Log In

Logged in as: vasan

Log out

Last-Visited Pages

1. LITMUS-rt SLOTSHIFTING ARCHITECTURE

Quick Edit a Wiki Page

Create/Edit

Menu

Home

Categories

MyTiki

Wiki

Image Galleries

File Galleries

Admin

Since your last visit...

Tue 23 of Feb., 2016

Nothing has changed

LITMUS-rt SLOTSHIFTING ARCHITECTURE

OVERVIEW

- we tried to implement Slot shifting as a platform independent framework,
  - where porting to any RTOS is made possible by simple filling of platform dependent plugin functions.
- The framework design approach is made scalable, i.e. framework is designed with
  - scalable data handling class which can be tuned to either Global/partioned/hybrid selection function.
- Algorithmic part is also made scalable,
  - i.e. the functions in algorithmic part are very much made disassociated enabling the replacing of any core functionality with other.

SLOT SHIFTING ALGORITHM ITSELF

- The Algorithm Works in 2 phases
  - Offline Phase
  - Online Phase
- In the following we define a **function** as follows x ↦ f(x). i.e. defined by a formula or algorithm that tells how to compute the output for a given input.
- Fundamentals of online **Selection Function is EDF**.
- The **Decision Function** is based on uniform time intervals called **SLOTS**.

- **OFFLINE PHASE**
  - Basic schedulable test is done on periodics .
  - A layer of certainty is applied on periodics called Interval notated as I defined with following terms.
    - End(I) : deadline(Ji). i ∈ all jobs with same deadline
    - Est(I) : min (est(Ji)). i ∈ all jobs with same deadline
    - Start(I): max(est(Ii), end(Ii-1))
    - Length(I): | Ii | ➔ End(Ii) – Start(Ii).
    - SC(I) : | Ii | - ∑WCET(Ji) + min(SC(Ii+1),0) , Spare Capacity of the interval I
  - So Basic record of interval looks like:

SNIPPET 1: Basic Overview of interval

```
struct interval {
   int id; /// unique ID for the interval
   int start; /// start of interval
   int end; ///end of interval
   int sc; /// spare capacity of interval
   list *jobs; /// list of jobs that belong to this interval
   list *nxt; /// reference to next interval / NULL
};

PS: This is a basic structure independent of platform, real implementation is much more elaborate.
```

SNIPPET2: **OFFLINE_PHASE:** Overview of Function Create interval

```
function CreateInterval()
input  T: T is schedulable periodic task-set
output t,I: t is list of jobs sorted based on dl.
         I is list of intervals

   st := sort jobs of task-set based on deadline.
   t := st  ///duplicate list of st on t
   while(st)  //CREATE INTERVAL

      I.nxt = create_new_interval(0,0);
      I.end = get_first_job_dl(st);

      while(I.end == get_first_job_dl(st))
            I.jobs.nxt =   Get_rmve_1st_job(st);

      est_i = min(I.jobs.est);
      I.start = max( est_i, Ii-1.end);
      I.sc = |I| - ∑WCET(I.jobs);

    while(Il)  //CREATE SC OF INTERVAL STARTING FROM LAST INTERVAL
       I.sc = I.sc + min(Ii+1 .sc, 0);
       I = I.nxt;

return t,I;
-----------------------------------------------------------------
function create_new_interval()
input s, e: s is the start of interval
            e is the end of interval
output i: i is the created new interval

    i := allocate memory for new record of type struct interval
    i.start := s;
    i.end := e;
    i.jobs := NULL;
    i.nxt := NULL;
return i;
----------------------------------------------------------------
```

```
function get_first_job_dl()
input j: j is list of jobs
output j0 : j0 is the 1 st job in the list / NULL.
     if(j) return j[0];
     else return NULL;
-----------------------------------------------------------------
function get_rmv_1 st _job()
input j: j is list of jobs
output j0 : j0 is the 1 st job in list , removed from list / NULL

    if(j)
       j0 = get_first_job_dl(j);
       remove_job(j, j 0 ); /// removes job j0 from list j
        return j0 ;
    else
       return NULL;
```

- ONLINE PHASE:
    - Basic selection function is based on EDF on Guaranteed ready list or FCFS on not Guaranteed list.
    - Selection function also manages updating intervals and its spare capacity.
    - It checks Firm aperiodics arrival; if the presence is detected then acceptance test is run and if it can be guaranteed then guarantee algorithm is run.
    - If Soft aperiodics arrives, then simply add the task to not-guaranteed list and schedule them when slot is empty after guaranteed EDF function.

