OS Project 2
Scheduling in Linux

Advisor: Prof. Tei-Wei Kuo

TA: Han-Yi Lin

Outline

- Review: Life of Process
- Scheduling in Linux
- Implementation
  - Part I: Invoke FIFO Scheduler
  - Part II: Weighted Round Robin Scheduler
- Submission Rules
- References

Process Life Cycle

- A process is not always ready to run.
- The scheduler must know the status of every process in the system when switching between tasks.
- A process may have one of the following states:
  - **Running** – The process is executing at the moment.
  - **Waiting** – The process is able to run but is not allowed to because the CPU is allocated to another process. The scheduler can select the process at the next task switch.
  - **Sleeping** – The process is sleeping and cannot run because it is waiting for an external event. The scheduler cannot select the process at the next task switch.
- The system saves all processes in a process table.

The Need of the Scheduler

- A unique description of each process is held in memory and is linked with other processes by means of several structures.
- This is the situation facing the scheduler, whose task is to share CPU time between the programs to create the illusion of concurrent execution.
- This task is split into two different parts –
  - One relating to the scheduling policy and
  - The other to context switching

Outline

- Review: Life of Process
- Scheduling in Linux
- Implementation
  - Part I: Invoke FIFO Scheduler
  - Part II: Weighted Round Robin Scheduler
- Submission Rules
- References

Scheduling in Linux (1/2)

- The schedule function is the starting point to an understanding of scheduling operations.
- It is defined in “kernel/sched.c” and is one of the most frequently invoked functions in the kernel code.
- Not only priority scheduling but also two other soft real-time policies required by the POSIX standard are implemented.
  - E.g., completely fair scheduling, real-time scheduling and scheduling of the idle task, etc.

Scheduling in Linux (2/2)

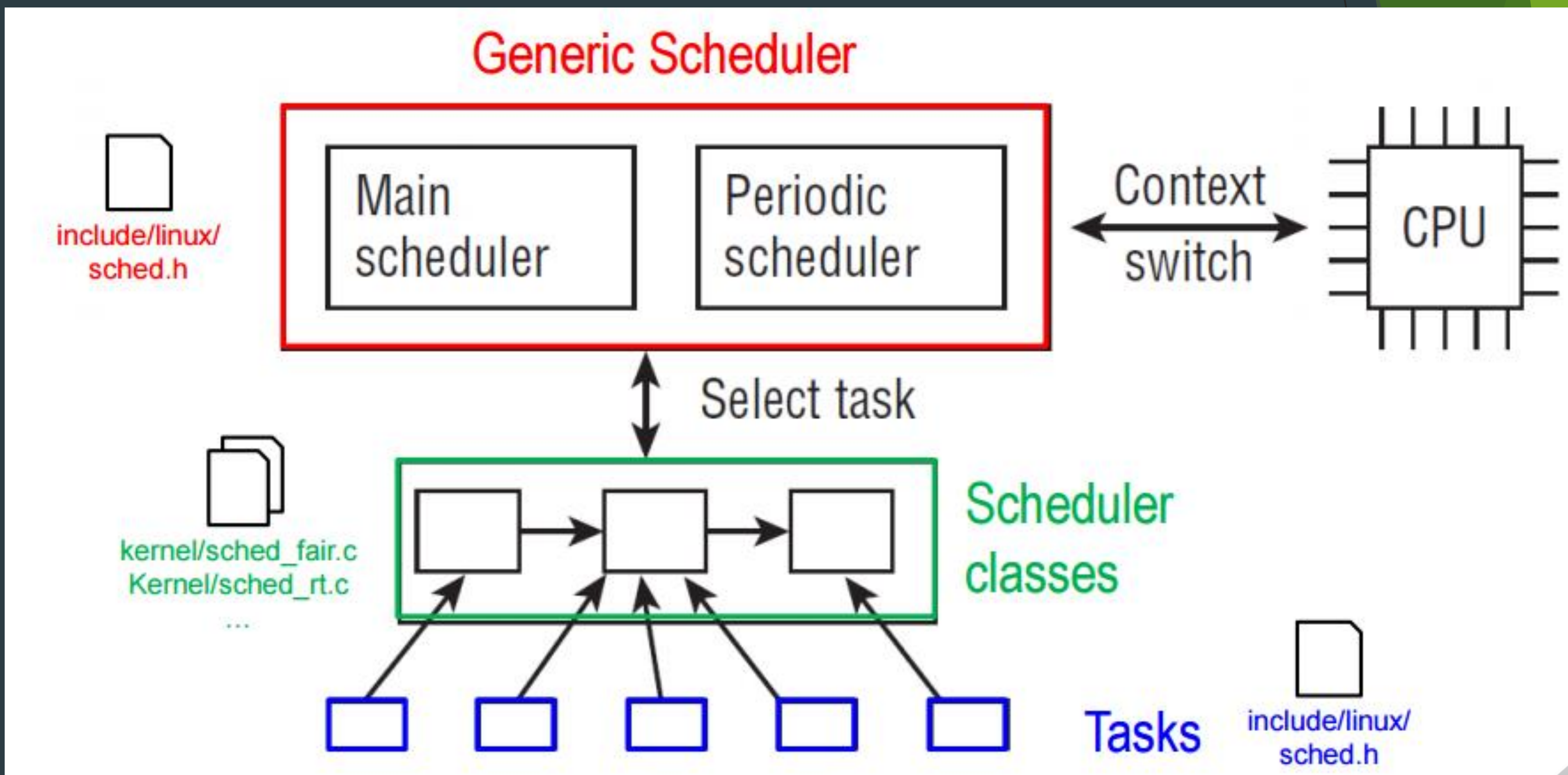
- The scheduler uses a series of data structures to sort and manage the processes in the system.
- Scheduling can be activated in two ways:
  - Main scheduler: Either directly if a task goes to sleep or wants to yield the CPU for other reasons,
  - Periodic scheduler: Or by a periodic mechanism that is run with constant frequency to check from time to time if switching tasks is necessary
- Generic scheduler = Main + Periodic schedulers

