<_nrk_prev_timer_val)
            {
            // It's an error for BASIC_TASK but for aperiodic CBS task it is possible to be the case.
            // We need make sure the CBC will not be forced to set its cpu_remaining to 0 if it hasnt finished its executions. Also we register to error only when it is BASIC_TASK
#ifdef NRK_STATS_TRACKER
            _nrk_stats_add_violation(nrk_cur_task_TCB->task_ID);
#endif
            if(nrk_cur_task_TCB->task_type == CBS_TASK && nrk_cur_task_TCB->task_state != SUSPENDED){
                nrk_cur_task_TCB->next_period = nrk_cur_task_TCB->period;
                nrk_cur_task_TCB->cpu_remaining = nrk_cur_task_TCB->cpu_reserve;
                printf("Replenish CBS of Task %d\n",nrk_cur_task_TCB->task_ID);
            }else{
                nrk_cur_task_TCB->cpu_remaining=0;
                nrk_kernel_error_add(NRK_RESERVE_ERROR,nrk_cur_task_TCB->task_ID);
            }
        }
        else
            nrk_cur_task_TCB->cpu_remaining-=_nrk_prev_timer_val;

        task_ID= nrk_cur_task_TCB->task_ID;
    //                  printf("cpu remaining of %d is %d \n",task_ID,nrk_task_TCB[task_ID].cpu_remaining);

        if (nrk_cur_task_TCB->cpu_remaining ==0 )
        {
            //printf("Task %d cpu remaining = 0\n", task_ID);
            //printf("Task type is %d\n", nrk_cur_task_TCB->task_type);
            // Here we dont need to suspend CBS
            if(nrk_cur_task_TCB->task_type == BASIC_TASK){
#ifdef NRK_STATS_TRACKER
                _nrk_stats_add_violation(nrk_cur_task_TCB->task_ID);
#endif
                nrk_kernel_error_add(NRK_RESERVE_VIOLATED,task_ID);
                nrk_cur_task_TCB->task_state = SUSPENDED;
                nrk_rem_from_readyQ(task_ID);
            }else if(nrk_cur_task_TCB->task_type == CBS_TASK
                     && nrk_cur_task_TCB->task_state != SUSPENDED
                      ){
                // We need replenish the budget for CBS
                printf("Task %d: Replenish CBS <---------- \n", task_ID);
                nrk_cur_task_TCB->next_period = nrk_cur_task_TCB->period;
                nrk_cur_task_TCB->cpu_remaining = nrk_cur_task_TCB->cpu_reserve;
                //printf("Next period is from %d\n", nrk_cur_task_TCB->next_period);
            }

        }
    }
```

Figure 2: Handler for missed deadlines

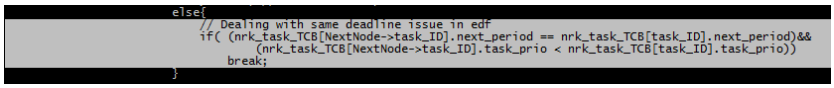
In order to simulate an aperiodic task, we generate different behaviour if the task is to be serviced by the CBS. We set up random functions to simulate an aperiodic tasks by implementing randomness in its execution time and period (Figure 3) (*CBS_TASK*). If a task to be run is not serviced by the CBS, the behaviour of the task remains periodic, and the execution time is also known (*BASIC_TASK*).

A small check statement was added to *nrk_add_to_readyQ* in order to deal with tasks that have the same deadline shown below.

```
/*
   To stay inline (no pun intended) with the nano-rk direction of using guards for
   optimization and potential performance, we use macros rather than methods.
   We could have created a method, but the call may be expensive and impede the performance of the kernel.
 */
#define TASK(n, taskPeriod, taskExecution)                                  \
    void task_##n##_func()                                                  \
{                                                                           \
    int task_ID =  nrk_cur_task_TCB->task_ID;                               \
    int k=0;                                                                \
    while (1)                                                               \
    {                                                                       \
        printf("\n------\nStart running task..%d\n", n);            \
        long int duty = rand()*4;                                           \
        if(duty<0) duty = 0-duty;                                           \
        if(n==2){                                                           \
            printf("Duty amount is %d\n",duty);                             \
            for(int i=0;i<=duty;i++){                          \
                if(i%2000==0){                                 \
                    printf("CBS budget is %d\n", nrk_task_TCB[task_ID].cpu_remaining);\
                    for(int j=0;j<9999;j++){k++;}                           \
                    printf("Busying with task 2(CBS) %d out of %d\n",i,duty);\
                }                                                           \
            }                                                  \
        }else{for(int i=0;i<10;i++){k++;}printf("Some busy tasks\n");}  \
        printf("Finishing the task %d\n------\n\n",n);          \
        nrk_wait_until_next_period();                          \
    }                                                  \
}                                                                   \
NRK_STK stack_##n[NRK_APP_STACKSIZE];                                       \
nrk_task_type task_##n;                                                     \
uint32_t task_##n##_period = taskPeriod;                                    \
uint32_t task_##n##_execution = taskExecution;                              \