SNIPPET3: function slot_shift_core

```
function slot_shift_core( )
input tprev : tprev is the previous task that was scheduled.
output tnxt  : tnxt is the next task to be scheduled.

    /// 1.INTERVAL UPDATE
    update_sc (tprev); /// update spare capacity of the current interval based on prev scheduled task.
    I = Get_current_interval();
    if(slot_count > I.end) /// check we are on end of interval, if so move to next interval
          move_to_next_interval();
    end if

     /// 2.APERIODIC CHECK
     If(aperiodic_task)  ///check on aperiodics task
         if(FIRM  == aperiodic_task.type)
               acceptance_guarantee_algorithm(aperiodic_task); /// run acceptance and guarantee algorithm.
          else if(SOFT  == aperiodic_task.type)
               add_to_notGuaranteed_list(aperiodic_task);
          end if
     end IF

     ///3.EDF
     tnxt  = get_nxt_guaranteed_ready_task();  /// EDF  on ready list
     if(! tnxt )
         tnxt  = get_nxt_notGuaranteed_ready_task();  ///FCFS on not guaranteed ready list
     end if

return  tnxt;
```

SNIPPET4: Function UPDATE_SC

```
function Update_sc()
input t: t is the job that got scheduled,

    I := get_current_interval();
    while(I.job)
       if(I.job == t)
           return;
       end if
       I.job = I.job.nxt;

    i_tmp = get_task_interval(t);
    if(i_tmp .sc  < 0)
         update_till_positive(i_tmp);
    end if

    negate_sc(I);
---------------------------------------------------------------------------------
function Update_till_positive()
Input I : I is the interval from where sc needs update in backwards.

   i_tmp = I;
   do{
      i_tmp.sc++;
      if(i_tmp == get_current_interval())
          break;
      end if
      i_tmp = get_prev_interval(i_tmp);
   }while(i_tmp.sc < 0);
   i_tmp.sc++;
```

- **FIRM APERIODIC ARRIVAL (JA)**:acceptance_guarantee_algorithm(JA)
    - On Firm aperiodic arrival acceptance test needs to run on task to check whether it can be admitted.
    - To run the acceptance test we traverse through 3 kinds intervals and sum there SC.

- 1 ← SC(IC) : Spare capacity of current interval.
- 2 ← SC(Ii) . C < i <= l. end(Il) <= dl(JA) ∧ end(Il+1) > dl(JA).
  - All intervals that are between current and last interval.
- 3 ← Min{SC(Il+1), dl(JA) – start(Il+1)}
  - Required spare capacity in the last interval within dl(JA).
- **If(1+2 +3 >= WCET(JA)) then JA can be accepted i.e. return true.**

SNIPPET5: Simple version of acceptance_guarantee_algorithm

```
function Acceptance_guarantee_algorithm(JA)
     if(acceptance_test(JA))
           guarantee_algorithm(JA);
     else
           move_to_notGuaranteed_list(JA);
```

- **TRADITIONAL GUARANTEE ALGORITHM:**

SNIPPET6: traditional_Guarantee_Algorithm

```
Function Guarantee_algorithm(JA)
     if(dl(JA)  < end(Il+1))
           split_interval(Il+1 , JA);
     end if
     Ii = Il+1;

     while(Ii != Ic)
         sc(Ii) = | Ii | - ∑WCET(Ji) +  min(SC(Ii+1),0);
         Ii  = get_prev_interval(Ii);
```

- Complexity : O(N.t) where N is the number of intervals and t is the jobs in that intervals.
- Redoing what we did offline and traversing both vertically and horizontally in a list is very cumbersome to implement.
- We tried to use this offline effort to recalculate online change in SC without having vertical traversal. Giving us O(N) guarantee algorithm.

- **O(N) GUARANTEE ALGORITHM**:
  - IDEA: either remove a part of delta or add negative version of it to the consecutive backward intervals, till delta become zero.
  - Delta is the WCET of the task

SNIPPET 7:

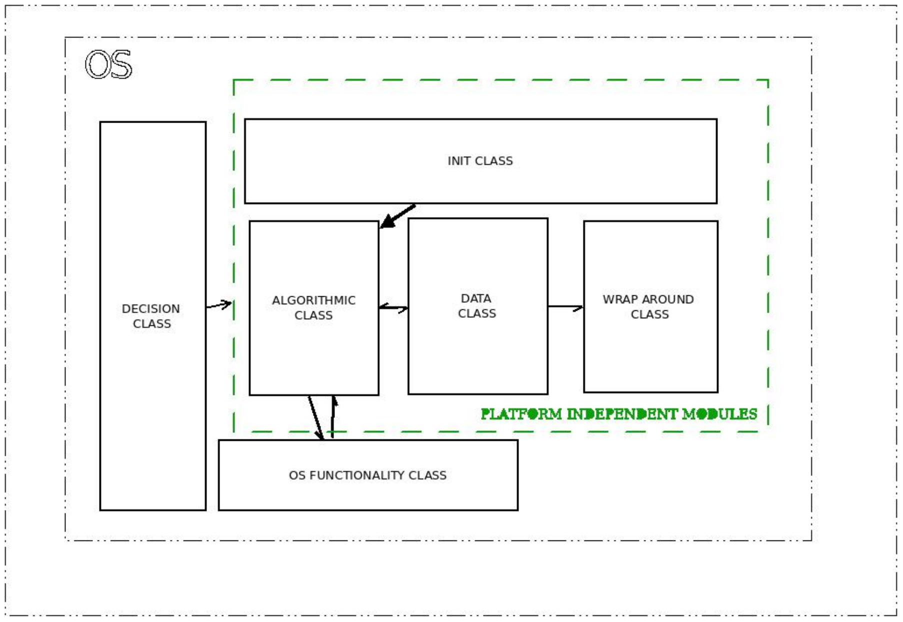
```
Function Guarantee_algorithm( )
Input Idl , JA, , slot_no :  Idl  is the interval in which aperiodic deadline,i.e. Il+1.
                  JA is the aperiodic task that cleared acceptance test.
                  slot_no is the current slot number.
Output :  NONE

      IF(dl(JA,) <  END(Idl,))
          split_point  = DL(JA,)  - START(Idl);
          Ileft  = split_interval (I dl,  split_point , slot_no);
      end IF

      JA.Interval = Ileft? Ileft : Idl;
      delta  := WCET(JA ) ;
      i_temp  :=  JA .Interval;

      while(delta)
        IF(SC(i_temp) > 0)
            IF(SC(i_temp) >= delta)
                SC(i_temp)  = SC(i_temp)  -  delta;
                delta  = 0;
            else
               delta = delta – SC(i_temp);
               SC(i_temp) = -delta;
            end IF
        else
            SC(i_temp) += -delta;
        end IF

        i_temp  = PREV(i_temp);

---------------------------------------------------------------------------------
Function split_interval ( )
Input : Iright , split_point , curr_slot
Output : Ileft

   If(Iright == get_current_interval())
       new_len = (split_point  + start(Iright)) – curr_slot;
   else
       new_len = split_point + start(Iright) – start(Iright);

   SC(Iright) = SC(Iright) – new_len;
   SC(Ileft) = new_len + min(0, SC(Iright));
   END(Ileft) = split_point + start(Iright);
   START(Ileft) =  start(Iright);
   START(Iright) = split_point + start(Iright);

   Add_intr_to_list (Ileft);

return  Ileft
```

IMPLEMENTATION OF SLOT SHIFTING FRAMEWORK

CLASSIFICATION OF SLOT SHIFTING FRAMEWORK

The framework is divided into many classes. This classification enables modular, scalable and seperation of functionality. figure 1 gives the overview of the placement of different clas within the framework.

FIG1: CLASS PLACEMENT OVERVIEW



- platform is divided into vertical and horizontal layers.
- vertical layers are highly dependent on its neighboring verticals.
- horizontals are mostly independent layers, which reacts to above layers.
- arrow ends decides on the communication direction.

Algorithmic class

Contains core algorithm of the slot shifting implementation.
which can be subdivided into 3 major parts.

1. selection functionality.
2. Acceptance and Guarantee functionality.
3. interval functionality.

