OS Project 2
Process Scheduling in Linux

Advisor: Tei-Wei Kuo
Speaker: Chien-Chung Ho

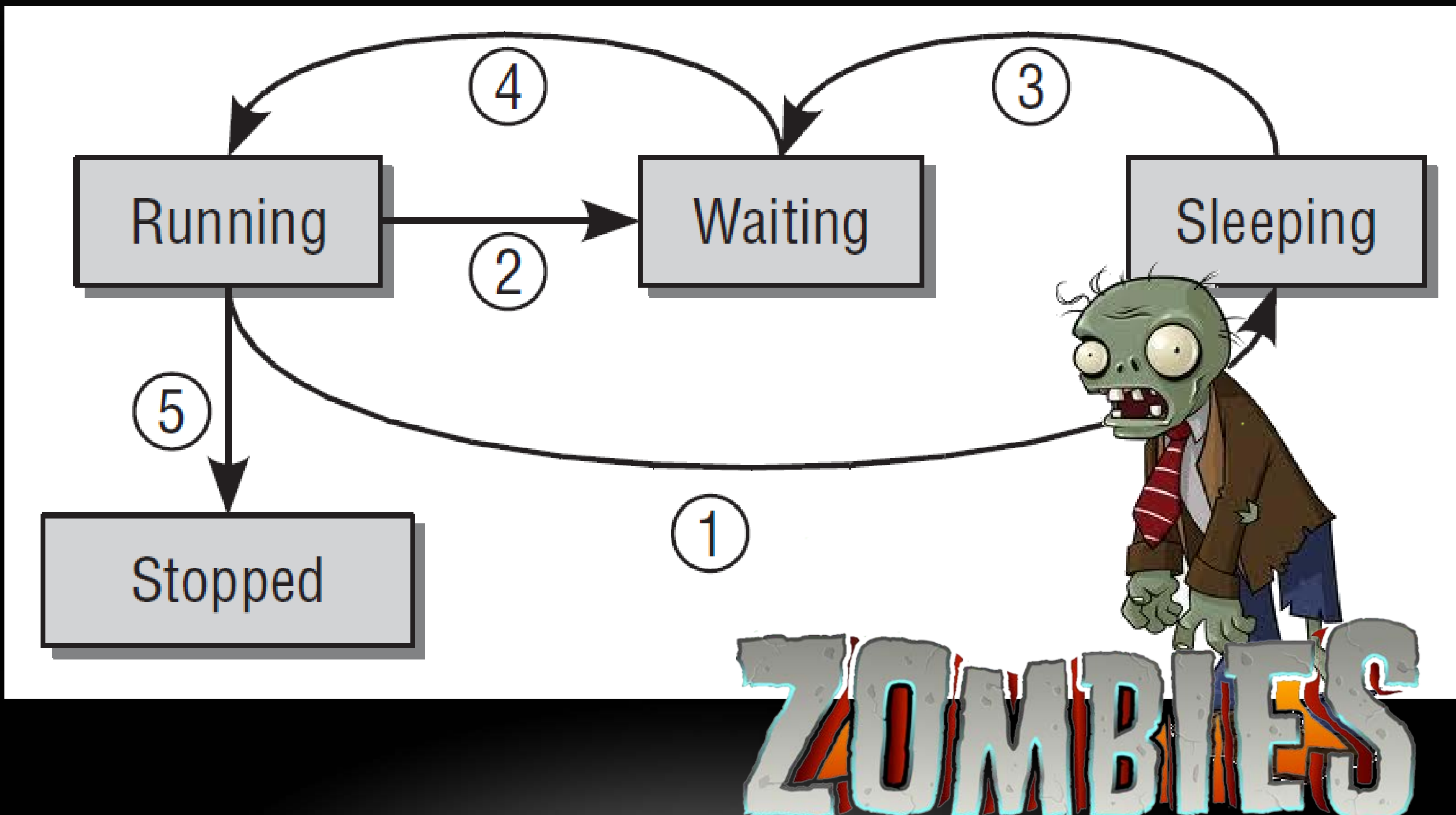
Outline

- Introduction
- Project Requirements
- 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.