Overview of the Scheduling Subsystem in Linux

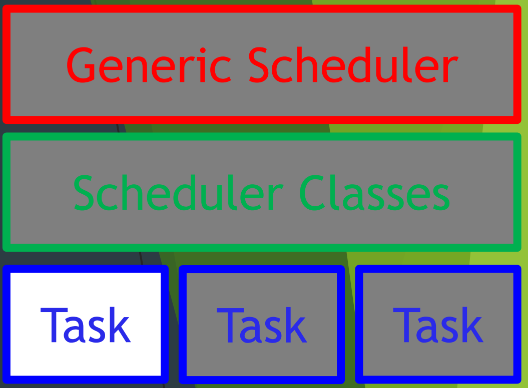
Generic Scheduler

Scheduler Classes

Task Task Task



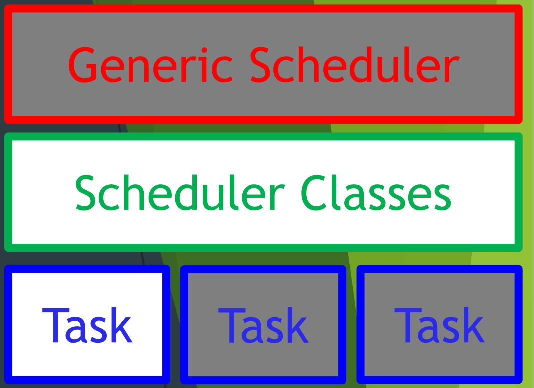
Task Representation



- In Linux, all concerned with processes and programs are built around a data structure: task_struct.

```
<sched.h>
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned long flags;    /* per process flags, defined below */
        unsigned long ptrace;
        int lock_depth;         /* BKL lock depth */

        int prio, static_prio, normal_prio;
        struct list_head run_list;
        const struct sched_class *sched_class;
        struct sched_entity se;

        ... see more in "include/linux/sched.h"
```