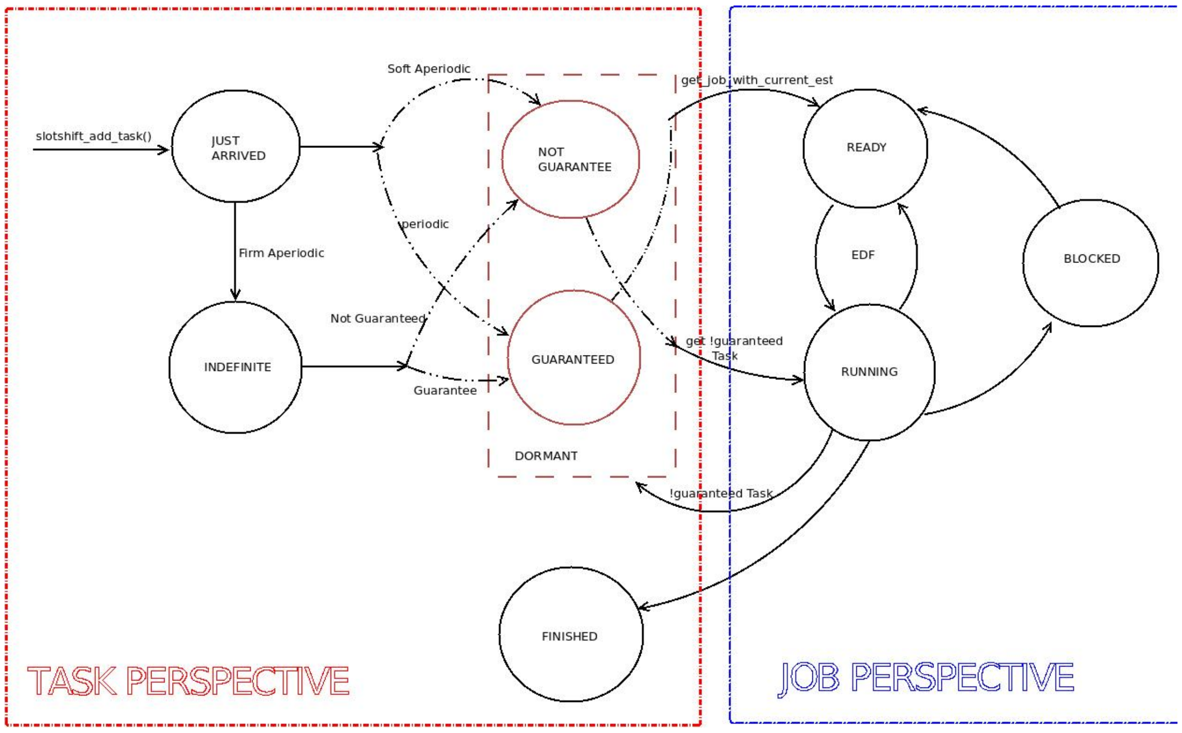
- 1.1. **selection Functionality :** selects who has to run next, this bascically runs through EDF on ready queue.
- 1.2. **Acceptance and Guarantee Functionality :** On a Firm Aperiodic task Arrival acceptance and Guarantee functionality is responsible for execution of the task within its deadline; for acheiving such a functionality first we need to search whether the task has enough exection time within its deadline, this functionality of checking availability is carried forward by acceptance test, but to make sure that once it is accepted it should meets its execution within its deadline which is carried forward by Guarantee Algorithm.These 2 functionality are 2 different algorithms that are closely bind as it performs one major task of accommodating a random arriving task to be part of normal EDF selection Functionality.
- 1.3. **Interval Functionality:** Main functionality of this module is to manage intervals Spare Capacity and update the current interval based on the slot number.
- Exposed API:
  - **.slot_shift_core**: this is the main selction functionality. along with selection of the next task this also runs acceptance and guarantee algorithms.

Data class

Manages the instance and data of tasks, intervals and execution.

- Manages Task through customized state Transition Graph.
- provides the Algorithmic class a abstract set of API's to fetch and put data.
- creates a great level of abstraction on internal data handling.
- further divided into following submodules.
  - Guaranteed Task handling
  - Non-Guaranteed Task handling
  - Interval Data Handling
- Exposed API:
  - update_tsk_state : moves the task to different states based on selction decision
  - update_tsk_quantum : updates the task execution time.
  - update_tsk_q_state : moves the task among Unconcluded/ guaranteed and not guaranteed queues.strictly works with state transition graph.

FIG2: STATE TRANSITION GRAPH:



state transition graph

FIG2 has 2 major seperations namely task perspective and job perspective.

- job perspective : Slot shifting scheduling is majorly based on jobs and interval association to enable such a mechanism selection function majorly works on job perspective
- task perspective : OS generally deals with tasks so once selection function decided then corresponding task of the selected job is returned to OS .

OS Functionality class

- intimates the OS on specific decision taken within the framework.
- This are plugin functions which is very much platform dependent.

Wrap around class

- This is an additional feature to wrap the Hyper period.
- Majorly called in Data handling functionality.

init class.

