CASH Approach for Capacity Sharing

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

November 5, 2012

1 Current Scheduler

Changes in task 4 extended the scheduler to EDF scheduling with CBS. CBS provides a basis on which we will add an overrun control.

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

Overrun: The amount of time a given task has exceeded its deadline

Capacity Sharing for Overrun Control: A policy that allows multiple task servers, in particular CBS, to share available, but unused budget

Required outcome from this task: extend the scheduler to support CASH

More on CASH: Whenever an aperiodic task runs on the CBS, there is a possibility that the task finishes before the budget is used up. When dealing with multiple CBS, we can implement a queue that keeps track of available, but unused budget. This allows task servers to consume unused resources first, increasing efficiency in resource consumption, and lowering overall execution times for aperiodic tasks.

Based on the requirements, we need to add a reclaiming mechanism to support CASH. We need to first implement a queue to store unused resources, and we need to store the budget and deadline of these resources. Instead of a queue, we have extended the TCB to support place holders for the budget and deadline (Figure 1).

In *nrk_scheduler.c*, we implement 3 mechanisms to adjust and update unused budget to cover 3 different scenarios. See figure 2, 3, and 4 for code.

1. We need to check and update the validity of unused budget. First, we determine whether the deadline of CBS has passed. If the deadline has passed, we can remove the resource, since we can no longer use them. This is done by setting *cash* and *cash_period*in the TCB to 0. Secondly, we also update the time values, specifically *cash_period* to reflect the new relative deadline each time the scheduler is called. We check to prevent the available budget from being larger than the relative deadline to avoid using non-existing budget.

```
/*
 ***********************************************************
 *                    TASK CONTROL BLOCK                   *
 ***********************************************************
 */

typedef struct os_tcb {
        NRK_STK         *OSTaskStkPtr;          /* Pointer to current top of stack */
        NRK_STK         *OSTCBStkBottom;        /* Pointer to bottom of stack    */


        bool      elevated_prio_flag;
        bool      suspend_flag;
        bool      nw_flag;                  // allows user to wake up on event or nw;
        uint8_t   event_suspend;            // event 0 = no event ; 1-255 event type;
        int8_t    task_ID;                  // For quick reference later, -1 means not active
        uint8_t   task_state;               // Task status
        uint8_t   task_prio;                // Task priority (0 == highest, 63 == lowest)
        uint8_t   task_prio_ceil;           // Task priority (0 == highest, 63 == lowest)
        uint8_t   errno;                    // 0 no error 1-255 error code
        uint32_t  registered_signal_mask;  // List of events that are registered
        uint32_t  active_signal_mask;      // List of events currently waiting on

        // Inside TCB, all timer values stored in tick multiples to save memory
        uint16_t  next_wakeup;
        uint16_t  next_period;
        uint16_t  cpu_remaining;
        uint16_t  period;
        uint16_t  cpu_reserve;
        uint16_t  num_periods;

        // Task type goes here
        uint8_t task_type;
        //bool FinishTask;

        #ifdef CBS_CASH
        // extend task for CBS CASH
        uint16_t cash;
        uint16_t cash_period;
        #endif

} NRK_TCB;
```

Figure 1: Updated TCB

2. We need to start keeping track of unused budget. Whenever an CBS task finishes running, we would like store the remaining budget of a the *CBS_TASK* as *cash* and update *cash_period* in the TCB to store the **relative** deadline of the resource.

