[Log o](http://rts-wiki/tiki-logout.php)



##### Log In

Logged in as: vasan

[Log out](http://rts-wiki/tiki-logout.php)

##### Last­Visited Pages

1. [LITMUS­rt SLOTSHIFTING ARCHITECTURE](http://rts-wiki/tiki-index.php?page=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE)

##### Quick Edit a Wiki Page

Create/Edit

[**LITMUS­rt SLOTSHIFTING ARCHITECTURE**](http://rts-wiki/tiki-index.php?page=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE) **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,

* 1. the functions in algorithmic part are very much made disassociated enabling the replacing of any core functionality with other.

[Home](http://rts-wiki/tiki-index.php)

##### Menu

**SLOT SHIFTING ALGORITHM ITSELF**

The Algorithm Works in 2 phases Offline Phase

[Categories](http://rts-wiki/tiki-browse_categories.php) [MyTiki](http://rts-wiki/tiki-my_tiki.php) [Wiki](http://rts-wiki/tiki-index.php)

[Image Galleries](http://rts-wiki/tiki-galleries.php) [File Galleries](http://rts-wiki/tiki-list_file_gallery.php) [Admin](http://rts-wiki/tiki-admin.php)

##### Since your last visit... Tue 23 of Feb., 2016

Nothing has changed

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(Ii­1))

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 task­set output t,I: t is list of jobs sorted based on dl.

I is list of intervals

st := sort jobs of task­set 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, Ii­1.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 not­guaranteed 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++;

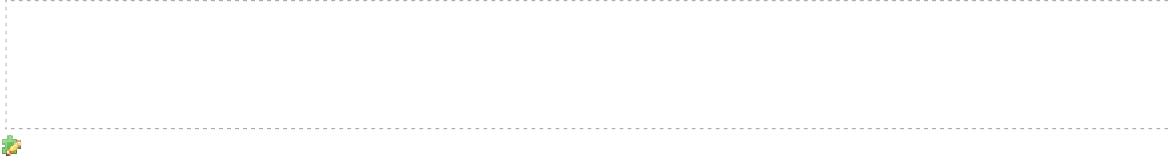
**FIRM APERIODIC ARRIVAL (JA)**:acceptance\_guarantee\_algorithm(JA)

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

else

delta = delta – SC(i\_temp); SC(i\_temp) = ­delta;

end IF

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);

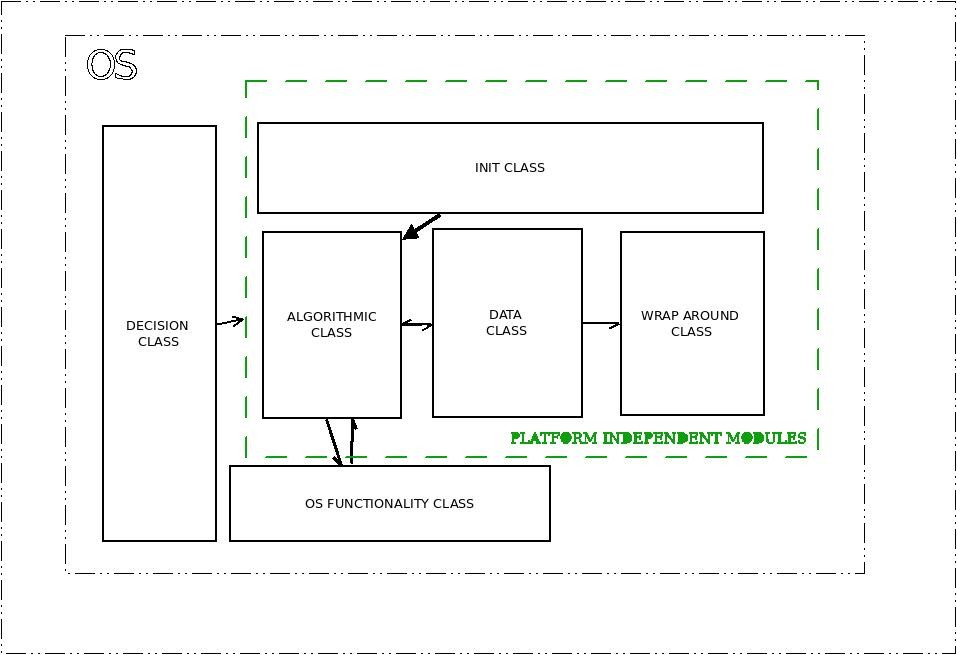
return Ileft

# 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. **selection Functionality :** selects who has to run next, this bascically runs through EDF on ready queue.
   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 ED selection Functionality.
   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 Non­Guaranteed 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:

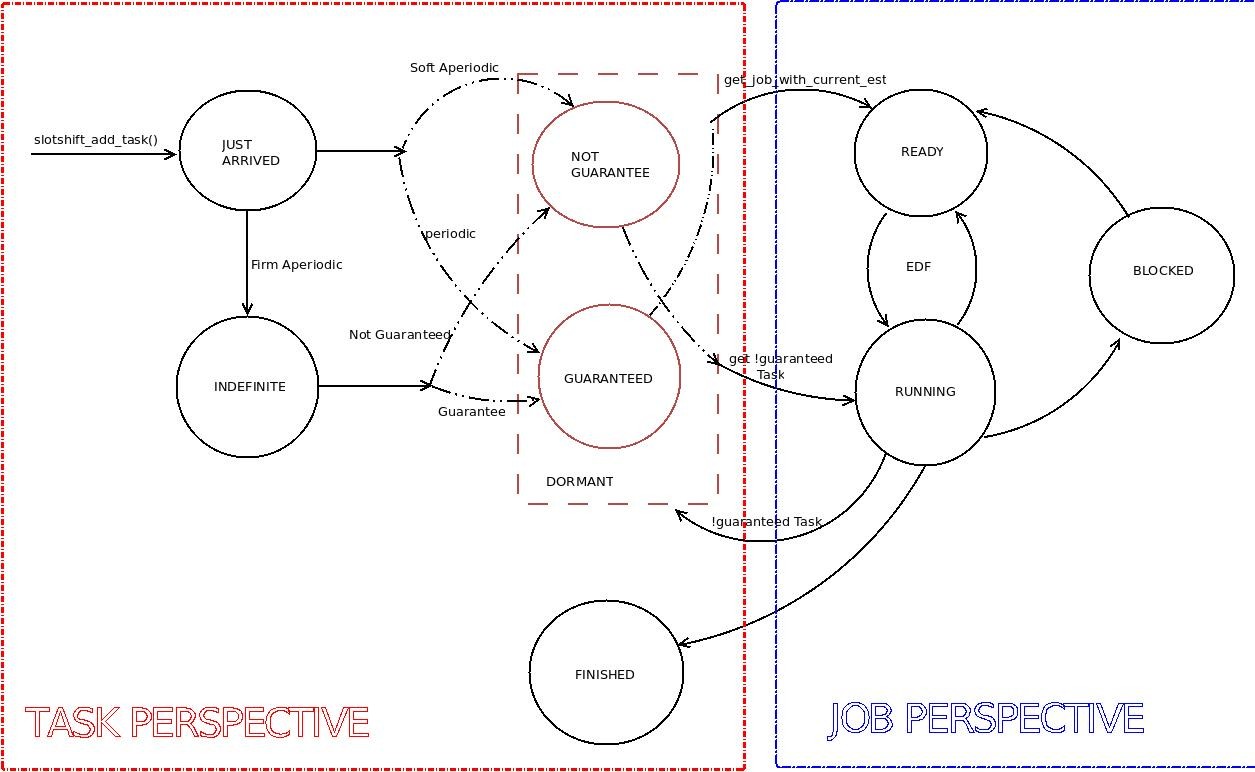




FIG2 has 2 major seperations namely task perspective and job perspective.

state transition graph

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 LITMUS­rt

We adapted Slot shifting framework for LITMUS­rt.

Following text talks about internals and thought behind LITMUS­rt and Wip­reservations.

### LITMUS­rt 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



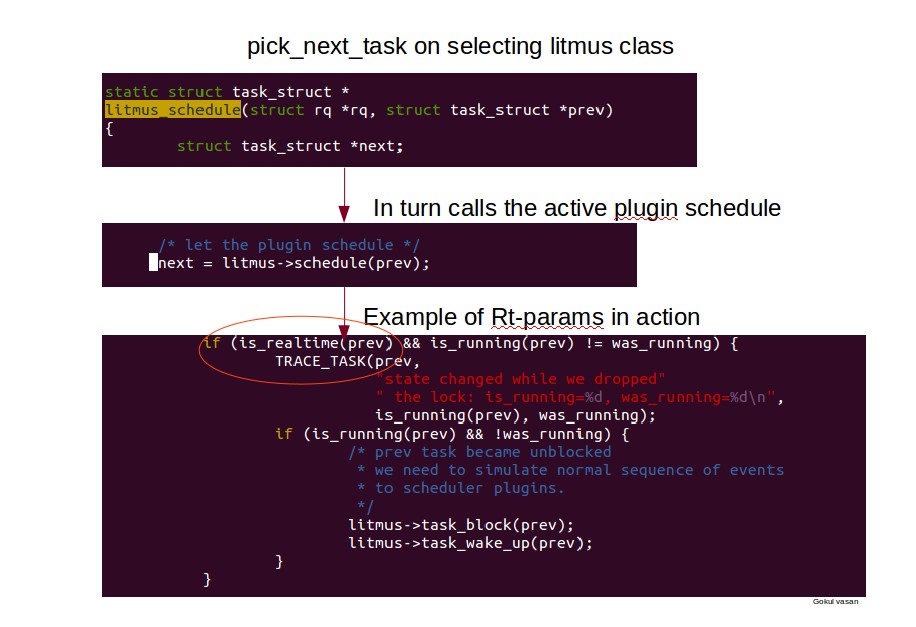
The Litmus is added as one among the scheduler class here. Some hacks are made to struct task\_struct to make the tasks rt.