Transitions between Process States



Example: Process Representation in Linux

- 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” in the kernel source

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

Scheduling in Linux (1/2)

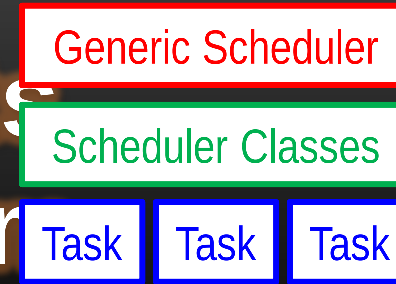
- The schedule function is the starting point to an understanding of scheduling operations.
- It is defined in “kernel/sched.c” and is one of the most frequently invoked functions in the kernel code.
- Not only *priority scheduling* but also two other soft real-time policies required by the POSIX standard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

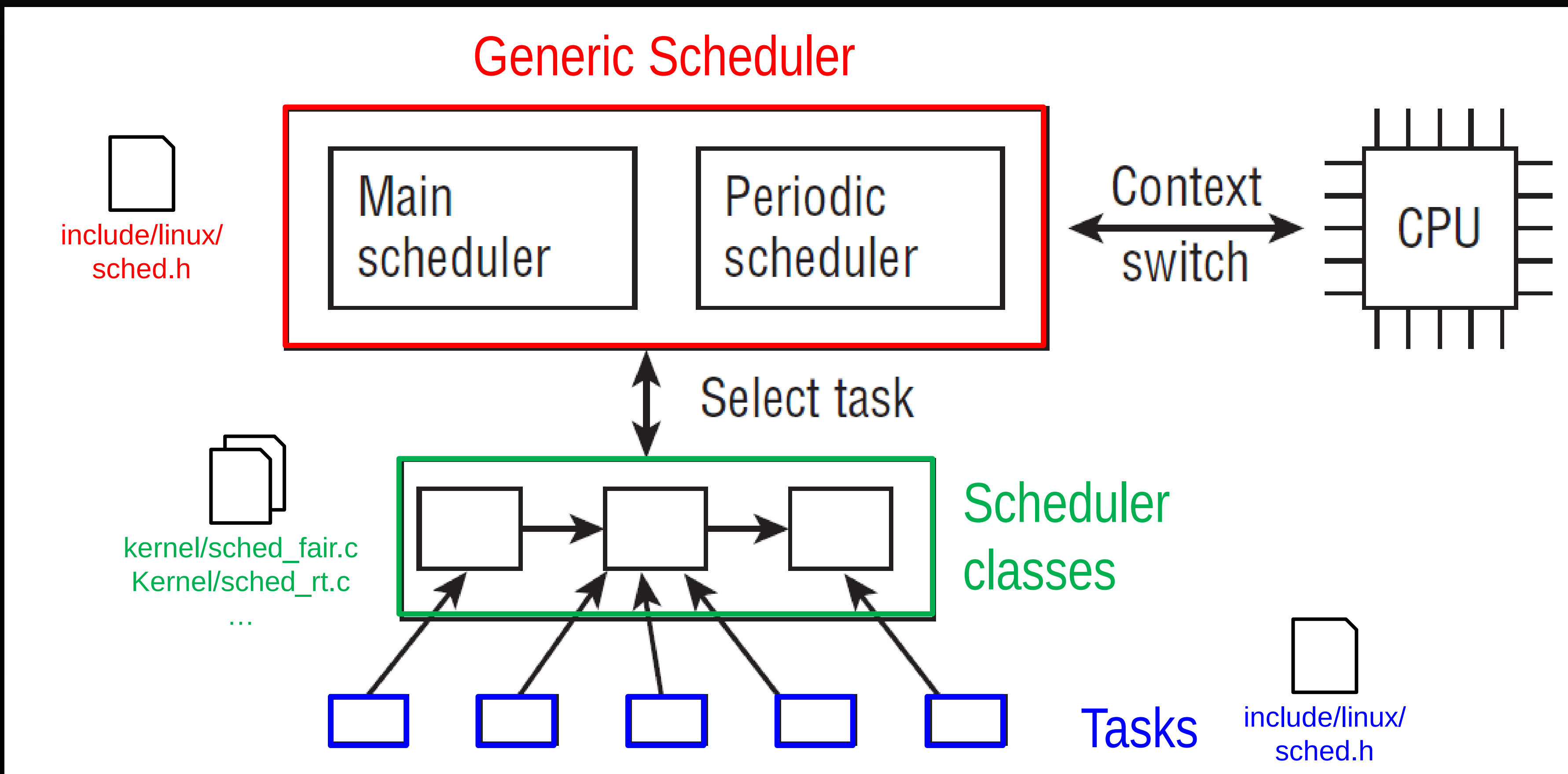
Scheduling in Linux (2/2)

