Design Documentation of LITMUS ^{RT} Reservation FrameWork.

The Following Document tries to provide a high level and implementation view of reservation framework implemented in LITMUS-RT.
I have tried to keep the Documentation more as a developer's handbook but in case of missing links the best approach is to browse the code with this document as reference.
The first part of this document start with the basic ideology behind the implementation and approach followed to achieve the same.The Second part of this document goes into specific details of the implementation which includes SUP: Single Uniprocessor, reservations and Clients

1.Ideology

The reservation based framework provides a method to implement and test different schedulers concurrently by abstracting them into a concept called **reservation**.
Each reservation could be a different scheduler / selection function with which additional parameter called budget is associated. The budget decides the state of the reservation, i.e. Active, Active-idle, depleted, Inactive (we will discuss about the state transition in much detail as we move more specific towards implementation.). Budget forms the basis of concurrency among reservations and moves the reservations through different states. The reservations that are in Active state are only eligible to schedule its tasks.
Reservations have many clients. The Clients are the scheduling entity,i.e. The concept of tasks and jobs are further abstracted with Clients. Client is a abstract data structure holding tasks with which the reservations make the selection function.
Reservations and clients are closely associated and there is no methodology to migrate a client from one reservation to other. The reservations provides a closed environment for the scheduling selection function,i.e. The reservations are hidden from the other reservations and have no communication with other reservations.

2. Architecture:

To Achieve reservation based methodology all active CPU's holds a state. The state would take care of decision functionality and selection functionality for each CPU individually. The decision functionality is through a timer. The trigger point is decided by the reservations budget. The select functionality of reservations are managed through an abstracted data structure named reservation environment, this Forms the basis for partitioned reservation. This reservation_environement is called SUP / Simple uni-processor framework. Each Active CPU has a SUP framework. SUP framework manages the state transition of the reservations for each CPU through queue infrastructure.
Reservations has a set of function pointers/virtual functions, This gives the framework a flexibility and customize the reservation to implement and test different selection algorithm.