- This is an optional feature used to initialize the aperiodic tasks.
- Majorly used in monolithic kernel like Linux but can also be used on other types of OS where the event based Task should be initialized before it can actually run

decision class.

- Decides on when the selction functionality should be called.
- This is OS specific and completely decoupled from other classes.
- In Linux this is majorly a HRTIMER based trigger, where the trigger is based on slot period.

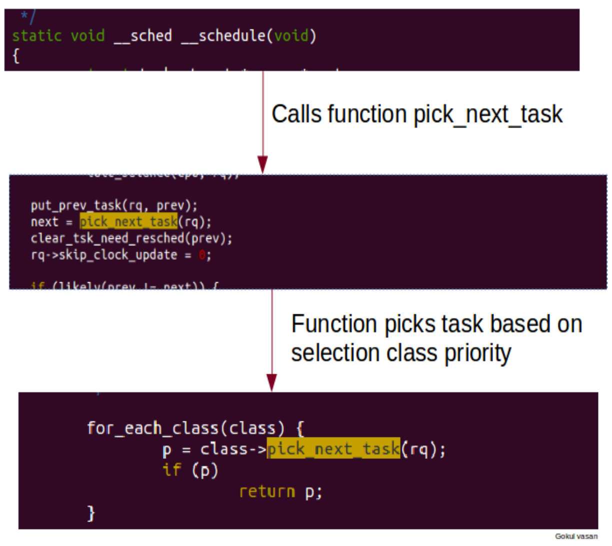
UNDERSTANDING LITMUS-rt

- We adapted Slot shifting framework for LITMUS-rt.
- Following text talks about internals and thought behind LITMUS-rt and Wip-reservations.

LITMUS-rt FRAMEWORK AND LINUX ADAPTATION:

- Implemented as a layer above Linux scheduler framework,i.e. Linux has multiple schedulers classified based on its priority of selection function.

FIG3: Linux scheduler classes


Calls function pick_next_task
Function picks task based on selection class priority
Gokul vasan

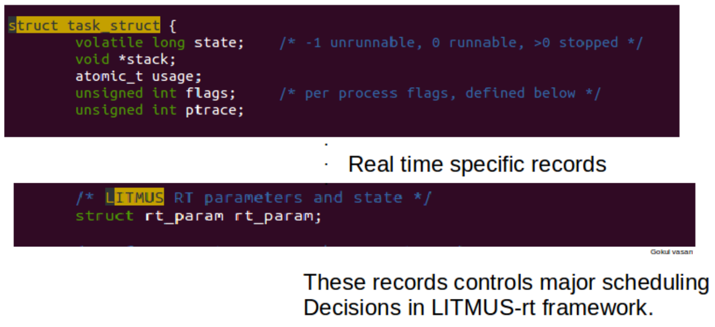
```
 */
static void __sched __schedule(void)
{
```

```
    put_prev_task(rq, prev);
    next = pick_next_task(rq);
    clear_tsk_need_resched(prev);
    rq->skip_clock_update = 0;
```

```
        for_each_class(class) {
                p = class->pick_next_task(rq);
                if (p)
                        return p;
        }
```

- The Litmus is added as one among the scheduler class here.
- Some hacks are made to struct task_struct to make the tasks rt.

FIG4: Linux adaptation for litmus-rt: Data Structure

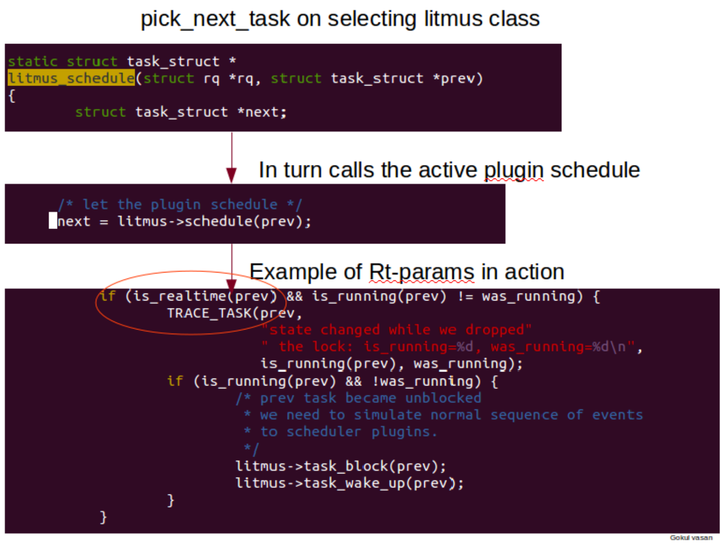

Real time specific records
Gokul vasan
These records controls major scheduling Decisions in LITMUS-rt framework.

