OS Project 2

第 51 組
組員：b02902015 梁智泓、b02902019 陳柏均

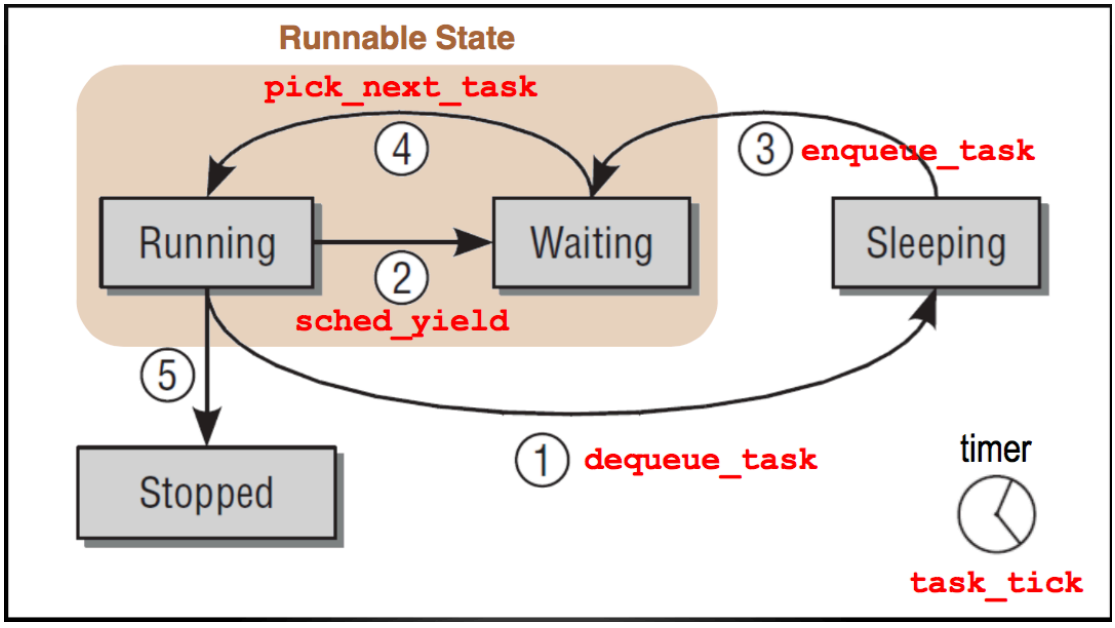
Implementation details：

```
static void enqueue_task_simple_rr(struct rq *rq, struct task_struct *p, int
wakeup, bool b)
{
	update_curr_simple_rr(rq);
	list_add_tail(&(p->simple_rr_list_item) , &((rq->simple_rr).queue) ) ;
	rq->simple_rr.nr_running++ ;
	先更新 running queue,把傳進來的 task 加進 running queue 的最後面
	更新 running task number
}
static void dequeue_task_simple_rr(struct rq *rq, struct task_struct *p, int
sleep)
{
	update_curr_simple_rr(rq);
	list_del(&(p->simple_rr_list_item) ) ;
	rq->simple_rr.nr_running-- ;
	先更新 running queue,把要踢掉的 task 從 queue 中移除
	更新 running task number
}
static void yield_task_simple_rr(struct rq *rq) {
	list_move_tail(&(rq->curr->simple_rr_list_item) ,
&rq->simple_rr.queue ) ;
	將 current  在跑的 task 放到 running queue 的最尾端
}
static struct task_struct *pick_next_task_simple_rr(struct rq *rq)
{
	if(list_empty(&rq->simple_rr.queue)){
		return NULL;
	}else{
		struct task_struct *p = list_first_entry(&rq->simple_rr.queue , struct
task_struct , simple_rr_list_item) ;
		p->se.exec_start = rq->clock;
```

```
        return p ;
    }
```
如果沒有任何 task 在 queue 裡則回傳空指針
否則則回傳用 list_first_entry 裡得到的 queue 中第一個 task

```
}
static void task_tick_simple_rr(struct rq *rq, struct task_struct *p,int queued)
{
    update_curr_simple_rr(rq);
    if(simple_rr_time_slice <= 0)
        return ;
    if ( p->task_time_slice > 0 ){
        p->task_time_slice-- ;
    }else if(p->task_time_slice == 0){
        p->task_time_slice = simple_rr_time_slice ;
        set_tsk_need_resched(p) ;
        requeue_task_simple_rr(rq , p) ;
    }
```
如果 task 中 time_slice 大於零則-1
等於零則重新賦值並把該 task 移到 running queue 的最後面
set_tsk_need_resched 設置 TIF_NEED_RESCHED 重新計算 queue 的優先級及 time_slice
```
}
```

yield_task_simple_rr 和 dequeue_task_simple_rr 有幾個基本上的差異
yield 是 time slice expired,所以會放回 running queue 的尾端等待下一次的執行
dequeue_task 則是因為 task 要等待 IO 或是其他 task 到達同樣的完成度才能繼續跑下去,或是任何 kernel 認定該 task 需要等待.所以會放到 sleeping section.



上述的圖其實就清楚的表示了一個 process 從開始到結束的過程，一開始 process 如果需要等待就會在 sleeping 的狀態，直到被某一個 signal 給叫醒，透過 enqueue_task 進入到 waiting 的狀態，此時的 process 會處於 runnable 的狀態，只是還沒分到ＣＰＵ而已，process 會一直等到他能夠使用到ＣＰＵ時才會真的開始 run，而一個 process 要用什麼規則在 waiting 的 queue 裡排隊等待，就是這次作業在做的事情（scheduler），當 process 排到了 running queue 的頭 call pick_next_task 進入ＣＰＵrun，該 process 會 time_slice 時間內被 requeue 回 waiting queue 的尾巴，反覆直到該 process 完成或著遇到 error 而中斷進到 stopped status

心得:

這次寫 project 剛開始的時候遇到很大的認知問題,雖然 hint 其實都寫得很清楚了,但因為在使用函式如 list_add_tail 的時候其實並不確定內部是否有幫我維護 running queue,因為隨便多一個新的 task 感覺都會需要 list_add_tail,函式也沒說會不會幫你動到 queue,很怕 list_add_tail 後 rq 的頭也跟著指到 queue 的最尾端,一直在想是否我需要再 call 一個函式才會把這個動作完成.再加上平常動到的 linked list 都是直接指向 structure,當 pointer 被包在 structure 裡面時很懷疑這樣的 implementation 要怎麼再使用 structure 裡的其他東西.
最後也是問了已經寫好的同學才發現這些問題其實不用顧慮到.
裡面有些 function 或是變數看了滿多文章才知道意思的,做完 project 覺得學到不少,但是感覺還是有點一知半解的 feeling,可能要看完 sched.c7000 多行才能融會貫通吧 XD