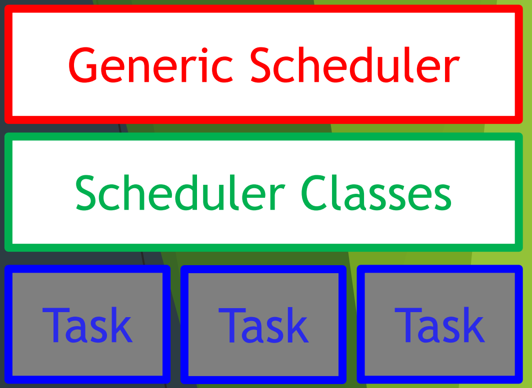
Policy Designation



<sched.h>

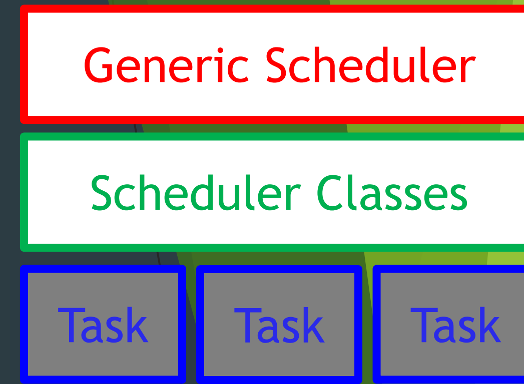
```
struct task_struct {
...
        int prio, static_prio, normal_prio;
        unsigned int rt_priority;

        struct list_head run_list;
        const struct sched_class *sched_class;
        struct sched_entity se;

        unsigned int policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice;
...
}
```

Scheduler Classes (1/3)



- Scheduler classes provide the connection between the generic scheduler and individual scheduling methods.
  - They are represented by several function pointers collected in a special data structure.
  - Each operation that can be requested by the global scheduler is represented by one pointer.
- This allows for creation of the generic scheduler without any knowledge about the internal working of different scheduler classes.

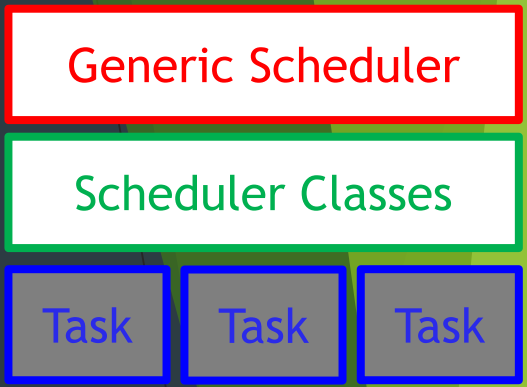
Scheduler Classes (2/3)



- An instance of struct sched_class must be provided for each scheduling class.