/*"Instantiate" the task*/
#define INITIALIZE_TASK(n,task_type)                                        \
task_##n.FirstActivation = TRUE;                                            \
task_##n.prio = n;                                                          \
task_##n.Type = task_type;                                                 \
task_##n.SchType = PREEMPTIVE;                                             \
task_##n.period.secs = task_##n##_period;                                   \
task_##n.period.nano_secs = 0;                                              \
task_##n.cpu_reserve.secs = task_##n##_execution;                         \
task_##n.cpu_reserve.nano_secs = 0;                                         \
task_##n.offset.secs = 0;                                                   \
task_##n.offset.nano_secs = 0;                                              \
nrk_task_set_entry_function(&task_##n, task_##n##_func);            \
nrk_task_set_stk(&task_##n, stack_##n, NRK_APP_STACKSIZE);                  \
nrk_activate_task(&task_##n)
```

Figure 3: Aperiodic task simulation



```
                    else{
                        // Dealing with same deadline issue in edf
                        if( (nrk_task_TCB[NextNode->task_ID].next_period == nrk_task_TCB[task_ID].next_period)&&
                                (nrk_task_TCB[NextNode->task_ID].task_prio < nrk_task_TCB[task_ID].task_prio))
                            break;
                    }
```

Figure 4: Evaluating tasks with same deadlines by priority

# 3 Testing

The main function as part of main.c, is responsible for setting up and initialization of the system. It also holds the task set required to verify the accuracy of the scheduler. Note that the task set now has an additional parameter that determines the type of task to be run.

Table 1: Task Set for CBS

| Task | $e_i$ | $P_i$ | Type |
|---|---|---|---|
| 1 | 3 | 6 | Basic |
| 2 | 1 (Budget) | 5 | CBS |
| 3 | 1 | 7 | Basic |

# 4 Expected Results

Since the task server runs as a normal period task, and the scheduler uses EDF to prioritize the tasks, we have already proven the accuracy of this algorithm in task 2 and task 3 of this assignment. We expect the scheduler to demonstrate two scenarios and maintain a EDF priority.

The aperiodic task has a:

1. Small execution time - the task meets its deadline, and no replenishment and deadline extension is needed and
2. Large execution time - the task does not meet its deadline, and a replenishment and deadline extension is needed

# 5 Results

The screen caption below shows the behaviour of two running aperiodic task, one with the intermediate steps of extending the deadline and budget, and one without. The changes to the code tracks the budget of a CBS and shows replenishment as the task misses its deadline. We can also see that the budget is decreasing at a steady rate until it reaches 0, at which the CBS resets the budget to its original value, and continues to handle the task.

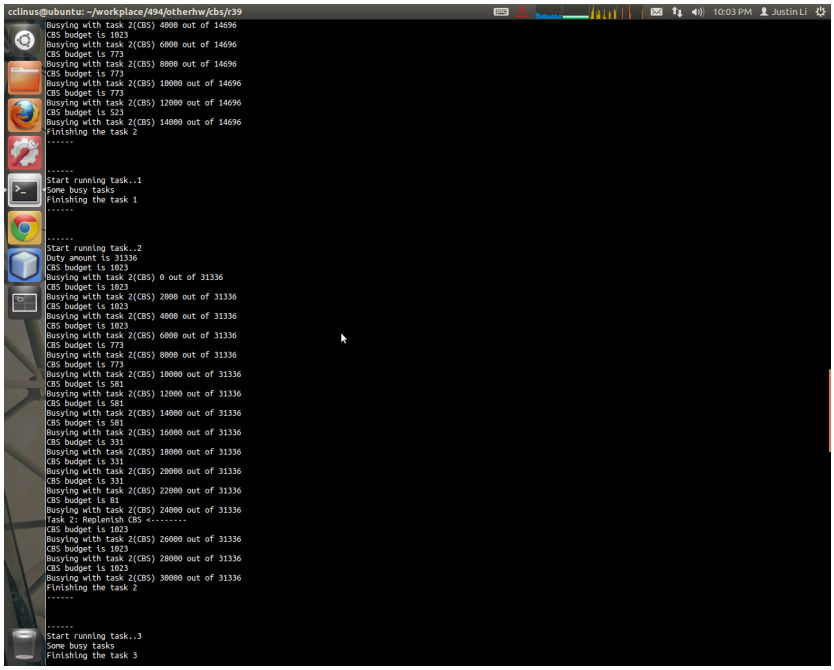



Figure 5: EDP with SRP screencap