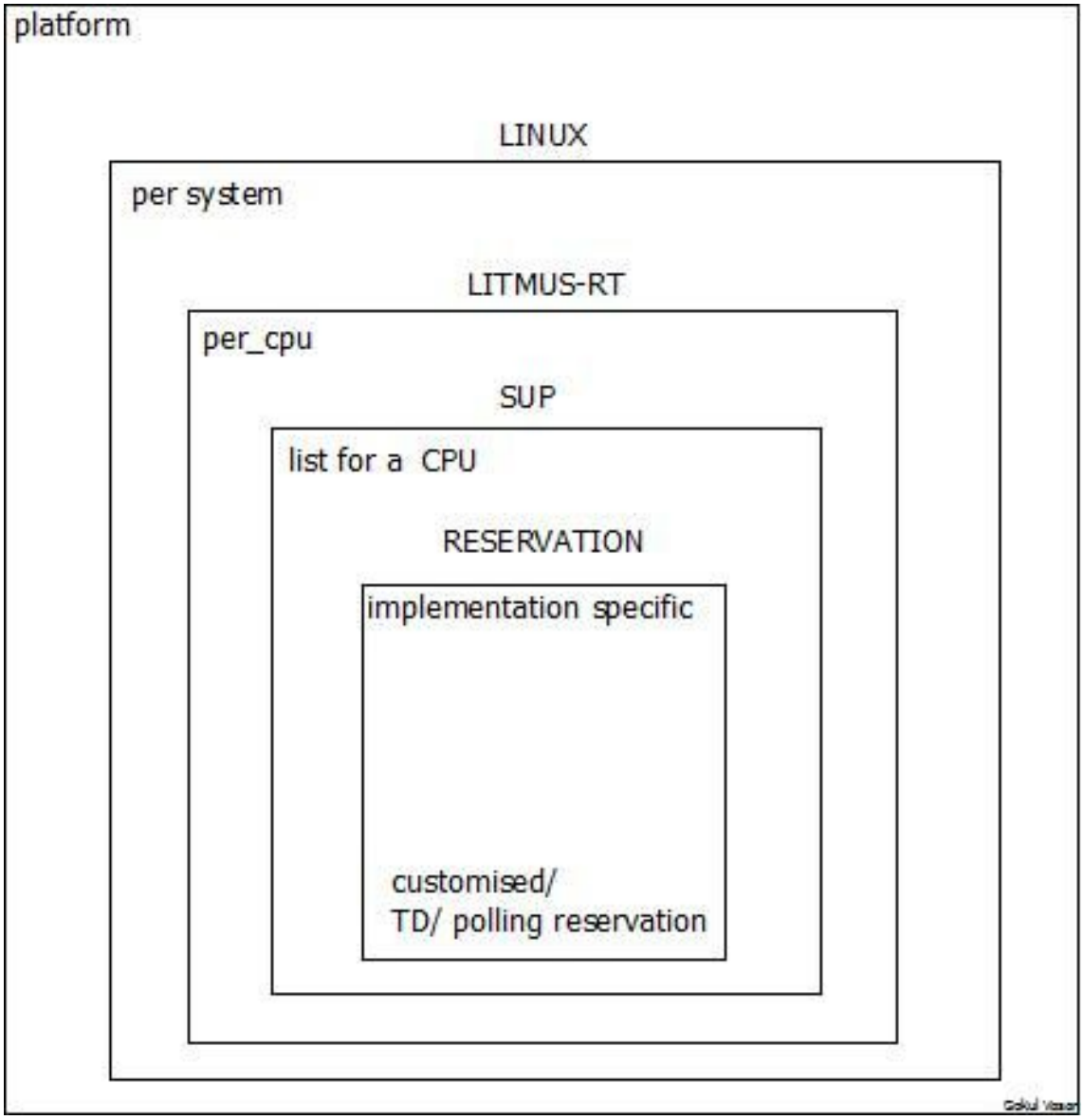

FIG 1 : Architecture View Of reservation framework.

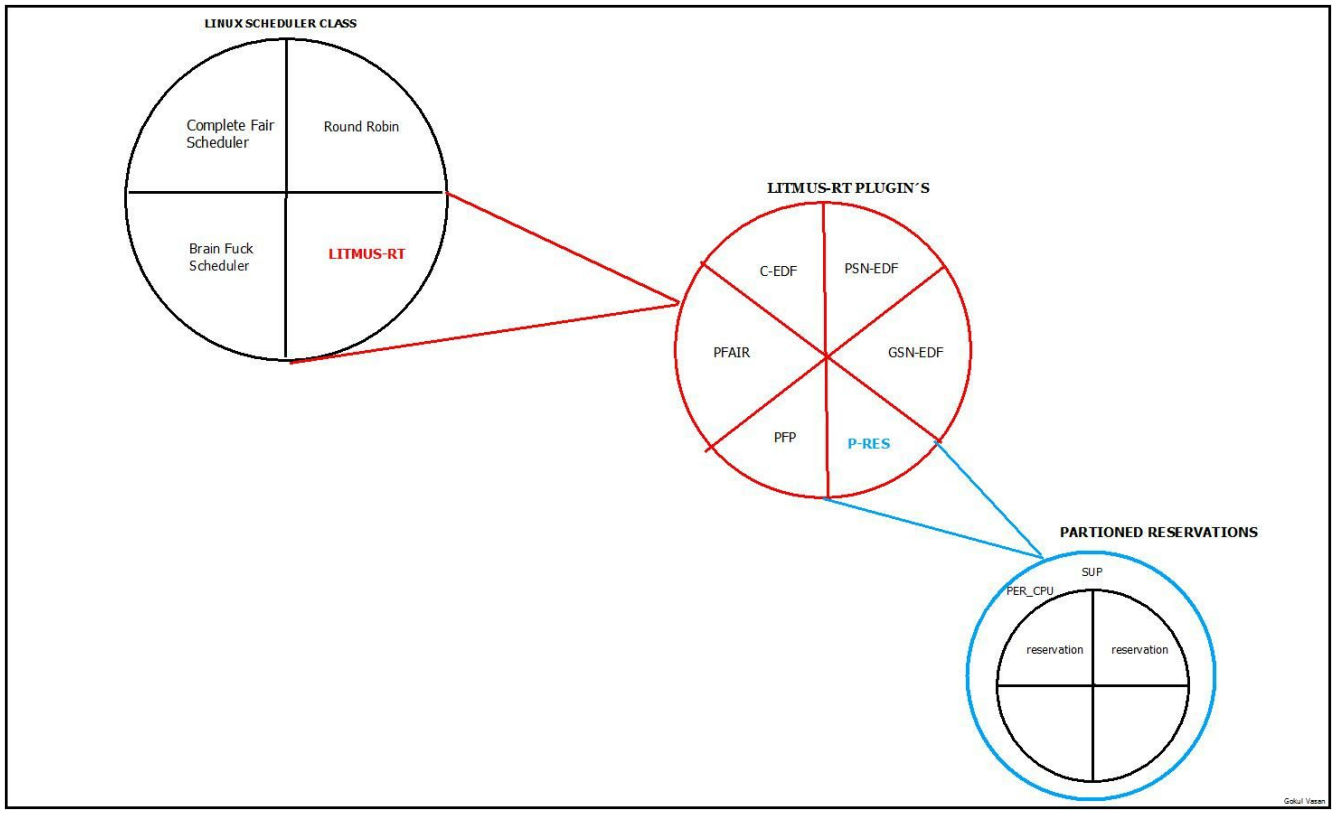

FIG 2 : P-RES LOCATION AND VIEW FROM LINUX

2.1. State Transitions of reservations.

Each reservation gets created with some customized parameters along with budget parameter which provides the amount of CPU time the reservation needs to execute all its clients. Budget and clients decides the state of the reservation. Initially on time of creation reservation is inactive and becomes active on arrival of client.

as the reservation is in active state the budget gets charged. SUP moves the reservation to depleted state once the budget reaches 0.

