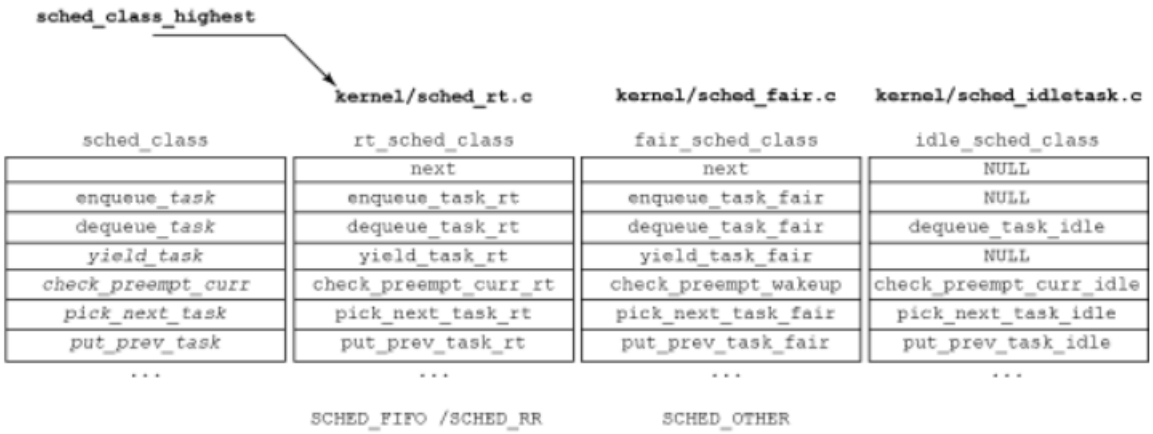
OPERATING SYSTEMS - SCHEDULING

SHUBHIKA GARG

(EURECOM)

For the Fair Share scheduling class drawn in Figure 3, basically explain the use of listed functions, both from the text related to this figure but also from what one could expect from those functions.

Figure 3. Graphical view of scheduling classes



- Each task is contained inside a scheduling class.
- The scheduling class determines how a task will be scheduled.
- It defines a common set of functions (via sched_class) that define the behaviour of the scheduler.
- Scheduling classes are implemented through the sched_class structure.

Functions:

enqueue_task()

This function is called when a task enters a "RUNNABLE" state.

It adds a task from the particular scheduling structures.

dequeue_task()

This function is called when a task is no longer in "RUNNABLE" state.

It removes a task from the particular scheduling structures.

check_preempt_curr()

This function checks if a task that entered the runnable state should pre-empt the currently running task.

pick_next_task(…)

This function chooses the most appropriate task eligible to run next.

set_curr_task(…)

This function is called when a task changes its scheduling class or changes its task group.

yield_task(…)

This function is just a dequeue followed by an enqueue, unless the compat_yield sysctl is turned on;

in that case, it places the scheduling entity at the right-most end of the red-black tree.

An instance of sched_class is given to each scheduling algorithm. Each scheduling algorithm connects the pointers of the function with their respective implementations.

rt_sched_class: It implements the real-time (RT) scheduler. The Real Time scheduler assigns a priority to each thread to schedule, and processes the threads in order of their priorities.

fair_sched_class: It implements the Completely Fair Schedular (CFS). The thread's priority is also assigned by the CFS.

The core logics of the kernel scheduler iterate over scheduler classes in order of their priority: rt_sched_class is processed prior to fair_sched_class.

Two scheduling policies of the RT Schedular:

SCHED_RR: The threads of SCHED_RR run one-by-one for a pre-defined time interval in their turn.

SCHED_FIFO: The threads of SCHED_FIFO run as first-in/first-out.

The main body of the core scheduler is the generic function schedule(). It puts the previously running thread into a run queue, then picks a new thread to run next, and at then, does context switching between the two threads.

Pick up at least two functions of previous function and provide a sequence of the main operations which are likely to be performed by these functions. Since the rb_tree structure is probably complex, just invent a basic data structure that is enough to explain these two functions.

The scheduler class is linked with each other in a linked list, allowing the classes to be iterated.