- 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 Components of the Scheduling Subsystem





How to Designate a Scheduler for Tasks?

Generic Scheduler

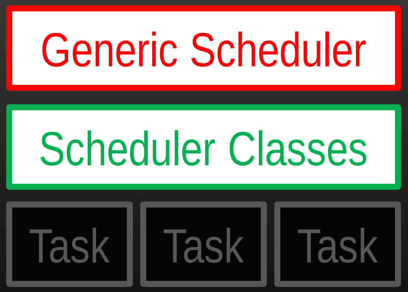
Scheduler Classes

Task | Task | Task

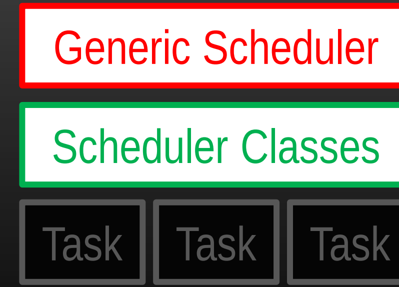
<sched.h>

```
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 (1/4)



- 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.

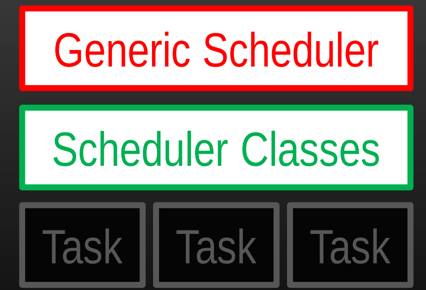


Scheduler Classes (2/4)

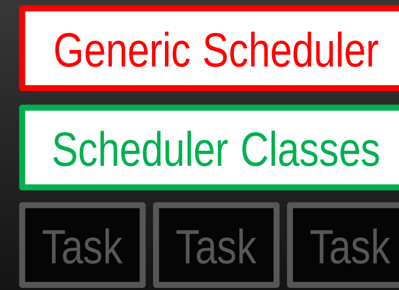
- An instance of **`struct sched_class`** must be provided for each scheduling class.

```
<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 (3/4)