Once budget is depleted, replenishment happens. replenishment is very much implementation specific, but on replenishment the reservation can go to ACTIVE_IDLE when there are no clients in spite replenished or can go to INACTIVE state. The Fig 2 provides the state transition graph flow.

The complete movement within the transition graph is implementation specific through virtual functions provided for reservation.

The Below fig3 transition is just a suggestion provided by looking into the ideology, but the complete transition can be customized based on the implementation of the reservation.

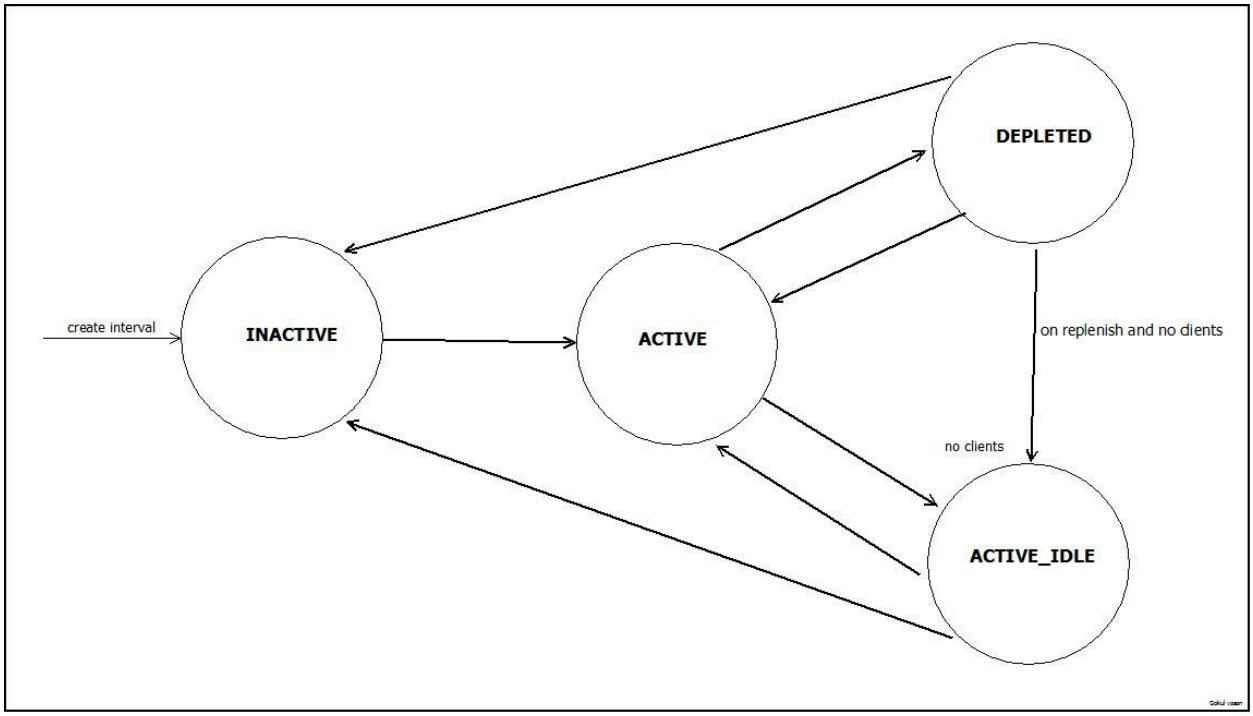

Fig 3: state transition Flow of reservations

3. Implementation:

3.1. Initialisation Phase:

- The P-RES framework is just another plugin sitting along with other schedulers in LITMUS^{RT} .
- The activation of P-RES is same as activation of other plugins , from user space using **setsched** command.
- On activation the P-RES initializes the **struct pres_cpu_state** for each CPU through DEFINE_PER_CPU macro in linux.
- It initialises SUP framework for every CPU through **sup_init** function.
- it also initializes **timer** that is associated with every CPU for decision function.