3. We need to check for unused budget to consume whenever a new CBS task is run. This can be determined by evaluating the *cash* and *cash_period* of all TCBs. An *cash* value larger than 0 indicate unused budget is available to be reclaim (Figure 5)). If unused resources are available, then the task can consume the resource held by the CBS, and adjust the budget to reflect the usage by decrementing the budget. As per the algorithm of CASH, we choose the CBS with the earliest deadline. We also handle cases when a single task requires more execution time than a single CBS remaining budget can provide; the algorithm allows a single running task to use up unused budgets from multiple CBS. In the case that there are no unused resources available, the task will follow the default protocol.

```
#ifdef CBS_CASH

    // 1:Add Cash
    if(nrk_cur_task_TCB->task_state == SUSPENDED && nrk_cur_task_TCB->task_type == CBS_TASK){
        nrk_task_TCB[nrk_cur_task_TCB->task_ID].cash = nrk_cur_task_TCB->cpu_remaining;
        nrk_task_TCB[nrk_cur_task_TCB->task_ID].cash_period = nrk_cur_task_TCB->next_period - _nrk_prev_timer_val;
        printf("Add cash to task%d to cash %d, period is %d <===\n", nrk_cur_task_TCB->task_ID, nrk_task_TCB[nrk_cur_task_TCB->task_ID].cash, nrk_cur_task_TCB->next_period);
    }
```

Figure 2: Code snippet: Adds remaining budget to CASH

```
// 1:Add Cash
if(nrk_cur_task_TCB->task_state == SUSPENDED && nrk_cur_task_TCB->task_type == CBS_TASK){
    nrk_task_TCB[nrk_cur_task_TCB->task_ID].cash = nrk_cur_task_TCB->cpu_remaining;
    nrk_task_TCB[nrk_cur_task_TCB->task_ID].cash_period = nrk_cur_task_TCB->next_period - _nrk_prev_timer_val;
    printf("Add cash to task%d to cash %d, period is %d <===\n", nrk_cur_task_TCB->task_ID, nrk_task_TCB[nrk_cur_task_TCB->task_ID].cash, nrk_cur_task_TCB->next_period);
}
// 2:Maintein CashQ
// CASH book keeping - update cash given prev timer val
// Loop through all nrk tasks - Delete CBS tasks that passed deadline or reduce cash_period.
int i = 0;
for(i = 0; i < NRK_MAX_TASKS; i++){
    if(nrk_task_TCB[i].task_type != IDLE_TASK) {
        // update all cash period
        if (nrk_task_TCB[i].task_type == CBS_TASK){
            if(nrk_task_TCB[i].cash_period < _nrk_prev_timer_val && nrk_task_TCB[i].cash != 0){
                // delete cash where deadline pass
                nrk_task_TCB[i].cash = 0;
                nrk_task_TCB[i].cash_period = 0;
                printf("CASH task%d passes deadline <====\n", nrk_cur_task_TCB->task_ID);
            }
            else{
                // if cash_period is greater than prev_timer_val, reduce cash_period
                nrk_task_TCB[i].cash_period -= _nrk_prev_timer_val;

                if(nrk_task_TCB[i].cash > nrk_task_TCB[i].cash_period){
                      nrk_task_TCB[i].cash = nrk_task_TCB[i].cash_period;
                }
            }
        }
    }
}
// 3:Use Cash
if(nrk_cur_task_TCB->cpu_remaining == nrk_cur_task_TCB->cpu_reserve
        && nrk_cur_task_TCB->task_type == CBS_TASK
        ){
```