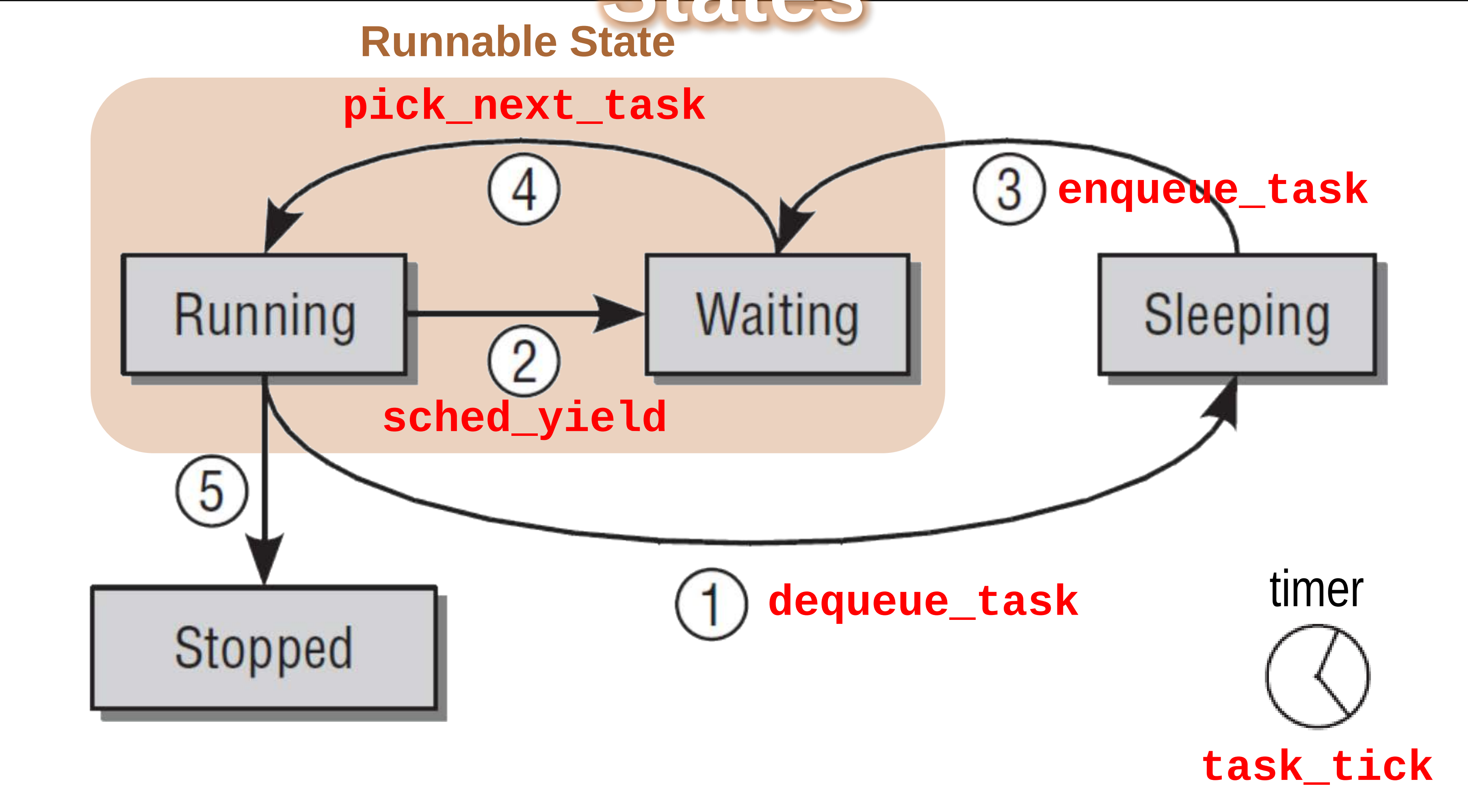
- **`enqueue_task`**: adds a new process to the run queue. This happens when a process changes from a sleeping into a runnable state.
- **`dequeue_task`**: provides the inverse operation: It takes a process off a run queue. Naturally, this happens when a process switches from a runnable into an un-runnable 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.
- **`check_preempt_curr`**: is used to preempt the current task with a newly woken task if this is necessary. The function is called, for instance, when a new task is woken up with wake_up_new_task.



Scheduler Classes (4/4)

- **pick_next_task**: selects the next task that is supposed to run
- **put_prev_task**: is called before the currently executing task is replaced with another one.
  - Note that these two operations are *not* equivalent to putting tasks on and off the run queue like **enqueue_task** and **dequeue_task**. Instead, they are responsible to give the CPU to a task, respectively, take it away. Switching between different tasks, however, still requires performing a low-level context switch.
- **set_curr_task**: is called when the scheduling policy of a task is changed. There are also some other places that call the function, but they are not relevant for our purposes.
- **task_tick**: is called by the periodic scheduler each time it is activated.
- **new_task**: allows for setting up a connection between the fork system call and the scheduler. Each time a new task is created, the scheduler is notified about this with **new_task**.