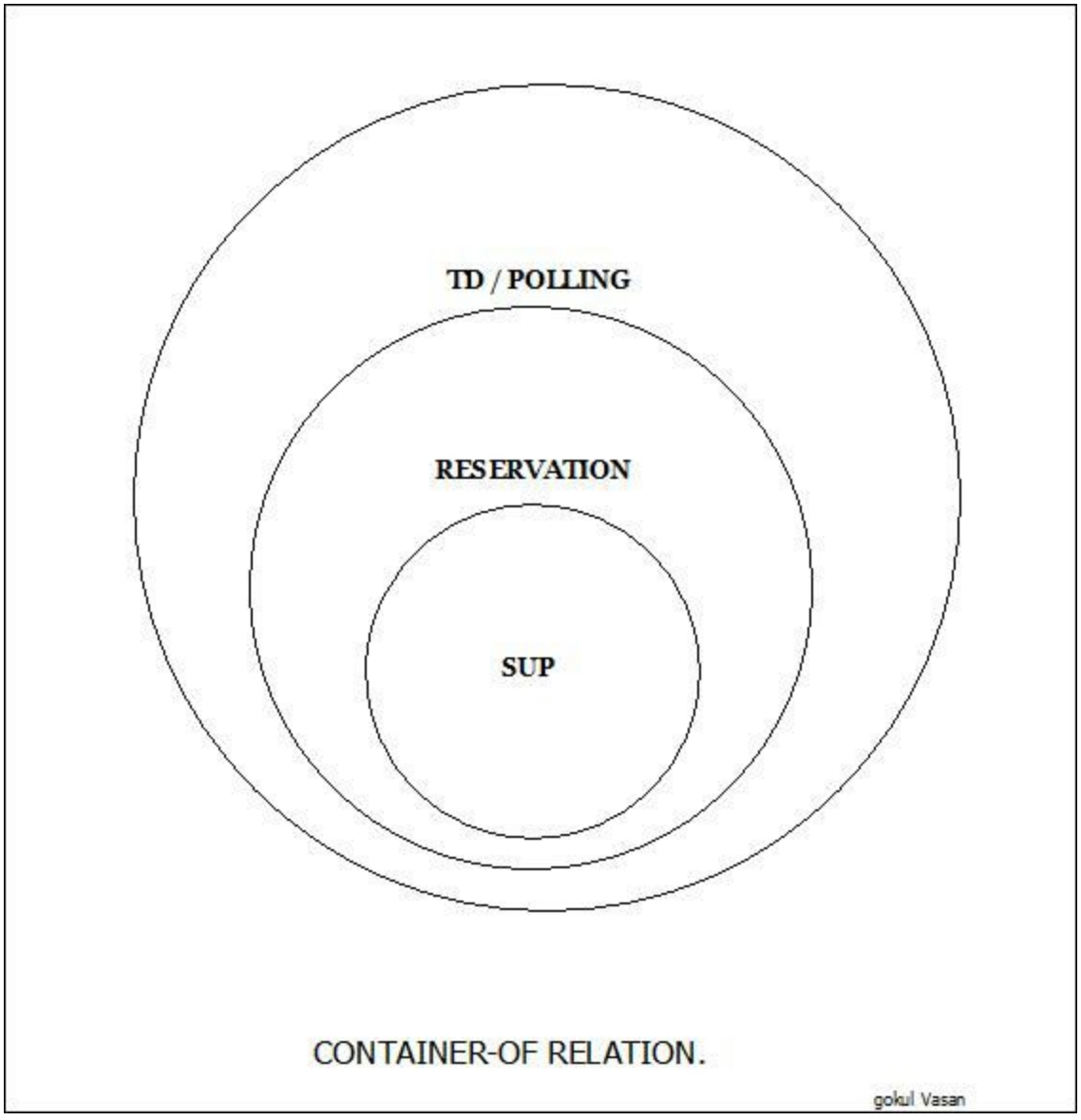

Fig4 : Implementation perspective of P-res.

3.2. CPU State:

- every active CPU has the struct pres_cpu_state once pres becomes active plugin
- each cpu has a lock to access this structure mutually exclusive.
- sup reservation environment which defines and abstracts reservation framework and provides selection functionality.
- struct hrtimer is used as a decision function, and its trigger is based on variable next_scheduler_update defined in sup_env
- the structural view is provided in snippet 1.

```
struct pres_cpu_state {
        raw_spinlock_t lock; /// lock for mutual exclusion

        struct sup_reservation_environment sup_env; /// sup_environment defined per cpu
        struct hrtimer timer; /// decision function, timeout is based on variable
                              /// next_scheduler_update

        int cpu; /// cpu id
        struct task_struct* scheduled; /// task that is currently scheduled
};
```