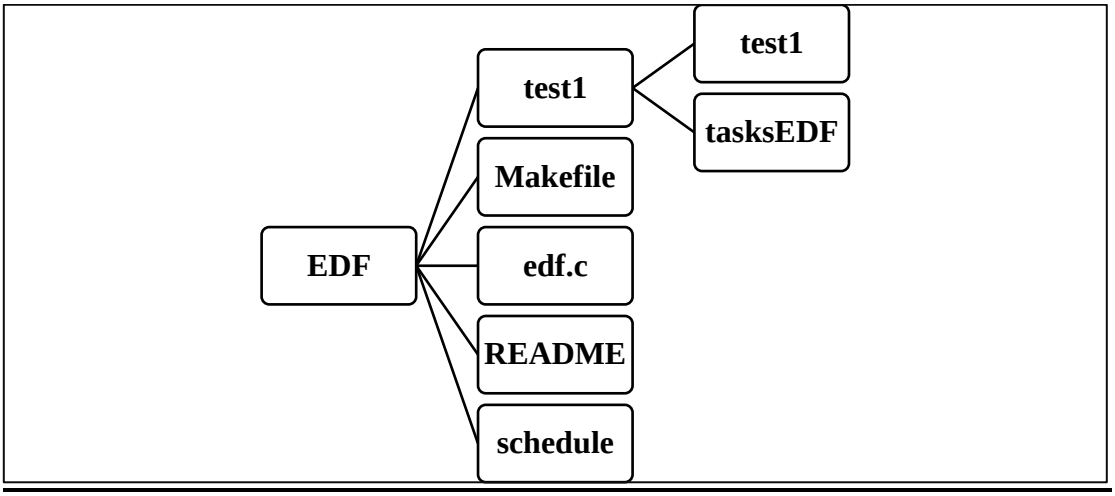
```
static void enqueue_task (struct t *t, struct tasks *p)
{
   p->curr-> enqueue_task (t, p);
}
static void pick_next_task (struct t *t, struct tasks *p)
{
   p->curr-> pick_next_task (t, p);
}
```

We can use a linked list to explain the two functions.

enqueue_task() to do insertion of tasks.

pick_next_task () to traverse through the linked list and pick the most appropriate task to be executed next.

SCHEDULING FOLDER:



FCFS (First-Come First-Served):

In this algorithm, the task that arrives first is executed first and the next task executes only after the previous task gets executed completely.

CODE DESCRIPTION:

To allocate memory to the name variable, we used malloc function.

This resolved the segmentation fault error we were encountering.

The malloc function will allocate the required memory and will return a pointer to it.

```
char *str=(char*)(malloc(sizeof(char)*MAX_TASK_NAME_SIZE));
tasks[nbOfTasks].name=str;
```

In FCFS, the task which arrives first is executed first.

We will implement a sorting function. The tasks are sorted based on their arrivalDate.

The function sorting_Tasks() accepts two arguments: the tasks and the number of tasks

and then sorts them in the order of their arrivalDate.

INPUT FILE FORMAT:

tasks

Task name; computationalTime; arrivalDate;

Example:

T1 2 10

EXPLANATION OF ONE CASE SCENARIO:

tasks:

T1 11 10

T2 12 2

T3 5 3

In the above scenario, we can see that T2 has arrived the earliest (at time 2).

Then comes T3 at time 3 and T1 arrived at time 10.

Initially, at time 0 and 1, there was no task to be scheduled.

At time 2, T2 arrives and starts executing till time 13 as its computationalTime is 12.

Meanwhile, T3 had arrived at time 3 and T1 at time 10. Once T2 finishes its execution,

as the arrivalDate(T3) <arrivalDate(T1)

T3 will start its execution from time 14 till time 18 (computationalTime=5).

Once T3 finishes, T1 starts its execution from time 19 till time 29 (computationalTime=11).

```
sg@DESKTOP-SVPVH04:~$ cd OS/LAB_Scheduling/FCFS/
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/FCFS$ ls
Makefile  README.md  fcfs.c  schedule  test1  test2
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/FCFS$ make
gcc -o schedule fcfs.c
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/FCFS$ make mytest2
running test2
./schedule test2/tasks2
Time 0: no task to schedule
Time 1: no task to schedule
Time 2: T2
Time 3: T2
Time 4: T2
Time 5: T2
Time 6: T2
Time 7: T2
Time 8: T2
Time 9: T2
Time 10: T2
Time 11: T2
Time 12: T2
Time 13: T2
Time 14: T3
Time 15: T3
Time 16: T3
Time 17: T3
Time 18: T3
Time 19: T1
Time 20: T1
Time 21: T1
Time 22: T1
Time 23: T1
Time 24: T1
Time 25: T1
Time 26: T1
Time 27: T1
Time 28: T1
Time 29: T1
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/FCFS$ cat test2/tasks2
T1 11 10
T2 12 2
T3 5 3sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/FCFS$
```

EDF (Earliest Deadline First) Algorithm:

In this algorithm, tasks are prioritized based on the deadline.

It is used in real-time operating systems where the tasks are placed in the priority queue.

CODE DESCRIPTION:

The tasks are assigned based on the order of deadlines. The task with deadline nearest in time is given the highest priority.

The tasks occur at regular time intervals and the tasks must complete within its deadline.

New tasks are admitted based on the current and arrival time.

For each task, we will check if the deadline is missed.

Also, we will check if the task is over or not. If yes, make it to state "RUNNABLE".

A warning message is displayed in case, a task cannot complete before its deadline.

We will check if the current running task terminates, if not we will make it RUNNABLE.

Calculate the least time to deadline out of all the tasks.

If the task is in RUNNABLE State, we compute the time to deadline value and then compare least time to deadline and time to deadline value.

In case, time to deadline value is lower than least time to deadline value, least time to deadline variable is initialized with the time to deadline value.