Relationships between Generics Functions and Process States



Outline

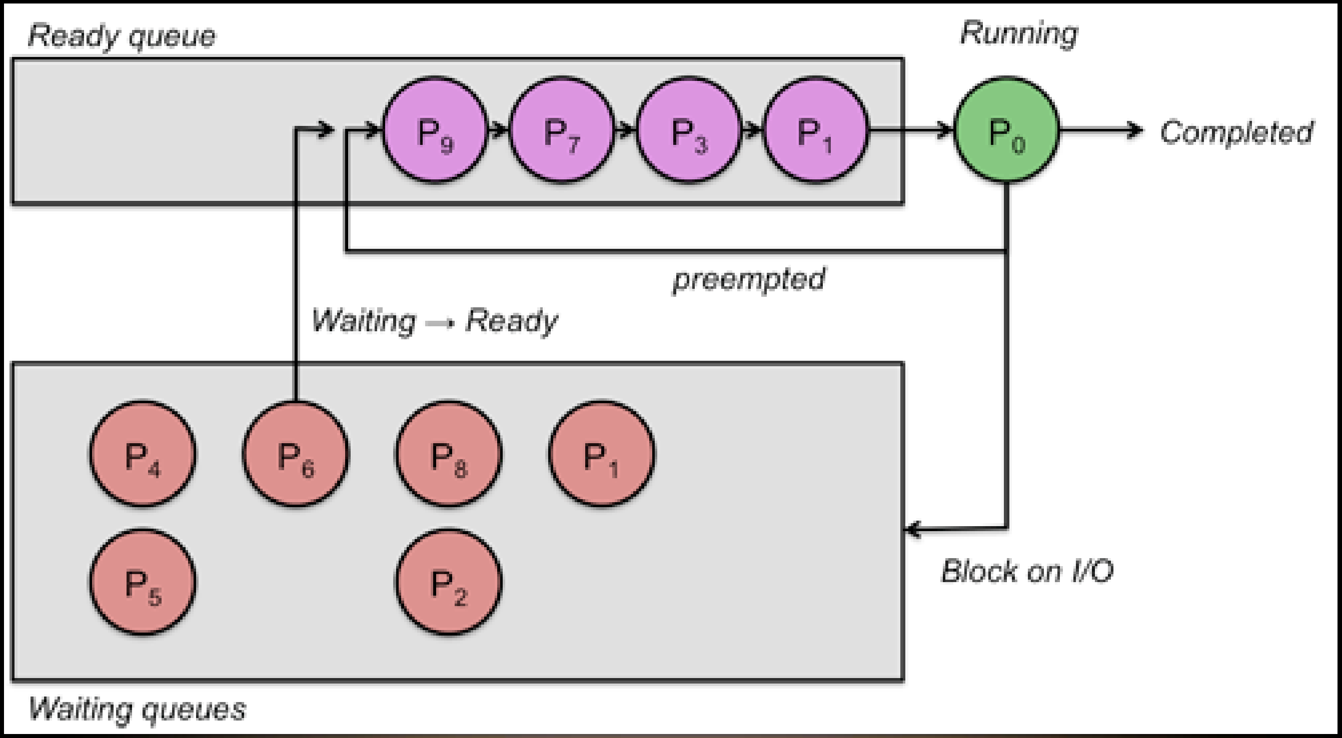
- Introduction
- Project Requirements
- Submission Rules
- References

Project Requirements

1) In this project, you should modify the Linux kernel source code (**2.6.32.60**) to implement a simple round robin scheduling policy.
2) Besides, you should add a system call, just like what you have done in project 1, to let user programs set the time_slice for the simple round robin scheduling policy.
3) Finally, you should use the provided test program to observe the behavior of your scheduler (and also to compare with the default schedulers).

Simple Round Robin Scheduling (Simple_RR)

- 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.



So, how to add a custom scheduler into Linux?

Note that I only list some major points in this slides. You can trace the provided source codes for more details.

Generic Scheduler

Scheduler Classes

Task Task Task

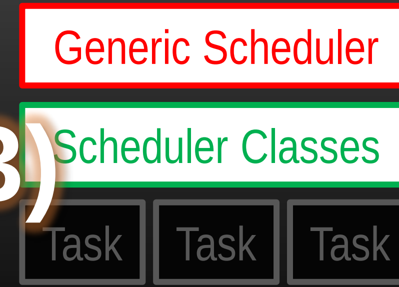
Generic Scheduler Side (1/3)

In “include/linux/sched.h”,

- Add #define SCHED_SIMPLE_RR 6 – to define your simple 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: simple_rr
#define SCHED_SIMPLE_RR   6
```

Generic Scheduler Side (2/3)



In “kernel/sched.c”,

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

```
//+ OS Proj2: simple_rr
if(p->prio == SCHED_SIMPLE_RR)
{
    printk("[OS PROJ2]: Simple RR scheduler has been set.");
    p->sched_class = &simple_rr_sched_class;
}
```