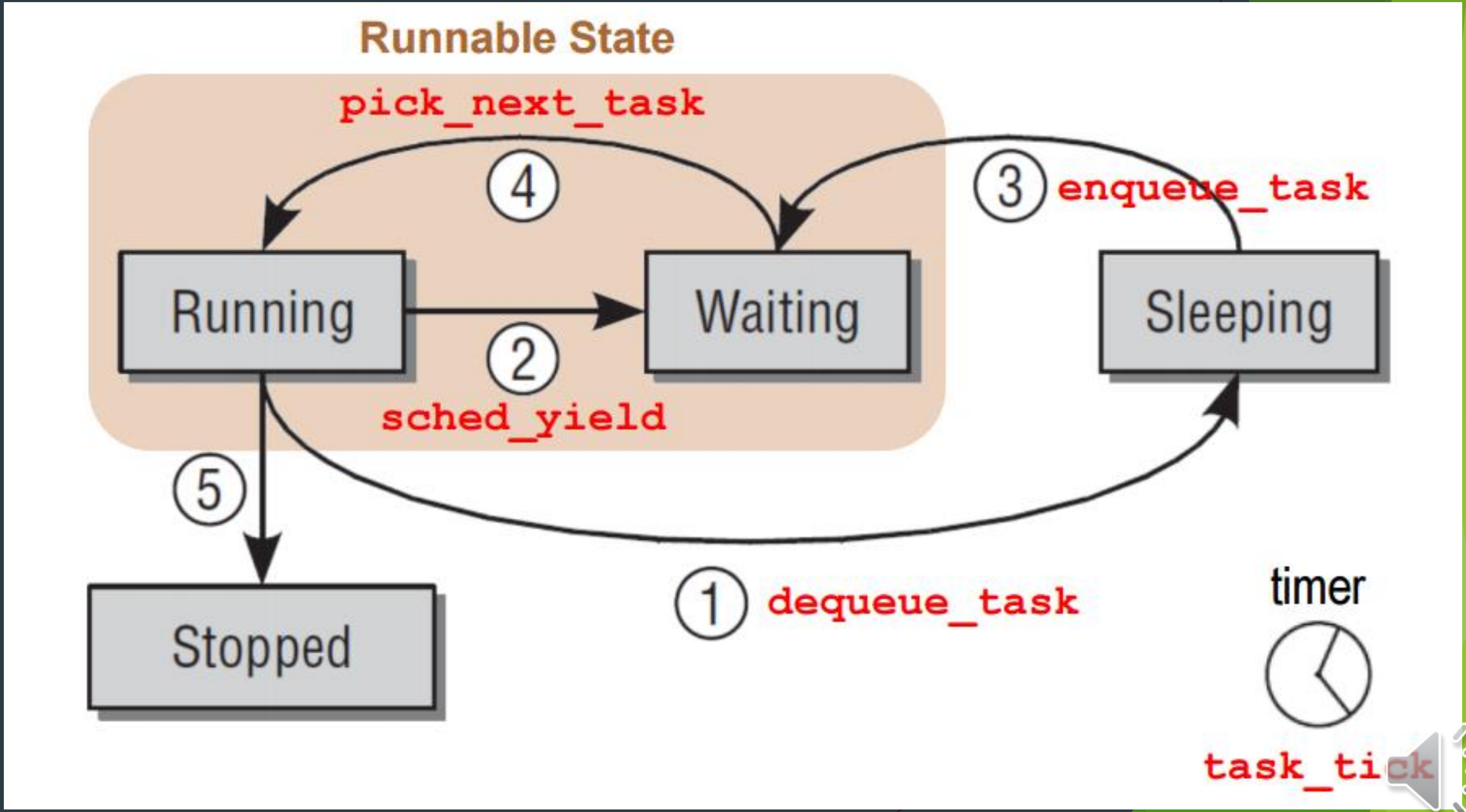
```
<sched.h>
struct sched_class {
        const struct sched_class *next;

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
        void (*yield_task) (struct rq *rq);

        void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);

        struct task_struct * (*pick_next_task) (struct rq *rq);
        void (*put_prev_task) (struct rq *rq, struct task_struct *p);
        void (*set_curr_task) (struct rq *rq);
        void (*task_tick) (struct rq *rq, struct task_struct *p);
        void (*task_new) (struct rq *rq, struct task_struct *p);
};
```

Scheduler Classes (3/3)



- enqueue_task: adds a new process to the run queue. This happens when a process changes from a sleeping into a runnable state.
- dequeue_task: provides the inverse operation: It takes a process off a run queue. Naturally, this happens when a process switches from a runnable into an un-runnable state, or when the kernel decides to take it off the run queue for other reasons.
- yield_task : when a process wants to relinquish control of the processor voluntarily, it can use the sched_yield system call. This triggers yield_task to be called in the kernel.
- pick_next_task: selects the next task that is supposed to run
- task_tick: is called by the periodic scheduler each time it is activated.

Relationships between Generics Functions and Process States



Outline

- Review: Life of Process
- Scheduling in Linux
- Implementation
  - Part I: Invoke FIFO Scheduler
  - Part II: Weighted Round Robin Scheduler
- Submission Rules
- References

Linux Scheduling Policies

- Linux Scheduling Policies
  - Normal Scheduling policies (Non-real-time)
    - SCHED_OTHER, SCHED_BATCH, SCHED_IDLE.
  - Real-Time policies
    - SCHED_FIFO, SCHED_RR.
- The default scheduling policy is non-real-time.
- In this part, using Linux real-time scheduling policy (FIFO) to schedule threads in a process.