Snippet1 : CPU state defined for each cpu

3.3. SUP: Single UniProcessor Framework:

```
#define SUP_RESCHEDULE_NOW (0)
#define SUP_NO_SCHEDULER_UPDATE (ULLONG_MAX)

/* A simple uniprocessor (SUP) flat (i.e., non-hierarchical) reservation
 * environment.
 */
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;
};
```

Snippet 2: struct sup_reservation_environment

- The first element is SUP is **env** which is used as reference in **struct reservation** to identify the SUP framework to which the reservation belongs.
- there are 3 list within which the reservations are shifted, namely active, depleted, inactive reservations. As already said this transitions is based on BUDGET. snippet 3 provides a superficial computational view.
    - sup_update_time function is called on decision function timer expiry
    - This timer charges all the reservations in ACTIVE and ACTIVE_IDLE state with the time difference between previous time update and current time.

- Once charged and budget reaches a value 0 or less, SUP moves the reservation to depleted state.
- sequentially depleted reservation's replenish through **sup_replenish_budget**. In function sup_replenish_budget **next_replenishment** variable is checked and if it holds value less than or equal to the current time then replenish function specific to reservation is called.

```
void sup_update_time( struct sup_reservation_environment* sup_env,
                                lt_t now)
{
                                  .
                                  .
                                  .
        delta = now - sup_env->env.current_time;
        sup_env->env.current_time = now;

        /* check if future updates are required */
        if (sup_env->next_scheduler_update <= sup_env->env.current_time)
                sup_env->next_scheduler_update = SUP_NO_SCHEDULER_UPDATE;

        /* deplete budgets by passage of time for ACTIVE reservations*/
        sup_charge_budget(sup_env, delta);

        /* check if any budgets of reservations are replenished */
        sup_replenish_budgets(sup_env);

}
```

SNIPPET 3: Budget charging and replenish.

- **next_scheduler_update**: this variable decides the next moment of time out for decision function. snippet 4 below gives a superficial view of this.
  - if the value is SUP_RESCHEDULE_NOW then dispatch now and SUP_NO_SCHEDULER_UPDATE means not to do anything.
  - else the value of next_scheduler_update is done through functions ***sup_scheduler_update_at*** and **sup_sheduler_update_after** functions.These functions are called by ***sup_queue_depleted, sup_replenish_budget, sup_dispatch, sup_queue_active, sup_charge_budget***.Most of the mentioned sup functions are called on timeout through **sup_update_time**.

```
static enum hrtimer_restart on_scheduling_timer(struct hrtimer *timer)
{
                          .
                          .
                          .
       update = state->sup_env.next_scheduler_update;
       now = state->sup_env.env.current_time;

       if (update <= now) { /*Obviously update value will be less than now val
when*/
                               /* SUP_RESCHEDULE_NOW  is set*/
               litmus_reschedule_local(); /*Calls the dispatch function */
```

```
        } else if (update != SUP_NO_SCHEDULER_UPDATE) {
                hrtimer_set_expires(timer, ns_to_ktime(update));
                restart = HRTIMER_RESTART;
        }
}
```

snippet 4: next_scheduler_update usage forms the basis for decision functionality.

3.4. Reservations:

```
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;
};
```

Snippet 5: struct reservation and its ops

- Each CPU can have a list of reservations which is managed by the element **list**.
- Each reservation has an unique **ID** which is used to identify the reservations.
- the **priority** enables selection of reservation to execute its client next.
- **curr_budget** : provides the budget still the reservation has to be in active state.
- **next_replenishment**: on exhaustion of budget next_replenishment decides when the reservation should be refilled.
- **env:** is the reference to SUP framework through container_of macro.
- **clients:** the list of clients that belongs to this reservation.
- **ops :** This structure makes the reservations customizable for user implementations, holds a set of virtual functions, ***enabling users to implement and test algorithms***.

```
struct reservation_ops {
        /*
         * on schedule() or
         * on_scheduling_timer()-> next_scheduler_update value
```

```
         */
        dispatch_client_t dispatch_client;

        /* Task arriving and departing handler
         * Arriving: new task assigned to reservation.
         *           or tasked that is assigned is moved to runqueue
         * Departing: on blocking or task completion
         */
        client_arrives_t client_arrives;
        client_departs_t client_departs;

        /*Budget based property update */
        on_replenishment_timer_t replenish;
        drain_budget_t drain_budget;
};
```

Snippet 6: struct reservation_ops

- ***dispatch_client:*** Called on selection functionality, The selection functionality is called when Linux **schedule()** is triggered or **decision timer** is out.
  - Schedule() in linux could be due to syscall sys_complete_job() which is triggered from user space by test tasks like rt-spin.
- ***client_arrives***: This method gets called whenever client either gets created or assigned to a reservation or when a client moves from blocked state to ready state.
- ***client_departs:*** Whenever client gets blocked or client gets terminated or finished.
- ***replenish:*** Called from SUP framework timer when the budget of this reservation is depleted, This is implementation specific of reservation implementation.
- ***drain_budget:*** On Each timeout of the SUP timer the reservations in ACTIVE state is drained along with that reservation's drain_budget is also called, where the implementation is specific to reservation implementation

- Snippet 5 holds the record for the explanation.