```
if (policy != SCHED_FIFO && policy != SCHED_RR &&
        policy != SCHED_NORMAL && policy != SCHED_BATCH &&
        //+ OS Proj2: simple_rr
        policy != SCHED_IDLE && policy != SCHED_SIMPLE_RR)
    return -EINVAL;
```

Generic Scheduler Side (3/3)

Generic Scheduler

Scheduler Classes

Task Task Task

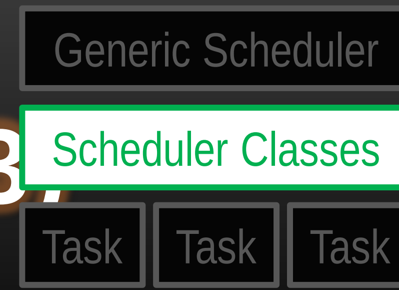
In “struct rq” of “kernel/sched.c”,

- Add struct simple_rr_rq simple_rr – to specify the run queue for your simple rr

```
struct rq {
    ...
    struct cfs_rq cfs;
    //+ OS Proj2: simple_rr
    struct simple_rr_rq simple_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)



In “kernel/sched.c”,

- Declare int simple_rr_time_slice – to define the time slice for your simple rr scheduling policy

```
//+ OS Proj2: simple_rr
#include "sched_simple_rr.c"
int simple_rr_time_slice = DEF_TIMESLICE;
```

Scheduler Classes Side (2/3)

As well in “kernel/sched.c”,

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

```
//+ OS Proj2: simple_rr
/* SIMPLE_RR classes' related field in a runqueue: */
struct simple_rr_rq {
    struct list_head queue;
    unsigned long nr_running;
};
```

Scheduler Classes Side (3/3)

In “kernel/sched_simple_rr.c”

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

```
const struct sched_class simple_rr_sched_class = {
    .next              = &idle_sched_class,
    .enqueue_task         = enqueue_task_simple_rr,
    .dequeue_task         = dequeue_task_simple_rr,
    .yield_task       = yield_task_simple_rr,

    .check_preempt_curr = check_preempt_curr_simple_rr,

    .pick_next_task       = pick_next_task_simple_rr,
    .put_prev_task        = put_prev_task_simple_rr,
    ...
```