Part I: Invoke FIFO Scheduler

- Write a C program (sched_test.c) to create two threads.
- Each thread will print who is running and busy for 0.5 second.
- Run the program by default time-sharing schedule policy and show the result.
  Ex. $ ./sched_test
- Run the program by real-time scheduling policy (FIFO) and show the result.
  Ex. $ ./sched_test SCHED_FIFO

```
int main(){
    set CPU affinity//all threads run on the same core
    invoke FIFO_SCHED

    for(i=0;i<2;i++)
        thread_create(i)
        print "Thread i was created"
    for(i=0;i<2;i++)
        thread_join(i)
}
thread_func(){
    for(i=0i<3;i++)
        print "Thread # is running"
        busy 0.5 second
}
```

Result

```
os2016@os2016-VM: ~
os2016@os2016-VM:~$ ./sched_test
Thread 1 was created.
Thread 2 was created.
Thread 2 is running.
Thread 1 is running.
Thread 1 is running.
Thread 2 is running.
Thread 1 is running.
Thread 2 is running.
os2016@os2016-VM:~$ sudo ./sched_test SCHED_FIFO
Thread 1 was created.
Thread 2 was created.
Thread 1 is running.
Thread 1 is running.
Thread 1 is running.
Thread 2 is running.
Thread 2 is running.
Thread 2 is running.
os2016@os2016-VM:~$
```

Hint for Part I

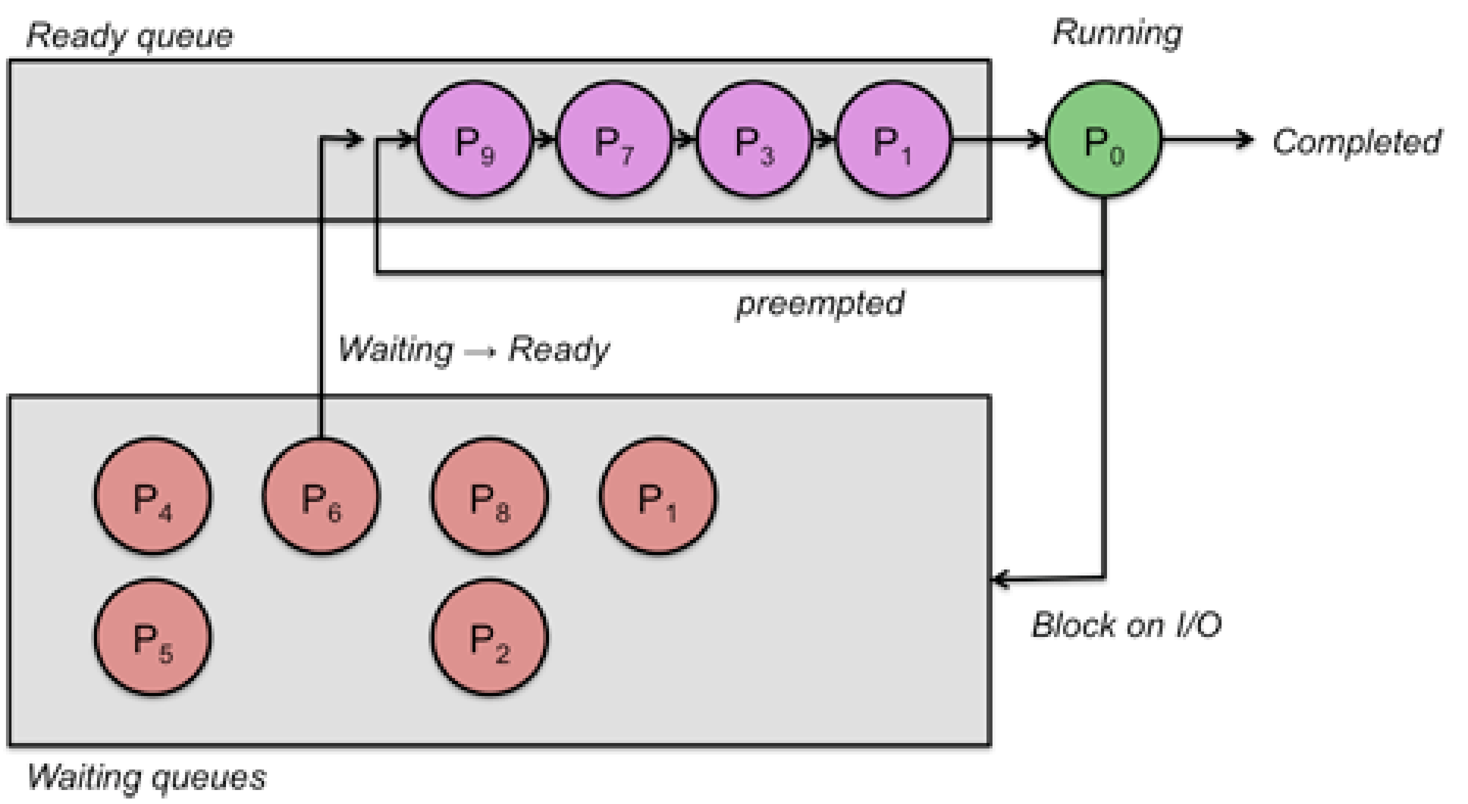
- Set CPU affinity
- sched_setscheduler();
- The policy corresponding value define in /include/linux/sched.h
- Set the priority of real-time process (sched_param *param)
- The permission to run real-time process

