

# Process Scheduling

Penny Lee, 8 Apr. 2002

NCTU CIS Operating System lab.

2002 Linux kernel trace seminar



# Outline

- I Introduction
- I Scheduling Policy
  - a. SCHED\_OTHER
  - b. SCHED\_FIFO
  - c. SCHED\_RR
  - d. SCHED\_YIELD
- I The primary function “schedule()”
- I Computing the “goodness” value
  - a. realtime process
  - b. conventional process

# Introduction

- | Two basic kind of scheduler
  - Complex schedulers
  - Quick -and-dirty schedulers--Linux scheduler is this kind
- | Time sharing and processes run “concurrent”
- | CPU-bound and I/O-bound
- | Process Preemption and SMP
- | Scheduling Algorithm
  - FIFO
  - Shortest-Job-First
  - Round-Robin (RR)
  - Etc.

# Scheduling Policy (1)

- | SCHED\_OTHER
- | SCHED\_FIFO
- | SCHED\_RR
- | SCHED\_YIELD

```
/* include/linux/sched.h */  
/* Scheduling policies */  
  
#define SCHED_OTHER      0  
#define SCHED_FIFO      1  
#define SCHED_RR        2  
  
/* This is an additional bit  
set when we want to yield  
the CPU for one re-schedule..*/  
  
#define SCHED_YIELD      0x10
```

# Scheduling Policy (2)

- | SCHED\_OTHER
  - Conventional process
  - Not real-time process
  
- | SCHED\_FIFO
  - Just do it !!
  - Yields the CPU
    - | Blocks on I/O
    - | Higher priority realtime process

# Scheduling Policy (3)

## I SCHED\_RR

- Realtime process
- Round-Robin schedule
- Example in next page.

## I SCHED\_YIELD

- Yield CPU if you want another processes run
- Maybe it cause nonrealtime process get CPU

# Round-Robin Scheduling

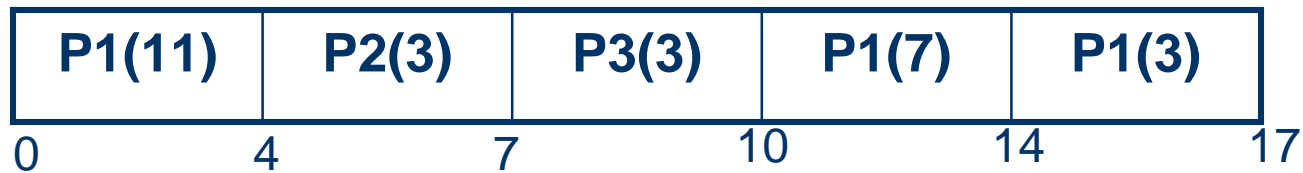
(1)

- | Define “Time quantum”
- | All process in a cycle queue
- | Add new process in the end of queue
- | Interrupt when a process exhausts its time quantum, and move this process to the end of queue
- | Example:

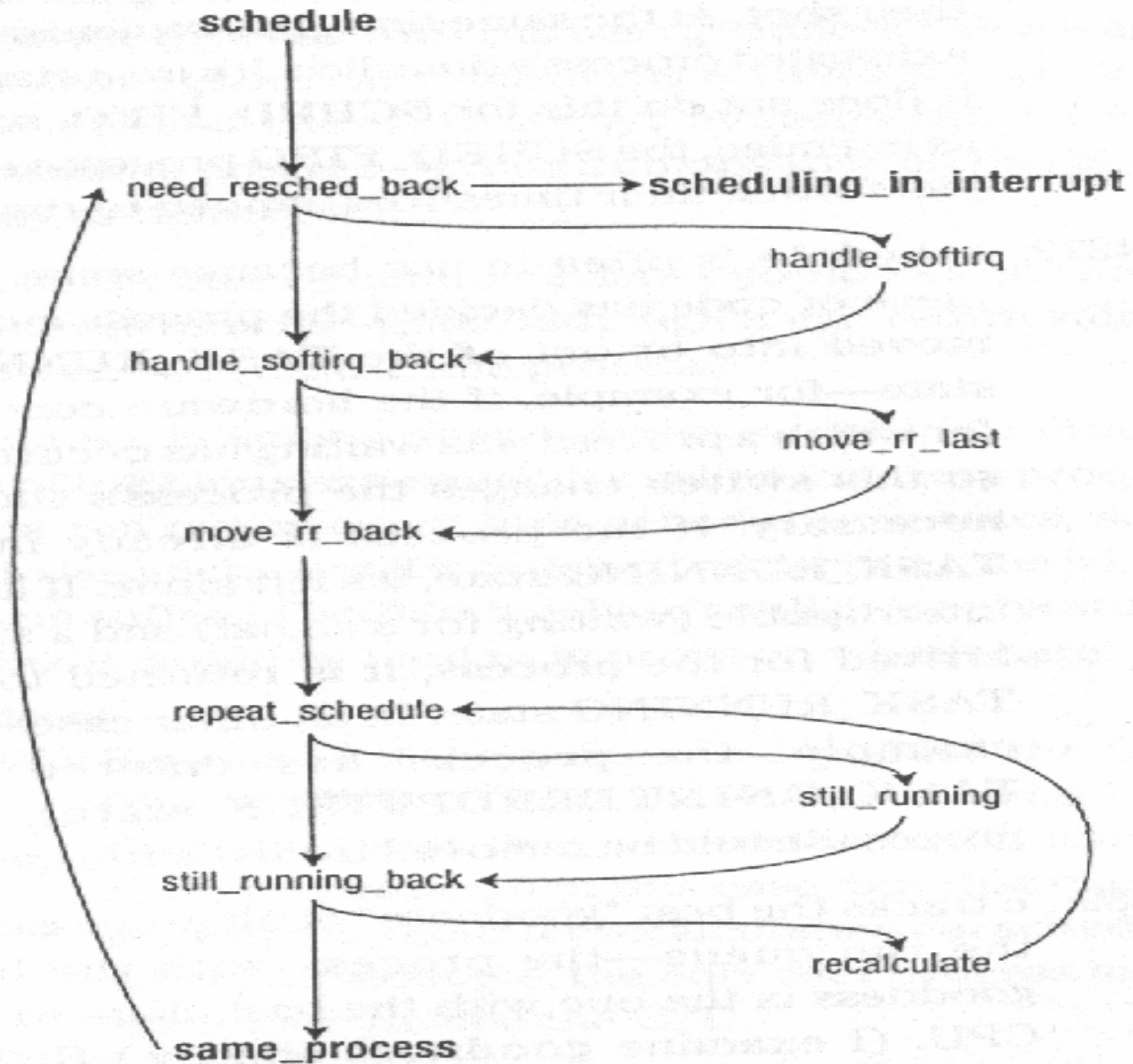
Time quantum  
= 4

Process	Time
P1	11
P2	3
P3	3

# Round-Robin Scheduling (2)



Process	Time
P1	11
P2	3
P3	3



# Schedule()

/kernel/sched.c

```
asmlinkage void schedule(void)
```

```
{
```

```
    need_resched_back:
```

```
    handle_softirq_back:
```

```
    move_rr_back:
```

```
    repeat_schedule:
```

```
    still_running_back:
```

```
    same_process:
```

```
    recalculate:
```

```
    still_running:
```

```
    handle_softirq:
```

```
    move_rr_last:
```

```
    scheduling_in_interrupt:
```

```
    return;
```

```
}
```

**/\* Use “goto” statement skillfully \*/**

# Schedule()

/kernel/sched.c

```
asmlinkage void schedule(void)
{
    struct schedule_data *sched_data;
    struct task_struct *prev, *next, *p;
    struct list_head *tmp;
    int this_cpu, c;
    .....

    need_resched_back:
        prev = current;
        this_cpu = prev->processor;
        if (in_interrupt())
            goto scheduling_in_interrupt;

    .....
    scheduling_in_interrupt:
        printk("Scheduling in interrupt\n");
        BUG();
        return;
}
```

# Schedule()

/kernel/sched.c

```
asmlinkage void schedule(void)
```

```
{
```

```
    .....
```

```
    if (softirq_active(this_cpu) & softirq_mask(this_cpu))  
        goto handle_softirq;
```

```
handle_softirq_back:
```

```
    .....
```

```
handle_softirq:
```

```
    do_softirq();
```

```
    goto handle_softirq_back;
```

```
    .....
```

```
}
```

# Schedule

/kernel/sched.c

asmlinkage void schedule(void)

{

.....

if (prev->policy == SCHED\_RR)  
goto move\_rr\_last;

move\_rr\_back:

.....

move\_rr\_last:

if (!prev->counter) {  
prev->counter = NICE\_TO\_TICKS(prev->  
move\_last\_runqueue(prev);

}

goto move\_rr\_back;

.....