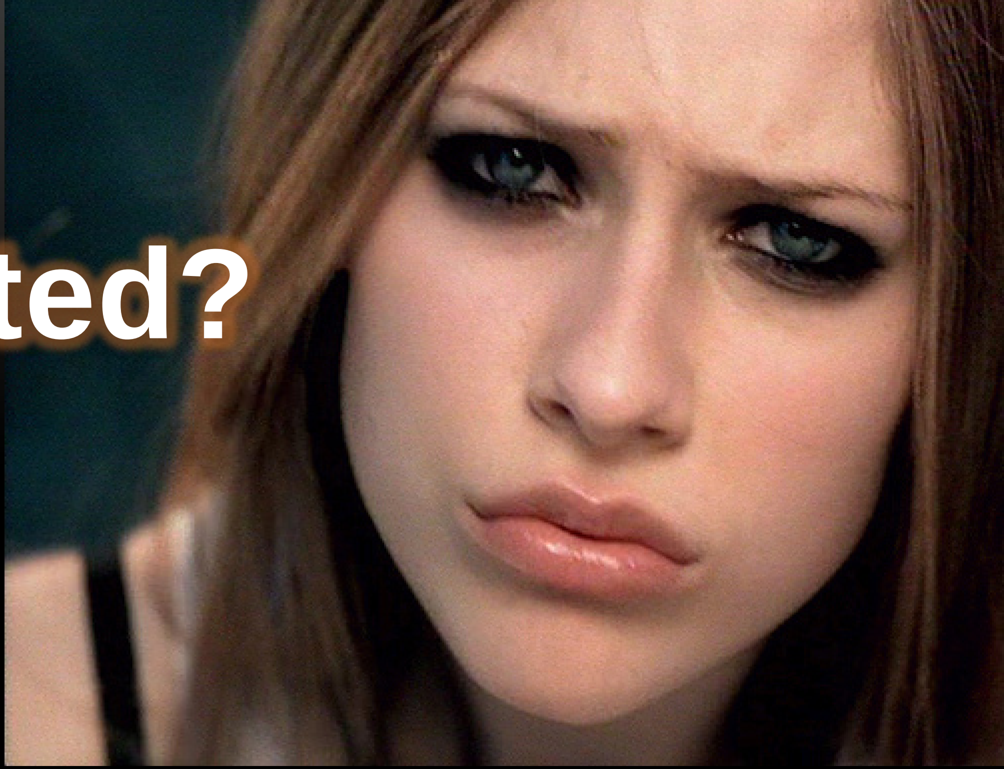
Task Side

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

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

```
struct task_struct {
    ...
    //+ OS Proj2: simple_rr
    struct list_head simple_rr_run_list;
    //+ OS Proj2: simple_rr
    unsigned int task_time_slice;
    ...
```

Too Complicated?



Why TAs have to go and make things so complicated?

Why TAs have to go and make things so complicated?

Why TAs have to go and make things so complicated?

Why TAs have to go and make things so complicated?

Why TAs have to go and make things so complicated?

We prepare two "Lazy Packages" for you…

- The first lazy package includes
  - http://newslab.csie.ntu.edu.tw/course/OS2015/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
  - One new added file
    - sched_simple_rr.c (incomplete, your job!)

We prepare two "Lazy Packages" for you...

- The second lazy package includes
  - http://newslab.csie.ntu.edu.tw/course/OS2015/files/test_simple_rr.zip
  - One test file (and its Makefile) to examine your simple rr scheduling policy
    - test_simple_rr.c
    - Makefile

Something about test_simple_rr.c

- The test program will first allocate a write buffer with size *bs.*
- Then, the test program will create *nt* 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 simple_rr_time_slice *ts*.

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

Possible Results

./test_simple_rr *scheduling_policy ts nt bs*

- ./test_simple_rr default *ts* 5 500000
  - dbcbceac… (frequently context-switch)

  Note that ts makes no effect on the default scheduling policy.
- ./test_simple_rr simple_rr *ts* 5 500000
  - Set *simple_rr_time_slice* as 0 (ts=0 means infinite)
  - abcde
  - Set small *simple_rr_time_slice* and large *bs*
  - abcdeabcdeabcde… (some tasks might finish early)

Note that the output results might not always be exactly the same.

Scoring of Project 2 (1/2)

- Implement the below FIVE incomplete functions in "simple_rr.c" (50%)
  - enqueue_task_simple_rr()
  - dequeue_task_simple_rr()
  - yield_task_simple_rr()
  - pick_next_task_simple_rr()
  - task_tick_simple_rr()

- Add a system call to let the test program can set different simple_rr_time_slice values (10%)

Scoring of Project 2 (2/2)

- Report (40%)
  - Your implementation details
  - At most 4 pages
- Bonus (at most 20%)
  - Any variation of the round-robin scheduling policy
    - E.g., priority-weighted round-robin, SJF, and so on.

Outline

- Introduction
- Project Requirements
- Submission Rules
- References

Submission Rules

- Project deadline: 2015/05/27 (Wednesday) 23:59
  - Delayed submissions yield severe point deduction
- Upload your team project to the FTP site.
  - FTP server: 140.112.28.118 (os2015ktw / ktw2015os)
- The team project should
  - Contain the whole “linux2.6.32.60/” directory
  - Contain your modified test program
  - Contain your report (PDF or DOC, within 4 pages)
  - Be packed as one file named “OSPJ2_Group##.tar.xz”
- **DO NOT COPY THE HOMEWORK**

Contact TAs

- If you have any problem about the projects, you can contact TAs by the following ways:
- Facebook: NTU OS2015 Spring Group
  - https://www.facebook.com/groups/920624997989865/
- E-mail:
  - Chien-Chung Ho: f99922110@csie.ntu.edu.tw

Outline

- Introduction
- Project Requirements
- Submission Rules
- References

References

- Reference Book
  - Professional Linux® Kernel Architecture, Wolfgang Mauerer, Wiley Publishing, Inc.

- Process Scheduling
  - http://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html