#### FIG4: Linux adaptation for litmus­rt: Data Structure



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: Litmus­rt schedule adaptation



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.

### LITMUS­rt 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 {

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

struct list\_head

list;

/\*

char

basic info

\*/

\*plugin\_name; /// STRING SEARCH TO ACTIVATE/DEACTIVATE

/\*

setup

activate\_plugin\_t

deactivate\_plugin\_t get\_domain\_proc\_info\_t

\*/

activate\_plugin; deactivate\_plugin; get\_domain\_proc\_info;

/\*

scheduler invocation

\*/

schedule\_t

finish\_switch\_t

schedule; /// FUNCTOR HOLDING SELECTION FUNCTION

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\_wake\_up\_t task\_block\_t

task\_new;

task\_wake\_up; task\_block;

task\_exit\_t

task\_cleanup\_t

task\_exit;

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;

141 #ifdef CONFIG\_LITMUS\_LOCKING

142

143

144 #endif

/\*

locking protocols

\*/

allocate\_lock\_t

allocate\_lock;

145 } \_\_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.

### LITMUS­rt ADAPTATION FOR Wip­reservations:

As any scheduler plugin, the P­RES 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 syst 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: table­driven and polling reservations. Both redefine the default operations and add new variables to store internal state. The P­RES plugin has a list of so­called "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 RT­task in a specific reservation, P­RES 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.

#### wip­reservation 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

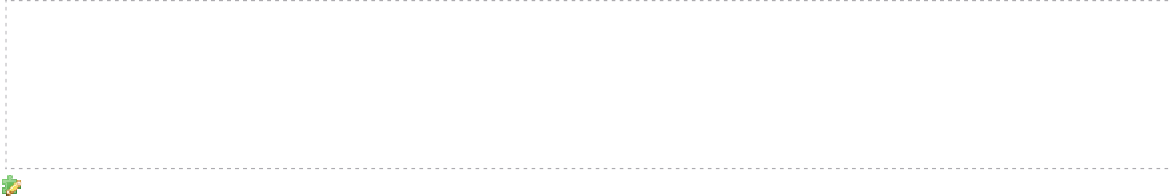
\*/

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 th reservations can call to request changing from one of the lists to another one. Its definition can be seen below

struct reservation\_environment {

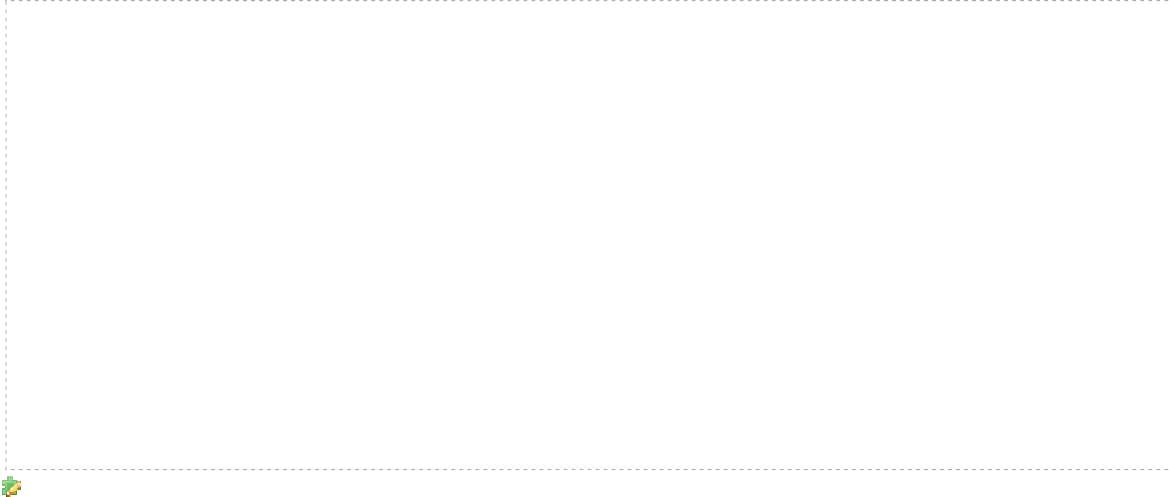
lt\_t time\_zero; lt\_t current\_time;

/\* 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;