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;
```

```
        /* LITMUS RT parameters and state */
        struct rt_param rt_param;
```

- scheduling decision is completely with linux kernel, but can be made to work for our needs by injecting tasks using liblitmus.
- selection function is called whenever linux schedule() is called.

FIG5: Litmus-rt schedule adaptation


pick_next_task on selecting litmus class
In turn calls the active plugin schedule
Example of Rt-params in action
Gokul vasan

```
static struct task_struct *
litmus_schedule(struct rq *rq, struct task_struct *prev)
{
        struct task_struct *next;
```

```
    /* let the plugin schedule */
    next = litmus->schedule(prev);
```

```
        if (is_realtime(prev) && is_running(prev) != was_running) {
                TRACE_TASK(prev,
                           "state changed while we dropped"
                           " the lock: is_running=%d, was_running=%d\n",
                           is_running(prev), was_running);
                if (is_running(prev) && !was_running) {
                        /* prev task became unblocked
                         * we need to simulate normal sequence of events
                         * to scheduler plugins.
                         */
                        litmus->task_block(prev);
                        litmus->task_wake_up(prev);
                }
        }
```

- basically the plugins are activated from userspace through proc interface.
- where the string search happens @

litmus_active_proc_write==> find_sched_plugin(name); ==> switch_sched_plugin

- the search is string based.

LITMUS-rt PLUGIN ARCHITECTURE:

- As said above different schedulers can be enabled through proc interface and this dynamic change is made by further classifying the selection function.
- such a classification is done through following structure

```
struct sched_plugin {
        struct list_head        list;
        /*      basic info              */
        char                    *plugin_name; /// STRING SEARCH TO ACTIVATE/DEACTIVATE

        /*      setup                   */
        activate_plugin_t       activate_plugin;
        deactivate_plugin_t     deactivate_plugin;
        get_domain_proc_info_t  get_domain_proc_info;

        /*      scheduler invocation    */
        schedule_t              schedule; /// FUNCTOR HOLDING SELECTION FUNCTION
        finish_switch_t         finish_switch;

        /*      syscall backend         */
        complete_job_t          complete_job;
        wait_for_release_at_t   wait_for_release_at;
        synchronous_release_at_t synchronous_release_at;

        /*      task state changes      */
        admit_task_t            admit_task;

        task_new_t              task_new;
        task_wake_up_t          task_wake_up;
        task_block_t            task_block;

        task_exit_t             task_exit;
        task_cleanup_t          task_cleanup;

        /* Reservation support */
        reservation_create_t    reservation_create;
        reservation_destroy_t   reservation_destroy;

        /* Add interval data
         * TODO: change this name! */
        slot_shift_add_interval_t       slot_shift_add_interval;

        slot_shift_aper_count_t         slot_shift_aper_count;
#ifdef CONFIG_LITMUS_LOCKING
        /*      locking protocols       */
        allocate_lock_t         allocate_lock;
#endif
} __attribute__ ((__aligned__(SMP_CACHE_BYTES)));
```

FUNCTORS EXPLANATION:

Schedule:

- when schedule decision needs to be made then schedule needs to be called.
- Input Parameters: List of tasks from which decision needs to be made.
  - it is of the type *task_struct.
- Return Value : Pointer to task_struct that is been decided as the next task to be scheduled.

Finish_switch:

- When context switch is completed then this function is called.

New Task arrival or admittance

admit_task:

- This method is called when a new task arrives.
- input Parameters: pointer to task_struct of the task that arrived.
- return value: 0 on success and negative values on failure.

task_new:

- This method is called when a existing task changes its class of scheduler.

Task State Change :

task_wakeup:

- called when a task is moved from wait queue to ready queue.

task_block :

- called when a task moves to wait queue

task_exit :

- called when task exits.

LITMUS-rt ADAPTATION FOR Wip-reservations:

As any scheduler plugin, the P-RES scheduler implements the functions needed for the `struct sched_plugin` data structure. It doesn't have, however, any code on deciding "which should be the next task to run". Rather, it manipulates a data structure called "reservation" (`struct reservation`) that can be created at will by the user space through some new syste calls (added to liblitmus).
Reservations are generic data structures. They carry a budget, a priority and a list of clients. Clients could be anything (other reservations, a task, ...). They also carry function pointe to defined operations, such as "what to do when a new client enters the reservation", "what to do when a client gets out of the reservation", "how/when to replenish its budget", ...
It is possible to use the reservation data structure to create new "flavors" of reservations. Two examples already exist: table-driven and polling reservations. Both redefine the default operations and add new variables to store internal state. The P-RES plugin has a list of so-called "active" reservations. When deciding what task runs next (i.e., when `schedule()` is called), it simply calls the highest priority active reservation's `dispatch()` function. It doesn't make, therefore, any scheduling decision, leaving this decision entirely for the reservation code. Note that this reservation code could have been overriden in new reservation types, and thus this provides a new abstraction upon which schedulers could be created.
When a task requests to become a RT-task in a specific reservation, P-RES simply associates that client to the given reservation (in `admit_task()`).
The `task_new()` function would then insert the client in the reservation. At this point, the reservation's `client_arrive()` function is called. Again, this code is reservation specific and could be overriden by new reservation types.

wip-reservation Data Structure:

all the existing reservations are stored in a data structure called `struct sup_reservation_environment` (which we will call "sup_environment" for short). The sup_environment stores t current time as well as four lists of reservations:

- The currently active reservations
- The currently depleted reservations
- The currently inactive reservations.

SNIPPET: holds the simple uniprocessor reservation record.