3.5. Client :

```
/* Reservation View */
struct reservation_client {
        struct list_head list;
        struct reservation* reservation;
        dispatch_t dispatch;
};

/* Linux View: task is returned on dispatch*/
struct task_client {
        struct reservation_client client;
        struct task_struct *task;
};
```

Snippet 7: Struct Client and task_client

- Provides an abstract View for tasks and associates clients with the reservation.
- reservations holds a list of struct reservatiopn_client
- which is embedded within task_client and is extracted through **dispatch** function in **reservation_client**

```c
static struct task_struct * task_client_dispatch(struct reservation_client *client)
{
	struct task_client *tc = container_of(client, struct task_client, client);
	return tc->task;
}
```

Snippet 8: reservation_client dispatch method.

- The client becomes associated with a reservation by updating task_struct's rt_params. which can be done through following ways
  - In case of existing task migration to reservation :
    - `resctrl -a ${PID_OF_EXISTING_TASK} -r 1234 -c 0`
  - In case of new task being created :
    - `rt_launch -p 0 -r 1234 100 100 -e task_name`
    - `rtspin -p 0 -r 1234 -v 10 100 60 -r $RESERVATION_ID`

3.6. Conclusion :

- customized budget management policies forms **container of** each reservation,
- for example: Table driven reservation manages replenish and drain through a concept called intervals.
  - On Depleted budget, replenish can happen only at the next cycle start.
  - This replenish update is given to variable next_replenishment in reservation.

```c
struct table_driven_reservation {
	/* extend basic reservation */
	struct reservation res;

	lt_t major_cycle;
	unsigned int next_interval;
	unsigned int num_intervals;
	struct lt_interval *intervals;

	/* info about current scheduling slot */
	struct lt_interval cur_interval;
	lt_t major_cycle_start;
};
```

3.7. HOW-TO reservation for dummies:

- **STEP 1:** Add new reservation type in enum reservation_type_t

```c
/* reservation support */
/* /kernel/include/rt-param.h */
typedef enum {
	PERIODIC_POLLING,
	SPORADIC_POLLING,
	TABLE_DRIVEN,
	SLOT_SHIFTING
} reservation_type_t;
```

- **STEP 2**: Add your specific Config in struct **reservation_config**.
    - This is the interface / structure that brings the data from user space to kernel space with respect to data on reservation handling.
    - reservation handling is mostly all about budget.
    - These config structure can be used to define on how to handle the reservation on depletion and replenishment of budget.
    - This should be copied to kernel space in a convenient structure for further usage.
    - In case of table driven the whole config parameter is copied into struct table_driven_reservation, as mentioned earlier the struct is the super container of reservations.
    - **Example**: A Simple reservation for slot shifting.

```
/*/kernel/include/rt-param.h */
struct reservation_config {
        unsigned int id;
        lt_t priority;
        int  cpu;

        union {
                struct {
                        lt_t period;
                        lt_t budget;
                        lt_t relative_deadline;
                        lt_t offset;
                } polling_params;

                struct {
                        lt_t major_cycle_length;
                        unsigned int num_intervals;
                        struct lt_interval __user *intervals;
                } table_driven_params;
               struct {
                       lt_t reservation_length;
                       lt_t slot_length;
               }slot_shift_params;
        };
};
```

        - In The above structure a simple variable reservation_length is added, This defines the length of reservation based on hyper period and repetitive count.
        - if the hyper period is 10 slots with 5ms each slot and repetitive count is 1 then reservation_length is 10*5*1 = 50ms should be the param.