Outline

- Review: Life of Process
- Scheduling in Linux
- Implementation
  - Part I: Invoke FIFO Scheduler
  - Part II: Weighted Round Robin Scheduler
- Submission Rules
- References

Part II: Weighted Round Robin Scheduling (1/2)

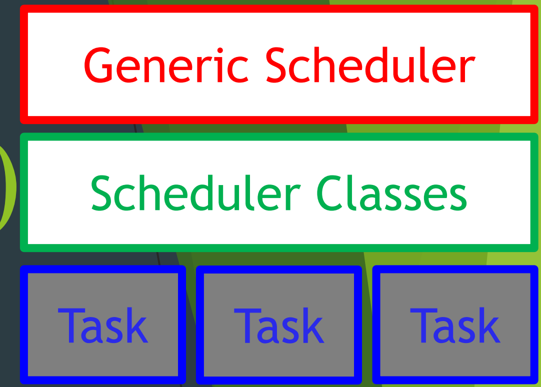
- Processes are dispatched in a FIFO sequence but each process is allowed to run for only a limited amount of time, a.k.a., time-slice or quantum.



Part II: Weighted Round Robin Scheduling (2/2)

- Implement kernel/sched_weighted_rr.c
  - enqueue_task_weighted_rr()
  - dequeue_task_weighted_rr()
  - yield_task_weighted_rr()
  - pick_next_task_weighted_rr()
  - task_tick_weighted_rr()
- Tasks with higher weights can finish their jobs earlier by having larger time slices.

How to add a custom scheduler into Linux?



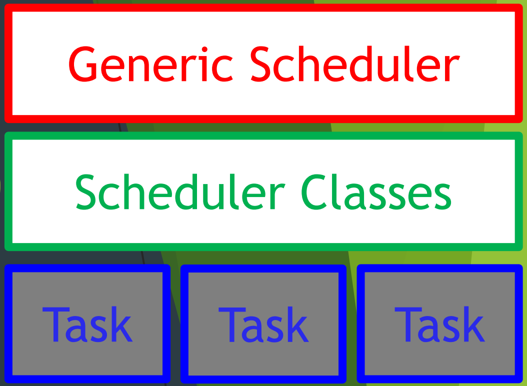
Generic Scheduler Side (1/3)

In “include/linux/sched.h”,

- Add #define SCHED_WEIGHTED_RR 6 – to define your weighted rr policy

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL        0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE      5
//+ OS Proj2: weighted_rr
#define SCHED_WEIGHTED_RR   6
```

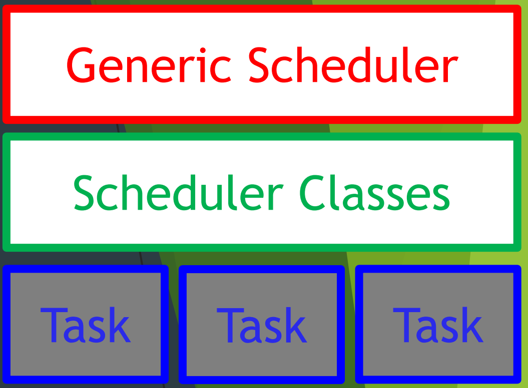
Generic Scheduler Side (2/3)



In “kernel/sched.c”

- Modify __setscheduler(), and __sched_setscheduler() functions – to let the generic scheduler can recognize your weighted rr scheduler