}

```
static int nice= 0;
```

```
#define NICE_TO_TICKS(nice)  
(TICK_SCALE(20-(nice))+1)
```

```
#if HZ < 200
```

```
#define TICK_SCALE(x) ((x) >> 2)
```

```
#elif HZ < 400
```

```
#define TICK_SCALE(x) ((x) >> 1)
```

```
#elif HZ < 800
```

```
#define TICK_SCALE(x) (x)
```

```
#elif HZ < 1600
```

```
#define TICK_SCALE(x) ((x) << 1)
```

```
#else
```

```
#define TICK_SCALE(x) ((x) << 2)
```

```
#endif
```

```
#define HZ 100 /* in i386 */
```

# Schedule()

/kernel/sched.c

```
asmlinkage void schedule(void)
```

```
{
```

```
.....
```

```
move_rr_back:
```

```
switch (prev->state) {
```

```
case TASK_INTERRUPTIBLE:
```

```
if (signal_pending(prev)) {
```

```
prev->state = TASK_RUNNING;
```

```
break;
```

```
}
```

```
case TASK_RUNNING:
```

```
break;
```

```
default:
```

```
del_from_runqueue(prev);
```

```
}
```

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_ZOMBIE          4
#define TASK_STOPPED          8
```

```
static inline int signal_pending(struct
task_struct *p)
```

```
{ return (p->sigpending != 0); }
```

/kernel/sched.c

asmlinkage void schedule(void)

{

.....

repeat\_schedule:

c = -1000;

if (prev->state == TASK

goto still

.....

still\_running:

c = goodness(prev, this\_

next = prev;

goto still\_running\_back;

```
#ifdef CONFIG_SMP
#define idle_task(cpu)
(init_tasks[cpu_number_map(cpu)])
#define can_schedule(p,cpu) \
    ((p)->cpus_runnable & (p)->cpus_allowed & (1 <<
cpu))
```

```
#else
```

```
#define idle_task(cpu) (&init_task)
```

```
#define can_schedule(p,cpu) (1)
```

```
#endif
```

```
still_running_back:
```

```
list_for_each(tmp, &runqueue_head) {
```

```
    p = list_entry(tmp, struct task_struct, run_list);
```

```
    if (can_schedule(p, this_cpu)) {
```

```
        int weight = goodness(p, this_cpu, prev-
```

```
>active_mm);
```

```
        if (weight > c)
```

```
            c = weight, next = p;
```

```
    }
```

```
}
```

# Schedule()

/kernel/sched.c

```
asmlinkage void schedule(void)
```

```
{
```

```
.....
```

```
if (!c)
```

```
    goto recalculate;
```

```
.....
```

```
recalculate:
```

```
{
```

```
    struct task_struct *p;
```

```
    spin_unlock_irq(&runqueue_lock);
```

```
    read_lock(&tasklist_lock);
```

```
    for_each_task(p)
```

```
        p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
```

```
    read_unlock(&tasklist_lock);
```

```
    spin_lock_irq(&runqueue_lock);
```

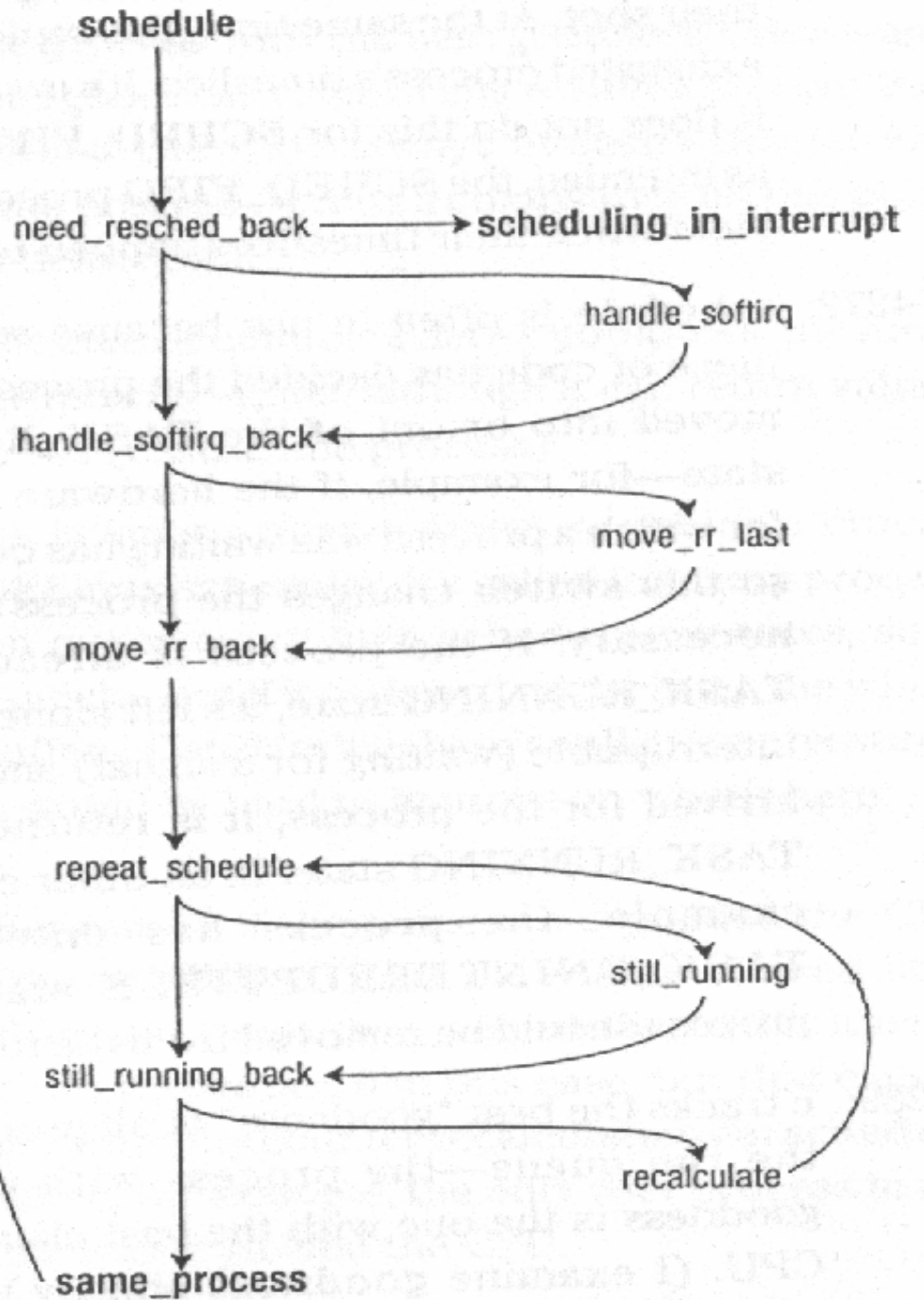
```
}
```

