Name: RUSHABH SHAH

Section: C

University ID: 728236509

Lab 5b Report

Lab Questions:

2.1:

2pts What does the *#define* on line 26 do?

It is a constant defined. And it needs to be updated for every patch or release. In the c file, the scheduler version is shown as 1.3.
Before compilation, compiler will replace that variable with its string value, wherever it is applicable

2pts Which lines are included and which lines are excluded due to the *#if* on line 43, assuming we are using kernel 2.4.06?

Lines outside of if – endif are *not* mentioned below, since they are included regardless the if conditions.
Included lines :
44 – 45, 64 – 100.
Excluded lines:
47 – 48, 102 – 137.

2pts What is the difference between the two? (Hint: describe the type of each declaration)

Line 50: This declaration is of an pointer to an array of sched_policy with NR_CPU elements in the array.
Line 51: This declaration is of a single sched_policy named round_robin.

2.3:

2pt What does the following code do (line 158)?

Line 158: This line creates a new pointer of type sched_policy and assigns to it the address of round_robin variable.

4pts What is the purpose of lines 161 and 162?

Line 161 & 162: These two lines assigns the task and preemptability of the process.
sp_choosetask is called every tick in order to determine the next task to run on the particular cpu. sp_preemtibility, as the name suggests, is called to determine whether the process can be preempted by another process.

2pts Find the lines that initialize this array.

Lines 164 – 166. These lines initializes the array, with the same policy for all CPUs in this case.

2pts Which lines handle the unregistration of the module?

The function cleanup_module contains the unregister_sched(CTRR_SCHED_VERSION). This line handles the un-registration of the module.

2.4:

3pts Based on line 161, what must this function do? *ctrr_choose_task* takes two parameters: the currently running task, and the number of the CPU running the scheduler.

This function chooses which task should be run next. In the implementation, assuming the linux version is greater than 2.4, the function checks the current task's state, policy, counter and CPU. If the conditions are satisfied then it will keep the current task running. Otherwise it goes through the queue and find out who currently has control of CPU. Then the function performs the preempt test, in which the number of ticks is checked for current task. If there are still ticks left, the current_task is returned otherwise it is removed and appended to the end of the queue. And it will look for another task, If nothing is found then the current task or the idle is returned, whichever is appropriate.

4pts Describe lines 67-70 in words. Translate the syntax into plain English one line at a time, but keep it brief.

Line 67: Check the current task's status. All possible tasks are defined in task_struct. If status is TASK_RUNNING then check next condition.
Line 68: Check current task's policy. The sched_policy object from (current_task→policy & SCHED_YIELD) must *NOT* be true. If so then check the next condition.
Line 69: Check if current task is NOT the idel_task, as returned by idel_task(this_cpu). If so, check next condition.
Line 70: Check the counter of current task. If we still have ticks left on current task, then keep the current task. (i.e. keep = 1).

3pts On what line is the counter replenished? Does this happen for all tasks every time this algorithm is invoked?

On line 93, the counter is replenished. Yes, it occurs for each process in the queue listed by list_for_each.

2pts Study lines 97-139. In what case is the idle task returned?

If there is no process available that can run then check whether we can keep the current process. If keep is set to false, (i.e. current_task is an idle task) then return the idle task. Ignoring the pre 2.4 linux version, if function could not find a valid process to run, then by default it returns idle task.

3pts On line 142, what is the purpose of the function *ctrr_preemptability* (in words)?

This function takes in three parameters. The current_task, thief task and this_cpu. It returns the difference between thief counter and current_task counter. This function is later invoked inside the init_module function, where it is assigned to the new sched_policy pointer.

3:

3pts Find all of the lines in *pset.c* where goodness is used. Note any differences in the parameters passed.

'goodness' is used in lines 132 and 142. In line 132 the first parameter passed is of type task_struct. In line 142 the first parameter passed in is of the same type. The difference is that line 132 gets its parameter from function pset_choose_task's parameter. Where line 142 gets a local variable as the parameter.

2pts What is *IDLE_WEIGHT* defined as? Why? You will need to look in the include files to answer the first part, and the definition of goodness for the second part.(Refer to the patched sched.h file we provide.)

IDLE_WEIGHT is defined as (-1000). By default high is initialized as IDLE_WEIGHT. Then it is assigned the value returned by the goodness function. This will determine the right of a particular process to run.

3pts Where is *can_choose* defined? Where is *is_visible* defined? When answering, pick the correct lines based on the values of *CONFIG_SMP* we gave you.

can_choose is defined in pset.c file on line 97. Because *CONFIG_CMP* is not defined meaning it is false.

4pts What are the initial values of *high* and *choice*?

The initial value for high is IDLE_WEIGHT, which is (-1000) according to line 1925 of sched.h file. Initial value for choice is idle, where idle is the value returned by idle_task(this_cpu).

3pts Under what conditions is choice set to *curr_task*? (lines 130-135)

Choice is set to curr_task, if the high is >= 0. That is the value returned by function goodness in line 132 is >= 0.

5pts In one sentence, which task does the code on lines 137-148 pick?

Line 137 – 148 goes through the list and picks the task who has the highest right to run, measured by the goodness function in line 142.

2pts If there are tasks that are runnable but have no ticks left, then what is the value of *high*? (see line 151)

In that case the value of high must be 0.

2pts Where does line 158 goto?

Line 158 goes to line 126.

5pts Will lines 150-159 ever run twice in a row? Why or why not? In what two cases will it not run, and what task does the algorithm return in those two cases? Look again at lines 130-135.

It will not run in these 2 cases.
Case 1: After 1st execution of lines 150 – 159, assuming we had runnable with no ticks left, if high value of that one of the tasks is >= 0 as assigned by line 132. Then 150 – 159 will not execute twice in a row. Algorithm will return the most runnable task.
Case 2: If we don't have any runnable with no ticks left. Then it will not run again after its first execution. Algorithm will return idle task in this case.

2pts - Extra If this *"if"* is removed (the whole thing: lines 130-135), in what case will the task chosen be different? Note that the list of tasks is not sorted in any particular manner. Think of cases where some of the tasks have the same goodness, and the current task is closer to the end of the list.

If the if block from line 130- 135 was removed, Then the task chosen will be different for the Case 1 explained above. The process with low but positive goodness value might never be picked.

2pts - Extra Why include this *"if"*? Think in terms of penalties incorporated into goodness.

The process that have a positive value returned by goodness function, but not the highest of all are never going to be picked because the list_for_each macro's inner code suggests that it will pick the most 'runnable' task in the pset.