```
    //+ OS Proj2: weighted_rr
    case SCHED_WEIGHTED_RR:
        p->sched_class = &weighted_rr_sched_class;
        break;
    }
```



Generic Scheduler Side (3/3)

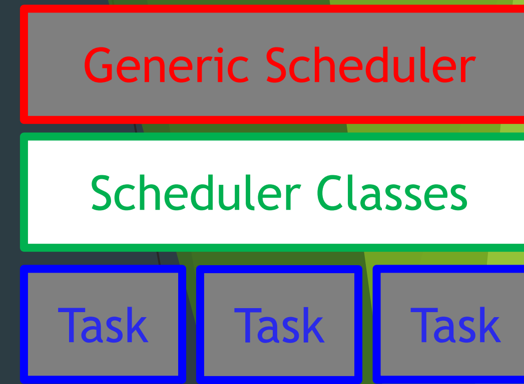
In “struct rq” of “kernel/sched.c”

- Add struct weighted_rr_rq weighted_rr – to specify the run queue for your weighted rr

```
struct rq {
        ...
        struct cfs_rq cfs;
        //+ OS Proj2: weighted_rr
        struct weighted_rr_rq weighted_rr;
        struct rt_rq rt;
        ...
```

Note that struct rq – the generic per-CPU run queue structure. However, this is NOT the queue structure you will work with. Rather, this structure contains a more specific run queue type for different scheduler classes.

Scheduler Classes Side (1/3)

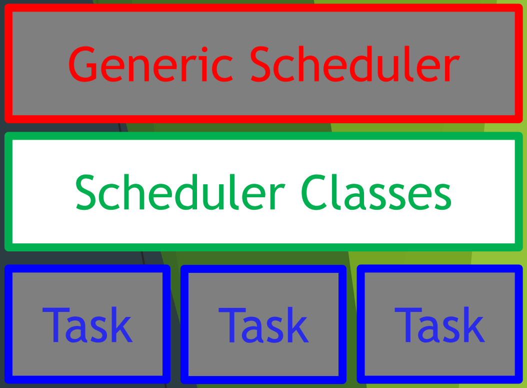


As well in “kernel/sched.c”

- Define weighted_rr_rq structure, which should contain
  - struct list_head queue - to denote the actual run queue for your weighted rr scheduler
  - unsigned long nr_running - to denote the number of processes which are now in the run queue

```
//+ OS Proj2: weighted_rr
struct weighted_rr_rq {
    struct list_head queue;
    unsigned long nr_running;
```

Scheduler Classes Side (2/3)

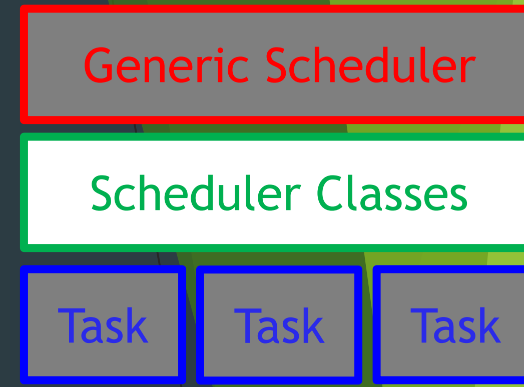


In “kernel/sched.c”,

- Declare int weighted_rr_time_slice – to define the time slice for your weighted rr scheduling policy

```
//+  OS  Proj2: weighted_rr
int weighted_rr_time_slice
```

```
//+ OS  Proj2: weighted_rr
SYSCALL_DEFINE1(sched_weighted_rr_setquantum, unsigned int, quantum)
{
    weighted_rr_time_slice = quantum;
    return;
}
```



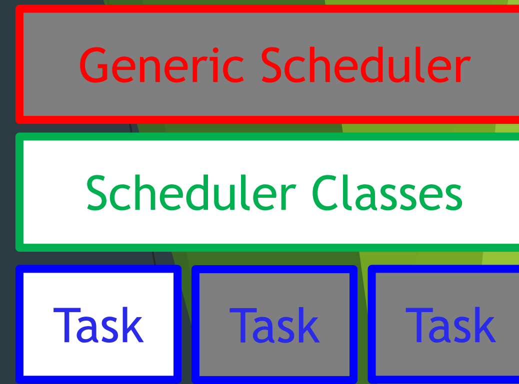
Scheduler Classes Side (3/3)

In “kernel/sched_weighted_rr.c”

- Accomplish the implementation of weighted rr scheduler
  - Recall that an instance of struct sched_class must be provided for each scheduling class.