```
goto repeat_schedule;
```

/kernel/sched.c

```
asmlinkage void schedule(void)
{
    .....
    kstat.context_swch++;
    .....
    prepare_to_switch();
    .....
    switch_to(prev, next, prev);
same_process:
    reacquire_kernel_lock(current); /*
    if (current->need_resched)
        goto need_resched_back;
    return;
}
```

S



**Compute the Goodness Value**



# Goodness

- | Two classes
  - Under 1000 : nonrealtime processes
  - Over 1000 : realtime processes
- | Nonrealtime processes usually use the value from 0 to 52.
- | realtime processes usually use from 1001 to 1099.
- | Never return -1000
- | Idle process is -99
- | Goodness function need execute quickly and correctly.

# Goodness()

/kernel/sched.c

```
static inline int goodness(struct task_struct *p, int this_cpu, struct mm_struct *this_mm)
{
    int weight;
    weight = -1;
    if (p->policy & SCHED_YIELD)
        goto out;

    if (p->policy == SCHED_OTHER) {
        weight = p->counter;
        if (!weight)
            goto out;

        .....
    out:
        return weight;
}
```

# Goodness()

/kernel/sched.c

```
static inline int goodness(struct task_struct *p, int this_cpu, struct mm_struct *this_mm)
{
    .....
    #ifdef CONFIG_SMP
        if (p->processor == this_cpu)
            weight += PROC_CHANGE_PENALTY; /* PROC_CHANGE_PENALTY is 15 */
    #endif

    if (p->mm == this_mm || !p->mm)
        weight += 1;
    weight += 20 - p->nice;
    goto out;
} /* nonrealtime process end */

    weight = 1000 + p->rt_priority;
out:
    return weight;
}
```

# System Call for Schedule

System Call	Description
<code>nice( )</code>	Change the priority of a conventional process.
<code>getpriority( )</code>	Get the maximum priority of a group of conventional processes.
<code>setpriority( )</code>	Set the priority of a group of conventional processes.
<code>sched_getscheduler( )</code>	Get the scheduling policy of a process.
<code>sched_setscheduler( )</code>	Set the scheduling policy and priority of a process.
<code>sched_getparam( )</code>	Get the scheduling priority of a process.
<code>sched_setparam( )</code>	Set the priority of a process.
<code>sched_yield( )</code>	Relinquish the processor voluntarily without blocking.
<code>sched_get_priority_min( )</code>	Get the minimum priority value for a policy.
<code>sched_get_priority_max( )</code>	Get the maximum priority value for a policy.
<code>sched_rr_get_interval( )</code>	Get the time quantum value for the Round Robin policy.

# Reference

- | **Linux Core Kernel Commentary**  
**second edition**
- | **Understanding the LINUX KERNEL**  
**O'reilly**
- | **Cross-Referencing Linux**
  - <http://lxr.linux.no/>