```
struct sup_reservation_environment {
        struct reservation_environment env;

        /* ordered by priority */
        struct list_head active_reservations;

        /* ordered by next_replenishment */
        struct list_head depleted_reservations;

        /* unordered */
        struct list_head inactive_reservations;

        /* - SUP_RESCHEDULE_NOW means call sup_dispatch() now
         * - SUP_NO_SCHEDULER_UPDATE means nothing to do
         * any other value means program a timer for the given time
         */
        lt_t next_scheduler_update;
        /* set to true if a call to sup_dispatch() is imminent */
        bool will_schedule;
};
```

- Notice that the sup_environment stores a `**struct reservation_environment**`, which contains the environment "**starting**" time, the current time, and a pointer to a function tha reservations can call to request changing from one of the lists to another one. Its definition can be seen below

```
struct reservation_environment {
                lt_t time_zero;
                lt_t current_time;

                /* services invoked by reservations */
                reservation_change_state_t change_state;
        };
```

- within which further separation called reservation is created.
- Enables association of clients with specific reservation, and provide different selection function within clients.
- struct reservation looks like.

```
struct reservation {
        /* used to queue in environment */
        struct list_head list;

        reservation_state_t state;
        unsigned int id;

        /* exact meaning defined by impl. */
        lt_t priority;
        lt_t cur_budget;
        lt_t next_replenishment;

        /* budget stats */
        lt_t budget_consumed; /* how much budget consumed in this allocation cycle? */
        lt_t budget_consumed_total;

        /* interaction with framework */
        struct reservation_environment *env;
        struct reservation_ops *ops;

        struct list_head clients;
};
```

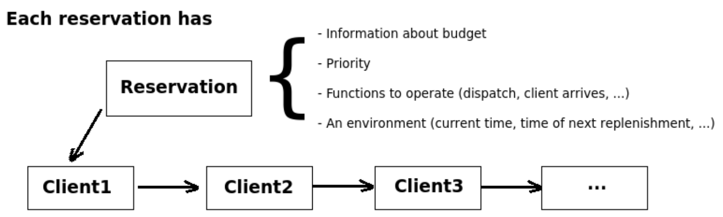
- associated with this reservation is a ops variable which is used for handling the data of this reservation.

```
struct reservation_ops {
        dispatch_client_t dispatch_client;

        client_arrives_t client_arrives;
        client_departs_t client_departs;