- **STEP3**: Add The corresponding type in syscall interface to bring and initialize data in kernel space.

```
 /*/litmus/sched_pres.c ==> pres_reservation_create() kernel version: 3.10.41*/
709 static long pres_reservation_create(struct res_data __user *res_type,
710                                      void __user *_config)
711 {
712         long ret = -EINVAL;
713         void *config;
714         struct res_data type;

718         if(copy_from_user(&type, res_type, sizeof(type))) {
719                 printk(KERN_ERR " invalid res_data\n");
720                 return -EFAULT;
721         }
722
723         config = kmalloc(type.param_len, GFP_KERNEL);
724
725         if (copy_from_user(config, _config, type.param_len))
726                 return -EFAULT;
735         .
            .
            .
736         switch (type.type) {
737                 case PERIODIC_POLLING:
738                 case SPORADIC_POLLING:
739                         ret = create_polling_reservation(type.type, config);
740                         break;
741                 case TABLE_DRIVEN:
742                         ret = create_table_driven_reservation(config);
743                         break;
744                 case SLOT_SHIFTING:
745                         ret = create_slot_shifting_reservation(&type, config);
746                         break;
747                 default:
748                         return -EINVAL;
749         };
750
751         return ret;
752 }
```

- Following function is an abstraction of syscall interface, here a new case statement called slot_shifting is added which in calls ***create_slot_shifting_reservation***.
- This function copies the config into the corresponding structure which is used for managing budget and adding the reservation to SUP environment.
    - In our Example let's make a struct slot_shift to which config is copied in function create_slot_shift_reservation, This record extends the simple reservation for our convenience.

```
/* kernel/include/polling_reservation.h OR a new file of convenience*/
struct slot_shift {
       struct reservation res;

       lt_t reservation_budget;
       lt_t slot_length;
};
```

- On This syscall a new reservation is created.

- **STEP4:** Creating a New Reservation in SUP environment
  - The function create_slot_shifting_reservation majorly should do 2 things.
    i. Creating reservation and adding it to SUP environment.
    ii. initialising reservation variables to manage the reservation.

```c
/*litmus/sched_pres.c or in a new file,
* This is completely left with user
*/
static long create_slot_shifting_reservation(
        struct res_data *type,
        struct reservation_config *config)
{
        struct slot_shift *slotShift = NULL;
                    .
                    .
                    .
        state = cpu_state_for(config->cpu);

        raw_spin_lock_irqsave(&state->lock, flags);
        /*check if res already exists to throw back or handle different
         * scenario on existence.
         */
        res = sup_find_by_id(&state->sup_env, config->id);
        raw_spin_unlock_irqrestore(&state->lock, flags);
      /* if !res already exists*/
      if(!res) {
                slotShift = kzalloc(sizeof(*slotShift), GFP_KERNEL);

                /*1. copy all config data to the corresponding structure.
                */
                 slotshift->reservation_budget =
                              config->slot_shift_params.reservation_length;
                 slotshift->slot_length =
                               config->slot_shift_params.slot_length;
               /*
                * 2. init the reservation and add the needed data that
                * reservation needs to make things work.
                */
                reservation_init(slotshift->res);
                slotshift->res.id = config->id;
                slotshift->res.priority = config->priority;

               /* 3. Add reservation specific ops*/
               slotshift->res.ops = &slotShift_ops;
               /*
                * 4. Add the reservation to the SUP environment
                */
               sup_add_new_reservation(&state->sup_env,  &(slotshift->res));
              /*
               * 5. Free the config previously allocated.
               * please Note: This is an example code, very simplified,
               * This could be adapted in a different way as well wout
               * freeing config, rather directly adapting it into kernel.
               */
               kfree(config);
        /*
         * 6. Finally initialize decision function, of per slot
         * Execution.
         * This is very specific to Slot shift and will spoil the
         * Generalized view of reservations so not adding the snippet.
        */

     }
     else {
         return -EEXIST;
```

```
    }
}
```