Figure 3: Code snippet: Updates and maintains CASH and relative deadline values

```
    // 3:Use Cash
    if(nrk_cur_task_TCB->cpu_remaining == nrk_cur_task_TCB->cpu_reserve
            && nrk_cur_task_TCB->task_type == CBS_TASK
            ){
        int min_id = getMinRelativeDeadlineTaskWithCacheRemainingId();
        if(min_id != -1 && nrk_cur_task_TCB->task_ID != min_id){
            if (nrk_task_TCB[min_id].cash > _nrk_prev_timer_val) {
                nrk_cur_task_TCB->cpu_remaining+=_nrk_prev_timer_val;
                nrk_task_TCB[min_id].cash-=_nrk_prev_timer_val;
            } else {

                int i;
                int cache_sum = 0;
                int residual_cache_time = 0;
                for(i = 0; i< NRK_MAX_TASKS; i++){
                        min_id = getMinRelativeDeadlineTaskWithCacheRemainingId();
                        if (cache_sum >= _nrk_prev_timer_val) {
                                break;
                        }
                        residual_cache_time = (_nrk_prev_timer_val - cache_sum);
                        if (nrk_task_TCB[min_id].cash > residual_cache_time) {
                                nrk_cur_task_TCB->cpu_remaining+=residual_cache_time;
                                nrk_task_TCB[min_id].cash-=residual_cache_time;
                                break;
                        }
                        nrk_cur_task_TCB->cpu_remaining+=nrk_cur_task_TCB[min_id].cash;
                        cache_sum+=nrk_cur_task_TCB[min_id].cash;
                        nrk_task_TCB[min_id].cash = 0;
                        nrk_task_TCB[min_id].cash_period = 0;

                }
            }
            printf("min_id%d' cash remaining is %d <$$$$$$$$$$$\n", min_id, nrk_task_TCB[min_id].cash);
            //printf("Then the task%d's cpu_remaining becomes %d <$$$$$$$$$\n", nrk_cur)
        }
    }
#endif
    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 )
    {
```

Figure 4: Code snippet: Tasks use CASH first

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.

Table 1: Task Set for CBS

Task	e_i	P_i	Type
1	3 (Budget)	8	CBS
2	1	5	Basic
3	2 (Budget)	9	CBS

We chose the execution time of aperiodic tasks to be around 1.3 seconds in order to simulate task completion prior to budget depletion shown in figure 6. This task will help us demonstrate the functionality, explained in the next section.

```
    // 3:Use Cash
    if(nrk_cur_task_TCB->cpu_remaining == nrk_cur_task_TCB->cpu_reserve
            && nrk_cur_task_TCB->task_type == CBS_TASK
            ){
        int min_id = getMinRelativeDeadlineTaskWithCacheRemainingId();
        if(min_id != -1 && nrk_cur_task_TCB->task_ID != min_id){
            if (nrk_task_TCB[min_id].cash > _nrk_prev_timer_val) {
                nrk_cur_task_TCB->cpu_remaining+=_nrk_prev_timer_val;
                nrk_task_TCB[min_id].cash-=_nrk_prev_timer_val;
            } else {

                int i;
                int cache_sum = 0;
                int residual_cache_time = 0;
                for(i = 0; i< NRK_MAX_TASKS; i++){
                        min_id = getMinRelativeDeadlineTaskWithCacheRemainingId();
                        if (cache_sum >= _nrk_prev_timer_val) {
                                break;
                        }
                        residual_cache_time = (_nrk_prev_timer_val - cache_sum);
                        if (nrk_task_TCB[min_id].cash > residual_cache_time) {
                                nrk_cur_task_TCB->cpu_remaining+=residual_cache_time;
                                nrk_task_TCB[min_id].cash-=residual_cache_time;
                                break;
                        }
                        nrk_cur_task_TCB->cpu_remaining+=nrk_cur_task_TCB[min_id].cash;
                        cache_sum+=nrk_cur_task_TCB[min_id].cash;
                        nrk_task_TCB[min_id].cash = 0;
                        nrk_task_TCB[min_id].cash_period = 0;

                }
            }
            printf("min_id%d' cash remaining is %d <$$$$$$$$$$$\n", min_id, nrk_task_TCB[min_id].cash);
            //printf("Then the task%d's cpu_remaining becomes %d <$$$$$$$$$\n", nrk_cur)
        }
    }
#endif
    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 )
    {
```

Figure 5: Code snippet: Helper function to get the *task_ID* of the smallest CASH remaining

```
#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 = 30000;                                 \
        if(duty<0) duty = 0-duty;                                \
        if(n==1 || n==3){                                                \
            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<30000;j++){k++;for(int j=0;j<30000;j++){}}            \
                    printf("Busying with task (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;                   \
```

Figure 6: Code snippet: CBS task simulation

4 Expected Results

The expected behaviour of the scheduler for a new CBS task is to check and use any remaining budget, before using CPU reserves. The output of the simulation should reflect the checking and usage, and decreasing the CASH budget accordingly. We should also expect to see unused budget get added to the CASH budget when a CBS task completes.