        on_replenishment_timer_t replenish;
        drain_budget_t drain_budget;
};
```

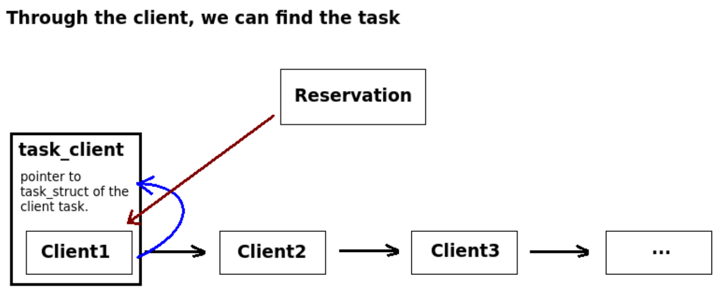
USING SUP_ Reservation ENVIRONMENT

CLIENTS AND TASKS in wip-reservation:

- wip-reservation has abstract concept called clients.
- These clients gets associated with its reservation.
- Through reservation we can find clients and vice-verse.


Each reservation has
Reservation
- Information about budget
- Priority
- Functions to operate (dispatch, client arrives, ...)
- An environment (current time, time of next replenishment, ...)
Client1
Client2
Client3
...

reservation and client relation


Through the client, we can find the task
Reservation
task_client
pointer to task_struct of the client task.
Client1
Client2
Client3
...

reservation and client relation

FUNCTIONS:

- 2 basic functions covers almost all the details of reservation framework.

sup_update_time():

Before any sup_ function is called, the scheduler plugin (in our case, P-RES) is expected to call `sup_update_time()`. The function does three things:

- Updates the current time in the environment;
- Drains the budgets of the reservations that should have their budget drained (i.e., calls the reservation's `drain_budget()` function);
- Replenishes the budgets of the reservations that should have their budget replenished (i.e., calls the reservations't `replenish()` function).

Step 2 acts only in the active reservations list. It calls `drain_budget()` in all reservations in the list (which is ordered by priority) until it finds the first non idle reservation. The following pseudo-code describes the operation:

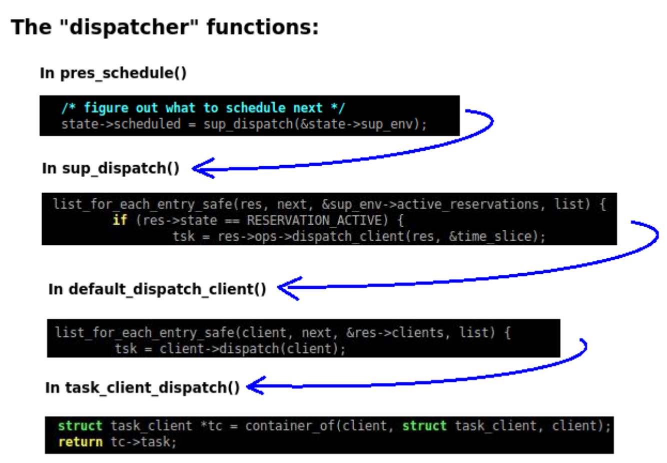
```
for_each reservation r in active_list
                r.drain_budget(res, delta);
                if (r.state = ACTIVE)
                        break
```

Notice that the reservation's `drain_budget()` function could do anything (i.e., it is independent of what the SUP_ Environment understands as budget).

Step 3 acts only in the depleted reservations list. It calls `replenish()` in all reservations whose `next_replenishment` time variable is smaller than (i.e., is in the past in relation to) the current time. Since the depleted list is ordered by next_replenishment, it stops when it finds the first reservation that still needs to wait some time until its next replenishment. The follow
pseudo-code describes the operation:

```
for_each reservation r in depleted_list
                if (r.next_replenishment <= current_time)
                        r.replenish(res);
                else
                        break;
```

sup_dispatch()



The "dispatcher" functions:
In pres_schedule()
In sup_dispatch()
In default_dispatch_client()
In task_client_dispatch()

```
/* figure out what to schedule next */
state->scheduled = sup_dispatch(&state->sup_env);
```

```
list_for_each_entry_safe(res, next, &sup_env->active_reservations, list) {
        if (res->state == RESERVATION_ACTIVE) {
                tsk = res->ops->dispatch_client(res, &time_slice);
```

```
list_for_each_entry_safe(client, next, &res->clients, list) {
        tsk = client->dispatch(client);
```

```
struct task_client *tc = container_of(client, struct task_client, client);
return tc->task;
```

Flow of dispatch function

- sup_dispatch is called by litmus_schedule() like where plugin is P-RES which calls pres_schedule() through litmus->schedule().
- Instead of calling directly a reservation's `dispatch()` function, the P-RES plugin calls the sup_environment's `sup_dispatch()` function:
- In case the highest priority reservation doesn't return a task, the next reservation is given the opportunity to dispatch one of its clients.
- After a client task is found, `sup_scheduler_update_after()` is called, so that the sup_environment updates itself again soon.

```
for_each_reservation r in active_list
                if (r.state = ACTIVE)
                        tsk = r.dispatch_client(res, &time_slice);
                        if (tsk)
                                sup_scheduler_update_after(sup_env,
                                                res->cur_budget);
                                return tsk;
```

Created by vasan. Last Modification: Friday 11 of December, 2015 17:02:27 CET by vasan.