```
const struct sched_class weighted_rr_sched_class = {
    .next               = &idle_sched_class,
    .enqueue_task           = enqueue_task_weighted_rr,
    .dequeue_task           = dequeue_task_weighted_rr,
    .yield_task         = yield_task_weighted_rr,

    .check_preempt_curr = check_preempt_curr_weighted_rr,

    .pick_next_task         = pick_next_task_weighted_rr,
    .put_prev_task          = put_prev_task_weighted_rr,
    ...
```

Task Side



In “struct task_struct” of “include/linux/sched.h”, add

- Declare unsigned int weighted_rr_task_time_slice – to denote the current time slice for this task
- Declare struct list_head weighted_rr_list_item – to denote the list item which will be inserted into the run queue of weighted_rr

```
struct task_struct {
    ...
    //+  OS Proj2: weighted_rr
    unsigned int task_time_slice;
    unsigned int weighted_time_slice;
    ...
        //+  OS Proj2: weighted_rr
    struct list_head weighted_rr_list_item;
```

“Lazy Package”

- The lazy package includes
  - http://newslab.csie.ntu.edu.tw/course/OS2016/files/project/linux-2.6.32.60.tar.gz
  - Six modified files (don’t modify, but read it)
    - include/linux/sched.h, kernel/sched.c, kernel/sched_fair.c, include/linux/syscalls.h, arch/x86/kernel/syscall_table_32.S, arch/x86/include/asm/unistd_32.h
- sched_weighted_rr.c (incomplete, your job!)

  enqueue_task_weighted_rr(), dequeue_task_weighted_rr()

  yield_task_weighted_rr(), pick_next_task_weighted_rr()

  task_tick_weighted_rr()

Testing Program

In linux-2.6.32.60\test_weighted_rr\test_weighted_rr.c

- The test program will first allocate a write buffer with size b.
- Then, the test program will create n user threads, each of which will write a unique character (e.g., a) into the buffer over and over.
  - Note that, every threads will write the same number of characters in to the buffer, based on the buffer size.
- Moreover, you can assign the scheduling policy, and the weighted_rr_time_slice t.

Note that, when dumpling the write buffer, the test program will aggregate the consecutive characters into one symbol.

Possible Results

- ./test_weighted_rr weighted_rr t 5 5000000

```
> ./test_weighted_rr weighted_rr 10 5 500000000
sched_policy: 6, quantum: 10, num_threads: 5, buffer_size: 500000000
abcdeabcdeabcdeabcdabcdabcabcabcabcababababababababa
```

'e' finish 'd' finish 'c' finish

test_weighted_rr.c

```
for (i = 0; i < num_threads; i++)
{
    ...
    syscall (SYS_weighted_rr_setquantum, quantum);
    pthread_create(&threads[i], &attr, run, (void *)targs);
    quantum*=2;
```

Scoring of Project 2

- Part I: Implementation of a program to invoke FIFO scheduler (30%)
- Part II: Implementation of the below FIVE incomplete functions in “sched_weighted_rr.c” (40%)

  enqueue_task_weighted_rr(), dequeue_task_weighted_rr()

  yield_task_weighted_rr(), pick_next_task_weighted_rr()

  task_tick_weighted_rr()
- Report (30%)
  - Your implementation details and results
  - At most 4 pages (6 pages if you have bonus)
- Bonus (at most 20%)
  - Any variation of the scheduling policy

Submission Rules

- Project deadline: 2016/05/18 23:59
- Upload to FTP Server
  - IP:Port : 140.112.28.132:5566
  - Account/Password: os2016/ktw2016os
- Be packed as one file named "OSPJ2_Team##.ZIP "

```
+---OSPJ2_Team##(directory)
    +---Report.pdf
    +---Part1(directory)
        +---sched_test.c
    +---Part2(directory)
        +---sched_weighted_rr.c
    +---Bonus(directory)
        +----your code and any modified files
```

- DO NOT COPY THE HOMEWORK

References

- Reference Book
  - Professional Linux® Kernel Architecture, Wolfgang Mauerer, Wiley Publishing, Inc.
- Linux Kernel: 強大又好用的 list_head 結構
  - http://adrianhuang.blogspot.tw/2007/10/linux-kernellisthead.html
- E-mail: d03922006@csie.ntu.edu.tw