5 Results

The screen captions below captures two scenarios. In figure 7, we can observe a running CBS task and its completion. Upon completion, we can see that it takes the remaining budget of 2071, and updates the CASH to 2071, as well as setting the relative deadline of the reserve. In figure 8, we can observe a

CBS task using CASH reserves. The budget is originally at a value of 2047. The unused budget is 4142, and is added to the current reserve, resulting in a new budget of 5939. These results successfully shows us the increment and decrement of the CASH budget.

Note: Further results are recorded in testResult.txt, submitted as part of this assignment.

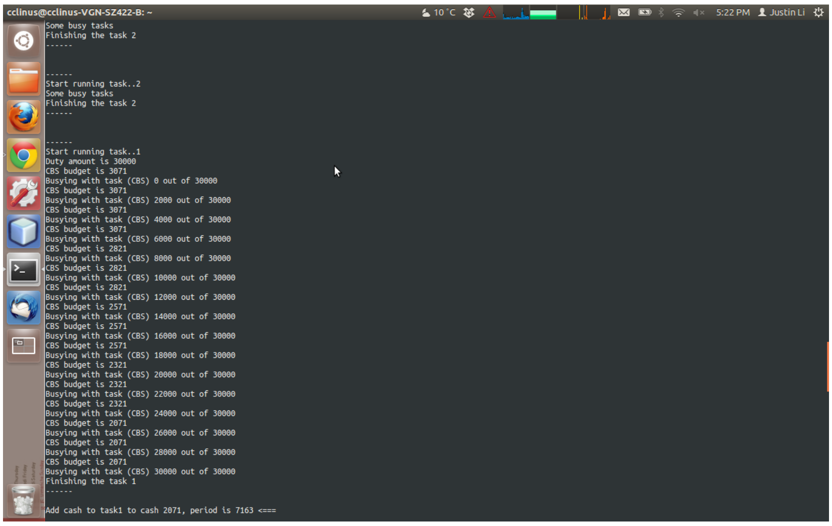

Figure 7: Handling remaining budget

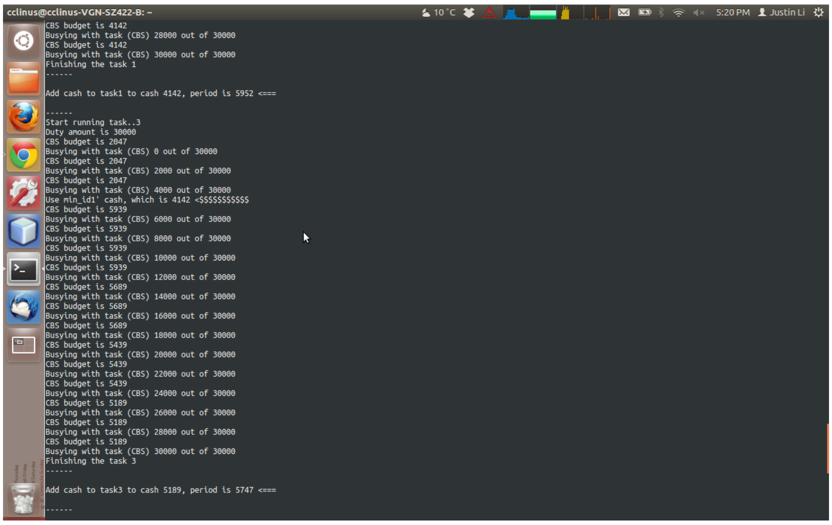

Figure 8: Using CASH reserves