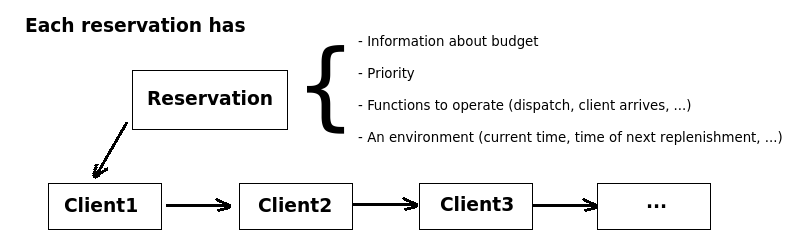
on\_replenishment\_timer\_t replenish; drain\_budget\_t drain\_budget;

};

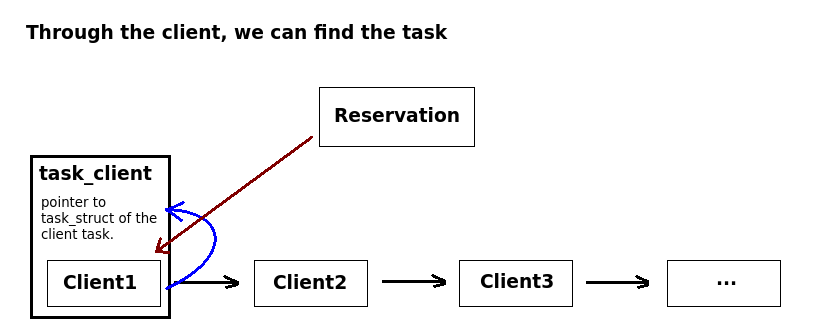
#### USING SUP\_ Reservation ENVIRONMENT CLIENTS AND TASKS in wip­reservation:



wip­reservation has abstract concept called clients. These clients gets associated with its reservation. Through reservation we can find clients and vice­verse.



reservation and client relation





#### FUNCTIONS:

reservation and client relation

1. 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, P­RES) 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 pseudo­code 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 pseudo­code describes the operation:



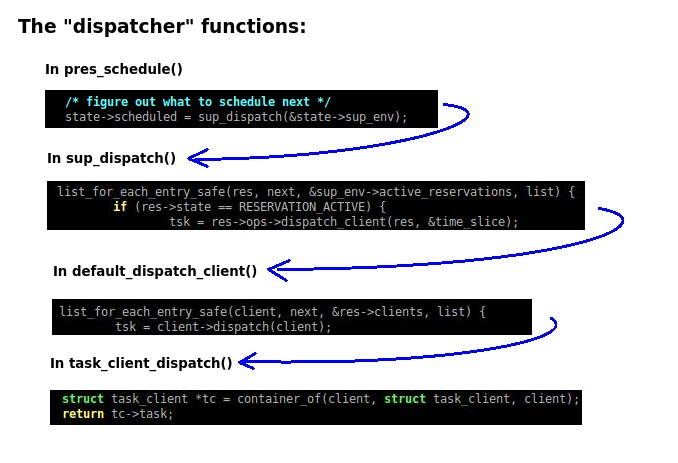
for\_each reservation r in depleted\_list

if (r.next\_replenishment <= current\_time) r.replenish(res);

else

break;

#### sup\_dispatch()



Flow of dispatch function

sup\_dispatch is called by litmus\_schedule() like where plugin is P­RES which calls pres\_schedule() through litmus­>schedule(). Instead of calling directly a reservation's `dispatch()` function, the P­RES 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.*

[Edit this page](http://rts-wiki/tiki-editpage.php?page=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE) [Source](http://rts-wiki/tiki-pagehistory.php?page=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE&amp;source=0) [Remove](http://rts-wiki/tiki-removepage.php?page=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE&amp;version=last) [Rename](http://rts-wiki/tiki-rename_page.php?page=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE) [Permissions](http://rts-wiki/tiki-objectpermissions.php?objectId=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE&amp;objectName=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE&amp;objectType=wiki%2Bpage&amp;permType=wiki) [History](http://rts-wiki/tiki-pagehistory.php?page=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE) [Comments](http://rts-wiki/tiki-ajax_services.php?controller=comment&amp;action=list&amp;type=wiki%2Bpage&amp;objectId=LITMUS-rt%2BSLOTSHIFTING%2BARCHITECTURE&amp;comment-container) 3 files attached

Powered by [Tiki Wiki CMS Groupware](http://tiki.org/) | Theme :[Twist](http://themes.tiki.org/)

[Wiki](http://rts-wiki/tiki-wiki_rss.php?ver=2) [Image Galleries](http://rts-wiki/tiki-image_galleries_rss.php?ver=2) [File Galleries](http://rts-wiki/tiki-file_galleries_rss.php?ver=2)