- **STEP 5:** Filling the reservation OPS:
  - Either in file /litmus/polling_reservation.c or in a new file define struct reservation_ops.
  - As mentioned in section 3.4 fill the function pointers as per the need.
  - This struct should be part of `struct reservation` and should be assigned during creation phase as mentioned in Step4 snippet.
  - Example snippet:

```
/* /litmus/polling_reservation.c  OR a new file */
/* Please note below Example is very specific to slot shifting
*  reservation and is just a example ,
*  This could very much change for another reservation implementation
*/
static struct reservation_ops slotshift_ops = {
         .dispatch_client = slotShift_dispatch_client,
         .client_arrives = slotShift_client_arrives,
         .client_departs = slotShift_client_departs,
         .replenish = slotShift_on_replenishment,
         .drain_budget = slotShift_drain_budget,
 };

static struct task_struct* slotShift_dispatch_client(
        struct reservation *res,
        lt_t *for_at_most)
{
    1. Update Slot progress.
    2. update interval progress.
    3. check on aperiodic in unconcluded list if so run acceptance and guarantee
       algorithm
    4. select next task on EDF from ready queue.
}


static void slotShift_client_arrives(
        struct reservation* res,
        struct reservation_client *client
)
{
       /*
       * 1. check the type of task and act accordingly.
       * 2.check state of reservation and change state if needed.
       */
       struct slot_shift *ss =
              container_of(res, struct slot_shift, res);
       struct task_client *tc =
               container_of(client, struct task_client, client);
       struct task_struct *t = tc->task;
       ss_task_struct *tsk = tsk_rt(t)->slot_shift_task;

       if(tsk->freshly_arrived) {  /*This is added as example and is
                                    * abstract from reservation
                                    * implementation
```

```
                                    */

                if((tsk->type != PERIODIC) {
                      1.if Firm aperiodic move to unconcluded list
                      2. else if soft aperiodic move to not guaranteed list
                }
                else {
                     move to guaranteed list.
                }
                tsk->freshly_arrived = 0;
        }

        switch (res->state) {

                 case RESERVATION_INACTIVE:
                        /*Simply move the reservation to depleted */
                        res->env->change_state(res->env, res,
                                RESERVATION_DEPLETED);
                        break;

                case RESERVATION_ACTIVE:
                case RESERVATION_DEPLETED:
                        /* do nothing */
                        break;

                case RESERVATION_ACTIVE_IDLE:
                        res->env->change_state(res->env, res,
                                RESERVATION_ACTIVE);
                        break;
        }
}

static void slotShift_client_departs(
         struct reservation *res,
         struct reservation_client *client,
         int did_signal_job_completion
)
{
   /*
    * 1.if needed remove from reservation list.
    * 2.change the state of the reservation if needed.
    * 3. Make necessary changes on internal task state if needed.
    */
      list_del(&client->list);

        switch (res->state) {
                case RESERVATION_INACTIVE:
                case RESERVATION_ACTIVE_IDLE:
                        BUG(); /* INACTIVE or IDLE <=> no client */
                        break;

                case RESERVATION_ACTIVE:
                        if (list_empty(&res->clients)) {
                                res->env->change_state(res->env, res,
                                                RESERVATION_ACTIVE_IDLE);
                        } /* else: nothing to do, more clients ready */
                        break;

                case RESERVATION_DEPLETED:
                        /* do nothing */
```

```
                        break;
        }

}

static void slotShift_on_replenishment(
         struct reservation *res
)
{
     /*
      *  slot shifting has no need for replenishment as we run
      *  continuously and once depleted it moves to inactive.
      */
        struct slot_shift *ss =
                container_of(res, struct slot_shift, res);

        switch(res->state) {
                case RESERVATION_DEPLETED:
                case RESERVATION_INACTIVE:
                        res->env->change_state(res->env, res,
                                               RESERVATION_ACTIVE);
                        break;
        }

}

static void slotShift_drain_budget(
                 struct reservation *res,
                 lt_t how_much)
{
          struct slot_shift *ss = container_of(res, struct slot_shift);
          ss->reservation_budget -= how_much;

          switch (res->state) {
                case RESERVATION_DEPLETED:
                case RESERVATION_INACTIVE:
                        BUG();
                        break;

                case RESERVATION_ACTIVE_IDLE:
                case RESERVATION_ACTIVE:
                        ss->reservation_budget -= how_much;
                        if(!ss->reservation_budget) {
                                res->env->change_state(res->env, res,
                                        RESERVATION_INACTIVE);
                        }
        }
}
```

- STEP6: **USER SPACE:**
    - within liblitmus/bin/resctrl.c==>main()==> case 't': add new reservation string decode, i.e. convert string to corresponding type defined in enum reservation_type_t. If needed add additional parameters that would help you configure your specific reservation.

```
/* liblitmus/bin/resctrl.c */
int main(int argc, char** argv)
{
        int ret, opt;
        char* parsed;
              .
              .
              .
       while ((opt = getopt(argc, argv, OPTSTR)) != -1) {

              switch (opt) {
                    .
                    .
                    .
                   case 't' :
                      if (strcmp(optarg, "polling-periodic") == 0) {
                                res_type = PERIODIC_POLLING;
                      } else if (strcmp(optarg, "polling-sporadic") == 0) {
                                res_type = SPORADIC_POLLING;
                      }  else if (strcmp(optarg, "table-driven") == 0) {
                           res_type = TABLE_DRIVEN;
                           /* Default for table-driven reservations to
                            * maximum priority. EDF has not meaning for
                            * table-driven reservations. */
                            if (config.priority == LITMUS_NO_PRIORITY)
                                        config.priority =
                                                LITMUS_HIGHEST_PRIORITY;
                      }
                      else if(0 == strcmp(optarg, "slot-shift")) {
                          res_type = SLOT_SHIFTING;
                      }
                      else {
                                usage("Unknown reservation type.");
                      }
                      break;

                    .
                    .
                    .

}
```

- Add Additional Parameters if needed based on the requirements.
- The walk through main() will give an idea adding additional config and is straightforward.