INPUT FILE FORMAT:

tasksEDF

Task name; computationalTime; arrivalTime; absoluteDeadline

Example:

T1 2 1 4

EXPLANATION OF ONE CASE SCENARIO:

tasksEDF:

T1 1 0 4

T2 2 0 6

T3 3 0 8

In the above case, we see all the three tasks T1, T2, T3 arrived at time 0.

The computationalTime for T1, T2, T3 is 1,2,3 respectively.

T1 has the lowest deadline.

It starts its execution from time 0 and ends at time 0 itself as its time of computations is 1.

Next lowest deadline is of T2. It starts its execution from time 1 till time 2 as its time of computations is 2.

Now, T3 starts its execution at time 3 as its deadline is the highest of all three tasks.

T3 has computationalTime of 3.

Now, as the absoluteDeadline for T1 is 4, it executes at time 4 and ends its execution.

Again, T3 starts its execution from time 5 to time 6.

```
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/EDF$ ls
Makefile  README.md  edf.c  schedule  test1  test2  test3
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/EDF$ cat test1/tasksEDF
T1 4 0 40
T2 5 3 10
T3 4 7 20sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/EDF$ make
gcc -o schedule edf.c
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/EDF$ make mytest1
running test1
./schedule test1/tasksEDF
Loading file of tasks
******** Time 0: T1

******** Time 1: T1

******** Time 2: T1

******** Time 3: T2

******** Time 4: T2

******** Time 5: T2

******** Time 6: T2

******** Time 7: T2

******** Time 8: T3

******** Time 9: T3

******** Time 10: T2

******** Time 11: T2

******** Time 12: T2

******** Time 13: T2

******** Time 14: T2

******** Time 15: T3

******** Time 16: T3

******** Time 17: T1

******** Time 18: no task to schedule
```

EDF (Earliest Deadline First) and FCFS (First-come and First-served) Implementation

CODE DESCRIPTION:

The real-time tasks are assigned based on the order of deadlines.

The task with deadline nearest in time is given the highest priority.

The tasks occur at regular time intervals and the tasks must complete within its deadline.

New tasks are admitted based on the current and arrival time.

For each task, we will check if the deadline is missed. Also, check if the task is over or not. If yes, make it to state RUNNABLE.

A warning message is displayed in case, a task cannot complete before its deadline.

We calculate the next deadline using:

next_deadline = arrivalTime + (Number_of_computations_performed + 1) * absolute_deadline

We will check if the current running task terminates, if not we will make it RUNNABLE.

Calculate the least time to deadline out of all the tasks.

If the task is in RUNNABLE State, we compute the time to deadline value and then compare

least time to deadline and time to deadline value.

In case, time to deadline value is lower than least time to deadline value,

least time to deadline variable is initialized with the time to deadline value.

We calculate the lowest time to deadline using:

lowest_time_to_deadline = (arrivalTime + (Number_of_executions + 1) * absolute_deadline) - current_time;

In case, two tasks have same deadline then according to FCFS, the task with earlier arrivalTime will be executed first.

INPUT FILE FORMAT:

tasksEDF_FCFS

Task name; computationalTime; arrivalTime; absolute deadline

Example:

T1 2 1 4

In case, we do not provide the deadline:

Example:

T1 2 0

T2 3 1 2

EXPLANATION OF ONE CASE SCENARIO:

T1 6 2

T2 4 4 18

T3 5 3 15

In the above case, we see all the three tasks T1, T2, T3 arrived at time 2,4,3 respectively.

The computationalTime for T1, T2, T3 is 6,4,5 respectively.

T1 has no deadlines while T2 and T3 have deadlines provided. T3 has lower deadline than T2.

At time 0 and 1, there is no task to be scheduled. At time 2, T1 starts its execution.

At time 3, T3 has arrived and an absolute deadline of 15 is provided. Now, the regular task is pre-empted and the real-time task executes. T3 executes from time 3 to 7 as its computationalTime is 5.

T2 will now execute from time 8 to time 11 as its computationalTime is 4.

Now, once T2 finishes its execution T1 starts executing from time 12 to 16.

```
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/EDF_FCFS$ ls
Makefile  README.md  edf_fcfs.c  schedule  test1  test2  test3  test4
sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/EDF_FCFS$ cat test1/tasksEDF_FCFS
T1 6 2
T2 4 4 18
T3 5 3 15sg@DESKTOP-SVPVH04:~/OS/LAB_Scheduling/EDF_FCFS$ make mytest1
running test1
./schedule test1/tasksEDF_FCFS
Loading file of tasks
******** Time 0: no task to schedule

******** Time 1: no task to schedule

******** Time 2: T1

******** Time 3: T3

******** Time 4: T3

******** Time 5: T3

******** Time 6: T3

******** Time 7: T3

******** Time 8: T2

******** Time 9: T2

******** Time 10: T2

******** Time 11: T2

******** Time 12: T1

******** Time 13: T1

******** Time 14: T1

******** Time 15: T1

******** Time 16: T1

******** Time 17: no task to schedule
```